Multi-party Business Process Compliance Monitoring through IoT-enabled Artifacts

Giovanni Meroni[1,], Luciano Baresi[1], Marco Montali[1], Pierluigi Plebani[1]

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

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 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 process models into a set of artifact-centric process models, rendered in (our extension of the 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.

*Corresponding Author

Keywords:
Business Process Compliance, Runtime Compliance Monitoring, Internet of Things, Guard-Stage-Milestone, E-GSM, Declarative Languages, Artifact-centric Languages, Business Process Model Transformation

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 proper coordination among organizations, 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 . In fact, today's include a monitoring module to oversee the execution of fully automated business processes that can be confined within a single party. also provide dashboards to inform the process owner of the current status, bottlenecks, and possible alerts.

Unfortunately, when moving to multi-party processes, the 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 , or by deploying a centralized 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 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 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 to 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 to, 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 (), and artifact-centric analysis, as defined using (), our extended version of the () notation [**?**]. The user starts defining the multi-party process in , a widely known process modeling language. Then, for each artifact, models suited to monitor the process and the lifecycle of the artifact are semi-automatically derived from the 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) to transform artifacts into active entities that can both enact the 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 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 paper extends the based process monitoring approach presented in [**?**] by proposing a structured approach to instruct the monitoring platform, and providing an implementation of the solution. It also extends the to translation presented in [**?**] and [**?**] by also taking artifacts and their lifecycle into consideration.

The rest of the paper is organized as follows. Sect. **??** 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. Sect. **??** describes the proposed extensions to the notation, then exploited in Sect. **??**, which presents our approach. Sect. **??** argues about the correctness of the automated transformation of models into ones, while Sect. **??** introduces the distributed architecture defined for supporting the presented process compliance monitoring solution. Sect. **??** analyzes the state of the art and Sect. **??** concludes the paper.

2. Motivations

Fig. **??** shows the 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 , 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 and the *Producer* verify the identity of the *Carrier* before granting it access to their sites.

This 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.

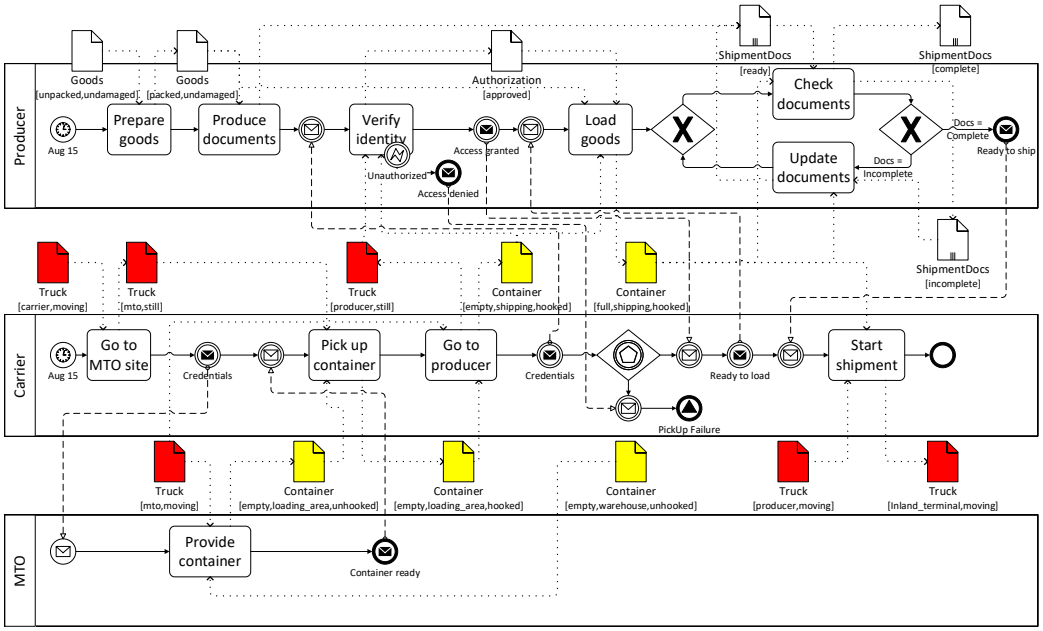

Figure 1: BPMN diagram of the running example (artifacts that should be monitored are highlighted).

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 , 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 *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 . 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 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 detects a failure in the execution since it does not comply with the model any longer. The 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 . The 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 needs explicit notifications to determine when activities outside its jurisdiction are executed. When a process involves multiple parties, and their are not configured to exchange information, or when one party does not have a , 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 about their execution. If they forget, or do not do it intentionally, the is not be able to know it.

Conformance checking tools try to overcome these limitations by inferring process models from execution traces, where 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, [**?**] 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 .

To balance ease of design and effectiveness of monitoring, our approach includes: *(i)* an extension of the notation [**?**], called , which augments the original notation with the control flow defined in [1] (see Sect. **??**); *(ii)* a complete, tool-supported solution for translating a model into an equivalent specification (see Sect. **??**).

2.2. IoT as monitoring means

The role of artifacts in multi-party business process is twofold. On the one hand, they define how an object, owned by one party, should also be

[1] We chose 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 or Activity Diagrams.

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. **??**) 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 [**?**] provides readily-available solutions for interacting with distributed objects remotely. In particular, the turns physical objects into smart objects, equipped with sensors, , and network connectivity interfaces. This is why we argue that the 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 Sect. **??**) shows how the models that result from transforming the original one are assigned to the smart objects. This requires that each smart object be equipped with an engine (see Sect. **??**) able to process the assigned model. The 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 notation [**?**] 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 rules that can predicate on control flow dependencies, external events, or data. Therefore, 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

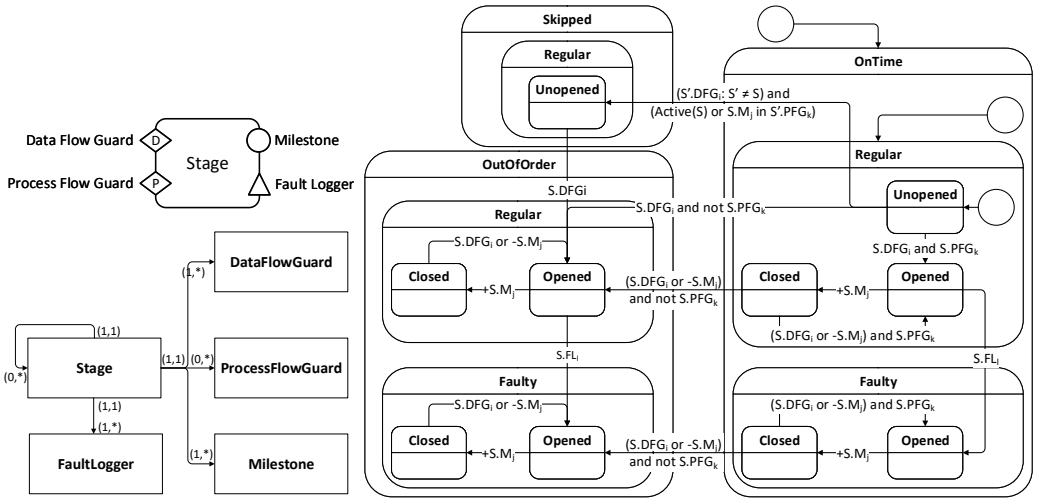

Figure 2: E-GSM meta-model (bottom left), graphical representation (top left) and life-cycle of a **Stage** (right).

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, 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 [**?**] aims to cover these limitations. Fig. **??** shows a graphical representation of the main elements proposed by that are relevant for the purposes of this paper[2].

While only uses **Guards** to decorate a **Stage**, distinguishes between **Data Flow Guards** and **Process Flow Guards**. The former borrow their semantics from **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

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

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.

also adds **Fault Loggers**, which are 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. **??** sketches the lifecycle of an **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 ($\texttt{S.DFG}_\texttt{i}$), if

its parent **Stage** is *opened*. An *opened* **Stage** becomes *closed* if either one of its **Milestones** is achieved (+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 (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 (S.DFG_i and not(S.PFG_k)). Once a **Stage** S' is declared *outOfOrder*, every other *onTime* **Stage** S that 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 and guides towards the definition of the models used to oversee the execution of both the process and its artifacts. More precisely, as shown in Fig. **??**, the starting point is a collaboration diagram [**?**], 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 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 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 artifact*s (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 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 engine fed with the two processes derived from the initial process. This

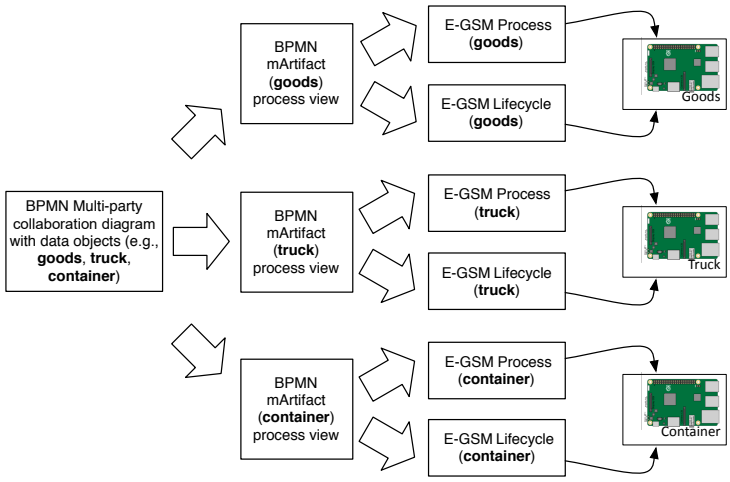

Figure 3: Overview of the transformation process.

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 Sect. **??**, Sect. **??**, and Sect. **??** respectively. Since steps (ii) and (iii) do not require human intervention, a dedicated tool[3] is in charge of them. Finally, Sect. **??** gives details on the characteristics of the 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 collaboration diagram and the following assumptions:

- Each artifact must be modeled in the process using 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

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

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, thank to Algorithm **??**, **??** and **??**, we derive a process model where the activities are organized according to the mArtifact's standpoint. This new representation will be the one monitored by the mArtifact. To derive this process, only the elements directly related to the mArtifactare 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 model representing the mArtifactview 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 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 mArtifactof interest that complies with a process diagram. Note that this new model may not be well-structured. Remarkably, when message exchanges in the original 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. **??** shows the process model obtained by applying these steps onto the process presented in Sect. **??**, 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 Sect. **??**, the derived mArtifact-oriented process view cannot be directly used to instrument the monitoring platform. However,

Algorithm 1 BPMN collaboration diagram to process view (simplified) translation

```
1:  targetBPMNModel = copy(sourceBPMNModel);
2:  discardedElements = new int[];
3:  for all i=1: targetBPMNModel.elements.count do
4:      if targetBPMNModel.elements.get(i).type == 'AtomicActivity' or targetBPMNModel.elements.get(i).type ==
    'Event' then
5:          keep = false;
6:          for all j=1: targetBPMNModel.elements.get(i).inputDataObjects.count do   ▷ keep element if it references
    artifact
7:              if targetBPMNModel.elements.get(i).inputDataObjects.get(j).artifact == artifact then
8:                  keep = true;
9:              end if
10:         end for
11:         for all j=1: targetBPMNModel.elements.get(i).outputDataObjects.count do  ▷ keep element if it references
    artifact
12:             if targetBPMNModel.elements.get(i).outputDataObjects.get(j).artifact == artifact then
13:                 keep = true;
14:             end if
15:         end for
16:         if targetBPMNModel.elements.get(i).type == 'StartEvent' or targetBPMNModel.elements.get(i).type ==
    'EndEvent' or targetBPMNModel.elements.get(i).type == 'IntermediateEvent' then          ▷ element is an event
    responsible for the collaboration
17:             if targetBPMNModel.elements.get(i).messageTo != null then          ▷ replace message throw event with
    parallel split gateway
18:                 target = targetBPMNModel.elements.get(i).messageTo;
19:                 replace(targetBPMNModel.elements.get(i),'ParallelSplitGateway');
20:                 targetBPMNModel.elements.get(i).successors.add(target);
21:                 target.predecessors.add(targetBPMNModel.elements.get(i));
22:                 keep = true;
23:             else
24:                 if targetBPMNModel.elements.get(i).messageFrom != null then  ▷ replace message catch event with
    parallel merge gateway
25:                     source = targetBPMNModel.elements.get(i).messageFrom;
26:                     replace(targetBPMNModel.elements.get(i),'ParallelMergeGateway');
27:                     targetBPMNModel.elements.get(i).predecessors.add(source);
28:                     target.successors.add(targetBPMNModel.elements.get(i));
29:                     keep = true;
30:                 end if
31:             end if
32:         end if
33:         if keep==false then                                                  ▷ discard element
34:             discardElement(targetBPMNModel,i);                          ▷ discard irrelevant elements
35:         end if
36:     end if
37: end for
38: removePools(targetBPMNModel);                                  ▷ remove pools from process model
39: removeOrphans(targetBPMNModel);                  ▷ remove discarded elements (i.e., not connected)
40: makeWellformed(targetBPMNModel);               ▷ make target model well-formed (if not already so)
```

this view can be translated into , which supports a higher level of flexibility than . In particular, control flow dependencies, which are prescriptive in , are relaxed in the model and used only to assess compliance. Similarly, data objects, which in 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 to achieve the same level of flexibility as the model without adopting a new semantics for the notation, and thus a new engine.

To perform such a translation, the following rules, which are automated in Algorithm **??**, are applied:

- We create an **Stage** S for each activity A in the model.

Algorithm 2 Helper function to discard elements not relevant to the process view

```
1:  function discardElement(BPMNModel,i)
2:      if BPMNModel.elements.get(i).boundaryEvents.count == 0 then    ▷ element has no boundary event, directly connect predecessors to successors
3:          for all j=1: BPMNModel.elements.get(i).predecessors.count do
4:              BPMNModel.elements.get(i).predecessors.get(j).successors.remove(BPMNModel.elements.get(i));
5:              BPMNModel.elements.get(i).predecessors.get(j).successors.add(BPMNModel.elements.get(i).successors);
6:          end for
7:          for all j=1: BPMNModel.elements.get(i).successors.count do
8:              BPMNModel.elements.get(i).successors.get(j).predecessors.remove(BPMNModel.elements.get(i));
9:              BPMNModel.elements.get(i).successors.get(j).predecessors.add(BPMNModel.elements.get(i).predecessors);
10:         end for
11:     else                                                    ▷ turn boundary events into gateways
12:         exclusiveGW = add(BPMNModel,'ExclusiveSplitGateway');                    ▷ create exclusive gateway
13:         inclusiveGW = add(BPMNModel,'InclusiveSplitGateway');                    ▷ create inclusive gateway
14:         for all j=1: BPMNModel.elements.get(i).boundaryEvents.count do
15:             if BPMNModel.elements.get(i).boundaryEvents.get(j).interrupting == true then    ▷ connect successors to interrupting events to exclusive split gateway
16:                 exclusiveGW.successors.add(BPMNModel.elements.get(i).boundaryEvents.get(j).successors);
17:                 for all k=1: BPMNModel.elements.get(i).boundaryEvents.get(j).successors.count do
18:                     BPMNModel.elements.get(i).boundaryEvents.get(j).successors.get(k).predecessors.remove( BPMNModel.elements.get(i).boundaryEvents.get(j));
19:                     BPMNModel.elements.get(i).boundaryEvents.get(j).successors.get(k).predecessors.add(exclusiveGW);
20:                 end for
21:             else                          ▷ connect successors to non interrupting events to inclusive split gateway
22:                 inclusiveGW.successors.add(BPMNModel.elements.get(i).boundaryEvents.get(j).successors);
23:                 inclusiveGW.successors.add(BPMNModel.elements.get(i).boundaryEvents.get(j).successors);
24:                 for all k=1: BPMNModel.elements.get(i).boundaryEvents.get(j).successors.count do
25:                     BPMNModel.elements.get(i).boundaryEvents.get(j).successors.get(k).predecessors.remove( BPMNModel.elements.get(i).boundaryEvents.get(j));
26:                     BPMNModel.elements.get(i).boundaryEvents.get(j).successors.get(k).predecessors.add(inclusiveGW);
27:                 end for
28:             end if
29:         end for
30:         if exclusiveGW.successors.count > 0 then          ▷ activity has one or more interrupting boundary events
31:             exclusiveGW.successors.add(BPMNModel.elements.get(i).successors); ▷ connect successors to activity to exclusive gateway
32:             for all j=1: BPMNModel.elements.get(i).successors.count do
33:                 BPMNModel.elements.get(i).successors.get(j).predecessors.remove(BPMNModel.elements.get(i));
34:                 BPMNModel.elements.get(i).successors.get(j).predecessors.add(exclusiveGW);
35:             end for
36:             if inclusiveGW.successors.count > 0 then    ▷ activity has also one or more non interrupting boundary events
37:                 exclusiveGW.predecessors.add(inclusiveGW);        ▷ connect inclusive gateway to exclusive gateway
38:                 inclusiveGW.successors.add(exclusiveGW);
39:                 inclusiveGW.predecessors.add(BPMNModel.elements.get(i).predecessors);  ▷ connect predecessors to activity to inclusive gateway
40:                 for all j=1: BPMNModel.elements.get(i).predecessors.count do
41:                     BPMNModel.elements.get(i).predecessors.get(j).successors.remove(BPMNModel.elements.get(i));
42:                     BPMNModel.elements.get(i).predecessors.get(j).successors.add(inclusiveGW);
43:                 end for
44:             else                                          ▷ activity has no non interrupting boundary events
45:                 exclusiveGW.predecessors.add(BPMNModel.elements.get(i).predecessors);  ▷ connect predecessors to activity to exclusive gateway
46:                 for all j=1: BPMNModel.elements.get(i).predecessors.count do
47:                     BPMNModel.elements.get(i).predecessors.get(j).successors.remove(BPMNModel.elements.get(i));
48:                     BPMNModel.elements.get(i).predecessors.get(j).successors.add(exclusiveGW);
49:                 end for
50:             end if
51:         else                                          ▷ activity has only non interrupting boundary events
52:             inclusiveGW.successors.add(BPMNModel.elements.get(i).successors); ▷ connect successors to activity to inclusive gateway
53:             for all j=1: BPMNModel.elements.get(i).successors.count do
54:                 BPMNModel.elements.get(i).successors.get(j).predecessors.remove(BPMNModel.elements.get(i));
55:                 BPMNModel.elements.get(i).successors.get(j).predecessors.add(inclusiveGW);
56:             end for
57:             inclusiveGW.predecessors.add(BPMNModel.elements.get(i).predecessors);       ▷ connect predecessors to activity to inclusive gateway
58:             for all j=1: BPMNModel.elements.get(i).predecessors.count do
59:                 BPMNModel.elements.get(i).predecessors.get(j).successors.remove(BPMNModel.elements.get(i));
60:                 BPMNModel.elements.get(i).predecessors.get(j).successors.add(inclusiveGW);
61:             end for
62:         end if
63:     end if
64:     BPMNModel.elements.get(i).successors = null;                    ▷ disconnect element to predecessors
65:     BPMNModel.elements.get(i).predecessors = null;                    ▷ disconnect element to successors
66: end function
```

Algorithm 3 Helper function to make the process model well-formed

```
1:  function makeWellformed(BPMNModel)
2:      startEvents = new element[];
3:      endEvents = new element[];
4:      for all i=1: BPMNModel.elements.count do
5:          if BPMNModel.elements.get(i).type = 'StartEvent' then    ▷ find start events already present in the model
6:              startEvents.add(BPMNModel.elements.get(i));
7:          else
8:              if BPMNModel.elements.get(i).type = 'EndEvent' then  ▷ find end events already present in the model
9:                  endEvents.add(BPMNModel.elements.get(i));
10:             end if
11:         end if
12:     end for
13:     if startEvents.count == 0 then                               ▷ if no start event is present, introduce a new one
14:         startEvent = add(BPMNModel,'StartEvent')
15:         startEvents.add(startEvent);
16:     end if
17:     for all i=0: startEvents.count do            ▷ connect to start events all the elements that have no predecessor
18:         for all j=1: BPMNModel.elements.count do
19:             if BPMNModel.elements.get(j).type != 'StartEvent' and BPMNModel.elements.get(j).predecessors.count
    == 0 then
20:                 BPMNModel.elements.get(j).predecessors.add(startEvents.get(i));
21:                 startEvents.get(i).successors.add(BPMNModel.elements.get(j));
22:             end if
23:         end for
24:     end for
25:     if endEvents.count == 0 then                                 ▷ if no end event is present, introduce a new one
26:         endEvent = add(BPMNModel,'EndEvent')
27:         endEvents.add(endEvent);
28:     end if
29:     for all i=0: endEvents.count do              ▷ connect to end events all the elements that have no successor
30:         for all j=1: BPMNModel.elements.count do
31:             if BPMNModel.elements.get(j).type != 'EndEvent' and BPMNModel.elements.get(j).successors.count ==
    0 then
32:                 BPMNModel.elements.get(j).successors.add(endEvents.get(i));
33:                 endEvents.get(i).predecessors.add(BPMNModel.elements.get(j));
34:             end if
35:         end for
36:     end for
37: end function
```

- The 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.

- We create an **Stage** S' for each start, end, or intermediate event. The 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.

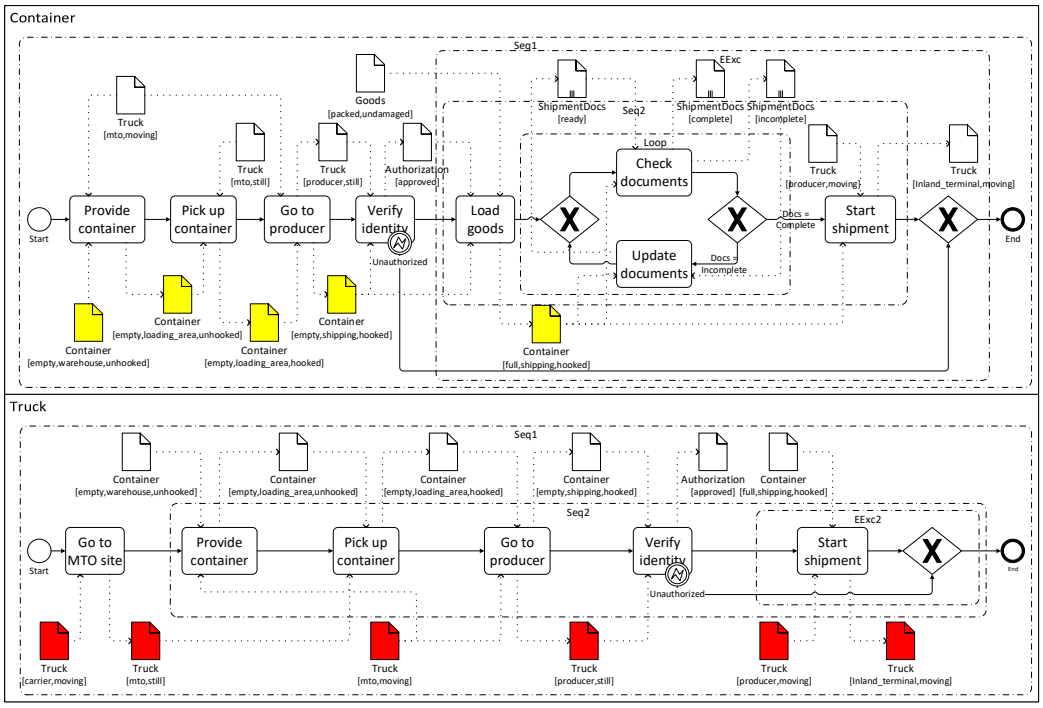

Figure 4: BPMN process model from the viewpoint of the container (top) and the truck (bottom).

> Control flow and exceptional flow patterns are managed accordingly [**?**].

Example. Fig. **??** shows the process model derived from the mArtifact-oriented process view of the container (see top of Fig. **??**). `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 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

Algorithm 4 BPMN to E-GSM (simplified) translation

```
1: processTree = decomposeProcess(sourceBPMNModel);  ▷ decompose source BPMN process model into nested blocks, return root block of process tree
2: targetEGSMModel = translate(processTree);
3: function TRANSLATE(block)
4:     stage = new stage();
5:     stage.name = block.name;
6:     if block.parent is not null then
7:         stage.processFlowGuard = generateProcessFlowGuard(block.parent) ▷ generate process flow guard based on parent block
8:     end if
9:     if block.type == 'AtomicActivity' then  ▷ block is an atomic activity
10:        dataFlowGuard = generateDataFlowGuard(block.inputDataObjects);  ▷ generate a data flow guard based on input data objects
11:        stage.dataFlowGuards.add(dataFlowGuard);
12:        milestone = generateMilestone(block.outputDataObjects);  ▷ generate a milestone based on output data objects
13:        stage.milestones.add(milestone);
14:        for all l=1: block.BoundaryEvents.count do  ▷ generate a fault logger for each boundary event attached to the activity
15:            faultLogger = generateFaultLogger(block.boundaryEvents.get(l));
16:            stage.faultLoggers.add(faultLogger);
17:            if block.boundaryEvents.get(l).type = 'interrupting' then  ▷ if the boundary event is interrupting, generate an additional milestone
18:                milestone = generateMilestone(block.boundaryEvents.get(l));
19:                stage.milestones.add(milestone);
20:            end if
21:        end for
22:    else
23:        if block.type == 'StartEvent' or block.type == 'EndEvent' or block.type == 'IntermediateEvent' then  ▷ block is an event
24:            dataFlowGuard = generateDataFlowGuard(block.trigger);  ▷ generate a data flow guard based on the event trigger
25:            stage.dataFlowGuards.add(dataFlowGuard);
26:            milestone = generateMilestone(block.trigger);  ▷ generate a data flow guard based on the event trigger
27:            stage.milestones.add(milestone);
28:        else  ▷ block contains child blocks
29:            for all k=1: block.children.count do
30:                child = translate(block.children.get(k));  ▷ recursively invoke the function for each child block
31:                stage.substages.add(child);  ▷ add substages produced from child blocks
32:                stage.dataFlowGuards.add(child.dataFlowGuards);  ▷ add data flow guard from child block
33:            end formilestone = generateMilestone(block);  ▷ generate milestone based on block type
34:            stage.milestones.add(milestone);
35:        end if
36:    end if
37: end function
```

`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 process model derived in Sect. **??** 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 Sect. **??**, 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 (Sect. **??**), 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. **??**). Such a state machine is obtained by adopting the approach in [**?**], 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 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 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.

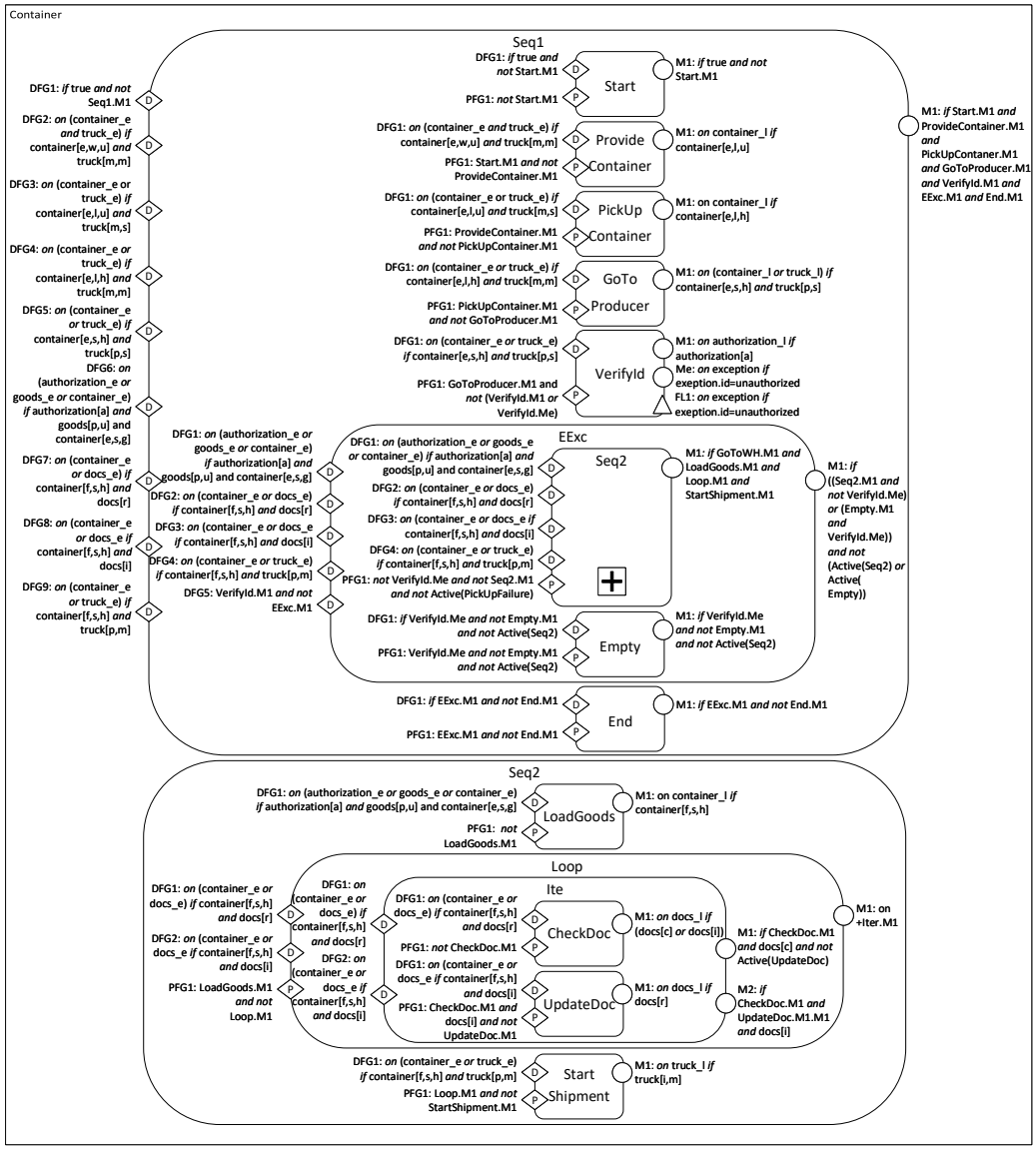

Figure 5: E-GSM process model of the container (top). For the sake of clarity, **Stages** inside `Seq2` are shown separately (bottom).

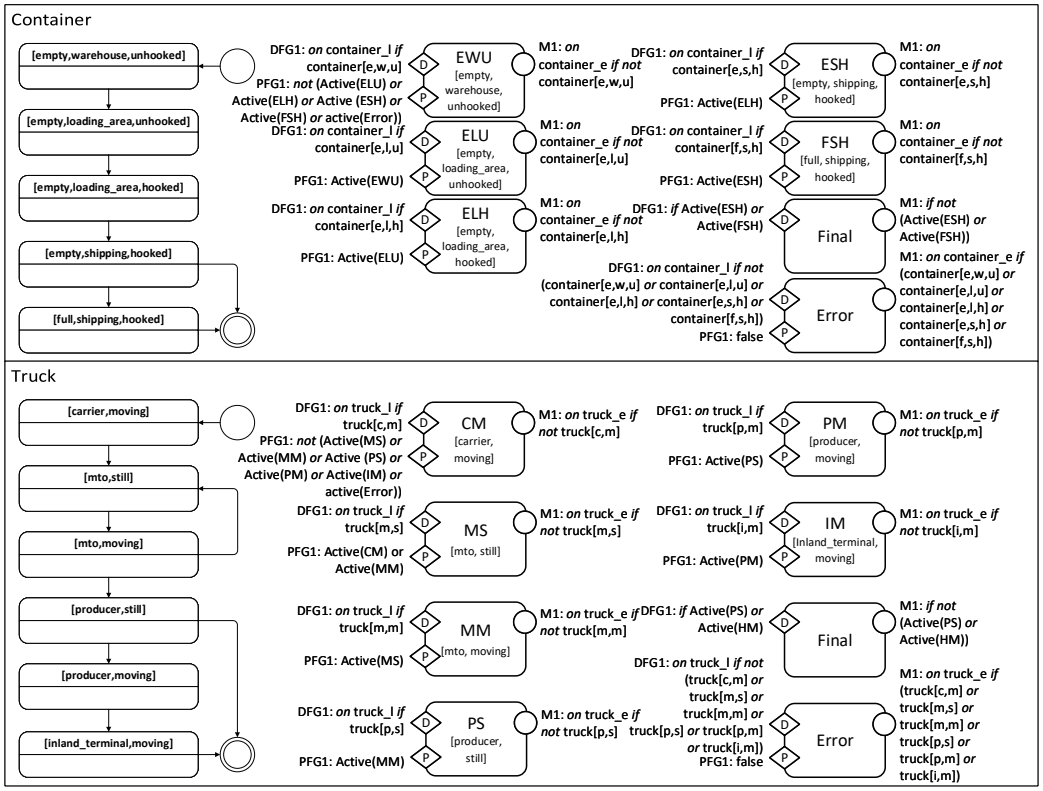

Figure 6: Finite State Machine (left) and E-GSM model (right) representing the lifecycle of the container (top) and the truck (bottom).

To monitor the lifecycle of the mArtifact, is again adopted to model the admissible state transitions. To this aim, Algorithm **??** translates the finite state machine to 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`.

Data Flow Guards and **Milestones** allow us to keep track of the current state of the mArtifact: when it is in a specific state, the corresponding **Stage** are 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, 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.

Algorithm 5 Finite state machine to E-GSM (simplified) translation

```
1: fsm = produceFSM(sourceBPMNModel, artifact);  ▷ derive artifact's lifecycle from source BPMN model as finite state machine
2: targetEGSMModel = new stage();
3: targetEGSMModel.name = artifact.name;
4: for all i=1: fsm.states.count do  ▷ generate a stage for each state
5:     stage = new stage();
6:     stage.name = fsm.states.get(i).name;
7:     dataFlowGuard = new dataFlowGuard();
8:     dataFlowGuard.condition = 'on '+artifact.name+'_l if '+artifact.name+' ['+fsm.states.get(i).name+']';  ▷ trigger data flow guard when the artifact assumes the state
9:     stage.dataFlowGuards.add(dataFlowGuard);
10:    milestone = new milestone();
11:    milestone.condition = 'on '+artifact.name+'_e if not '+artifact.name+' ['+fsm.states.get(i).name+']';  ▷ trigger milestone when the artifact assumes a different state
12:    state.milestones.add(milestone);
13:    processFlowGuard = new processFlowGuard();
14:    if fsm.states.get(i).initial == true then
15:        nonpredecessors = new string[];  ▷ state is initial
16:        for all j=1: fsm.states.count do  ▷ find states that are not predecessors
17:            if ( thenfsm.states.get(j).successor!=fsm.states.get(i))
18:                nonpredecessor = 'Active('+fsm.states.get(j).name()+')';
19:                nonpredecessors.add(nonpredecessor);
20:            end if
21:        end for
22:        processFlowGuard.condition = 'not('+concatStrings('or',nonpredecessors)+')';  ▷ require no non-predecessor to be active
23:    else  ▷ state is not initial
24:        predecessors = new string[];
25:        for all j=1: fsm.states.get(i).predecessors.count do
26:            predecessor = 'Active('+fsm.states.get(i).predecessors.get(j).name()+')';
27:            predecessors.add(predecessor);
28:        end for
29:        processFlowGuard.condition = concatStrings('or',predecessors);  ▷ require at least one predecessor to be active
30:    end if
31:    stage.processFlowGuard = processFlowGuard;
32:    targetEGSMModel.substages.add(stage);  ▷ add state to E-GSM model
33: end for
34: errorStage = new stage();
35: errorStage.name = 'Error';  ▷ generate error stage
36: states = new string[];
37: for all i=1: fsm.states.count do  ▷ find all states
38:    state = '+artifact.name+' ['+fsm.states.get(i).name()+']';
39:    states.add(state);
40: end for
41: dataFlowGuard = new dataFlowGuard();
42: dataFlowGuard.condition = 'on '+artifact.name+'_l if not (' + concatStrings('or',states)+')';  ▷ trigger data flow guard when artifact assumes a state different from the ones in finite state machine
43: errorStage.dataFlowguards.add(dataFlowGuard);
44: milestone = new milestone();
45: milestone.condition = 'on '+artifact.name+'_e if '+concatStrings('or',states);  ▷ trigger milestone when artifact assumes a state from the ones in finite state machine
46: errorStage.milestones.add(milestone);
47: processFlowGuard = new processFlowGuard();
48: processFlowGuard.condition = 'false';  ▷ error stage is always non-compliant
49: errorStage.processFlowGuard = processFlowGuard;
50: targetEGSMModel.substages.add(errorStage);  ▷ add error stage to E-GSM model
51: finalStage = new stage();
52: finalStage.name = 'Final';  ▷ generate final stage
53: finalStates = new string[];
54: for all i=1: fsm.states.count do  ▷ find final states
55:    if fsm.states.get(i).final == true then
56:        state = 'Active('+fsm.states.get(i).name()+')';
57:        finalStates.add(state);
58:    end if
59: end for
60: dataFlowGuard = new dataFlowGuard();
61: dataFlowGuard.condition = 'if (' + concatStrings('or',finalStates)+')';  ▷ trigger data flow guard when artifact assumes a final state
62: finalStage.dataFlowguards.add(dataFlowGuard);
63: milestone = new milestone();
64: milestone.condition = 'if not ('+concatStrings('or',finalStates)+')';  ▷ trigger milestone when artifact asumes a non-final state
65: finalStage.milestones.add(milestone);
66: targetEGSMModel.substages.add(finalStage);  ▷ add final stage to E-GSM model
```

Example. The right portion of Fig. **??** shows the 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.

5. Correctness of the Translation

The transformation from to presented in Sect. **??** and **??** must be *correct*. To this end, we first need to precisely define what to we mean by "correctness". Intuitively, in our setting correctness captures the fact that, given a model and a process execution trace, the trace deviates from the model if and only if the 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 as soon as it actually occurs. More specifically, let: (i) $\mathcal{B}$ be the input process model of interest, obtained as a result of the methodological step shown in Sect. **??**, and obeying to the well-structuredeness assumptions mentioned before; (ii) $\mathcal{G}^P_{\mathcal{B}}$ be the process model encoding the control-flow of $\mathcal{B}$ for monitoring purposes, i.e., the result of the methodological step discussed in Sect. **??**; (iii) $\mathcal{G}^A_{\mathcal{B}}$ be the model encoding the lifecycle of the mArtifact, i.e., the result of the methodological step discussed in Sect. **??**. 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. **??**); conversely, if the trace contains a deviation, then such a deviation is promptly 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. This 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 [**?**]. Here 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 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 as a workflow net), 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 (see, e.g., [**?**]). This straightforwardly extends to partial traces, by simply checking whether the partial trace is a prefix of a complete, conforming trace.

In our setting, we leverage the fact that $\mathcal{B}$ is well-structured, and adopt an alternative definition of conformance that has three advantages: *(i)* it is modularly defined over the different types of blocks that may be employed to structure $\mathcal{B}$; *(ii)* it is applicable also to , thus providing the basis for comparing $\mathcal{B}$ and $\mathcal{G}^P_{\mathcal{B}}$ in terms of control flow alignment; *(iii)* it is fully compatible with the aforementioned definition of conformance, i.e., a trace conforms to $\mathcal{B}$ in our sense *iif* it is the prefix of a trace leading from the input to the output place of the workflow net corresponding to $\mathcal{B}$[4].

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

[4]The workflow net is constructed using standard mechanisms, leveraging the well-structuredeness of $\mathcal{B}$, and introducing special transitions for events.

part of Fig. ?? 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 the first task can be activated as soon as the sequence block is activated, that the second task can activated as soon as the first task is completed, and that the sequence block is deactivated as soon as the second task is completed.

5.2. Control Flow Alignment

Given the notions of acceptance of an execution step and of activation of a block sketched in Sect. ??, it is possible to derive a direct, modular correspondence between $\mathcal{B}$ and $\mathcal{G}^P_{\mathcal{B}}$. More specifically, the translation procedure (cf. Sect. ??) 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 $\mathcal{B}$ (possibly introducing single, intermediate stages to handle the semantics of the corresponding block in $\mathcal{B}$). See Fig. ?? 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 Sect. ??, we then proceed by induction on the structure of the process tree. In particular, we show that, given an

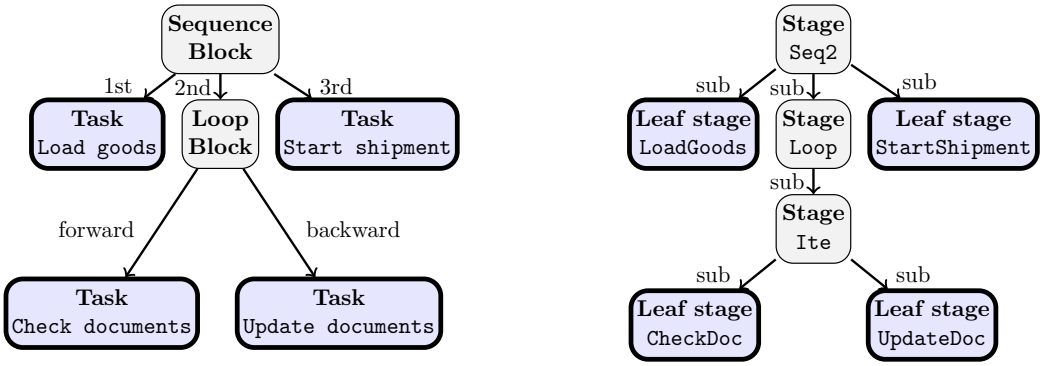

Figure 7: Block structure of the `Seq2` fragment of the model in Fig. **??** and of its corresponding translation in Fig. **??**

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 `t`:

- if `t` is considered as a deviation by $\mathcal{B}$ in s, then the execution of `t` in s causes a **Stage** of $\mathcal{G}^P_\mathcal{B}$ to become *outOfOrder*;
- if `t` is accepted by $\mathcal{B}$ in s, then the new execution state s' resulting from the execution of `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*, *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 only when the currently processed execution step is indeed considered so by $\mathcal{B}$.

5.3. Lifecycle Alignment

Lifecycle alignment amounts to check whether $\mathcal{G}^A_\mathcal{B}$ is constructed by properly considering the mArtifact transitions induced by $\mathcal{B}$. However, $\mathcal{G}^A_\mathcal{B}$ incorporates **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** and to simultaneously open the **Stage** corresponding to s_2. In this light, lifecycle alignment amounts to check 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, can be proven in two steps. In the first step, we rely on the correctness of the method proposed in [**?**], which encodes the state-transitions of the input 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 $\mathcal{M}$; *(ii)* produces exactly $\mathcal{G}^A_{\mathcal{B}}$ from $\mathcal{B}$. Correctness of lifecycle then directly follows.

6. Implementation

Fig. **??** 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 models derived in Sect. **??**. 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, thank 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[5]. 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

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

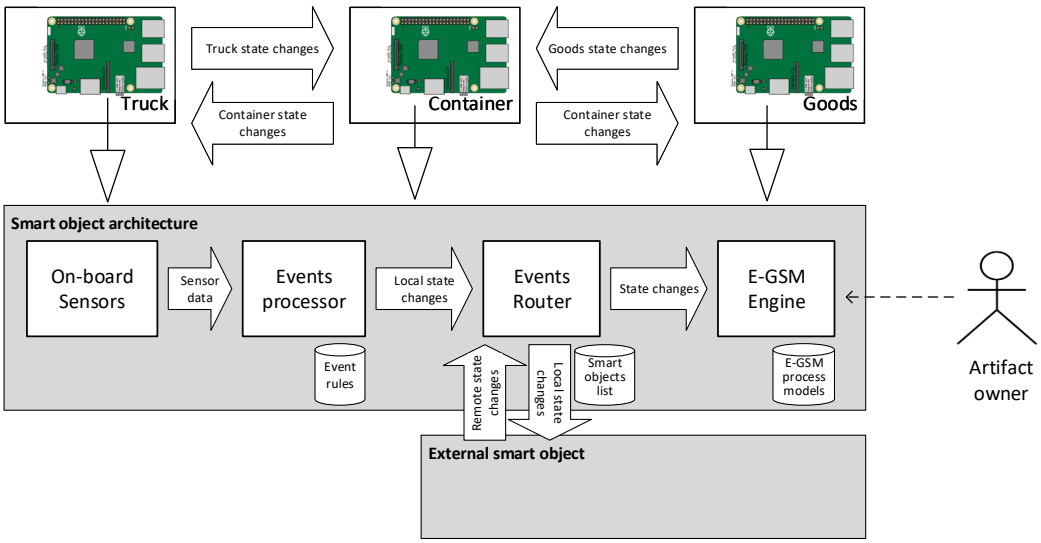

Figure 8: Infrastructure of our IoT-based monitoring platform.

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 [**?**]) that runs on a dedicated server or in the cloud[6]. As soon as the mArtifact changes state, the *Event Processor* produces two eventsto 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*[7] 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.

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

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

The *Events Router* then feeds the *Engine*[8] with the actual state changes of the mArtifact. The engine hosts the two 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 process model. The Engine also provides a simple , 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[9] 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 [10], 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 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[11]. 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 representation of the process[12]. 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 start, end or intermediate events). This way, we can demonstrate that, given proper inputs, the implemented architecture can 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

[8]Source code of the Engine is available at `https://bitbucket.org/polimiisgroup/egsmengine`.

[9]See `https://nodejs.org`.

[10]See `http://mqtt.org`.

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

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

with an 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 Sect. **??** relevant for a container and a truck. To do so, for both of them, we fed their 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 scanner detects the tag that identifies the loading area, the scales detects a weight less 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 process model of the container, `Start`, `ProvideContainer` and `PickUpContainer` are closed. On the other hand, in the 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 engine of the container, that detects a violation both in the control flow and in the lifecycle. In the 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 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 engine of the truck, that detects a violation only in the con-

trol flow. Similarly to the container, in the 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 lifecycle model.

Fig. **??** shows the interface reporting the status of the smart device related to the container according to the described scenario. On the left, stages are represented with different colors with respect to their status, whereas on the right 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

According to [**?**], 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 [**?**], BPath [**?**], Mobucon EC [**?**], and SeaFlows [**?**]. 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, [**?**] proposes the integration of a and a engine to detect when activities are executed based on external events. is extended with to identify which events produced by the CEP engine determine the activation or termination of activities, gateways, and events. [**?**] proposes an architecture that implements this solution, while [**?**] presents an approach that relies

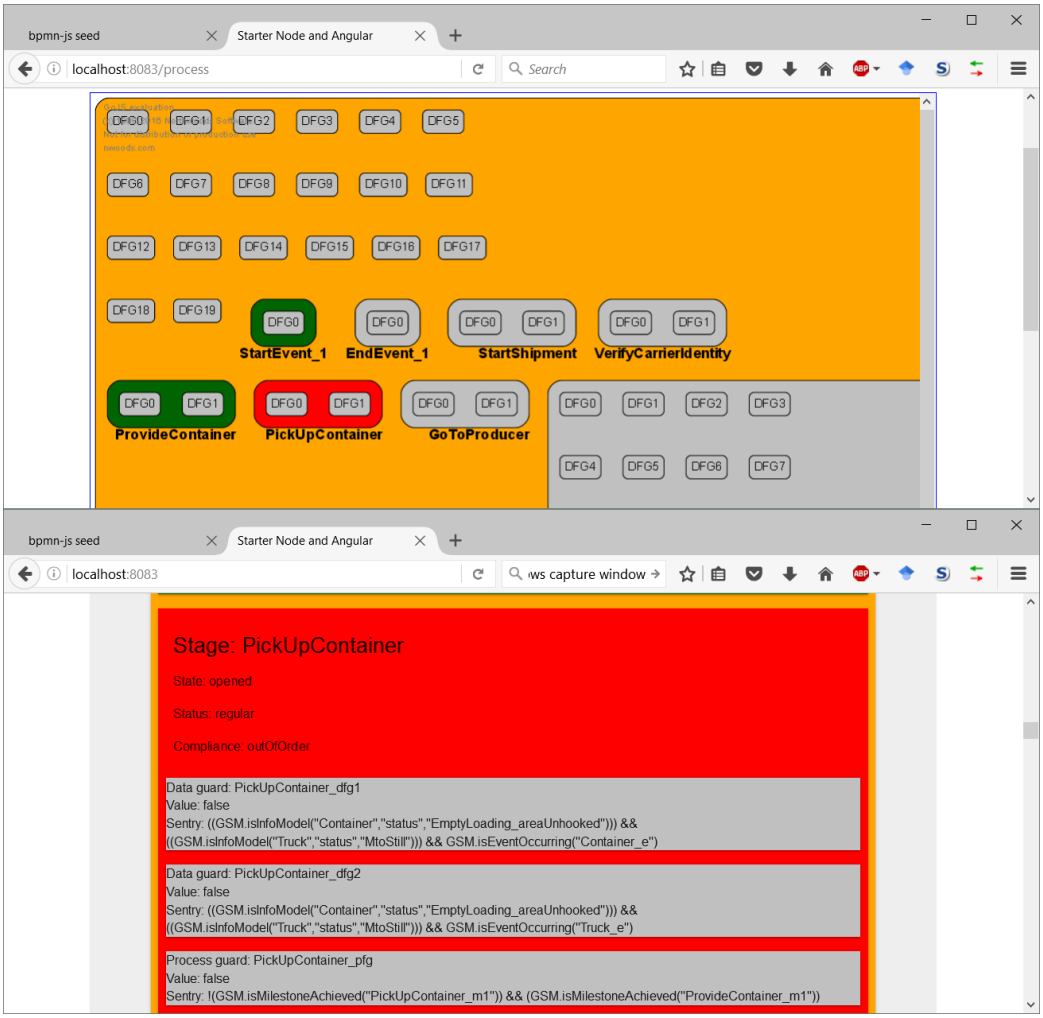

Figure 9: Screenshot of the monitoring solution.

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, [**?**] proposes transformation rules that transform a process model into a 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 model is monitoring-oriented, rather than execution-oriented. Therefore, our model must not allow to detect only the executions performed exactly as defined in the model, but potentially every possible execution.

[**?**] defines a semi-automated approach to transform a process modeled using Activity Diagrams into a model that captures the lifecycle of each involved artifact. Similarly, [**?**], [**?**] and [**?**] 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.

[**?**] defines a translator from Petri Nets to . 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 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 models, [**?**] proposes to start modeling the process with graphs, and then translate them into . While 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 [**?**]).

Concerning the usage of to monitor multi-party processes, [**?**] proposes a collaboration hub running a 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. [**?**] overcomes this limitation by adopting the 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 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 paradigm only to collect and forward sensor data to a centralized 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, [**?**] and [**?**] 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. , in contrast, does not enforce any predefined flow and uses control flow information only to detect compliance violations.

8. Conclusion

This paper explained how monitoring multi-party business processes is a challenging activity: needs for coordination among the involved , 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 add new challenges.

The proposed approach shows how a properly mix between imperative languages, used to easily model the process, and declarative languages, used to configure a monitoring system, and the adoption of the paradigm can overcome these limitations. In particular, starting from a multi-party process defined using , the paper proposed a methodology to translate this process to the notation. Smart objects are properly instrumented with the code needed to monitor different portion 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 to has been proved, thus guaranteeing that a violation is promptly detected by the engine if and only if the last processed execution step deviates from the input model.

Currently, the proposed approach requires organization 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 to 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 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.