Information Systems 73 (2018) 61–78

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Multi-party business process compliance monitoring through IoT-enabled artifacts

Giovanni Meroni[a,*], Luciano Baresi[a], Marco Montali[b], Pierluigi Plebani[a]

[a] Politecnico di Milano, Piazza Leonardo da Vinci, 32, Milano 20133, Italy
[b] Free University of Bozen-Bolzano, Piazza Università, 1, Bolzano 39100, Italy

ARTICLE INFO

Article history:
Received 21 February 2017
Revised 23 November 2017
Accepted 29 December 2017
Available online 30 December 2017

Keywords:
Business process compliance
Runtime compliance monitoring
Internet of things
Guard-Stage-Milestone
E-GSM
Declarative languages
Artifact-centric languages
Business process model transformation

ABSTRACT

Monitoring the compliance of the execution of multi-party business processes is a complex and challenging task: each actor only has the visibility of the portion of the process under its direct control, and the physical objects that belong to a party are often manipulated by other parties. Because of that, there is no guarantee that the process will be executed — and the objects be manipulated — as previously agreed by the parties.

The problem is usually addressed through a centralized monitoring entity that collects information, sent by the involved parties, on when activities are executed and the artifacts are altered. This paper aims to tackle the problem in a different and innovative way: it proposes a decentralized solution based on the switch from control- to artifact-based monitoring, where the physical objects can monitor their own conditions and the activities in which they participate.

To do so, the Internet of Things (IoT) paradigm is exploited by equipping physical objects with sensing hardware and software, turning them into smart objects. To instruct these smart objects, an approach to translate classical Business Process Model and Notation (BPMN) process models into a set of artifact-centric process models, rendered in Extended-GSM (E-GSM) (our extension of the Guard-Stage-Milestone (GSM) notation), is proposed.

The paper presents the approach, based on model-based transformation, demonstrates its soundness and correctness, and introduces a prototype monitoring platform to assess and experiment the proposed solution. A simple case study in the domain of advanced logistics is used throughout the paper to exemplify the different parts of the proposal.

1. Introduction

Modern organizations are more and more required to become open, reactive, and flexible entities able to satisfy the ever-changing needs of their customers. This is why they are redesigning their internal structures and business processes to increase dynamism and be open to cooperate with new organizations. Many business processes — once internal to single organizations — now cross the boundaries of single organizations and require the coordination among different, potentially changing actors. This transformation heavily impacts on how the process is executed. Organization no longer have full control on the whole process. Instead, they control only the portion of that process that is assigned to them. At the same time, the physical objects belonging to an organization can now be manipulated by the other actors, and the ownership of these objects can change while the process is performed.

To ensure that organizations coordinate properly, and that physical objects are correctly handled, the correctness and compliance of these distributed processes has to be monitored. In particular, the execution order and the successful execution of the activities composing the process have to be checked. To automate and keep track of business processes, organizations deploy Business Process Management Systems (BPMSs). In fact, today's BPMSs include a monitoring module to oversee the execution of fully automated business processes that can be confined within a single party. BPMSs also provide dashboards to inform the process owner of the current status, bottlenecks, and possible alerts.

Unfortunately, when moving to multi-party processes, the BPMS of each organization can only manage the activities under its control, but it has no jurisdiction on the activities carried out by the other parties. Consequently, it can only monitor the process portions carried out by the organization. This limitation is traditionally addressed by federating the BPMSs, or by deploying a centralized

* Corresponding author.
E-mail addresses: giovanni.meroni@polimi.it (G. Meroni), luciano.baresi@polimi.it (L. Baresi), montali@inf.unibz.it (M. Montali), pierluigi.plebani@polimi.it (P. Plebani).

https://doi.org/10.1016/j.is.2017.12.009
0306-4379/

one. However, these solutions lack flexibility, as whenever a new party is introduced, leaves, or the process changes, the underlying infrastructure must be heavily reconfigured.

When activities are automated, the BPMS is in charge of executing them. Therefore, it exactly knows when such activities start and when they finish, and which is their outcome. However, when dealing with non-automated activities, a BPMS relies on human operators to know about the outcome of such activities. As these operators could forget to notify the events of interest, they could make mistakes, or they could even intentionally postpone, fake, or alter provided inputs, monitoring manual activities can be unreliable. This has an impact not only to the party in charge of executing these activities, but also to the other connected parties.

To overcome these issues, this paper proposes a novel approach to autonomously and continuously monitor multi-party business processes in a distributed way. To this aim, we move the monitoring tasks directly onto the artifacts, i.e., the physical objects that participate in the process, which are equipped with sensors and computing devices, thus becoming "smart".

By doing so, these smart objects can autonomously keep track of all the activities in which they were involved, regardless of the organization performing them. Additionally, smart objects can track all the changes in their states, i.e., their conditions, throughout the execution of the process. This way, a smart object can autonomously monitor the compliance of the process it participates in, as well as its own lifecycle, that is, the transitions from one state to a new one that are expected to occur while the process is executed. On this basis, the characterizing contributions of the proposed solution are the following:

- We combine control-flow analysis, as defined using Business Process Model and Notation (BPMN), and artifact-centric analysis, as defined using Extended-GSM (E-GSM), our extended version of the Guard-Stage-Milestone (GSM) notation [1]. The user starts defining the multi-party process in BPMN, a widely known process modeling language. Then, for each artifact, E-GSM models suited to monitor the process and the lifecycle of the artifact are semi-automatically derived from the BPMN model. The combination of these two perspectives allows one to predicate on both executions and involved artifacts. If we say that an execution is *compliant* if it evolves through the foreseen control flow, and an artifact is *compliant* if it evolves according to its lifecycle, our solution can distinguish among *(i)* compliant executions that produce compliant artifacts, *(ii)* non-compliant executions that lead to compliant artifacts, and *(iii)* non compliant executions that lead to non-compliant artifacts.
- We adopt *smart* objects (a-la IoT) to transform artifacts into active entities that can both enact the E-GSM models and communicate with the others. The former capability means that each artifact (smart object) can: *(i)* infer its current state, *(ii)* know the admissible next states, and *(iii)* know the order in which the process' activities should manage it.
- We propose an innovative architecture for the distributed execution and monitoring of multi-party processes that embed the characteristics highlighted above. The proposed architecture is based on the use of simple Single-board Computer (SBC) boards, such as the Raspberry PI and the Intel Galileo, and exploits `Node.js` as implementation language.

All the key features of the proposed solution are exemplified through a (simplified without being trivial) real example process borrowed from the domain of advanced logistics. The same process is also used for accessing the solution.

With respect to our previously published articles, this article extends the E-GSM based process monitoring approach presented in [2] by proposing a structured approach to instruct the monitoring platform, and providing an implementation of the solution. It also extends the BPMN to E-GSM translation presented in [3] and [4] by formalizing how to extract the portion of the process relevant to each participating object, and to also represent in E-GSM the lifecycle of each object. Compared to the monitoring approach described in [5] and [6], which focuses mostly on runtime conformance checking, the main focus of this article is on monitoring how physical objects are impacted by the process execution. Therefore, it extends the approach to also monitor the lifecycle of these objects, and to detect compliance violations even if they do not explicitly violate the control flow. The monitoring architecture presented in this article differentiates from mArtifact [7] by running directly on top of the smart objects participating in the process.

The rest of the paper is organized as follows. Section 2 discusses the limitations of current monitoring approaches and presents the main elements of our solution by means of a concrete case study, then used consistently throughout the paper. Section 3 describes the proposed extensions to the E-GSM notation, then exploited in Section 4, which presents our approach. Section 5 argues about the correctness of the automated transformation of BPMN models into E-GSM ones, while Section 6 introduces the distributed architecture defined for supporting the presented process compliance monitoring solution. Section 7 analyzes the state of the art and Section 8 concludes the paper.

2. Motivations

Fig. 1 shows the BPMN representation of a real multi-party process, taken from the logistics domain. It describes the initial phase of a multimodal transport. At first, the *Carrier*, the entity responsible for the physical shipment of the goods, collects an empty shipping container from the warehouse of the *Multimodal Transport Operator (MTO)*, which is in charge of organizing the entire shipment, and ships it to the *Producer* of the goods. In parallel, the *Producer* prepares the goods and produces the documentation for the shipment. Once the *Carrier* reaches the *Producer*'s site, the *Producer* loads the goods onto the container, verifies that all the documents are correct and, if not, updates them. Finally, the *Carrier* starts the shipment. Both the *MTO* and the *Producer* verify the identity of the *Carrier* before granting it access to their sites.

This BPMN model is treated as an agreement between the various organizations. Process portions carried out by each organization (i.e., the ones inside the pools) are disclosed, and the other organizations agree on how the whole process is executed. Therefore, no privacy restriction holds on this process model, which is shared among the participating organizations.

To know if this process is correctly executed, organizations have to both monitor their internal activities (i.e., the ones under their control) and verify that their objects were correctly manipulated by the other organizations. Since each party can already monitor its own process portions, the focus of this paper is on monitoring the objects. To this aim, for each object (e.g., the goods, the container), it is necessary to monitor the activities involving that object. This way, it is possible to know the exact steps that caused an object to be in its current conditions. It is worth noting that, since an object may be manipulated by organizations other than the owner, monitoring only internal activities is not sufficient. For example, although it belongs to the *MTO*, the container is manipulated by both the *Carrier* and the *Producer*. Therefore, activities belonging to other organizations, as long as they interact with the objects, should be monitored as well.

Additionally, the conditions of the objects have to be also monitored, and anomalies have to be promptly notified. For example, in case of drugs, the producer may want to be sure that the temperature of the goods remains stable during the whole transportation. These objects (i.e., those that must be monitored) are rendered as

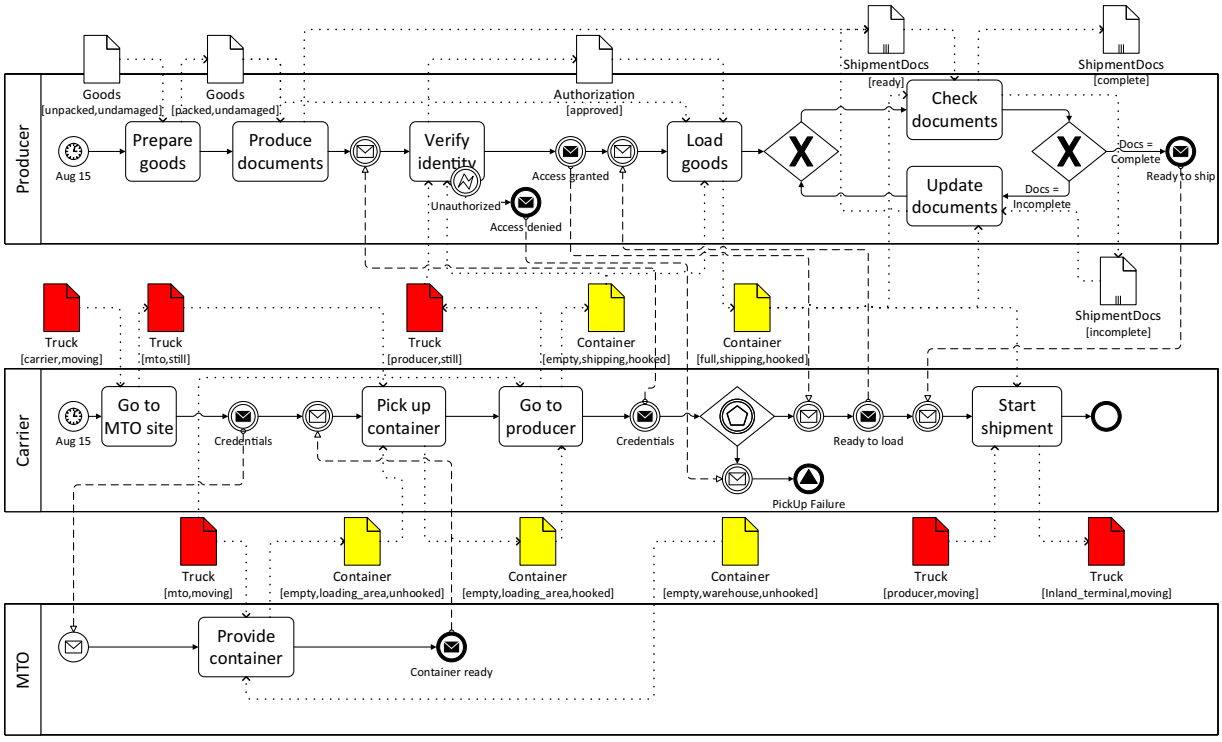

Fig. 1. BPMN diagram of the running example (artifacts that should be monitored are highlighted).

artifacts, while their conditions are discretized into a set of *states*. For example, a container may be `empty`, at the `wharehouse`, and `unhooked`, thus having state *[empty,warehouse,unhooked]*. The states that an artifact can assume at a certain point in time and the transitions from one state to another one represent the lifecycle of that artifact (i.e., how it can evolve while the process is executed).

A comprehensive solution must be able to monitor both the execution of processes and the evolution of their artifacts, and to reason on the mutual effects since a deviation in the execution may have no impact on the artifacts, and vice versa.

2.1. Imperative vs. declarative process models

The vast majority of business processes is modeled by means of *imperative* languages, like BPMN. These languages require the user to model all the possible execution flows, and consider any other unspecified flow as *non-compliant* (with the model). Such models can then render standardized business processes properly and guide the BPMS to enact the process.

When a deviation in the process' execution occurs (e.g., an activity that is not supposed to be executed begins), a BPMS detects a failure in the execution since it does not comply with the model any longer. The BPMS does not know how to continue the execution, and it can only stop it or continue with no guarantees. This is not an issue when the process is managed by a single BPMS. The BPMS autonomously decides when activities should be executed, and as such it knows when they were started or finished. However, when the process is executed by multiple parties, each BPMS needs explicit notifications to determine when activities outside its jurisdiction are executed. When a process involves multiple parties, and their BPMSs are not configured to exchange information, or when one party does not have a BPMS, this information is not available. Therefore, the monitoring either cannot take place or is partial and inaccurate. In addition, the process may comprise activities carried out by humans, who are also in charge of informing the BPMS about their execution. If they forget, or do not do it intentionally, the BPMS is not be able to know it.

Conformance checking tools try to overcome these limitations by inferring process models from execution traces, where BPMSs have clearly marked the beginning and end of each activity. These models are then compared against the original ones to identify both correct executions and deviations. Most of these tools work offline and thus can only perform post-mortem analysis, while it would be desirable that organizations were informed as soon as violations materialize to be able to take countermeasures. In addition, conformance checking tools can only know about activities whose execution was recorded in the traces. Therefore, the problem of identifying when activities start or end without relying on explicit notifications remains unsolved.

The aforementioned issues do not apply to *declarative* languages as they do not impose the complete identification of all the possible execution flows, but they only require the identification of the conditions that determine when an activity should be executed. This way, it is possible to detect the execution of activities regardless of the execution flow, and it is also possible to define under which conditions an activity will be executed, thus not relying solely on start and end notifications. However, [8] points out that declarative models are usually more complex to define and harder to understand than imperative ones. This is why most of the existing process models have been defined using imperative languages, either to ease their documentation or to enable their execution by means of conventional BPMSs.

To balance ease of design and effectiveness of monitoring, our approach includes: *(i)* an extension of the GSM notation [1], called E-GSM, which augments the original notation with the control flow defined in BPMN[1] (see Section 3); *(ii)* a complete, tool-supported

[1] We chose BPMN because it is the de-facto standard for imperative process modeling. However, our approach can also be applied to some extent to other imperative process models, such as Event-driven Process Chains (EPCs) or Unified Modeling Language (UML) Activity Diagrams.

solution for translating a BPMN model into an E-GSM equivalent specification (see Section 4).

2.2. IoT as monitoring means

The role of artifacts in multi-party business processes is twofold. On the one hand, they define how an object, owned by one party, should also be managed by other parties. On the other hand, their state can suggest if an activity has started or finished. For example, an empty container parked at the loading area (state *[empty,loading_area,hooked]* in Fig. 1) suggests that the activity in charge of picking up the container (*Pick up container*) has terminated, while activity *Go to producer* is ready to start when container has the same state as before and truck is in state *[producer,still]*).

Nowadays, the IoT [9] provides readily-available solutions for interacting with distributed objects remotely. In particular, the IoT turns physical objects into smart objects, equipped with sensors, SBCs, and network connectivity interfaces. This is why we argue that the IoT can be used to "augment" the artifacts of interest, and become able to trace and control how they evolve, that is, their state. The sensing capabilities help collect all the information relevant to the state of artifacts. The computational capabilities allow one to run the monitoring solution directly onto the objects, thus removing the bottleneck of centralized monitoring. Network connectivity capabilities allow objects to communicate both with each other in a peer-to-peer way and with the information systems of involved parties to easily distribute state information.

As result, instead of having a centralized entity in charge of enabling the communication among the parties needed to monitor the artifacts, we propose an approach where the monitoring is directly performed on the artifact themselves, by turning them into smart objects. The proposed approach (see Section 4) shows how the E-GSM models that result from transforming the original BPMN one are assigned to the smart objects. This requires that each smart object be equipped with an E-GSM engine (see Section 6) able to process the assigned model. The E-GSM engine is then able to monitor the execution of the process, as well as the lifecycle of the artifact.

3. E-GSM

We selected the GSM notation [1] as starting point for our solution since it natively provides constructs — **Guards** and **Milestones** — to identify when a process portion, called **Stage**, should start or end, respectively. **Guards** and **Milestones** are Event-Condition-Action (ECA) rules that can predicate on control flow dependencies, external events, or data. Therefore, GSM can use both the information that is generated from inside the engine that runs the model (e.g., execution state) and the one coming from the outside (e.g., sensor data) to infer when **Stages** are executed. **Stages** represent the units of work that are carried out during the execution of a process. **Stages** can be nested and, if a **Stage** has no nested ones, it is *atomic* and represents a single task. As soon as one of the associated **Guards** is satisfied, a **Stage** is declared opened, that is, it starts its execution. When one of its **Milestones** is met, the **Stage** is declared closed, that is, its execution ends.

However, GSM lacks constructs to define the expected execution flow, which is required to model the execution order among activities (i.e., *Stages*) and thus to detect compliance violations. Also, there is no way to detect if something goes wrong while executing a **Stage**. Our extension, called E-GSM [2] aims to cover these limitations. Fig. 2 shows a graphical representation of the main elements proposed by E-GSM that are relevant for the purposes of this paper.[2]

While GSM only uses **Guards** to decorate a **Stage**, E-GSM distinguishes between **Data Flow Guards** and **Process Flow Guards**. The former borrow their semantics from GSM **Guards**. The latter are boolean expressions that predicate on the activation of **Data Flow Guards** and **Milestones**. As such, they allow one to define control flow dependencies among **Stages. Process Flow Guards** are evaluated when one of the **Data Flow Guards** associated with the same **Stage** is triggered, and before the **Stage** becomes opened, that is, it starts executing. If the predicate is true, the **Stage** complies with the expected execution, otherwise the activation of the **Stage** does not respect the execution flow. Note that **Process Flow Guards** differ from **Data Flow Guards** as they do not cause a **Stage** to become opened, and as such they do not force any execution flow.

For example, to determine when the carrier starts going to the producer based on the state of the container and the truck, **Stage** `GoToProducer` is decorated with a **Data Flow Guard** requiring the truck to have state *[mto,moving]* and the container *[empty,loading_area,hooked]*. This way, when the truck and the container assume these states, `GoToProducer` is opened. On the other hand, to indicate that the carrier should normally start going to the producer after the container has been picked up, `GoToProducer` is decorated with a **Process Flow Guard** requiring the achievement of one of the milestones of stage `PickUpContainer`. This way, whenever `GoToProducer` starts, the execution and completion of `PickUpContainer` is verified and, if `PickUpContainer` was not executed or completed, `GoToProducer` can be flagged as incorrectly executed.

E-GSM also adds **Fault Loggers**, which are ECA rules that cause the associated **Stages** to be declared faulty. In other words, they define when the execution of **Stages** should be considered irregular. The satisfaction of a **Fault Logger** does not imply the termination of the **Stage**, as the termination is only determined by **Milestones**.

For example, to determine when the producer stops verifying the identity of the carrier, **Stage** `VerifyIdentity` is decorated with a **Milestone** requiring the authorization to have state *[approved]*. On the other hand, to determine if the verification was unsuccessful, a **Fault Logger** predicating on the occurrence of event *unauthorized* is added to `VerifyIdentity`.

Finally, each **Stage** must be augmented with at least one **Data Flow Guard** and one **Milestone**, and may have one or more **Process Flow Guards** and **Fault Loggers**.

The right portion of Fig. 2 sketches the lifecycle of an E-GSM **Stage**, that is, all the possible states that a stage may assume, organized around three main orthogonal execution perspectives: status, outcome, and compliance:

- The *status* is driven by **Data Flow Guards** and **Milestones**. Initially, every **Stage** is *unopened*. An *unopened* or *closed* **Stage** becomes *opened* once one of its **Data Flow Guards** is triggered ($\texttt{S.DFG}_i$), if its parent **Stage** is *opened*. An *opened* **Stage** becomes *closed* if either one of its **Milestones** is achieved ($\texttt{+S.M}_j$), or its parent **Stage** becomes *closed*.
- The *outcome* is driven by **Fault Loggers**. Initially, every **Stage** is *regular* and becomes *faulty* when one of its **Fault Loggers** is triggered ($\texttt{S.FL}_l$).
- The *compliance* is driven by **Process Flow Guards**. Initially, every **Stage** is *onTime*. An *onTime* **Stage** becomes *outOfOrder* when one of its **Data Flow Guards** is triggered, but none of its **Process Flow Guards** holds ($\texttt{S.DFG}_i$ and not($\texttt{S.PFG}_k$)). Once a **Stage** `S'` is declared *outOfOrder*, every other *onTime* **Stage** `S` that

[2] The interested reader can refer to [10] for a detailed presentation of the proposed notation.

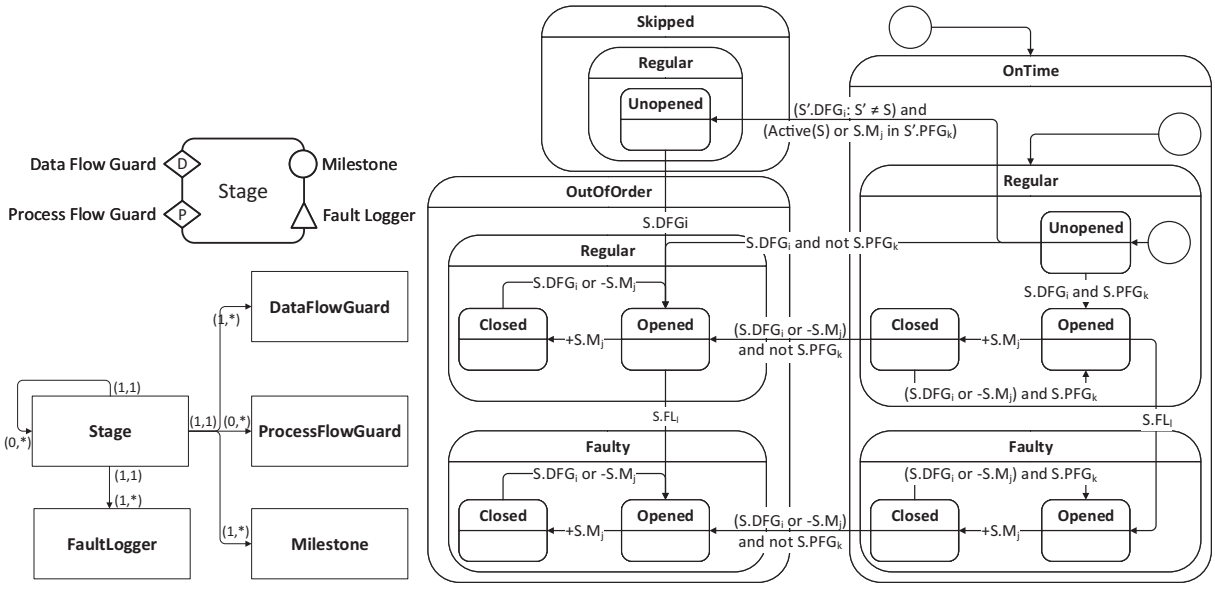

Fig. 2. E-GSM meta-model (bottom left), graphical representation (top left) and lifecycle of a **Stage** (right).

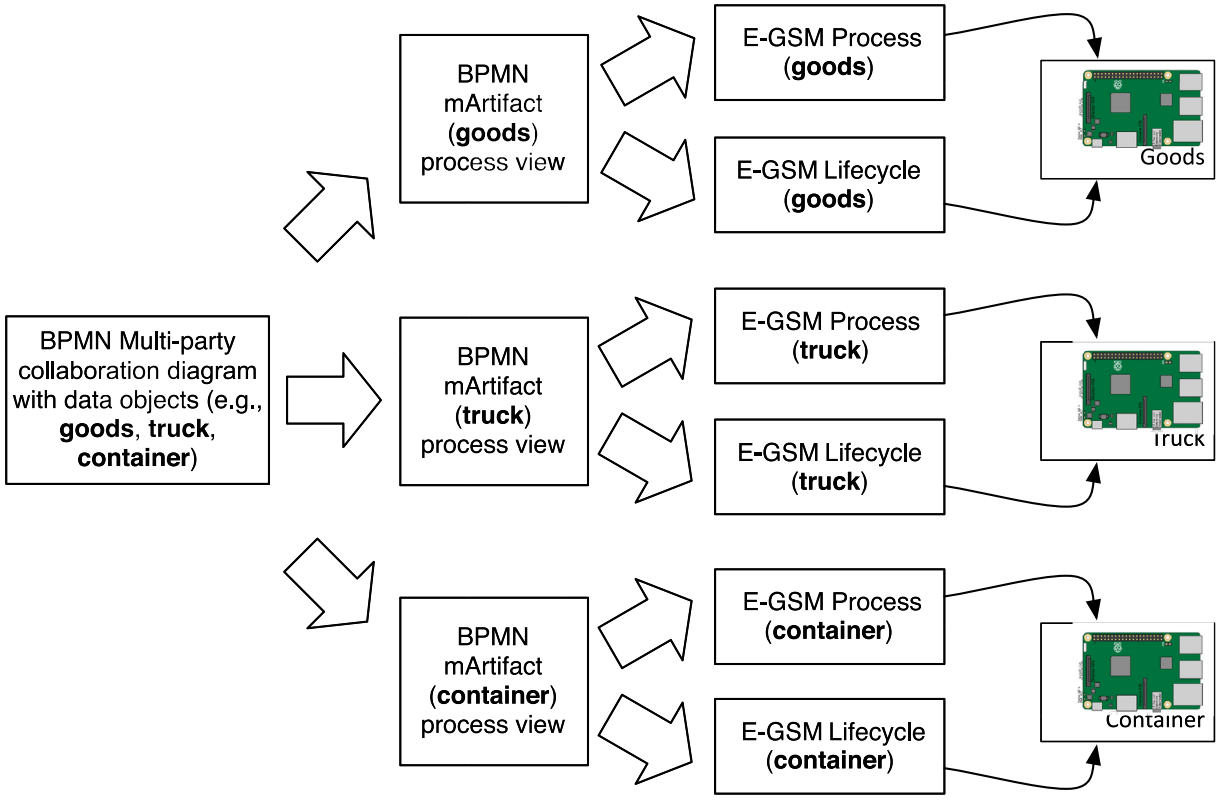

Fig. 3. Overview of the transformation process.

should precede S' ($S.M_j$ or Active(S) $\in$ S'.PFG_k) is declared *skipped*. A *skipped* **Stage** becomes *outOfOrder* once one of its **Data Flow Guards** is triggered ($S.DFG_i$).

4. From a BPMN process to the monitored artifacts

The proposed approach starts from a multi-party process modeled in BPMN and guides towards the definition of the E-GSM models used to oversee the execution of both the process and its artifacts. More precisely, as shown in Fig. 3, the starting point is a BPMN collaboration diagram [11], that specifies which portions of the process are carried out by which organizations, and how these organizations coordinate with each other. Artifacts are included in the BPMN collaboration diagram and represent resources that are manipulated and exchanged by the parties. Since the whole process has to be transparently shared among organizations, the BPMN collaboration diagram has to include all the activities interacting with artifacts.

Besides monitoring their process portions, organizations may be interested in knowing how an artifact is managed, and if its state evolves as expected. We call these artifacts *monitored artifacts* (mArtifacts hereafter). For each mArtifact, our solution requires that: *(i)* the mArtifact-oriented view of the process (i.e., the portion of the process relevant for that artifact) be extracted and, based on that process view, the E-GSM models representing *(ii)* the process model (i.e., the activities and their relationships) and *(iii)* the lifecycle of the mArtifact (i.e., all the admissible states and transitions) be generated. To monitor the mArtifact, smart objects related to it embed an E-GSM engine fed with the two E-GSM processes derived from the initial process. This way, the mArtifact can autonomously monitor the correct execution of the process and the evolution of its lifecycle.

The transformations performed in these steps are described in Sections 4.1, 4.2, and 4.3 respectively. Since steps (ii) and (iii) do not require human intervention, a dedicated tool[3] is in charge of them. Finally, Section 6 gives details on the characteristics of the E-GSM engine running on the smart object.

4.1. Extraction of the mArtifact-oriented process view

This step identifies the minimum set of information needed for monitoring a mArtifact, thus it excludes all the activities that do not affect or are not affected by the mArtifact. We start from a BPMN collaboration diagram and the following assumptions:

- Each artifact must be modeled in the process using BPMN data objects. The different states that the artifact may assume when the process is executed are expressed with the data state property of data objects.
- Each activity must have at least one input (output) data object. The activity is supposed to start (finish) only when all its input (output) data objects exist and are in the specified state. If an activity has two input (output) data objects that refer to the same artifact in different states, the artifact must assume one of these states.
- Data associations must not contradict the semantics of the control flow, as they are used to identify when activities start or end. For example, two activities cannot share the same set of data objects in the same data state as input if they belong to a sequence. They would always be detected to be executed at the same time, which would contradict the control flow. In contrast, if they were executed in parallel, they could share the same inputs.
- In case of parallel or inclusive branches, output data objects that refer to the same artifact must only be associated with activities that belong to the same branch. Otherwise, it would not be possible, based only on the process model, to determine which change in the state of the artifact occurs first. Therefore, the behavior of the artifact would be non-deterministic.

Given these hypotheses and a mArtifact, we derive a BPMN process model where the activities are organized according to the mArtifact's standpoint. Fig. 4 shows the main steps to be performed in order to obtain such a process view. This new representation will be the one monitored by the mArtifact. To derive this process, only the BPMN elements directly related to the mArtifact are considered. So, we:

- Keep only those activities that act as inputs or outputs to the mArtifact.
- Maintain the events that refer to the mArtifact, as well as all the events responsible for the collaboration among stakeholders (i.e., message events that have an inbound or outbound message flow).
- Keep those data objects that do not represent the mArtifact, but that are inputs to (outputs from) the activities that refer to the mArtifact.

In addition, to ensure that the BPMN model representing the mArtifact view is well-formed and, except for the absence of activities and events not related to the mArtifact, dependencies among activities are analogous to the ones in the original BPMN collaboration diagram, we:

- Replace the boundary blocking (non-blocking) events attached to discarded activities with exclusive (inclusive) split gateways with no branch condition.
- Remove pools, as well as the control flow that directly connects a message throw event to a message catch event, and replace message flows with control flows.
- Delete message throw (catch) events or replace them with parallel split (merge) gateways if they have multiple outbound (inbound) flows.
- Add a generic start (end) event if the resulting model has no start (end) event, and connect it to elements with only outbound (inbound) control flows.

The result of this transformation is a process model for each mArtifact of interest that complies with a BPMN process diagram. Note that this new model may not be well-structured. Remarkably, when message exchanges in the original BPMN collaboration diagram are not synchronous (i.e., after sending a message the process executes an activity instead of waiting for a response), the resulting process is always unstructured. In such a case, manual intervention is required to make it well-structured.

Example. Fig. 5 shows the process model obtained by applying these steps onto the process presented in Section 2, where the shipping container and the truck are the mArtifacts of interest. For the container, activities `Go to MTO site`, `Prepare goods`, `Produce documents`, and `Verify identity` have been removed since they do not use the container. Similarly, all timer and signal events have been removed. Data objects `Truck[carrier,moving]` and `Goods [unpacked,undamaged]` have also been removed, since none of the remaining activities uses them. Likewise, for the truck, activities `Prepare goods`, `Produce documents`, `Load goods`, `Verify identity`, `Check documents`, and `Update documents`, all timer and signal events, and data objects `Goods [unpacked,undamaged]`, `Goods[packed,undamaged]`, `ShipmentDocs[ready]`, `ShipmentDocs[complete]`, and `ShipmentDocs[incomplete]` have been removed.

4.2. Generation of the E-GSM process model

Due to the limitations of using imperative languages for monitoring purposes, as discussed in Section 2.1, the derived mArtifact-oriented BPMN process view cannot be directly used to instrument the monitoring platform. However, this view can be translated into E-GSM, which supports a higher level of flexibility than BPMN. In particular, control flow dependencies, which are prescriptive in BPMN, are relaxed in the E-GSM model and used only to assess compliance. Similarly, data objects, which in BPMN are only used for documentation, are used to define the conditions that determine the activation or termination of associated activities. It would then be impossible to use BPMN to achieve the same level of flexibility as the E-GSM model without adopting a new semantics for the notation, and thus a new engine.

To perform such a translation, the following rules are applied:

[3] The translator is publicly available at https://bitbucket.org/polimiisgroup/bpmn2egsm.

```
1:  function PRODUCEPROCESSVIEW(collaborationDiagram,artifact)
2:      processView = copy(collaborationDiagram);                    ▷ duplicate the source BPMN model
3:      for all i=1: processView.elements.count do                   ▷ examine each element in the source BPMN model
4:          keep = false;
5:          if processView.elements.get(i).type == 'Activity' or processView.elements.get(i).type == 'Event' then
6:              if processView.elements.get(i).inputDataObjects.contains(artifact) or processView.elements.get(i).outputDataObjects.contains(artifact) then   ▷ element references the artifact
7:                  keep = true;
8:              end if
9:              if processView.elements.get(i).type == 'StartEvent' or processView.elements.get(i).type == 'EndEvent' or processView.elements.get(i).type == 'IntermediateEvent' then
10:                 if processView.elements.get(i).messageTo != null or processView.elements.get(i).messageFrom != null then   ▷ element is not responsible for the collaboration
11:                     keep = true;
12:                 end if
13:             end if
14:         end if
15:         if keep == false then                                    ▷ make discarded element an orphan
16:             BPMNModel.elements.get(i).predecessor.successor = BPMNModel.elements.get(i).successor;
17:             BPMNModel.elements.get(i).successor.predecessor = BPMNModel.elements.get(i).predecessor;
18:             BPMNModel.elements.get(i).predecessor = null;
19:             BPMNModel.elements.get(i).successor = null;
20:         end if
21:     end for
22:     removePools(processView);                                    ▷ remove pools from process model
23:     removeOrphans(processView);                                  ▷ remove discarded elements (i.e., orphans)
24:     makeWellformed(processView);                                 ▷ make target model well-formed (if not already so)
25:     return processView;
26: end function
```

Fig. 4. (Simplified) algorithm to produce, given a BPMN collaboration diagram and an artifact, the process view.

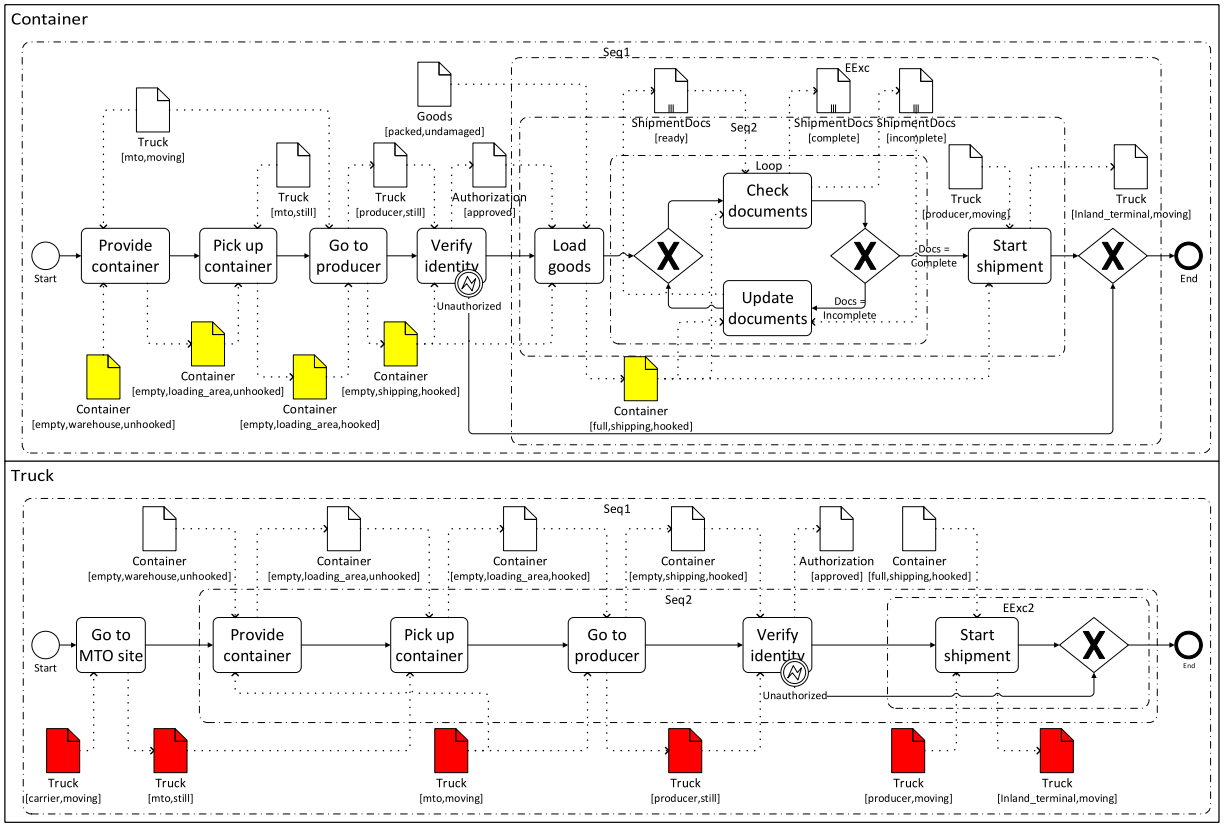

Fig. 5. BPMN process model from the viewpoint of the container (top) and the truck (bottom).

- We create an E-GSM **Stage** S for each BPMN activity A in the model.
- The ECA rule that defines the **Data Flow Guard** (**Milestone**) of S is triggered when a change in the state of one of the artifacts Ar associated with each input (output) data object of A occurs, and Ar enters the current (leaves the previous) state. It will only be fired if the state assumed by all artifacts Ar is the one indicated by the input (output) data objects of A.
- If the activity has a boundary event, we add a **Fault Logger** triggered by the event, attach it to the **Stage**, and, if the boundary event is interrupting, we define an additional **Milestone** triggered by the event.

```
lazy rule produceDFG {
  from expr: String, stage: String
  to rel: XGSM!DataFlowGuardType (
    id <- stage + '_dfg1',
    expression <- expr )}

lazy rule produceM {
  from expr: String, stage: String
  to res: XGSM!MilestoneType (
    id <- stage + '_m1',
    expression <- expr )}

lazy rule produceFL {
  from expr: String, stage: String
  to rel: XGSM!FaultLoggerType (
    id <- stage + '_fl1',
    expression <- expr )}

rule activity2substage {
  from s: BPMN!Activity
  to
    tpfg: EGSM!ProcessFlowGuardType (
      id <- s.id + '_pfg',
      expression <- s.producePFGExpression() ),
    tss: EGSM!SubStageType (
      id <- s.id,
      processFlowGuard <- tpfg
      dataFlowGuard <- OrderedSet{produceDFG(s.id, 'on␣' + 'or'.concatStrings(
        self.dataInputAssociations -> iterate (g; ret: OrderedSet(String) =
        OrderedSet{} | ret -> including(g.sourceRef.name+'_e')) + '␣if␣' + ret
        -> including('('+ 'or'.concatStrings(self.dataInputAssociations
          -> iterate(f; ret: OrderedSet(String) = OrderedSet{} |
          if (f.sourceRef.name=e.sourceRef.name) then
            ret -> including(f.sourceRef.name+ '[' +
              f.sourceRef.dataState.name + ']')
          else
            ret
          endif
        )) + ')' ))},
      milestone <- OrderedSet{produceM(s.id, 'on␣' + 'or'.concatStrings(
        self.dataOutputAssociations -> iterate (g; ret: OrderedSet(String) =
        OrderedSet{} | ret -> including(g.targetRef.name+'_l')) + '␣if␣' + ret
        -> including('('+ 'or'.concatStrings(self.dataOutputAssociations
          -> iterate(f; ret: OrderedSet(String) = OrderedSet{} |
          if (f.targetRef.name=e.targetRef.name) then
            ret -> including(f.targetRef.name + '[' +
              f.targetRef.dataState.name + ']')
          else
            ret
          endif
        )) + ')' ))} -> including(s.getBlockingBoundaryEvents()
        -> collect(b | produceM(s.id + '_' + b.id, 'on␣' + b.id)))),
      faultLogger <- s.boundaryEventRefs ->
        collect(e | produceFL(s.id + '_' + e.id ,'on␣' + e.id))
    )
}
```

Fig. 6. (Simplified) ATL translation rule to derive from a BPMN Activity the corresponding E-GSM construct.

- We create an E-GSM **Stage** S' for each BPMN start, end, or intermediate event. The ECA rules that correspond to the **Data Flow Guard** and **Milestone** of S' will be fired as soon as the event occurs.
- Finally, if the original process model is organized in nested blocks, a new **Stage** is created and wraps the **Stages** derived from the inner blocks. Control flow and exceptional flow patterns are managed accordingly [10].

To formalize these rules, and, consequently, automate the translation, we adopted ATLAS Transformation Language (ATL) [12]. Fig. 6 shows an excerpt of the implemented translator,[4] related to the transformation from a BPMN activity to the correspond-

[4] For the sake of clarity, in this paper we report only a simplified excerpt of the source code. A complete version is publicly available at https://bitbucket.org/polimiisgroup/bpmn2egsm/src/206dd0270c4f32a7997d847356b0397fb283aac4/BPMN2GSM/BPMN2XGSM.atl.

ing E-GSM constructs. In particular, rule `activity2substage` is responsible for transforming each activity in the source BPMN process view into a **Stage** in the target E-GSM process model. `activity2substage` also produces, for each activity, a **Process Flow Guard** that is attached to the corresponding **Stage** (the boolean expression associated to the **Process Flow Guard** is determined by invoking the helper function `producePFGExpression`).

Lazy rules `produceDFG`, `produceM` and `produceFL` are also defined to dynamically generate, respectively, **Data Flow Guards, Milestones**, and **Fault Loggers**. When `activity2substage` is triggered, `produceDFG` and `produceM` are invoked once, and their ECA is determined by the input and output data objects associated to the activity. Then, `produceM` is invoked as many times as the number of interrupting boundary events associated to the activity, and `produceFL` as the number of boundary events (both interrupting and non-interrupting), and their ECA predicates on the occurrence of the boundary event.

Example. Fig. 7 shows the E-GSM process model derived from the mArtifact-oriented process view of the container (see top of Fig. 5). `StartShipment.DFG1` is triggered whenever artifacts `Truck` or `Container`, respectively, enter a new state (which is represented by $truck^e$ or $container^e$). Furthermore, `StartShipment.DFG1` requires that `Truck` and `Container` are in states *[producer,moving]* and *[full,shipping,hooked]*, respectively. `StartShipment.M1`, on the other hand, is triggered whenever the truck leaves the current state ($truck^l$), and requires that `Truck` be in state *[highway,moving]*.

This E-GSM process model allows one to detect control flow violations. For example, if we assumed that once the goods are loaded in the container, the carrier leaved the producer's site without waiting for the shipment documents to be checked, `StartShipment` would become *outOfOrder* (being `StartShipment.DFG1` triggered before `StartShipment.PFG1` becomes active) and `Loop` would become *skipped* (being `Loop.M1` required in `StartShipment.PFG1`). Since these **Stages** are not *onTime*, a compliance violation is detected and, given the model, we can understand that `StartShipment` was executed before `Loop`.

4.3. Generation of the E-GSM lifecycle model

The E-GSM process model derived in Section 4.2 does not consider the lifecycle of the mArtifact, and this limits the possibility of monitoring its compliance. For example, still referring to the process presented in Section 2, let assume that, while reaching the producer's site, instead of having an empty container, the container is used for an unauthorized shipment. In this case, no compliance violation is detected, since all activities are executed in the right order, even if the container was not handled as expected: instead of being filled once, it has been filled somewhere, then emptied before reaching the producer, and finally filled again at the producer's premises.

To identify when the mArtifact is not handled as defined in the model, its lifecycle is taken into account. Starting from the mArtifact-oriented process model (Section 4.1), we produce a finite state machine that considers all the possible states that the mArtifact can assume, along with admissible transitions, (see left side of Fig. 8). Such a state machine is obtained by adopting the approach in [13], without labeling transitions as we are not interested in identifying the activities responsible for causing them to fire.

For the container, there is a single initial state, *[empty,warehouse,unhooked]*. This means that the process should start with an empty container that resides in the MTO warehouse and is not hooked to any means of transport. The two final states, *[empty,shipping,hooked]* and *[full,shipping,hooked]*, correspond to the process that terminates with a container that is either empty or full, outside the MTO site, and hooked to a means of transport. Transitions among states allow the container to evolve linearly, as the container is supposed not to enter a state it had left previously. Similarly, for the truck, there is only a single initial state, *[carrier,moving]*, indicating that the process should start when the truck is at the carrier's site, and is moving. The two final states, *[producer,still]* and *[inland_terminal,moving]*, are expected to be reached when the process ends with the truck either still at the producer's site, or moving to an inland terminal. The evolution of the truck has a cyclic portion, since it is possible to move from state *[mto,still]* to *[mto,moving]* and vice versa.

To monitor the lifecycle of the mArtifact, E-GSM is again adopted to model the admissible state transitions. To this aim, the finite state machine is translated to E-GSM as follows:

- Each state is translated into a **Stage** *S* whose **Data Flow Guard** (**Milestone**) is triggered when the mArtifact assumes the state represented (a state different from the one represented) by that **Stage**.
- The **Process Flow Guard** of each **Stage** *S* requires that at least one of the **Stages** that represent the states that directly precede the one represented by *S* be opened (i.e., active). If *S* represents the initial state, its **Process Flow Guard** requires that no **Stage** be active.
- A **Stage** named `Error` is introduced to identify when the artifact is in a state not included in the state machine. The condition on its **Data Flow Guard** (**Milestone**) is triggered when the mArtifact assumes a state not defined in the finite state machine. The condition on its **Process Flow Guard** is never satisfied (thus always raising a compliance violation whenever the mArtifact assumes this state).
- A **Stage** named `Final` is introduced to identify which **Stages** represent a final state. The condition on its **Data Flow Guard** (**Milestone**) requires that at least one (none) of the **Stages** that are translated from final states be active. This way, it is possible to know when the lifecycle of the mArtifact is concluded: when the mArtifact reaches a final state, the corresponding **Stage** becomes active, together with `Final`.

To automatically derive the E-GSM lifecycle model, ATL was adopted again. In particular, Fig. 9 shows a simplified version of the ATL translation rules to derive from the finite state machine obtained from [13] the corresponding E-GSM constructs.[5] Rule `state2substage` is responsible for transforming each stage in the source finite state machine into a **Stage** in the target E-GSM lifecycle model. `state2substage` also produces, for each **Stage**, one **Process Flow Guard**, one **Data Flow Guard**, and one **Milestone**, that are attached to the corresponding **Stage**. The ECA of the **Data Flow Guard** and the **Milestone** is triggered, respectively, when the artifact assumes the state, and when it assumes a different state. The boolean expression of the **Process Flow Guard** is determined by the helper function `producePFGExpression`, which identifies the predecessor of the state and builds the expression accordingly. Finally, rule `fsm2stage` is responsible for producing the **Final** and **Error Stages**, and their associated **Process Flow Guard, Data Flow Guard**, and **Milestone**.

Data Flow Guards and **Milestones** allow us to keep track of the current state of the mArtifact: when the mArtifact is in a specific state, the corresponding **Stage** is opened and the other ones, but `Final`, closed. When the mArtifact changes state, the **Data Flow Guard** attached to the **Stage** that represents the new state is triggered, and the corresponding **Stage** is opened. At the same time,

[5] The complete version of these translation rules is available at https://bitbucket.org/polimiisgroup/bpmn2egsm/src/206dd0270c4f32a7997d847356b0397fb283aac4/FSM2GSM/fsm2egsm.atl.

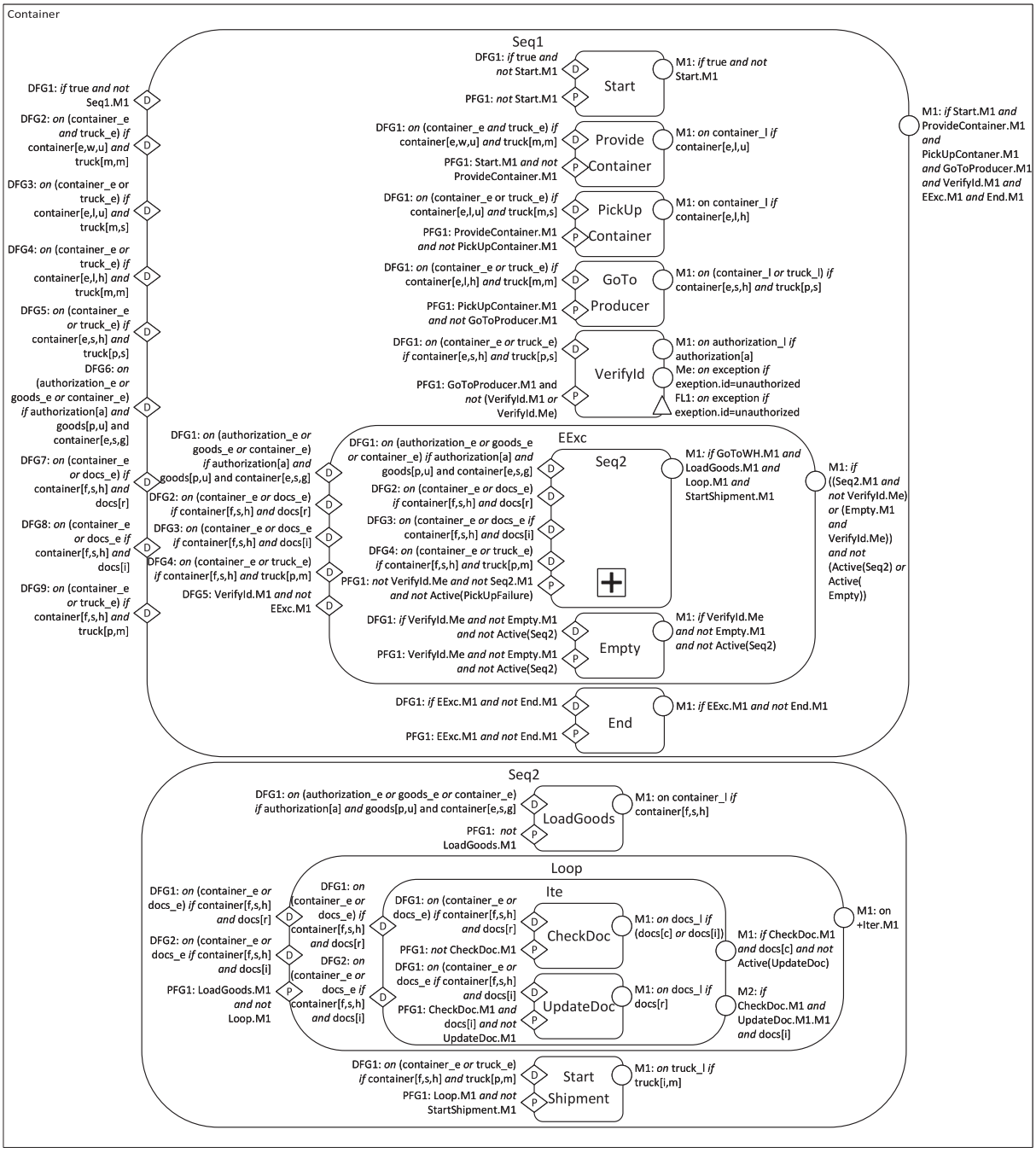

Fig. 7. E-GSM process model of the container (top). For the sake of clarity, **Stages** inside Seq2 are shown separately (bottom).

the **Milestone** attached to the **Stage** that represents the previous state is achieved, and the corresponding **Stage** closed. When an admissible state change occurs, both the **Data Flow Guard** and the **Process Flow Guard** of the **Stage** that represents the new state are expected to be triggered. On the other hand, when a non admissible state change occurs, the condition on the **Process Flow Guard** is not fulfilled, and therefore only the **Data Flow Guard** is triggered.

Example. The right portion of Fig. 8 shows the E-GSM lifecycle model of the container (top), and of the truck (bottom). For the container, the **Data Flow Guard** of **Stage** Final requires that **Stages** ESH or FSH be active, since these **Stages** are obtained from states *[empty,shipping,hooked]* and *[full,shipping,hooked]*, which are final. For the same reason, its **Milestone** requires that both be not active. The **Process Flow Guard** of **Stage** ELU requires that **Stage** EWU be active, since the container is expected to enter state *[empty,loading_area,unhooked]* only if it exits state *[empty,warehouse,unhooked]*, as there is only one transition between these two states in the state machine. On the other hand, the **Process Flow Guard** of **Stage** EWU requires that none of the other **Stages** be active, since state *[empty,warehouse,unhooked]* should be the initial state. For the truck, the **Process Flow Guard** of **Stage** MS requires that either **Stage** CM or MM be active. This way, the truck is expected to enter state *[mto,still]* only if it exits either state *[carrier, moving]* or *[mto, moving]*, the latter allowing the cyclic behavior in the lifecycle of the truck.

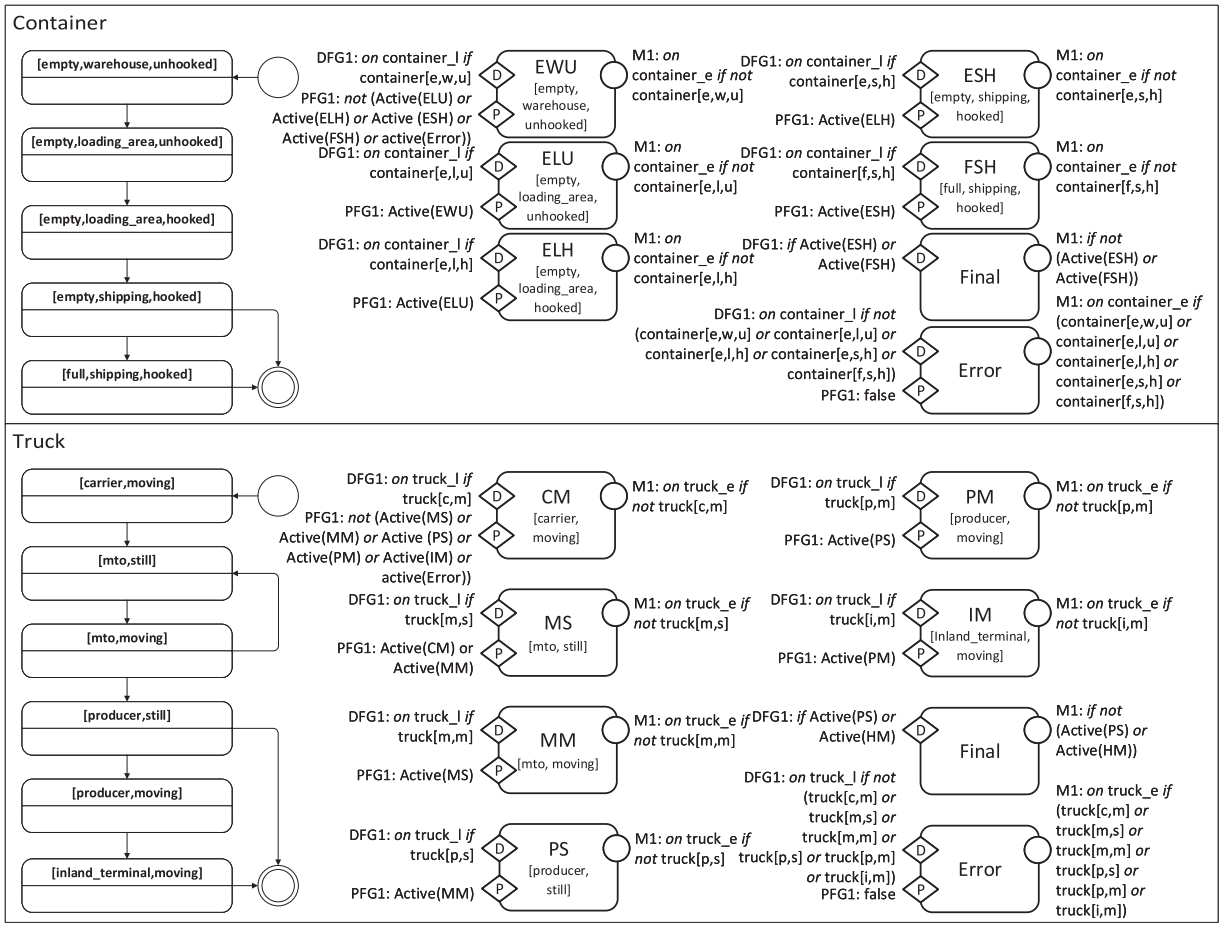

Fig. 8. Finite State Machine (left) and E-GSM model (right) representing the lifecycle of the container (top) and the truck (bottom).

5. Correctness of the translation

The transformation from BPMN to E-GSM presented in Sections 4.2 and 4.3 must be *correct*. To this end, we first need to precisely define what to we mean by "correctness". Intuitively, in our setting correctness captures that, given a BPMN model and a process execution trace, the trace deviates from the model if and only if the E-GSM translation detects so. Since our framework is meant to be used at runtime, we also require this to be prompt, that is, the deviation is detected when it actually occurs. More specifically, let: (i) $\mathcal{B}$ be the input BPMN process model of interest, obtained as a result of the methodological step shown in Section 4.1, and obeying to the well-structuredeness assumptions mentioned before; (ii) $\mathcal{G}^P_{\mathcal{B}}$ be the E-GSM process model encoding the control-flow of $\mathcal{B}$ for monitoring purposes, i.e., the result of the methodological step discussed in Section 4.2; (iii) $\mathcal{G}^A_{\mathcal{B}}$ be the E-GSM model encoding the lifecycle of the mArtifact, i.e., the result of the methodological step discussed in Section 4.3. Correctness asserts that for every (possibly partial) execution trace over the tasks and events of $\mathcal{B}$:

- If the trace conforms to $\mathcal{B}$, then none of the **Stages** of $\mathcal{G}^P_{\mathcal{B}}$ are *outOfOrder* (cf. Fig. 2); conversely, if the trace contains a deviation, then such a deviation is recognized by $\mathcal{G}^P_{\mathcal{B}}$, i.e., $\mathcal{G}^P_{\mathcal{B}}$ has at least one *outOfOrder, opened* **Stage** when the deviation actually occurs. Since this property must hold for every trace, then it must hold also for the minimal, deviating trace. This property is called the *control flow alignment* between $\mathcal{G}^P_{\mathcal{B}}$ and $\mathcal{B}$.
- By projecting away **Data Flow Guards** in $\mathcal{G}^A_{\mathcal{B}}$ (i.e., by keeping **Process Flow Guards** only), $\mathcal{G}^A_{\mathcal{B}}$ has an evolution from one **Stage** to another if and only if there exists a corresponding transition in the lifecycle of the mArtifactthat is induced by $\mathcal{B}$. This is called the *lifecycle alignment* between $\mathcal{B}$ and $\mathcal{G}^A_{\mathcal{B}}$.

The formal proof showing that our translation mechanism is indeed correct is given in [14]. In this section we report a relevant excerpt where a high-level discussion of the proof is presented.

5.1. Trace conformance

Before proving that the translation preserves control flow and lifecycle alignment, we need to define what does it mean for a trace to *conform* to (and *deviate* from) the BPMN model $\mathcal{B}$, considering in particular the control-flow constraints present in $\mathcal{B}$. Typically, conformance is tackled by transforming the process model of interest into a formal behavioral model (such as a workflow net [15]), then checking whether a complete trace can be *replayed* in the model starting from its initial state, executing all tasks in the order they are present in the trace, finally reaching the ending state of the process in a clean way (see, e.g., [16]). This straightforwardly extends to partial traces, by simply checking whether the given partial trace is a prefix of a complete, conforming trace.

In our context, we do not involve further translation mechanisms. Instead, we directly provide a formal definition of conformance, exploiting the fact that $\mathcal{B}$ is well-structured. This definition obeys to three important properties:

- It is modularly defined over the different types of blocks that may be employed to structure $\mathcal{B}$, and allows one to assess conformance by simultaneous induction on the length of the trace and on the sub-block relation of $\mathcal{B}$.

```
rule fsm2stage {
  from m: UML!StateMachine
  to
    tfdfg: EGSM!DataFlowGuardType (id <- 'Final_dfg1',
      expression <- 'on␣' + m.name + '_l␣if␣(' +
      'or'.concatStrings(m.getFinalStates()
      -> collect(s | 'Active(' + s.name + ')')) + ')'),
    tfm: EGSM!MilestoneType (id <- 'Final_m1',
      expression <- 'on␣' + m.name + '_e␣if␣not␣(' +
      'or'.concatStrings(m.getFinalStates()
      -> collect(s | 'Active(' + s.name + ')')) + ')'),
    tfs: EGSM!SubStageType (id <- 'Final',
      dataFlowGuard <- OrderedSet{tfdfg},
      milestone <- OrderedSet{tfm}),
    tedfg: EGSM!DataFlowGuardType (id <- 'Error_dfg1',
      expression <- 'on␣' + m.name + '_l␣if␣not␣(' +
      'or'.concatStrings(m.getAllStates()
      -> collect(s | m.name + '[' + s.name + ']' )) + ')'),
    tepfg: EGSM!DataFlowGuardType (id <- 'Error_pfg1', expression <- 'false'),
    tem: EGSM!MilestoneType (id <- 'Error_m1',
      expression <- 'on␣' + m.name + '_e␣if␣(' +
      'or'.concatStrings(m.getAllStates()
      -> collect(s | m.name + '[' + s.name + ']' )) + ')'),
    tes: EGSM!SubStageType (id <- 'Error',
      dataFlowGuard <- OrderedSet{tedfg},
      processFlowGuard <- OrderedSet{tepfg},
      milestone <- OrderedSet{tem})}

rule state2substage {
  from s: UML!State
  to
    tpfg: EGSM!ProcessFlowGuardType (id <- s.name + '_pfg1',
      expression <- s.producePFGExpression()),
    tdfg: EGSM!DataFlowGuardType (id <- s.name + '_dfg1',
      expression <- 'on␣' + s.getParentStateMachine().name + '_l␣if␣' +
      s.getParentStateMachine().name + '[' s.name + ']' ),
    tm: EGSM!MilestoneType (id <- s.name + '_m1',
      expression <- 'on␣' + s.getParentStateMachine().name + '_e␣if␣not␣' +
      s.getParentStateMachine().name + '[' s.name + ']' ),
    ts: EGSM!SubStageType (id <-  s.name,
      dataFlowGuard <- OrderedSet{tdfg},
      milestone <- OrderedSet{tm},
      processFlowGuard <- OrderedSet{tpfg})}

helper context UML!State def: producePFGExpression(): String =
  'or'.concatStrings(self.getPredecessor() -> collect(t | 'Active(' + t.name
   + ')') -> including(
     if (self.isInitialState()) then
       OrderedSet{ 'not␣(' +
         'or'.concatStrings((self.getParentStateMachine().getAllStates()
         -> excluding(self) -> collect(s | 'Active(' + s.name + ')'))
           -> including('Active(Error)')) + ')' }
     else
       OrderedSet{}
     endif ) -> flatten());
```

Fig. 9. (Simplified) ATL translation rules to derive from a finite state machine the E-GSM lifecycle model.

- It is also applicable to E-GSM, thus providing the basis for comparing $\mathcal{B}$ and $\mathcal{G}^p_{\mathcal{B}}$ in terms of control flow alignment;
- It is fully compatible with the standard definition of conformance over workflow nets, in the sense that a trace conforms to $\mathcal{B}$ according to our definition *if and only if* it is the prefix of a trace leading from the input to the output place of the workflow net that encodes $\mathcal{B}$. The encoding is the one obtained using standard translation mechanisms (see, e.g., [17], noticing that since $\mathcal{B}$ is well-structured, the obtained Petri net is indeed a workflow net).

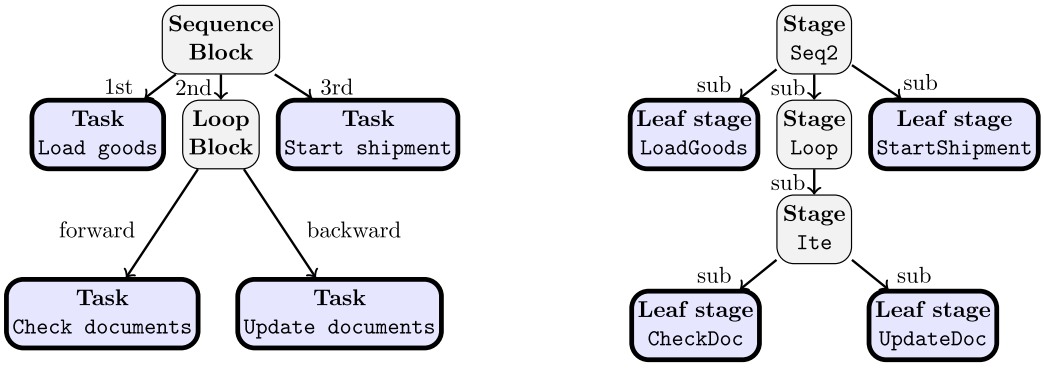

Fig. 10. Block structure of the Seq2 fragment of the BPMN model in Fig. 5 and of its corresponding E-GSM translation in Fig. 7.

To define conformance, we start by noting that no block is repeated twice in $\mathcal{B}$. This guarantees that tasks/events are unambiguous, and at the same time ensures that no block directly or indirectly embeds itself (see the left part of Fig. 10 for an example). As a consequence, we get that the sub-block relation in $\mathcal{B}$ induces a tree-structure, rooted in the top-level, end-to-end process, and whose leaves are atomic tasks and events. We call such a tree the *process tree* of $\mathcal{B}$. On top of this structure, a notion of *execution state* is introduced, so as to keep track of the currently active blocks, and of those that can be activated next. The initial execution state declares that the top process block is active, and that the start event can be activated next. When the start event occurs, the immediately consequent block can be activated next. Given the current activation state, a new activation state is computed when the next execution step is performed, i.e., an event occurs, a task is started, or a task is completed. How the new state is computed depends on the specific types of the active blocks, and is done in two phases. In the first phase, it is checked whether the execution step is accepted by $\mathcal{B}$ in the current activation state. The start of a task or the occurrence of an event are accepted only if that task/event can be activated next. The completion of a task is accepted instead only if that task is currently active. If the execution step is not accepted, then a deviation occurs. If it is accepted, the execution step is enforced, leading to update the current state of $\mathcal{B}$, by deactivating active blocks, and by making new blocks active. Both the "check" and the "update" phases depend on the semantics of active blocks and of those that can be activated next. For example, a sequence block containing two tasks is managed by ensuring that: *(i)* the first task can be activated as soon as the sequence block is activated, *(ii)* the second task can be activated as soon as the first task is completed, *(iii)* the sequence block is deactivated as soon as the second task is completed.

5.2. Control flow alignment

Given the key notion of acceptance of an execution step, and that of activation of a block (both sketched in Section 5.1), it is possible to derive a direct, modular correspondence between $\mathcal{B}$ and $\mathcal{G}^P_{\mathcal{B}}$. More specifically, the translation procedure (cf. Section 4.2) guarantees that the process tree of $\mathcal{B}$ is modularly mirrored in $\mathcal{G}^P_{\mathcal{B}}$, in the sense that the sub-stage relation of $\mathcal{G}^P_{\mathcal{B}}$ reconstructs the structure of $\mathcal{B}$. The correspondence is not direct, as single, intermediate stages may be inserted in $\mathcal{G}^P_{\mathcal{B}}$ so as to handle the execution semantics of their corresponding blocks in $\mathcal{B}$. See Fig. 10 for an example. Thanks to the fact that the translation is modular w.r.t. the blocks of $\mathcal{B}$, and so is the notion of conformance sketched in Section 5.1, we then proceed by induction on the structure of the process tree. In particular, we show that, given an execution state s where all active blocks in $\mathcal{B}$ correspond to *opened, onTime* **Stages** in $\mathcal{G}^P_{\mathcal{B}}$, and given an execution step $\mathtt{t}$:

- if $\mathtt{t}$ is considered as a deviation by $\mathcal{B}$ in s, then the execution of $\mathtt{t}$ in s causes a **Stage** of $\mathcal{G}^P_{\mathcal{B}}$ to become *outOfOrder*;
- if $\mathtt{t}$ is accepted by $\mathcal{B}$ in s, then the new execution state s' resulting from the execution of $\mathtt{t}$ in s is such that a block b is active in s' if and only if the corresponding **Stage** in $\mathcal{G}^P_{\mathcal{B}}$ is *opened* and *onTime*.

The combination of these two properties implies that $\mathcal{G}^P_{\mathcal{B}}$ correctly monitors the control flow of $\mathcal{B}$, promptly detecting a deviation when the currently processed execution step is indeed considered so by $\mathcal{B}$.

5.3. Lifecycle alignment

Lifecycle alignment amounts to checking whether $\mathcal{G}^A_{\mathcal{B}}$ is constructed by properly incorporating the mArtifact transitions induced by $\mathcal{B}$. However, $\mathcal{G}^A_{\mathcal{B}}$ contains **Data Flow Guards** that are not synthesized from $\mathcal{B}$, but are used to actually monitor the physical reality and obtain the mArtifact state accordingly. Hence, alignment is circumscribed to **Process Flow Guards**.

It is immediate to see that $\mathcal{G}^A_{\mathcal{B}}$ is such that each possible mArtifact state corresponds to a **Stage**, and that at each moment one and only one of such **Stages** is *opened*. We say that *mArtifact may change from state s_1 to state s_2 according to $\mathcal{G}^A_{\mathcal{B}}$* if $\mathcal{G}^A_{\mathcal{B}}$ foresees an execution step that is applicable when the **Stage** corresponding to s_1 is *opened*, and whose effect is to close that **Stage** while simultaneously opening the **Stage** that corresponds to s_2. In this light, lifecycle alignment amounts to checking that, for every pair of mArtifact states s_1 and s_2, the mArtifact may change from state s_1 to state s_2 according to $\mathcal{G}^A_{\mathcal{B}}$ if and only if $\mathcal{B}$ foresees such a transition. This property, in turn, is proven in two steps. In the first step, we rely on the correctness of the method proposed in [13], which encodes the state-transitions of the input BPMN model $\mathcal{B}$ into a corresponding state machine $\mathcal{M}$. In the second step, we reformulate the lifecycle alignment by considering the explicit description provided by $\mathcal{M}$ instead of the implicit one obtained from $\mathcal{B}$. It is then straightforward to see that this reformulation: *(i)* faithfully encodes the behavior of $\mathcal{M}$; *(i)* produces exactly $\mathcal{G}^A_{\mathcal{B}}$ from $\mathcal{B}$. Correctness of lifecycle then follows directly.

6. Implementation

Fig. 11 presents the architecture of the monitoring solution we implemented to assess the effectiveness of our approach. For the sake of clarity, the figure only shows three (out of five) of the mArtifacts included in our running example, but it is only a matter of replicating elements. Each mArtifact is monitored by a dedicated smart object, which embeds the components in the gray box (bottom part of the figure). Before the process starts, the owner of the artifact instructs the smart object with the E-GSM models derived

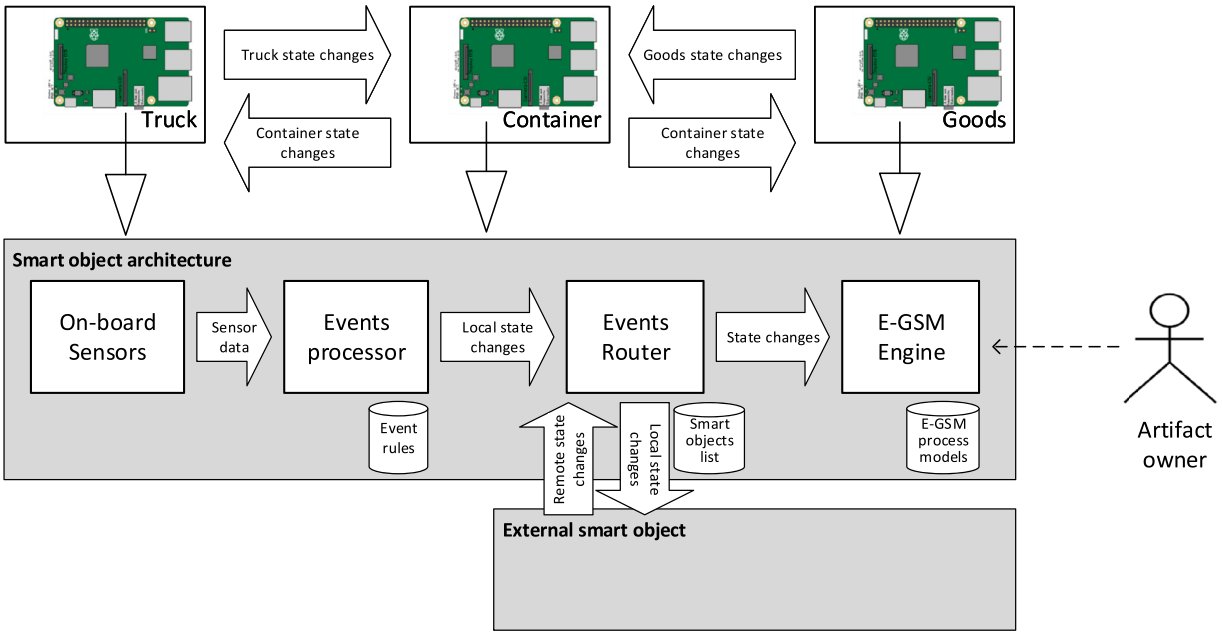

Fig. 11. Infrastructure of our IoT-based monitoring platform.

in Section 4. Then, by querying the smart object, the owner can be aware of violations in the process or the lifecycle and, in case, take countermeasures. It is worth noting that, if the ownership of the smart object changes, thanks to the monitoring solution the new owner can be aware of the history of the mArtifact.

6.1. Architecture

The *On-board Sensors Gateway* is responsible for periodically collecting the values coming from the sensors attached to the mArtifact and transforming them into messages.[6] These messages feed the *Event Processor*, which is a rule engine that transforms the values from sensors into the states that the mArtifact may assume (e.g., geographical coordinates can help infer the state of the Truck). This component, given the complexity of required rules and the computational power of the smart object, can either execute locally or outsource the computation to a full-fledged Complex Event Processor (CEP) [18]) that runs on a dedicated server or in the cloud.[7] As soon as the mArtifact changes state, the *Event Processor* produces two events to indicate that the mArtifact has left the old state and entered the new one.

The different mArtifacts involved in the same inter-organizational process must evolve in a coordinated way, and the correct monitoring of the process requires that mArtifacts exchange information about their states. For this reason, the events produced by the *Event Processor* are sent to the *Events Router*[8] to inform the other mArtifacts about state changes, and be informed by them. The *Events Router* must be configured to know the smart objects it is supposed to communicate with.

The *Events Router* then feeds the *E-GSM Engine*[9] with the actual state changes of the mArtifact. The engine hosts the two E-GSM models derived according to our methodology and checks whether the mArtifact embodied by the smart object complies with both the control and data flows defined in the original BPMN process model. The E-GSM Engine also provides a simple Graphical User Interface (GUI), which allows the owner of the mArtifact to check the evolution of the process and the artifact.

These components are deployed on the different smart objects, which then interact through the usual communication means (WiFi or 4G networks). The whole platform is based on the Node.js runtime environment[10] to support resource-constrained devices. Node.js eases the execution of hardware and operating system-agnostic JavaScript code, is available for most hardware and software platforms, and is optimized for systems low on CPU power and RAM. The communication among smart objects is based on Message Queue Telemetry Transport (MQTT)[11], which is optimized for low-bandwidth and resource-constrained environments. In addition, to allow the software modules to communicate with each other and with the parties, we exposed them as Representational State Transfer (REST) services.

6.2. E-GSM monitoring platform in action

To test our solution, we used several process models taken from the logistics domain, which were validated by domain experts. To simulate sensor values, as well as external events, we built a web-based test interface.[12] This interface allows one to manually define the values that the *On-board Sensors Gateway* should receive (upper part of the screen), and also to interact with the BPMN representation of the process.[13] Changes in the state of the artifacts can be reported to the monitoring solution (by clicking on the data objects), as well as events coming from the parties (by clicking on BPMN start, end or intermediate events). This way, we can demonstrate that, given proper inputs, the implemented architecture can

[6] Sampling time and type of interaction (pull/push) can be configured given the types of the attached sensors.

[7] Our prototype adopts the WSO2 CEP running on the GIoTTO cloud platform (http://www.almaviva.it/EN/OurOffering/Information_Technology/Pagine/giotto.aspx).

[8] Source code of the Events Router is available at https://bitbucket.org/polimiisgroup/eventsrouter.

[9] Source code of the E-GSM Engine is available at https://bitbucket.org/polimiisgroup/egsmengine.

[10] See https://nodejs.org.

[11] See http://mqtt.org.

[12] Source code of the test interface is available at https://bitbucket.org/polimiisgroup/testclient.

[13] Rendered through the bpmn.js library (https://bpmn.io/toolkit/bpmn-js).

monitor the execution of the process and the lifecycle of artifacts according to our approach.

We assumed that a container is mimicked by a smart object equipped with an Radio Frequency Identification (RFID) scanner to identify its position on the MTO premises, scales to detect the load weight, and a switch that closes once the container is hooked to a means of transport. Similarly, we assume that a truck is a smart object equipped with a GPS receiver to identify if it is moving and, if so, its geographical position. Such smart objects can then be used to monitor the portion of the process described in Section 2 relevant for a container and a truck. To do so, for both of them, we fed their E-GSM engine with the models derived according to our methodology, and provided the Event Processor with the rules to derive their state based on sensed data. For example, we infer state *[empty,loading_area,unhooked]* if the RFID scanner detects the tag that identifies the loading area, the scales detect a weight lower than 5 kg and the hooking switch is open. Finally, we instructed their Events Router to listen to the events coming from each other, plus the ones coming from other artifacts involved in the process (i.e., the goods, the shipment documents, etc.).

These experimental, instrumented container and truck allowed us to assess how our solution can be used to assess the compliance of process executions. For example, suppose that the container has a defective hooking mechanism, and it detaches from the truck after being loaded.

Before the container detaches, in the E-GSM process model of the container, `Start`, `ProvideContainer` and `PickUpContainer` are closed. On the other hand, in the E-GSM lifecycle model of the container, `EWU` and `ELU` are closed, while `ELH` is opened. When the container detaches, its Event Processor detects that the hooking switch opens, so it infers that the container is in state *[empty,loading_area,unhooked]* and emits a new event. The Events Router of the container captures this event and forwards it to the E-GSM engine of the container, that detects a violation both in the control flow and in the lifecycle. In the E-GSM process model, event *[empty,loading_area,unhooked]* triggers `PickUpContainer.DFG1`, which causes `PickUpContainer` to be opened a second time. However, being `PickUpContainer.PFG1` not active, `PickUpContainer` becomes *outOfOrder*. In the E-GSM lifecycle model, on the other hand, event *[empty,loading_area,unhooked]* triggers `ELH.M1` and `ELU.DFG1`, causing `ELH` to close and `ELU` to reopen. However, being `ELU.PFG1` not active, a non admissible transition in the lifecycle of the container (from *[empty,loading_area,hooked]* to *[empty,loading_area,unhooked]*) is detected.

The Events Router of the container also propagates the event to the other smart objects. The Events Router of the truck then receives the event, and forwards it to the E-GSM engine of the truck, that detects a violation only in the control flow. Similarly to the container, in the E-GSM process model, `PickUpContainer` becomes *outOfOrder*, since it is executed twice. On the other hand, being the truck compliant with its lifecycle, no compliance violation is detected in its E-GSM lifecycle model.

Fig. 12 shows the interface reporting the status of the smart device related to the container according to the described scenario. In the upper window, **Stages** are represented with different colors with respect to their status, whereas in the lower window the detail of the **Stage** is reported. As shown, the process correctly started and the container provisioning occurred as expected. Differently from the expected behavior, the `PickUpContainer` **Stage** is highlighted as non correctly executed (i.e., *outOfOrder*) and the details of the involved **Data Flow Guards**, **Process Flow Guards**, and **Milestones** are reported.

7. Related work

Traditionally, to monitor multi-party business processes, commitments have been used. Commitments are formal contracts that specify how the interactions between the organization and the service provider should be performed [19]. However, their main focus is on the outcome of the process portions carried out by the organizations. Consequently, their granularity is coarser than the one of our approach, which is capable of detecting violations at the activity level.

To provide a more fine-grained monitoring, Baouab et al. [20] and Berry and Milosevic [21] propose solutions that monitor the choreography. However, these solutions are limited to the messages exchanged among the organizations. Therefore, to monitor a process at the activity level, they require the modeler to define message exchanges before and after each activity is executed. Other solutions targeting multi-party business processes, such as [22] and [23], monitor the compliance only at design time, but offer no direct support for monitoring the execution. To monitor multi-party business processes at the activity level, Kikuchi et al. [24] proposes a framework that collect and distributes process logs generated by Web Services Business Process Execution Language (WS-BPEL) process engines. However, by expecting all activities to be automated (i.e., invoked web services) and by requiring links to be established among all the involved organizations, this solutions presents the same limitations as the traditional approaches described in Section 1.

According to Ly et al. [25], compliance monitoring approaches that analyze both the execution flow and managed data are able to continuously monitor a process even when violations are detected, and can also discriminate compliance violations according to the impact on the execution. For example, one can think of ECE Rules [26], BPath [27], Mobucon EC [28], and SeaFlows [29]. Since ECE rules are not specifically tailored to business processes, they do not explicitly support the lifecycle of data artifacts or control flow constraints. BPath, which was conceived to monitor the execution of workflow-like service compositions, do not deal with the problem of determining when activities are executed as the lifecycle is captured by the connected service. Mobucon EC and SeaFlows describe compliance rules with very powerful yet complex languages, Event Calculus and Compliance Rule Graphs respectively, but they do not offer advanced mechanisms to determine the degree of compliance of process instances. All these solutions require that compliance rules be defined by hand, and none of them offers mechanisms to derive such rules from the process model.

To detect the execution order of activities without relying on explicit messages, Baumgraß et al. [30] proposes the integration of a CEP and a BPMN engine to detect when activities are executed based on external events. BPMN is extended with Process Event Monitoring Points (PEMPs) to identify which events produced by the CEP engine determine the activation or termination of activities, gateways, and events. Bülow et al. [31] proposes an architecture that implements this solution, while [32] presents an approach that relies on external data to identify when activities, annotated with attributes that have to be monitored, are incorrectly executed.

About the usage of artifact-centric process models and the relation with imperative languages, Köpke and Su [33] proposes transformation rules that transform a BPMN process model into a GSM equivalent enriched with additional Stages that group activities according to business goals. While our work borrowed from these rules the idea of transforming blocks into nested Stages, the way expressions on Guards and Milestones are derived is completely different from the one presented in these articles. The main reason behind such a discrepancy is that our E-GSM model is monitoring-

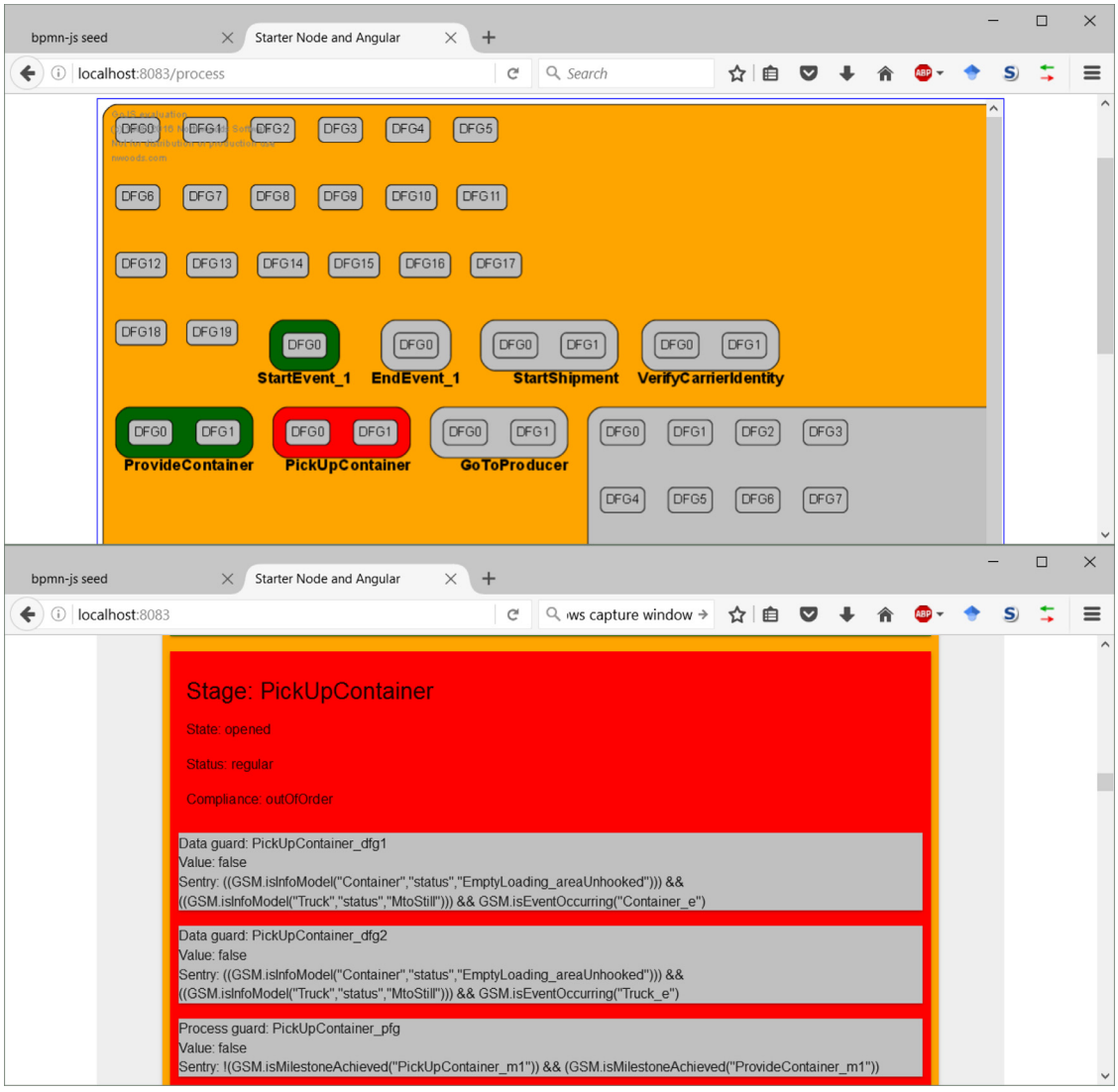

Fig. 12. Screenshot of the monitoring solution.

oriented, rather than execution-oriented. Therefore, our E-GSM model must not allow to detect only the executions performed exactly as defined in the BPMN model, but potentially every possible execution.

Eshuis and Van Gorp [34] defines a semi-automated approach to transform a process modeled using UML Activity Diagrams into a GSM model that captures the lifecycle of each involved artifact. Similarly, Kumaran et al. [35], Meyer and Weske [36] and Eid-Sabbagh et al. [13] propose a language-agnostic algorithm to derive the lifecycle of artifacts based on an imperative process model. This is possible as long as each activity has input and output information entities explicitly defined in the model. Our work differentiates from the one presented in these articles, which use control flow information to model the interactions among data artifacts, by keeping such information in the target process model to assess compliance.

Popova and Dumas [37] defines a translator from Petri Nets to GSM. The main purpose of that translator is to transform the outcome of process mining algorithms, which is often represented as a Petri Net, to a GSM model. This way, process mining techniques can be used to identify business artifacts that the translator represents in a language that is easier to understand by domain experts than Petri Nets.

To ease the definition of GSM models, Eshuis et al. [38] proposes to start modeling the process with Dynamic Condition Response (DCR) graphs, and then translate them into GSM. While DCR graphs allow to model the process in a completely graphical way, they are still way more complex to understand than imperative languages (as mentioned in [8]).

Concerning the usage of GSM to monitor multi-party processes, Meijler et al. [39] proposes a collaboration hub running a GSM engine to facilitate the coordination of logistics processes. With respect to our work, information on the execution of activities must be explicitly notified to the hub either by interacting with its interface or via web service calls. Gnimpieba et al. [40] overcomes this limitation by adopting the IoT paradigm: they take advantage of Guards and Milestones to identify when Stages are being executed by predicating on sensor data coming from smart objects. However, with respect to our work, they lack a methodology to ease the definition of the GSM model. Furthermore, they do not decouple the process model from the rules to infer the state of an artifact based from sensor data. Finally, they use the IoT paradigm only to collect and forward sensor data to a centralized GSM engine. All these solutions require the parties to rely on a single entity who owns the monitoring infrastructure, thus not allowing to independently check the process compliance.

Also, they lack mechanisms to detect deviations in the execution of the process with respect to the model.

As for the integration of both an activity-centric and a data-centric perspectives in business processes, Künzle and Reichert [41] and Meyer et al. [42] propose to use information about the process control flow to define how data should be manipulated. Both approaches use such information in a prescriptive way, and assume that the process respects the execution order of activities. E-GSM, in contrast, does not enforce any predefined flow and uses control flow information only to detect compliance violations.

8. Conclusions

This paper explains how monitoring multi-party business processes is a challenging activity: needs for coordination among the involved BPMSs, limited visibility on the whole process by the different parties, differences in monitoring artifacts and control flows are some of the aspects that traditional monitoring solutions are not able to cope with. Moreover, when physical objects are exchanged by the parties, monitoring these objects adds new challenges.

The proposed approach shows how a proper mix between imperative languages, used to easily model the process, and declarative languages, used to configure a monitoring system, and the adoption of the IoT paradigm can overcome these limitations. In particular, starting from a multi-party process defined using BPMN, the paper proposed a methodology to translate this process to the E-GSM notation. Smart objects are properly instrumented with the code needed to monitor different portions of the public process shared by the parties. As each smart object is referring to one of the physical artifacts involved in the multi-party process, it will keep the owner of the artifact informed about the progression of its state and how the process is evolving. Correctness of the transformation from BPMN to E-GSM has been proved, thus guaranteeing that a violation is promptly detected by the E-GSM engine if and only if the last processed execution step deviates from the input BPMN model.

Currently, the proposed approach requires organizations to transparently share among the other participants how their process portions are performed. Additionally, smart objects exchange information on their state to the other ones participating in the same execution. This does not allow organizations to keep part of their process private: either they share this information, or the private portion is omitted from the BPMN collaboration diagram and, consequently, it cannot be monitored with our approach. To address these issues, future work will focus on integrating our approach with security and privacy frameworks.

Future work will also concentrate on categorizing violations with respect to their impact on the execution, and on using such a classification to determine the overall health of process instances. Finally, we aim to extend the methodology considering costs, and scalability issues which may become important when a process involves numerous artifacts or a given artifact has to deal with numerous events.

Acknowledgments

This work has been partially funded by the Italian Project ITS Italy 2020 under the Technological National Clusters program. We also thank Marco Spelta for his help on implementing the architecture.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi 10.1016/j.is.2017.12.009.

References

[1] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F.F.T.H. III, S. Hobson, M.H. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, R. Vaculín, Introducing the guard-stage-milestone approach for specifying business entity lifecycles, in: M. Bravetti, T. Bultan (Eds.), Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16–17, 2010. Revised Selected Papers, *Lecture Notes in Computer Science*, 6551, Springer, 2010, pp. 1–24, doi:10.1007/978-3-642-19589-1_1.

[2] L. Baresi, G. Meroni, P. Plebani, A gsm-based approach for monitoring cross-organization business processes using smart objects, in: M. Reichert, H.A. Reijers (Eds.), Business Process Management Workshops - BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 - September 3, 2015, Revised Papers, *Lecture Notes in Business Information Processing*, 256, Springer, 2015, pp. 389–400, doi:10.1007/978-3-319-42887-1_32.

[3] L. Baresi, G. Meroni, P. Plebani, Using the guard-stage-milestone notation for monitoring bpmn-based processes, in: R. Schmidt, W. Guédria, I. Bider, S. Guerreiro (Eds.), Enterprise, Business-Process and Information Systems Modeling - 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13–14, 2016, Proceedings, *Lecture Notes in Business Information Processing*, 248, Springer, 2016, pp. 18–33, doi:10.1007/978-3-319-39429-9_2.

[4] L. Baresi, G. Meroni, P. Plebani, On handling business process anomalies through artifact-based modeling, in: S. España, M. Ivanovic, M. Savic (Eds.), Proceedings of the CAiSE'16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016), Ljubljana, Slovenia, June 13–17, 2016., *CEUR Workshop Proceedings*, 1612, CEUR-WS.org, 2016, pp. 9–16.

[5] G. Meroni, C. Di Ciccio, J. Mendling, Artifact-driven process monitoring: dynamically binding real-world objects to running processes, in: X. Franch, J. Ralyté, R. Matulevicius, C. Salinesi, R. Wieringa (Eds.), Proceedings of the Forum and Doctoral Consortium Papers Presented at the 29th International Conference on Advanced Information Systems Engineering, CAiSE 2017, Essen, Germany, June 12–16, 2017, *CEUR Workshop Proceedings*, 1848, CEUR-WS.org, 2017, pp. 105–112.

[6] G. Meroni, C. Di Ciccio, J. Mendling, An artifact-driven approach to monitor business processes through real-world objects, in: E.M. Maximilien, A. Vallecillo, J. Wang, M. Oriol (Eds.), Service-Oriented Computing - 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, Proceedings, *Lecture Notes in Computer Science*, 10601, Springer, 2017, pp. 297–313, doi:10.1007/978-3-319-69035-3_21.

[7] L. Baresi, C. Di Ciccio, J. Mendling, G. Meroni, P. Plebani, martifact: an artifact-driven process monitoring platform, in: R. Clarisó, H. Leopold, J. Mendling, W.M.P. van der Aalst, A. Kumar, B.T. Pentland, M. Weske (Eds.), Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, September 13, 2017., *CEUR Workshop Proceedings*, 1920, CEUR-WS.org, 2017.

[8] D. Fahland, D. Lübke, J. Mendling, H.A. Reijers, B. Weber, M. Weidlich, S. Zugal, Declarative versus imperative process modeling languages: the issue of understandability, in: T.A. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, R. Ukor (Eds.), Enterprise, Business-Process and Information Systems Modeling, 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8–9, 2009. Proceedings, *Lecture Notes in Business Information Processing*, 29, Springer, 2009, pp. 353–366, doi:10.1007/978-3-642-01862-6_29.

[9] L. Atzori, A. Iera, G. Morabito, The internet of things: a survey, Comput. Networks 54 (15) (2010) 2787–2805, doi:10.1016/j.comnet.2010.05.010.

[10] G. Meroni, L. Baresi, P. Plebani, Translating BPMN to E-GSM: specifications and rules, Technical Report, Politecnico di Milano, 2016. http://hdl.handle.net/11311/976678.

[11] M. Chinosi, A. Trombetta, BPMN: an introduction to the standard, Comput. Stand. Interfaces 34 (1) (2012) 124–134, doi:10.1016/j.csi.2011.06.002.

[12] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: a model transformation tool, Sci. Comput. Program 72 (1) (2008) 31–39.

[13] R. Eid-Sabbagh, M. Hewelt, A. Meyer, M. Weske, Deriving business process data architectures from process model collections, in: S. Basu, C. Pautasso, L. Zhang, X. Fu (Eds.), Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2–5, 2013, Proceedings, *Lecture Notes in Computer Science*, 8274, Springer, 2013, pp. 533–540, doi:10.1007/978-3-642-45005-1_43.

[14] G. Meroni, M. Montali, L. Baresi, P. Plebani, Translating BPMN to E-GSM: proof of correctness, Technical Report, Politecnico di Milano, 2016. http://hdl.handle.net/11311/990248.

[15] W.M. Van der Aalst, Verification of workflow nets, in: Application and Theory of Petri Nets 1997, Springer, 1997, pp. 407–426.

[16] A. Rozinat, W.M.P. van der Aalst, Conformance checking of processes based on monitoring real behavior, Inf. Syst. 33 (1) (2008) 64–95, doi:10.1016/j.is.2007.07.001.

[17] R.M. Dijkman, M. Dumas, C. Ouyang, Semantics and analysis of business process models in BPMN, Inf. Software Technol. 50 (12) (2008) 1281–1294.

[18] G. Cugola, A. Margara, The complex event processing paradigm, in: F. Colace, M.D. Santo, V. Moscato, A. Picariello, F.A. Schreiber, L. Tanca (Eds.), Data Management in Pervasive Systems, Springer, 2015, pp. 113–133, doi:10.1007/978-3-319-20062-0_6.

[19] P.R. Telang, M.P. Singh, Specifying and verifying cross-organizational business models: an agent-oriented approach, IEEE Trans. Serv. Comput. 5 (3) (2012) 305–318, doi:10.1109/TSC.2011.4.
[20] A. Baouab, W. Fdhila, O. Perrin, C. Godart, Towards decentralized monitoring of supply chains, in: C.A. Goble, P.P. Chen, J. Zhang (Eds.), 2012 IEEE 19th International Conference on Web Services, Honolulu, HI, USA, June 24–29, 2012, IEEE Computer Society, 2012, pp. 600–607, doi:10.1109/ICWS.2012.13.
[21] A. Berry, Z. Milosevic, Extending choreography with business contract constraints, Int. J. Cooperative Inf. Syst. 14 (2-3) (2005) 131–179, doi:10.1142/S0218843005001109.
[22] G. Governatori, Z. Milosevic, S.W. Sadiq, Compliance checking between business processes and business contracts, in: Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16–20 October 2006, Hong Kong, China, IEEE Computer Society, 2006, pp. 221–232, doi:10.1109/EDOC.2006.22.
[23] D. Knuplesch, W. Fdhila, M. Reichert, S. Rinderle-Ma, Detecting the effects of changes on the compliance of cross-organizational business processes, in: P. Johannesson, M. Lee, S.W. Liddle, A.L. Opdahl, O.P. López (Eds.), Conceptual Modeling - 34th International Conference, ER 2015, Stockholm, Sweden, October 19–22, 2015, Proceedings, *Lecture Notes in Computer Science*, 9381, Springer, 2015, pp. 94–107, doi:10.1007/978-3-319-25264-3_7.
[24] S. Kikuchi, H. Shimamura, Y. Kanna, Monitoring method of cross-sites' processes executed by multiple WS-BPEL processors, in: 9th IEEE International Conference on E-Commerce Technology (CEC 2007) / 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2007), 23–26 July 2007, National Center of Sciences, Tokyo, Japan, IEEE Computer Society, 2007, pp. 55–64, doi:10.1109/CEC-EEE.2007.72.
[25] L.T. Ly, F.M. Maggi, M. Montali, S. Rinderle-Ma, W.M.P. van der Aalst, Compliance monitoring in business processes: functionalities, application, and tool-support, Inf. Syst. 54 (2015) 209–234, doi:10.1016/j.is.2015.02.007.
[26] S. Bragaglia, F. Chesani, P. Mello, M. Montali, D. Sottara, Fuzzy conformance checking of observed behaviour with expectations, in: R. Pirrone, F. Sorbello (Eds.), AI*IA 2011: Artificial Intelligence Around Man and Beyond - XIIth International Conference of the Italian Association for Artificial Intelligence, Palermo, Italy, September 15–17, 2011. Proceedings, *Lecture Notes in Computer Science*, 6934, Springer, 2011, pp. 80–91, doi:10.1007/978-3-642-23954-0_10.
[27] S. Sebahi, Monitoring business process compliance : a view based approach. (Monitoring de la conformité des processus métiers : approche à base de vues), Claude Bernard University Lyon 1, France, 2012 Ph.D. thesis.
[28] M. Montali, F.M. Maggi, F. Chesani, P. Mello, W.M.P. van der Aalst, Monitoring business constraints with the event calculus, ACM TIST 5 (1) (2013) 17, doi:10.1145/2542182.2542199.
[29] L.T. Ly, S. Rinderle-Ma, D. Knuplesch, P. Dadam, Monitoring business process compliance using compliance rule graphs, in: R. Meersman, T.S. Dillon, P. Herrero, A. Kumar, M. Reichert, L. Qing, B.C. Ooi, E. Damiani, D.C. Schmidt, J. White, M. Hauswirth, P. Hitzler, M.K. Mohania (Eds.), On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17–21, 2011, Proceedings, Part I, *Lecture Notes in Computer Science*, 7044, Springer, 2011, pp. 82–99, doi:10.1007/978-3-642-25109-2_7.
[30] A. Baumgraß, N. Herzberg, A. Meyer, M. Weske, BPMN extension for business process monitoring, in: F. Feltz, B. Mutschler, B. Otjacques (Eds.), Enterprise modelling and information systems architectures - EMISA 2014, Luxembourg, September 25–26, 2014, *LNI*, 234, GI, 2014, pp. 85–98.
[31] S. Bülow, M. Backmann, N. Herzberg, T. Hille, A. Meyer, B. Ulm, T.Y. Wong, M. Weske, Monitoring of business processes with complex event processing, in: Lohmann et al. [43]., pp. 277–290. 10.1007/978-3-319-06257-0_22.
[32] C. Cabanillas, C. Di Ciccio, J. Mendling, A. Baumgrass, Predictive task monitoring for business processes, in: S.W. Sadiq, P. Soffer, H. Völzer (Eds.), Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7–11, 2014. Proceedings, *Lecture Notes in Computer Science*, 8659, Springer, 2014, pp. 424–432, doi:10.1007/978-3-319-10172-9_31.
[33] J. Köpke, J. Su, Towards quality-aware translations of activity-centric processes to guard stage milestone, in: M.L. Rosa, P. Loos, O. Pastor (Eds.), Business Process Management - 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings, *Lecture Notes in Computer Science*, 9850, Springer, 2016, pp. 308–325, doi:10.1007/978-3-319-45348-4_18.
[34] R. Eshuis, P. Van Gorp, Synthesizing data-centric models from business process models, Computing 98 (4) (2016) 345–373, doi:10.1007/s00607-015-0442-0.
[35] S. Kumaran, R. Liu, F.Y. Wu, On the duality of information-centric and activity-centric models of business processes, in: Z. Bellahsene, M. Léonard (Eds.), Advanced Information Systems Engineering, 20th International Conference, CAiSE 2008, Montpellier, France, June 16–20, 2008, Proceedings, *Lecture Notes in Computer Science*, 5074, Springer, 2008, pp. 32–47, doi:10.1007/978-3-540-69534-9_3.
[36] A. Meyer, M. Weske, Activity-centric and artifact-centric process model roundtrip, in: Lohmann et al. [43], pp. 167–181. 10.1007/978-3-319-06257-0_14.
[37] V. Popova, M. Dumas, From petri nets to guard-stage-milestone models, in: M.L. Rosa, P. Soffer (Eds.), Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers, *Lecture Notes in Business Information Processing*, 132, Springer, 2012, pp. 340–351, doi:10.1007/978-3-642-36285-9_38.
[38] R. Eshuis, S. Debois, T. Slaats, T.T. Hildebrandt, Deriving consistent GSM schemas from DCR graphs, in: Q.Z. Sheng, E. Stroulia, S. Tata, S. Bhiri (Eds.), Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10–13, 2016, Proceedings, *Lecture Notes in Computer Science*, 9936, Springer, 2016, pp. 467–482, doi:10.1007/978-3-319-46295-0_29.
[39] T.D. Meijler, M. Stollberg, M. Winkler, K. Erler, Coordinating variable collaboration processes in logistics, in: 13th International Conference on Modern Information Technology in the Innovation Processes of Industrial Enterprises, 2011.
[40] Z.D.R. Gnimpieba, A. Nait-Sidi-Moh, D. Durand, J. Fortin, Using internet of things technologies for a collaborative supply chain: application to tracking of pallets and containers, in: The 10th International Conference on Future Networks and Communications (FNC 2015) / The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015) / Affiliated Workshops, August 17–20, 2015, Belfort, France, in: *Procedia Computer Science*, 56, Elsevier, 2015, pp. 550–557, doi:10.1016/j.procs.2015.07.251.
[41] V. Künzle, M. Reichert, A modeling paradigm for integrating processes and data at the micro level, in: T.A. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, I. Bider (Eds.), Enterprise, Business-Process and Information Systems Modeling - 12th International Conference, BPMDS 2011, and 16th International Conference, EMMSAD 2011, held at CAiSE 2011, London, UK, June 20–21, 2011. Proceedings, *Lecture Notes in Business Information Processing*, 81, Springer, 2011, pp. 201–215, doi:10.1007/978-3-642-21759-3_15.
[42] A. Meyer, L. Pufahl, D. Fahland, M. Weske, Modeling and enacting complex data dependencies in business processes, in: F. Daniel, J. Wang, B. Weber (Eds.), Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26–30, 2013. Proceedings, *Lecture Notes in Computer Science*, 8094, Springer, 2013, pp. 171–186, doi:10.1007/978-3-642-40176-3_14.
[43] N. Lohmann, M. Song, P. Wohed (Eds.), Business process management workshops - BPM 2013 international workshops, Beijing, China, August 26, 2013, revised papers, vol. 171 Lecture Notes in Business Information Processing, Springer, 2014. 10.1007/978-3-319-06257-0.