Using the Guard-Stage-Milestone Notation for Monitoring BPMN-based Processes

Luciano Baresi, Giovanni Meroni(✉), and Pierluigi Plebani

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133 Milano, Italy
{luciano.baresi,giovanni.meroni,pierluigi.plebani}@polimi.it

Abstract. Business processes are usually designed by means of imperative languages to model the acceptable execution of the activities performed within a system or an organization. At the same time, declarative languages are better suited to check the conformance of the states and transitions of the modeled process with respect to its actual execution. To avoid defining models twice from scratch to cope with both the process enactment and its monitoring, this paper proposes an approach for translating BPMN process models to E-GSM ones: an extension of the Guard-Stage-Milestone artifact-centric notation. The paper also shows how a monitoring engine based on E-GSM specifications can detect anomalies during the execution of the process and classify them according to different levels of severity, that is, with respect to the impact on the outcome of the process.

Keywords: Guard-Stage-Milestone · Artifact-centric processes · BPMN · Process execution monitoring

1 Introduction

Process modeling represents one of the most crucial activities in Business Process Management and the goal of the resulting model is twofold. On the one hand, a business process model describes a portion of the world as it is (or as we want it to be) using a formalism easy to understand by all the relevant stakeholders (e.g., process owners and process users). On the other hand, a business process model — if properly defined in all of its parts — feeds the engine that will enact its execution. To this aim, imperative control-flow based languages are widely adopted, as their constructs and the underlying semantics are very intuitive. Among them, BPMN nowadays represents one of the most used notation adopted by both business and technical people.

However, when used for monitoring the execution of a process at run-time, imperative languages manifest a significant limitation: when a violation in the control flow occurs, an imperative process engine treats such a violation as an unhandled exception and stops monitoring the process until a user manually fixes the issue. This is not always desirable, especially when the engine has no control

R. Schmidt et al. (Eds.): BPMDS/EMMSAD 2016, LNBIP 248, pp. 18–33, 2016.
DOI: 10.1007/978-3-319-39429-9_2

on the monitored process, which would continue its execution even though the engine stopped. Declarative languages, on the other hand, do not have the notion of strict control flow. Therefore, declarative engines can both report deviations in the control flow and continue monitoring the process.

The goal of this paper is to mediate between these two perspectives by proposing a solution to monitor the execution of distributed control-flow processes modeled in BPMN, that relies on a monitoring system based on the artifact-centric Guard-Stage-Milestone (GSM) declarative language [5]. In particular, we start from a BPMN process, which is easy to conceive, and we transform it into a model defined using E-GSM, our extension of GSM. This transformation preserves control flow information, but such information, which is prescriptive in BPMN, becomes descriptive. Deviations from the "original" execution flow can easily be detected at run-time during the process enactment by analyzing the artifacts, that contain information about how the process is evolving, and represent the states through which the process should evolve during execution.

The adoption of E-GSM to drive the process monitoring introduces the following advantages. E-GSM allows one to define conditions both on the process and on external data to trigger the execution and termination of activities. Therefore, the monitoring platform can infer when activities are executed based on information coming from the environment, thus being not limited to explicit messages. Furthermore, E-GSM allows one to identify the results of the execution of the activities within the process model, and consequently it permits the identification of the activities that are incorrectly executed, if any.

The rest of the paper is structured as follows. Section 2 discusses how we extended GSM into E-GSM to enable a data-artifact driven process monitoring solution. Section 3 introduces the set of rules we defined to translate BPMN elements into equivalent E-GSM ones. Section 4 validates our work by showing how to apply the approach on a real business process in the domain of logistics. Section 5 surveys the state of the art, and Sect. 6 concludes the paper.

2 E-GSM

The GSM notation is a declarative language that allows one to model artifact-centric processes by defining conditions that determine the activation and termination of activities, called **Stages**. With respect to other declarative languages, like Declare [13], such conditions are not limited to dependencies among activities. Instead, they are based on events, which can be *external* (e.g., sent or received messages), or *internal* (e.g., termination of activities), to the process. Starting from the standard GSM notation and our preliminary work [2], we propose E-GSM, an extension to GSM where we distinguish between **Data Flow Guards** and **Process Flow Guards** and we add **Fault Loggers**.

The goal of this extension is to include information on the normal flow, that is, the expected behavior of the process, or happy path, in the artifact-centric process model. To this aim, the process model includes the dependencies among activities in terms of control flow. Being a declarative language, E-GSM does not

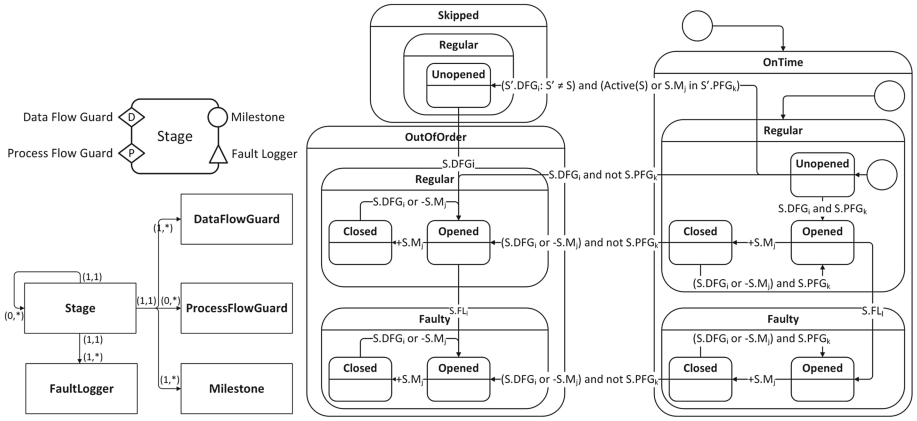

Fig. 1. E-GSM meta-model (bottom left), graphical representation (top right) and life-cycle of a Stage (right).

use control flow information to enforce a specific execution path among activities. Instead, it uses such information to let the process engine detect deviations between the happy path and how the process is actually executed.

The left portion of Fig. 1 shows a simplified version of the meta-model behind E-GSM, along with the graphical representation of its main elements. The original definition of GSM comprises **Stages**, **Guards**, and **Milestones**. A **Stage** represents the unit of work that can be executed in a process instance. A **Stage** can have one or more nested **Stages**, or it can be *atomic*, thus representing a single task. A **Stage** may be decorated with one or more **Guards** and **Milestones**.

A **Guard** (**Data Flow Guard** in E-GSM) is an Event-Condition-Action (ECA) rule[1]. If true, the associated **Stage** is declared opened. A **Milestone** is another ECA rule. If true, the **Stage** is declared closed. A **Milestone** may also have an *invalidator*: a boolean expression that can invalidate the **Milestone** and reopen the **Stage**.

In the proposed extension, a **Stage** can now also be decorated with **Process Flow Guards** and **Fault Loggers**. A **Process Flow Guard** is a boolean expression that predicates on the activation of the **Data Flow Guards** and **Milestones** used to map the expected control flow. The expression is evaluated once one of the **Data Flow Guards** of the associated **Stage** is triggered, and before the **Stage** becomes opened. If the expression is true, the **Stage** complies with the expected execution, otherwise the **Stage** has been activated without respecting the normal flow.

A **Fault Logger** is an ECA rule. If true, the associated **Stage** is declared faulty because something went wrong during the execution of the activity.

[1] An ECA rule is an `[on e] [if c]` expression, that is triggered when an event e occurs and the condition c is true. When `[one]` is missing, the ECA is triggered once c becomes true, when `[if c]` is missing, the ECA is triggered once e occurs.

A faulty **Stage** does not imply its termination, as the termination is only determined by **Milestones**.

The right portion of Fig. 1 sketches the lifecycle of an E-GSM **Stage** organized around three main orthogonal execution perspectives: outcome, compliance, and status[2].

The *Execution outcome* captures the situation of a **Stage**, which can be either *regular* (none of its **Fault Loggers** has ever been triggered) or *faulty* (at least one of its **Fault Loggers** has been triggered, $\texttt{A.FL}_\texttt{l}$).

The *Execution compliance* captures the compliance of each **Stage** with the normal flow. A **Stage** is declared *onTime* by default. It can become *outOfOrder* (according to the normal flow) when one of its **Data Flow Guards** is triggered but none of its **Process Flow Guards** holds (`A.DFG and not(A.PFG)`). If a **Stage** `S'` is declared *outOfOrder*, every other *onTime* **Stage** `S` that would trigger one of the **Process Flow Guards** of `S'` ($\texttt{S.M}_\texttt{j}$ `or Active(S)` $\in$ $\texttt{S'.PFG}_\texttt{k}$) is declared *skipped*. If a **Stage** is *skipped*, once one of its **Data Flow Guards** is triggered ($\texttt{S.DFG}_\texttt{i}$), it becomes *outOfOrder*.

The *Execution status* captures the status of a **Stage**: *unopened*, *opened* or *closed*. A **Stage** is *unopened* if its **Data Flow Guards** have never been triggered. A **Stage** can become *opened* only if it is *unopened* or *closed* and the parent **Stage** is *opened*. In addition, at least one of its **Data Flow Guards** must be triggered ($\texttt{S.DFG}_\texttt{i}$). A **Stage** becomes *closed* if it is *opened* and a **Milestone** is achieved ($\texttt{+S.M}_\texttt{j}$), or if the parent **Stage** becomes *closed*.

The combination of these three perspectives says that the whole lifecycle assumes that a **Stage** is initially *onTime*, *regular*, and *unopened*. **Data Flow Guards** drive the change of execution status. **Fault Loggers** drive the outcome, while **Process Flow Guards** are in charge of the compliance. With respect to Standard GSM, E-GSM interprets reopening a *closed* **Stage** as a new iteration of that process portion. Therefore, once a parent **Stage** is reopened (i.e., it moves from *closed* to *opened*), the lifecycle of all its child **Stages** will restart from scratch.

Thank to these three perspectives, it is possible to detect at runtime when a deviation in the execution of a process occurs and which stages are involved. This enables a classification that predicates on the lifecycle of all stages to evaluate how severely variations during execution affect the outcome of the process. For example, Table 1 reports a possible classification of severity that can be modified according to any specific scenario: *None*, if all activities are executed at the right time and their execution was successful. *Low*, if the process terminated, the expected control flow was not respected, yet no activity was skipped and they were all successfully executed. *Medium-low*, if an activity was incorrectly executed, but the expected control flow was respected. *Medium*, if the process

[2] In this paper we use the notation introduced in [5], so we write $\texttt{S.DFG}_\texttt{i}$, $\texttt{S.PFG}_\texttt{k}$, $\texttt{S.FL}_\texttt{l}$ to indicate the activation of a Data Flow Guard, Process Flow Guard, or a Fault Logger associated with Stage `S`, $\texttt{+S.M}_\texttt{j}$ ($\texttt{-S.M}_\texttt{j}$) to indicate the achievement (invalidation) of a Milestone $\texttt{M}_\texttt{j}$, $\texttt{S.M}_\texttt{j}$ to indicate that Stage `S` is closed and a Milestone $\texttt{M}_\texttt{j}$ is achieved, and `Active(S)` to indicate that Stage `S` is opened.

Table 1. Severity levels. $S_x.o$, $S_x.c$ and $S_x.s$ indicate the state of **stage** S_x, along with the execution outcome, compliance and status respectively.

Severity	Execution outcome ($S_y.o$)	Execution compliance ($S_z.c$)	Execution status ($S_x.s$)
None	$\forall S_y : S_y.o = regular$	$\forall S_z : S_z.c = onTime$	$\forall S_x : S_x.s = unopened$ $\vee S_x.s = opened$ $\vee S_x.s = closed$
Low	$\forall S_y : S_y.o = regular$	$\exists S_z : S_z.c = outOfOrder$	$\forall S_x : S_x.s = unopened$ $\vee S_x.s = closed$
Medium-low	$\exists S_y : S_y.o = faulty$	$\forall S_z : S_z.c = onTime$	$\forall S_x : S_x.s = unopened$ $\vee S_x.s = opened$ $\vee S_x.s = closed$
Medium	$\forall S_y : S_y.o = regular$	$\exists S_z : S_z.c = outOfOrder$ $\vee S_z.c = skipped$	$\exists S_x : S_x.s = opened$
Medium-high	$\forall S_y : S_y.o = regular$	$\exists S_z : S_z.c = skipped$	$\forall S_x : S_x.s = unopened$ $\vee S_x.s = closed$
High	$\exists S_y : S_y.o = faulty$	$\exists S_z : S_z.c = outOfOrder$ $\vee S_z.c = skipped$	$\forall S_x : S_x.s = unopened$ $\vee S_x.s = opened$ $\vee S_x.s = closed$

is still in progress and, during execution, the expected control flow was not respected. *Medium-high*, if the process terminated and an activity was skipped. *High*, if an activity was incorrectly executed and no corrective action was taken (i.e., at least another activity was either skipped or incorrectly executed).

This classification assumes that all stages have the same importance. However, weights can be introduced to differentiate the influence of each specific stage on the process, or metrics taken from the conformance checking domain [16] can be adopted.

3 Transformation Rules

The aforementioned semantics of E-GSM is then used in 13 transformation rules [10] to translate a BPMN process model into an E-GSM one.

These transformation rules are applicable to every BPMN process model that complies with a workflow net [1], that is, the process has only one start event and only one end event, and it always terminates (soundness). Note that the control flow is always captured by Process Flow Guards, and as such it is never enforced. This allows the E-GSM model to continue monitoring a process even if violations in the control flow occur.

3.1 Basic Elements

The transformation rules defined for basic elements are presented in Fig. 2.

Rule 1. *A BPMN Activity* `A` *is translated into a Stage* `A` *with one or more Data Flow Guards (*$A.DFG_i$*) and one or more Milestones (*$A.M_j$*).*

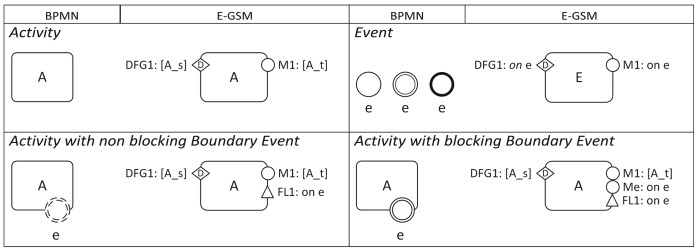

Fig. 2. BPMN to E-GSM transformation rules for basic elements.

Producing the conditions associated with those **Data Flow Guards** and **Milestones** is far from trivial [3]. They depend on the associated data objects and, if the activity is a task, on its type (i.e., *receive* or *user* task). In case of a generic task, placeholders `A_s` and `A_t` are associated with, respectively, `A.DFG1` and `A.M1` to represent the explicit start and termination of the activity. If the activity is a sub-process, $\texttt{A.DFG}_\texttt{i}$ and $\texttt{A.M}_\texttt{j}$ are then derived from the structure of the sub-process and from its elements, as explained in the following.

Rule 2. *A BPMN Start, End or Intermediate Event* `e` *is translated into a Stage* `E` *where* `E.DFG1` *and* `E.M1` *have the occurrence of the event as condition.*

Rule 3. *A BPMN Activity* `A` *with a non-interrupting Boundary Event* `e` *attached is translated into a Stage* `A` *according to Rule 1 with* `A.FL1` *having the occurrence of the event as condition (i.e.,* `on e`*).*

Rule 4. *A BPMN Activity* `A` *with an interrupting Boundary Event* `e` *attached is translated into a Stage* `A` *according to Rule 1 with an additional Milestone* `A.Me` *and* `A.FL1` *having the occurrence of the event as condition.*

3.2 Normal Flow

The combination of the above rules for basic elements allows one to translate well-structured business process models [15]. In particular, we focus on five types of blocks, defined starting from the classical control flow patterns [17]:

- A *sequence block* is made of linked activities, events and other blocks without splits or merges. It corresponds to pattern *sequence*.
- A *parallel block* organizes activities, events, and other blocks in two or more parallel threads resulting from the combination of patterns *parallel split* and *synchronization*.
- A *conditional exclusive block* organizes activities, events, and other blocks in two or more branches resulting from a combination of patterns *exclusive choice* and *simple merge*.

– A *conditional inclusive block* organizes activities, events, and other blocks in two or more branches resulting from a combination of patterns *multi-choice* and *structured synchronized merge*.
– A *loop block* organizes activities, events, and other blocks according to pattern *structured loop*.

For each of these blocks, we delivered proper transformation rules in [10]. A graphical representation of them is reported in Fig. 3. Due to space constraints, in this paper we will only describe in detail how sequence, conditional exclusive, and loop blocks are translated.

Rule 5. *A sequence block corresponds to a Stage* `Seq` *that includes* S_x *inner Stages obtained by applying the transformation rules to all the elements (i.e., Activities, Events, inner blocks) that belong to the block.*

– *In addition to the existing Process Flow Guards, each inner stage has* S_x`.PFG1` *to state that none of its Milestones is achieved, and at least one of the Milestones of the element that directly precedes it (if present) is achieved.*
– `Seq` *has a set* `Seq.DFG` *that includes all* S_x`.DFG`$_i$*, and a Milestone* `Seq.M1` *that requires that, for all* S_x*, at least one* S_x`.`M_j *be achieved.*

Rule 6. *A conditional exclusive block is translated into a Stage* `Exc` *that includes all the Stages obtained by applying Rule 5 to all its branches, which result in* S_x *inner Stages.*

– *For each* S_x*,* S_x`.PFG1` *is added to check that no* S_x`.`M_j *has already been achieved, that the condition on the branch from which* S_x *is produced (if present) is satisfied, and that none of the other inner Stages is opened (i.e.,* `not Active(`S_y`)` *where* $y \neq x$*).*
– `Exc` *has a set* `Exc.DFG` *that includes all* S_x`.DFG`$_i$*, and a Milestone* `Exc.M1` *that requires that, for at least one* S_x*, one* S_x`.`M_j *be achieved, and the condition on the branch from which* S_x *is produced (if present) be satisfied, as long as none of the other inner Stages is opened.*

Rule 7. *A loop block is translated into two Stages,* `Ite` *and* `Loop`*.* `Ite` *includes* S_x *inner Stages obtained by applying Rule 5 to all the branches within the loop block. One of these stages is a forward Stage, that is, its control flow goes in the same direction as the control flow that includes the loop block. The others are backward Stages.*

– *For all the inner Stages,* S_x`.PFG1` *is added to check that no* S_x`.`M_j *is already achieved. Moreover, if* S_x *is a backward stage,* S_x`.PFG1` *also requires that the condition on the branch (if present) be satisfied, and that one of the Milestones of the forward stage be achieved.*
– `Ite` *has a set* `Ite.DFG` *that includes all* S_x`.DFG`$_i$*, and two Milestones, where:*
  - `Ite.M1` *requires that one of the Milestones of the forward Stage be achieved and the exit condition of the loop (if present) be satisfied, as long as no backward Stage is opened.*

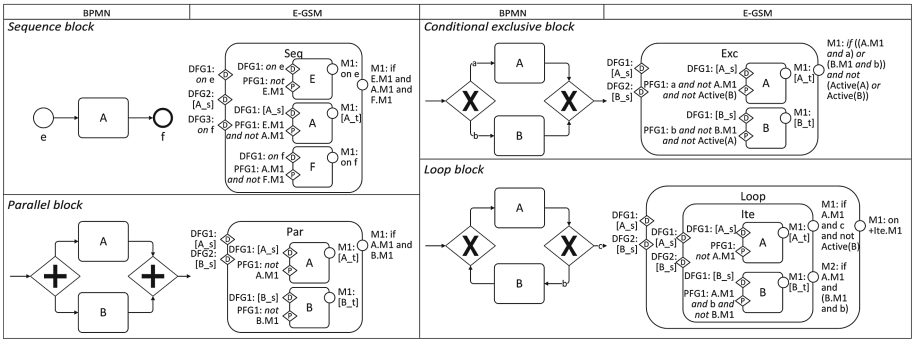

Fig. 3. BPMN to E-GSM transformation rules for normal flow blocks (due to space constraints, the conditional inclusive block is not presented).

- *`Ite.M2` requires that one of the Milestones of the forward Stage be achieved and, for at least a backward Stage, one of its Milestones be achieved and the condition on that branch (if present) be satisfied, as long as none of the other backward Stages is opened.*

Stage `Loop` *includes* `Ite` *and has* `Loop.DFG = Ite.DFG` *and* `Loop.M = on Ite.M1` *(i.e., the process can exit the loop).*

The iteration **Stage** `Ite` has no **Process Flow Guards** since it is supposed to be executed multiple times and, every time it becomes opened, a new iteration of the loop is carried out. Thus, `Ite` is opened when at least one of its inner Stages can be opened too, and it is closed when either the process can exit the loop (`Ite.M1` is achieved), or when an iteration is complete (`Ite.M2` is achieved).

3.3 Exceptional Flow

BPMN supports the management of foreseen exceptions through boundary events, that is, events directly attached to activities. These events, like split gateways, determine a branching of the control flow into an *exceptional* flow, which leaves the boundary event, and a *normal* flow, to continue the execution from the activity. If the foreseen exception occurs while executing the activity, the attached boundary event activates the exceptional flow. A dedicated set of rules shown in Fig. 4 is thus required to preserve this behavior in E-GSM models. Again, we refer to [10] for the details.

Interrupting boundary events cause the normal and exceptional flows to be mutually exclusive, therefore we expect them to be merged by an exclusive merge gateway at the end. This requires that two additional blocks, called *forward exception handling* and *backward exception handling*, respectively, be defined. The forward exception handling block comprises an interrupting boundary event, and a *simple merge*, defined with a BPMN exclusive gateway, that merges the exceptional control flow and the portion of the normal control flow that follows the activity to

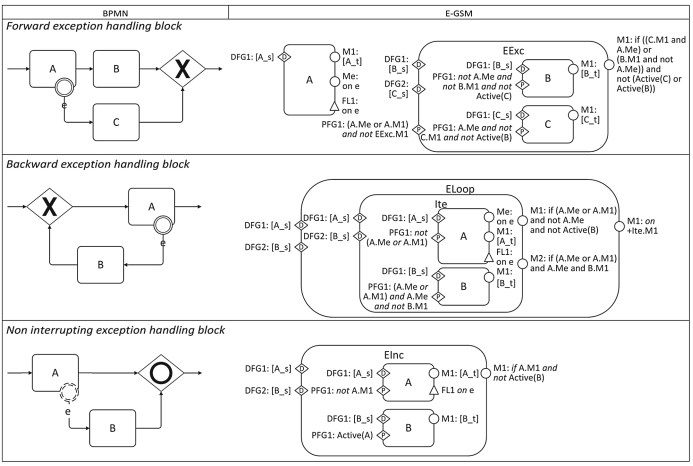

Fig. 4. BPMN to E-GSM transformation rules for handling exceptions.

which the boundary event is attached. Its behavior is similar to the one of the conditional exclusive block, with the exception of the branch condition, which predicates on the achievement of the milestone derived from the boundary event. The backward exception handling block comprises an interrupting boundary event and a *simple merge*, defined with a BPMN exclusive gateway, that merges the exceptional control flow and the portion of the normal control flow that precedes the activity to which the boundary event is attached. This block produces a loop that allows one to re-execute part of the normal control flow if the boundary event is triggered, and therefore it is translated similarly to a loop block.

In BPMN, boundary events could also be non interrupting, that is, they activate the exceptional control flow without terminating the associated activity. Therefore, the elements within the exceptional control flow can run in parallel with the normal flow that starts from the activity the boundary event is associated with. Since we expect these potentially simultaneous control flows be merged by an inclusive merge gateway, the transformation requires an additional block, called *non interrupting exception handling block*. This new block comprises a non interrupting boundary event to split the execution flow into an exceptional flow and the continuation of the normal one, and a *structured synchronized merge*, defined with a BPMN inclusive gateway, to merge the two flows in case the exception occurred.

4 Validation

The transformation rules introduced in the previous section allow any well-structured BPMN process model to be translated into E-GSM. To prove it, we developed a BPMN to E-GSM prototype translator[3], where the transformation rules

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

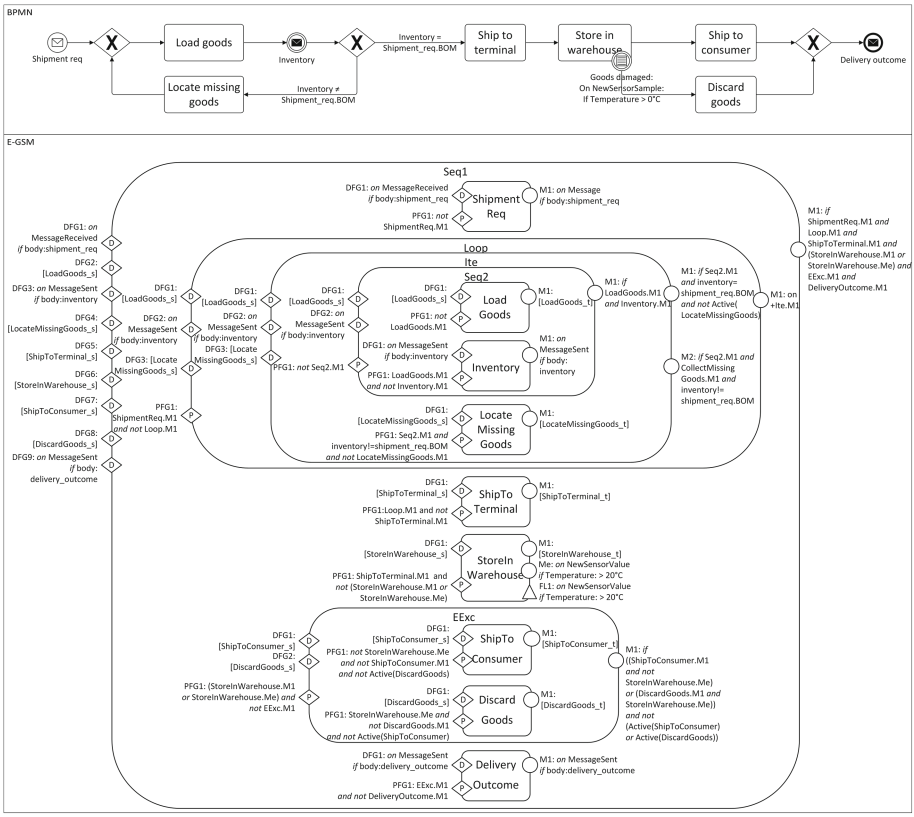

Fig. 5. BPMN and E-GSM models of the example shipping process.

are implemented in ATL (ATLAS Transformation Language [6]), and validated —and refined— the proposed rules against several BPMN business processes with different levels of complexity. A formal verification about the equivalence between BPMN processes and their correspondent E-GSM is under study and it aims to check if all the traces that a BPMN process can produce are also considered as satisfied in the E-GSM model.

Among these test processes, here we concentrate on an example taken from the logistics domain, which is shown at the top of Fig. 5, to better explain the advantages of adopting E-GSM to monitor the execution of complex (distributed) processes. A pharmaceutical company M has to ship drugs (that are highly susceptible to temperature variations) to one of its customers N. To do so, it relies on two shipping companies R and T for, respectively, rail and truck transportation, and on an inland terminal I for changing means. The shipping process starts when a shipment request by N is received, and comprises four main phases: (i) loading goods into a thermally-insulated shipping container; (ii) shipping such a container to I by rail; (iii) temporarily storing the container in a temperature-controlled warehouse;

(iv) delivering the goods to the customer's site by truck. Before starting phase (ii), an inventory report of the contents of the container must be produced, and it must be compared with the bill of materials included with the shipment request. If some products are missing, they must be located and loaded onto the container, and a new report must be produced. Furthermore, if the goods are exposed to a temperature higher than 20°C during phase (iii), they must be discarded and the whole process must be aborted. Our translator produces the E-GSM model shown at the bottom of Fig. 5.

Since all these activities interact with the shipping container, we can think of it as the process coordinator (i.e., the element that interacts with all the parties and has complete visibility on the whole process). To make the container process-aware, we can exploit the Internet of Things paradigm by equipping it with a single board computing device, sensors and a network interface, thus transforming it into a smart object (i.e., smart container). However, being the container completely passive, it cannot enforce the execution flow modeled in the process, and it needs information to identify when each activity is being executed. For this reason, a traditional process engine would be unsuited to monitor this process. On the other hand, an E-GSM engine[4] running onto the smart container would solve this problem: By predicating on on-board sensor values or explicit messages, **Data Flow Guards**, **Process Flow Guards**, **Milestones** and **Fault Loggers** can be triggered, and the execution of the process be monitored. This way, once a violation in the execution occurs, the E-GSM engine can report to stakeholders which activities are affected, and how severely the whole process is affected by such an incident.

To show how process monitoring can take advantage of the E-GSM model, we describe three possible scenarios.

4.1 An Error-Free Execution

Once the shipment request is received, `Seq1.DFG1` is triggered and, consequently, `Seq1` becomes opened (thus starting the process). This first triggers `ShipmentReq.PFG1`, then `ShipmentReq.DFG1`, which causes `ShipmentReq` to become *opened*, and finally `ShipmentReq.M1` be achieved, which moves `ShipmentReq` to the *closed* state, and triggers `Loop.PFG1`.

When R loads the goods onto the container, a notification is sent to the engine, which triggers `Loop.DFG1`, then `Ite.DFG1`, `Seq2.PFG1`, `Seq2.DFG1`, `LoadGoods.PFG1`, and finally `LoadGoods.DFG1`, which moves `Loop`, `Ite`, `Seq2` and `LoadGoods` to the *opened* state. After finishing loading the goods, the operator sends another notification, thus making `LoadGoods.M1` be achieved, which triggers `Inventory.PFG1` and moves `LoadGoods` to the *closed* state. It then produces the inventory of the loaded goods, which triggers `Inventory.DFG1`, and then makes `Inventory.M1` be achieved, which makes `Seq2.M1` achieved too, causing `Inventory` and `Seq2` to move to the *closed* state. Being the inventory consistent with the bill of materials included in the shipment request, also `Ite.M1` and,

[4] A prototype E-GSM engine is currently under development.

consequently, `Loop.M1`, are achieved, moving `Ite` and `Loop` to the *closed* state, and triggering `ShipToTerminal.PFG1`.

Once the rail shipping begins, R sends a notification, which triggers `ShipToTerminal.DFG1` and moves `ShipToTerminal` to the *opened* state. When the container is delivered to I, another notification is sent, which makes `ShipToTerminal.M1` become achieved, moving `ShipToTerminal` to the *closed* state and triggering `StoreInWarehouse.PFG1`. Similarly I sends a notification when the container is put in the warehouse and when T is ready to pick it up, thus triggering `StoreInWarehouse.DFG1`, achieving `StoreInWarehouse.M1`, triggering `EExc.PFG1`, and moving `StoreInWarehouse` to the *opened* state at first, and then to the *closed* state. After hooking the container to its truck, a notification is sent by T. That notification triggers `EExc.DFG1`, then `ShipToCustomer.PFG1`, and finally `ShipToCustomer.DFG1`, thus moving stages `EExc` and `ShipToCustomer` to the *opened* state. Once T delivers the goods to N, another notification is sent. That notification causes the achievement of `ShipToCustomer.M1`, which makes `EExc.M1` become achieved too, thus moving `ShipToCustomer` and `EExc` to the *closed* state and triggering `DeliveryOutcome.PFG1`.

Finally, once the goods are inspected by N, a report of the shipment is produced, which triggers `DeliveryOutcome.DFG1`, moving `DeliveryOutcome` to the *opened* state, and then causes the achievement of `DeliveryOutcome.M1`, which causes the achievement of `Seq1.M1` too, thus moving `DeliveryOutcome` and `Seq1` to the *closed* state and, since `Seq1` represents the whole process, terminating the monitoring activity. Once the process concludes, N queries the smart container and finds out that the severity level of the process is *None*, since all stages are in state either *unopened* or *closed*, their compliance is *onTime*, and their outcome is *regular*. Therefore, N accepts the goods.

4.2 A Catastrophic Execution

A second example shows how the system can monitor an incorrect execution of the process. During phase (iii), the warehouse cooling system breaks down, and the temperature of the goods goes beyond 20°C. Being the container equipped with a temperature sensor, the E-GSM engine is able to detect such an event and consequently triggers both `StoreInWarehouse.FL1` and `StoreInWarehouse.Me`, which move `StoreInWarehouse` to the *faulty* and *closed* states. This changes the severity level of the process from *none* to *medium-low*, since a *faulty* stage exists, but all stages are still *onTime*. Being `StoreInWarehouse` closed and `StoreInWarehouse.Me` achieved, `DiscardGoods.PFG1` is also triggered.

Instead of discarding the goods, I ignores that accident, and delivers the goods to N. This moves `ShipToConsumer` to state *outOfOrder*, since `ShipToConsumer.DFG1` is triggered before `ShipToConsumer.PFG1` becomes active. This causes the severity level of the process to become *high*, since there are both a *faulty* stage (`StoreInWarehouse`), and an *outOfOrder* one (`ShipToConsumer`).

Once N receives the goods, it queries the smart container and, since the severity level of the process is *high*, decides to immediately inspect its content,

thus discovering that the goods have been spoiled. Therefore, it sends them back to M. In turn, M identifies that `StoreInWarehouse` is in the *faulty* state, and that `ShipToConsumer` is *outOfOrder*. Thank to this information, M is able to charge I a penalty for having spoiled the goods and not having reported that accident. Note that had T queried the smart container, it would have seen that the severity level was *medium-low*, since `StoreInWarehouse` was in *faulty* state, and could have avoided delivering the container to N.

4.3 A Troublesome yet Recoverable Execution

Let us now focus on a less critically incorrect execution of the process. In this case, the inventory of the container is not consistent with the bill of materials, which causes `LocateMissingGoods.PFG1` to be triggered. However, R does not check the inventory and immediately begins shipping the container to I, which moves `ShipToTerminal` to the *outOfOrder* state, since `ShipToTerminal.DFG1` is triggered before `ShipToTerminal.PFG1` becomes active. This changes the severity level of the process from *none* to *medium*, as there are both *opened* stages (`Seq1`, `Loop` and `Ite`) and an *outOfOrder* one (`ShipToTerminal`).

Once N receives the goods, it queries the smart container and finds out that the severity level is still *medium* (since the missing goods were not collected and loaded onto the container, stages `Seq1`, `Loop` and `Ite` are still *opened*). So, it inspects the contents, discovers that some of them are missing, and asks M to ship the missing ones for free. By querying the smart container, M finds out that, even though the inventory did not match the shipping request, missing goods were never collected and loaded onto the smart container (i.e., `LocateMissingGoods` has not been executed even though `LocateMissingGoods.PFG1` was satisfied), and the shipment continued anyway (i.e., `ShipToTerminal` is in state *outOfOrder*). Because of this information, M can blame R for having shipped the goods without checking the inventory first. Note that the severity level (*medium*) reflects the results of the process: being at least part of the goods successfully delivered, M did not experience a complete loss as in the previous case, where all the goods were spoiled, and the truck shipment was done pointlessly.

5 Related Work

Köpke et al. [7] propose transformation rules that transform a BPMN process model into a GSM equivalent. While we have borrowed from these rules the idea of transforming blocks into nested Stages, our transformation rules produce completely different expressions for Guards and Milestones. The reason behind such a discrepancy is that we are interested in identifying control flow violations, and not in forcing the process to rigidly follow a given execution flow, which is what is pursued in [7]. Eshuis et al. [4] define a semi-automated approach that starts from UML Activity Diagrams and produces a data-centric process model in GSM. They capture the lifecycle of the data artifacts referred to in

the UML process model, and exploit control flow information to render it in GSM. Similarly, Kumaran et al. [8] and Meyer et al. [12] 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 [4,8,12], 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 et al. [14] define 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.

Concerning the integration of both activity and data-centric perspectives in business processes, Künzle et al. [9] propose a framework that maps portions of data structures to activities and use control flow information to define how such data objects should be manipulated. Similarly, Meyer et al. [11] propose a methodology to model both the control flow and data dependencies by extending BPMN data artifacts to define dependecines among all data items manipulated in a process. Both [9,11] use control flow information in a prescriptive way, while E-GSM uses it in a descriptive way to detect deviations from the original definitions during execution.

Conformance checking is the discipline that aims at identifying inconsistencies among a process model and its execution [16]. To do so, the process model is checked against high level execution logs, which report when and if activities have been executed. Our solution differs from this approach as it is able to autonomously identify when activities start or end, without relying on an execution log. Furthermore, it is able to detect deviations at runtime, whereas most process compliance techniques are applicable only when the process terminates.

6 Conclusions and Future Work

This paper extends the Guard-Stage-Milestone (GSM) notation to embed control flow information in the process model definition, presents a solution for transforming BPMN models into equivalent E-GSM ones, and shows how the derived E-GSM process model can be used to identify when activities are executed, to keep track of violations in the execution flow, and to evaluate the overall execution of a process along with different levels of severity.

As for our future work, we will investigate how to improve Rule 1 and Rule 2 by taking into account the nature of activities (i.e., receive tasks or user tasks) and events (i.e., timer, signal, etc.), and their associated data objects. We will also propose additional transformation rules to derive the *E-GSM Information Model*, which is not considered in this work, from data objects and implicit information defined in BPMN process models, which may also influence the definition of severity levels. In parallel, we will continue applying the proposed solution and assessing it on real industrial examples.

Acknowledgments. This work has been partially funded by the Italian Project ITS Italy 2020 under the Technological National Clusters program.

References

1. Van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) Application and Theory of Petri Nets 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. Baresi, L., Meroni, G., Plebani, P.: A gsm-based approach for monitoring cross-organization business processes using smart objects (2015). Accepted for publication
3. Cabanillas, C., Baumgrass, A., Mendling, J., Rogetzer, P., Bellovoda, B.: Towards the enhancement of business process monitoring for complex logistics chains. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013 Workshops. LNBIP, vol. 171, pp. 305–317. Springer, Heidelberg (2014)
4. Eshuis, R., Van Gorp, P.: Synthesizing data-centric models from business process models. Computing **98**, 1–29 (2015)
5. Hull, R., Damaggio, E., Fournier, F., Gupta, M., Heath III, F.T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P., Vaculin, R.: Introducing the guard-stage-milestone approach for specifying business entity lifecycles. In: Bravetti, M. (ed.) WS-FM 2010. LNCS, vol. 6551, pp. 1–24. Springer, Heidelberg (2011)
6. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1), 31–39 (2008)
7. Köpke, J., Su, J.: Towards ontology guided translation of activity-centric processes to GSM (2015). Accepted for publication
8. Kumaran, S., Liu, R., Wu, F.Y.: On the duality of information-centric and activity-centric models of business processes. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 32–47. Springer, Heidelberg (2008)
9. Künzle, V., Reichert, M.: Philharmonicflows: towards a framework for object-aware process management. J. Softw. Maintenance Evol: Res. Pract. **23**(4), 205–244 (2011)
10. Meroni, G., Baresi, L., Plebani, P.: Translating BPMN to E-GSM: specifications and rules. Technical report, Politecnico di Milano (2016). http://hdl.handle.net/11311/976678
11. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer, Heidelberg (2013)
12. Meyer, A., Weske, M.: Activity-centric and artifact-centric process model roundtrip. In: Lohmann, N., Song, M., Wohed, P. (eds.) Business Process Management Workshops. Lecture Notes in Business Information Processing, vol. 171, pp. 167–181. Springer, Switzerland (2013)
13. Pesic, M., Schonenberg, H., Van der Aalst, W.M.: Declare: full support for loosely-structured processes. In: Enterprise Distributed Object Computing Conference Proceedings. p. 287. IEEE (2007)
14. Popova, V., Dumas, M.: From Petri Nets to Guard-Stage-Milestone models. In: La Rosa, M., Soffer, P. (eds.) BPM Workshops 2012. LNBIP, vol. 132, pp. 340–351. Springer, Heidelberg (2013)

15. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems: Challenges, Methods, Technologies. Springer Science & Business Media, Heidelberg (2012)
16. Rozinat, A., van der Aalst, W.M.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008)
17. Russell, N., Hofstede, A., Mulyar, N.: Workflow controlflow patterns: A revised view. Technical report BPM-06-22, BPM Center Report, BPMcenter.org (2006)