TRANSLATING BPMN TO E-GSM: SPECIFICATIONS AND RULES

Giovanni Meroni, Luciano Baresi, Pierluigi Plebani

POLITECNICO
MILANO 1863

Politecnico di Milano
Dipartimento di Elettronica Informazione e Bioingegneria
Piazza Leonardo da Vinci 32
20133 Milano - Italy
`http://www.deib.polimi.it`

Feb. 20, 2016

Technical Report

Unless otherwise indicated, the content is available under the terms of the Creative Commons Attribution-ShareAlike license (CC-BY-SA) v3.0 or any later version.

Acknowledgments

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

Contents

1 Introduction **2**

2 E-GSM **2**

3 Transformation Rules **5**

3.1 Basic Elements 5

3.2 Normal flow 7

3.3 Exceptional flow 11

4 Validation **14**

5 Conclusions **16**

List of Figures

1 E-GSM meta-model and its graphical representation. 3

2 Lifecycle of an E-GSM Stage `S`. 4

3 Transformation rule for activities. 5

4 Transformation rule for events. 6

5 Transformation rule for non-interrupting boundary events. 6

6 Transformation rule for interrupting boundary events. 6

7 Transformation rule for sequence blocks. 8

8 Transformation rule for parallel blocks. 8

9 Transformation rule for conditional exclusive blocks. 9

10 Transformation rule for conditional inclusive blocks. 9

11 Transformation rule for loop blocks. 10

12 Transformation rule for blocks with empty branches. 11

13 Transformation rule for forward exception handling blocks. 12

14 Transformation rule for backward exception handling blocks. 12

15 Transformation rule for non-interrupting exception handling blocks. . 13

16 BPMN of the example shipping process. 14

17 Corresponding E-GSM model of the example shipping process. 15

1 Introduction

This report discusses the details of our approach on how to translate a BPMN process, which is easy to conceive, into an equivalent model in E-GSM, an extension to the Guard-Stage-Milestone (GSM) artifact-centric modeling notation [3]. Goal of this translation is to feed a lightweight distributed process monitoring system, that is under development, able to check anomalies during the execution of the process. Indeed, the resulting E-GSM model can be shared by the involved parties to allow them to keep track of the order in which activities are executed and of the status of each activity. Deviations from the "original" execution flow can easily be detected at run-time during the process enactment. Using a declarative model, instead of an imperative model, to instrument the monitoring system, makes the process monitoring activities more flexible as it does not enforce any specific execution flows and in case of anomalies can log the failure and keeps continuing the monitoring. An implementation of the BPMN2EGSM translator, based on ATL (ATLAS Transformation Language) is currently available at `https://bitbucket.org/polimiisgroup/bpmn2egsm`.

Section 2 discusses E-GSM, an extension of 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. Finally, an example of translation is reported in Section 4.

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**, based on events. GSM events can be *external*, like sent or received messages, or *internal*, like the termination of activities, to the process. Starting from the standard GSM notation and our preliminary work [1], we propose E-GSM, an extension where we distinguish between **Data Flow Guards** and **Process Flow Guards** and we add **Fault Loggers**.

The goal of this extension is to include in the artifact-centric definition of the process information on the normal flow, that is, the expected behavior of the process or happy path. To this aim, the process model includes the dependencies among activities in terms of control flow. Being a declarative language, E-GSM does not use control flow information to enforce a specific execution path among activities. Instead, it uses such information to let the process engine detect deviations among the happy path and how the process is actually executed.

Figure 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**.

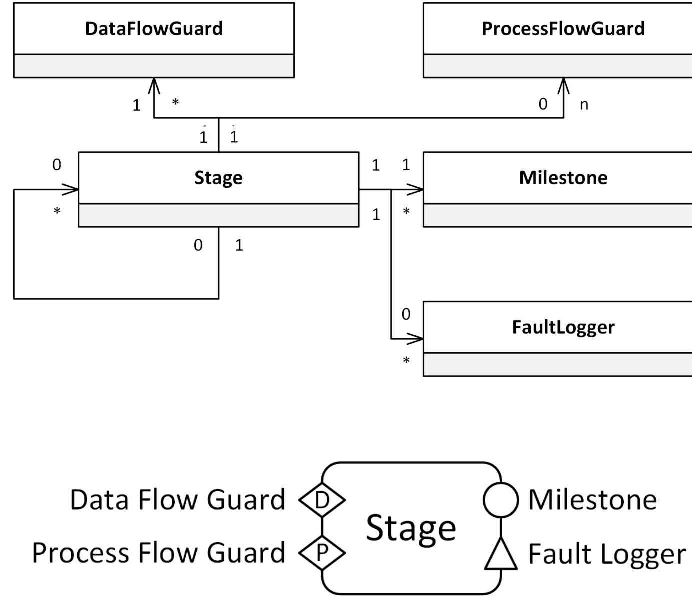

Figure 1: E-GSM meta-model and its graphical representation.

A **Guard** (**Data Flow Guard** in E-GSM) is an Event-Condition-Action (ECA)[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 as faulty because something went wrong during the execution of the activity. A faulty **Stage** does not imply its termination, as the termination is only determined by **Milestones**.

Figure 2 sketches the lifecycle of an E-GSM **Stage** organized around three main orthogonal, execution perspectives: state, status, and compliance[2].

The *Execution state* captures the state of a **Stage**: *unopened*, *opened* or *closed*. A **Stage** is *unopened* if its **Data Flow Guards** have never been triggered. A **Stage**

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

[2] In this paper we use the the notation introduced in [3], so we write $\mathtt{S.DFG_i}$, $\mathtt{S.PFG_k}$, $\mathtt{S.FL_l}$ to indicate the activation of a Data Flow Guard, Process Flow Guard, or a Fault Logger associated with Stage `S`, $\mathtt{+S.M_j}$ ($\mathtt{-S.M_j}$) to indicate the achievement (invalidation) of a Milestone $\mathtt{M_j}$, $\mathtt{S.M_j}$ to indicate that Stage `S` is closed and a Milestone $\mathtt{M_j}$ is achieved, and `Active(S)` to indicate that Stage `S` is opened.

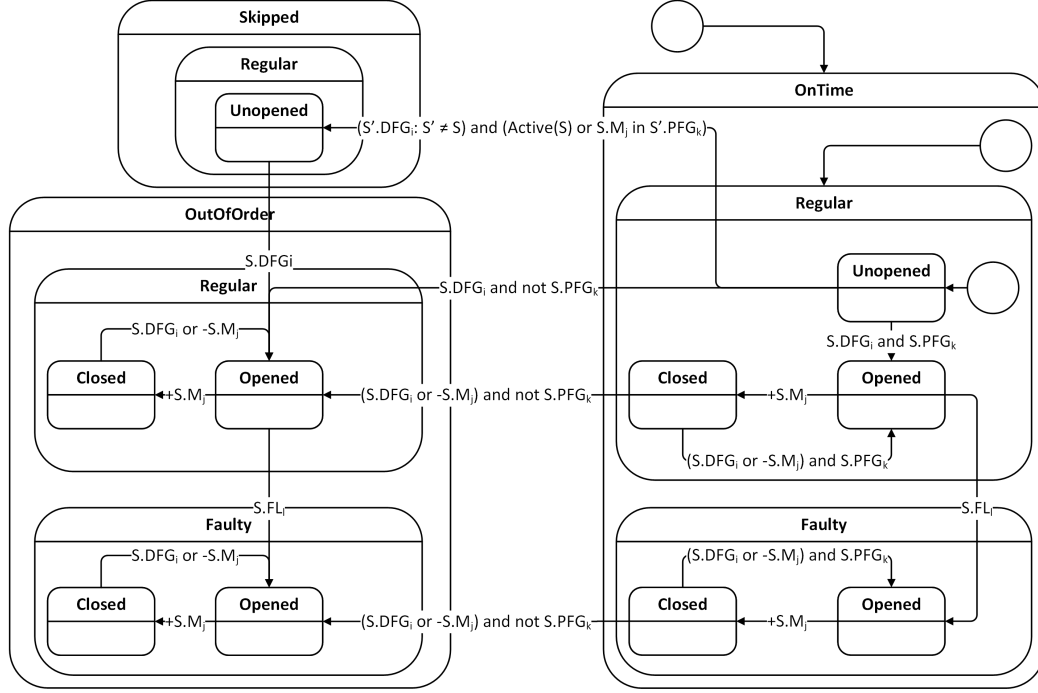

Figure 2: Lifecycle of an E-GSM Stage S.

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 *Execution status* 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 as *outOfOrder*, another **Stage** `S'` is declared as *skipped* if its **Process Flow Guards** require that `S` be activated ($\texttt{S.M}_{\texttt{j}}$ `or Active(S)` $\in$ $\texttt{S'.PFG}_{\texttt{k}}$). If a **Stage** is *skipped*, once one of its **Data Flow Guards** is triggered ($\texttt{S.DFG}_{\texttt{i}}$), it becomes *outOfOrder*.

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 state. **Fault Loggers** drive the status, 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*, the lifecycle of all its child **Stages** will restart from scratch.

3 Transformation Rules

The transformation rules we have defined are applicable to every BPMN process model that complies with a workflow net [7], that is, the process has only one start event and only one end event, and it always terminates (soundness). Another assumption is that activities are not duplicated, meaning that an activity can only be defined once in the process model. We make this assumption to avoid ambiguity in detecting which activity instance is currently running. Given these assumptions, the rules we have derived to translate BPMN in E-GSM are defined in the rest of this section.

3.1 Basic Elements

The transformation rules defined for basic elements are the following four.

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`$_j$).*

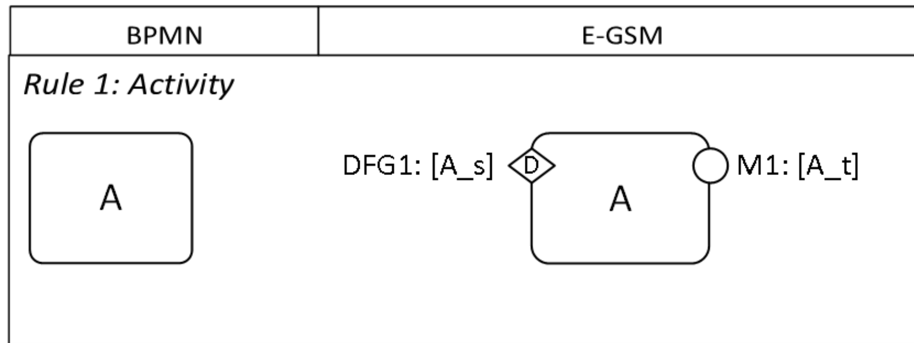

Figure 3: Transformation rule for activities.

Producing the conditions associated with those **Data Flow Guards** and **Milestones** is far from trivial [2]. 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, `A.DFG`$_i$ and `A.M`$_j$ 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.*

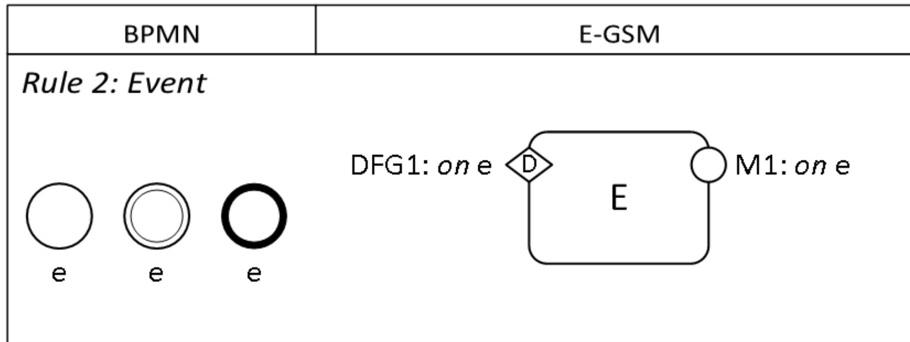

Figure 4: Transformation rule for events.

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`*).*

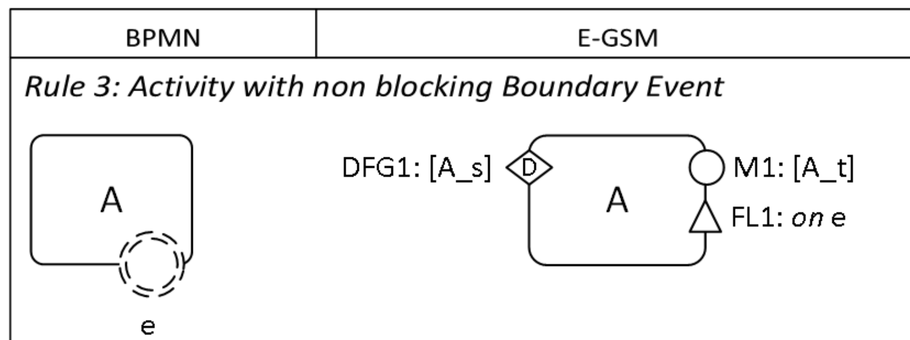

Figure 5: Transformation rule for non-interrupting boundary events.

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.*

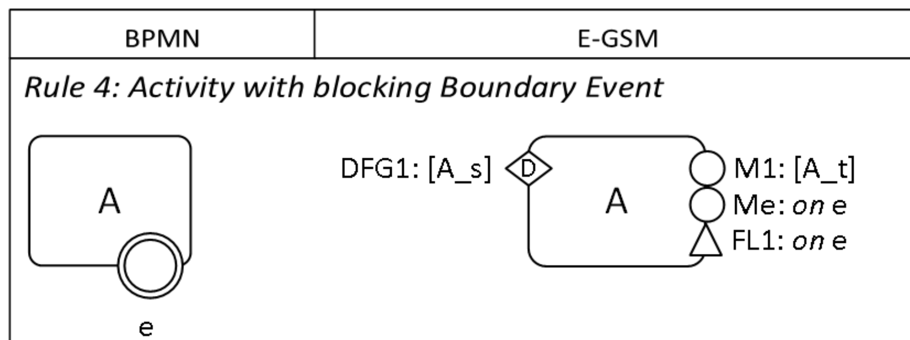

Figure 6: Transformation rule for interrupting boundary events.

3.2 Normal flow

The combination of the above rules for basic elements allows one to translate well-structured business process models. A process model is well-structured if it is made of process-portions, named blocks, that have a single inbound control flow and a single outbound control flow, that can be nested, and that must not overlap [5]. In particular, we focus on five types of blocks, defined starting from the classical control flow patterns [6]:

- 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 the following transformation rules have been defined.

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. This way, inner Stages are expected to be opened only once, and only after their direct predecessor is closed.*
- `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. This way,* `Seq` *is opened when at least one of its* S_x *is opened too, and – as achieving a Milestone is enough to close a Stage –* `Seq` *is closed when all* S_x *are closed.*

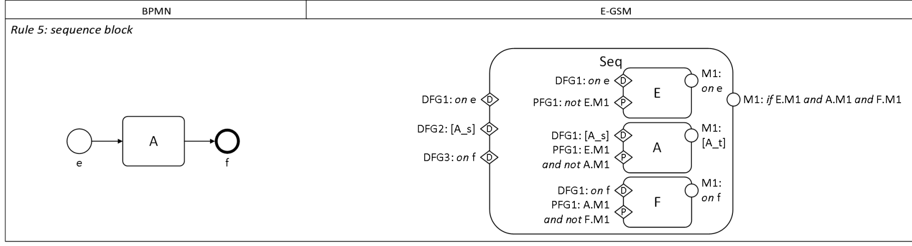

Figure 7: Transformation rule for sequence blocks.

Rule 6 *A parallel block is translated into a Stage* **Par** *that includes all the Stages obtained by applying Rule 5 to all its threads, 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 (i.e., inner stages must be opened only once).*
- *Similarly to the sequence block,* **Par** *has a set* **Par.DFG** *that includes all* $S_x.DFG_i$*, and a Milestone* **Par.M1** *that requires that, for all* S_x*, at least one* $S_x.M_j$ *be achieved.*

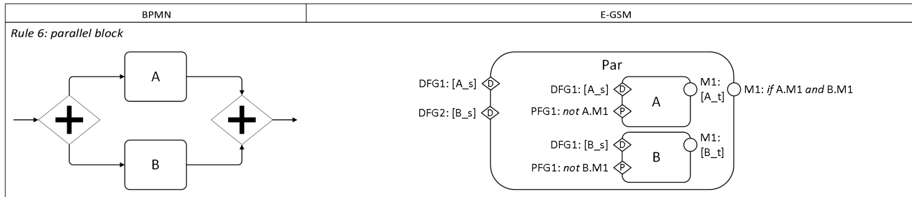

Figure 8: Transformation rule for parallel blocks.

Based on these rules sequence and parallel blocks differ only in the conditions on the **Process Flow Guards** of their inner **Stages**. In case of a sequence block, the conditions require that the execution order defined in the BPMN model be respected. In case of a parallel block, no order is imposed.

Rule 7 *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* $y \neq x$*). This way,* S_x *are expected to be opened only once, and only when their branch is taken and no other branch is.*

- *`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. This way, `Exc` is opened when at least one of its inner Stages can be opened too, and closed when the activated inner Stages become closed, as long as no other Stage is opened.*

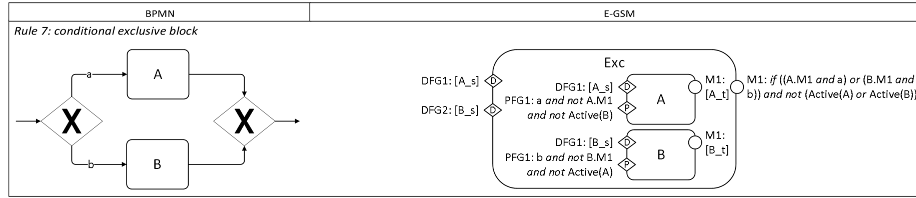

Figure 9: Transformation rule for conditional exclusive blocks.

Rule 8 *A conditional inclusive block is translated into a Stage `Inc` 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 is already achieved and that the condition on the branch from which S_x is produced (if present) is satisfied.*

- *`Inc` has a set `Inc.DFG` that includes all S_x`.DFG`$_i$, and a Milestone `Inc.M1` that requires that, for all S_x whose branch condition (if present) is satisfied, one of S_x`.`M_j be achieved.*

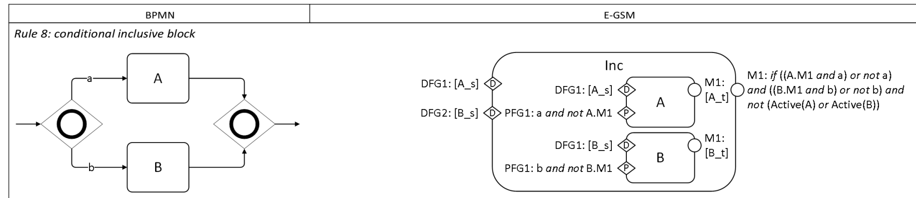

Figure 10: Transformation rule for conditional inclusive blocks.

Rule 9 *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 one of 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. This way, both Stages are expected to be opened only once and, for the backward Stages, only after the forward Stage is closed, the branch they represent is taken and no other branch is.

- ***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.*
  - ***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).*

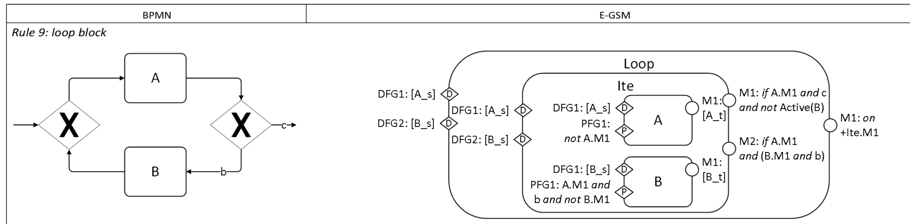

Figure 11: Transformation rule for loop blocks.

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).

Conditional exclusive, conditional inclusive, and loop blocks may have an *empty branch*, that is, a branch with no activities, events, or inner blocks. That block can then be skipped entirely. To handle this situation, a specific rule is defined.[3].

Rule 10 *An empty branch corresponds to a Stage **Empty** that has **Empty.DFG = Empty.M = if Empty.PFG**. The conditions in **Empty.PFG** are defined by the rules related to the block in which the empty branch is included. Moreover, for any stage **S** where **Empty** is an inner Stage at any level, the condition **if S.PFG** is added to **S.DFG**, and **S.DFG** must not include **Empty.DFG**.*

[3]Due to page limit, Figure **??** reports only the case that applies to the conditional exclusive block, other cases can be easily derived from this.

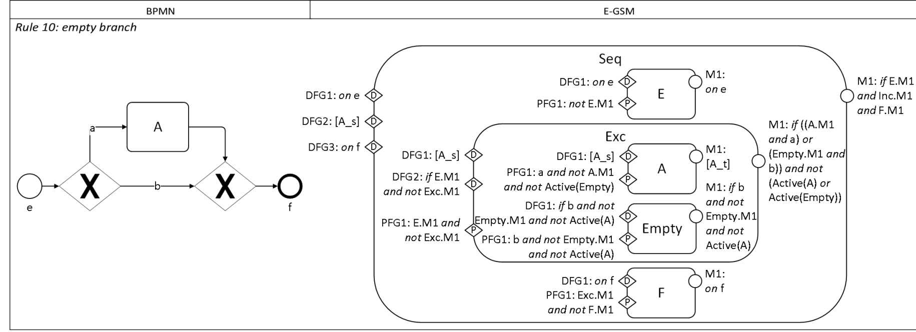

Figure 12: Transformation rule for blocks with empty branches.

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 is thus required to preserve this behavior in E-GSM models.

Like split gateways, also boundary events determine a branching of the control flow. If the attached event is an interrupting event, it interrupts the normal execution flow that follows the activity to which the event is attached, and switch to the exceptional flow. Since normal and exceptional flows are mutually exclusive, 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, along with two new transformation rules, 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 which the boundary event is attached. Its behavior is similar to the one of the conditional exclusive block, therefore we have defined a similar transformation rule.

Rule 11 *A forward exception handling block is translated by applying Rule 7 to that block.*

- S_x*.PFG1 and* *EExc.M1* *replace the satisfaction of the condition that activates the branches with the following statements. For the branch that represents the exceptional control flow, the Milestone* `Me` *derived from the Boundary Event by Rule 4 must be achieved. For the branch that corresponds to the normal control flow,* `Me` *must not be achieved.*

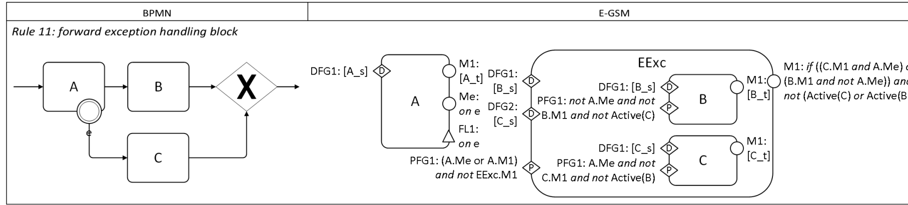

Figure 13: Transformation rule for forward exception handling blocks.

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.

Rule 12 *A backward exception handling block is translated by applying Rule 9 to that block, with the following differences:*

- *`Ite.M1` replaces the satisfaction of the exit condition of the loop with the statement that the Milestone `Me`, derived from the Boundary Event by Rule 4, must not be achieved.*
- *S_x`.PFG1` and `Ite.M2` replace the satisfaction of the condition activating branches with the statement that `Me` must be achieved.*

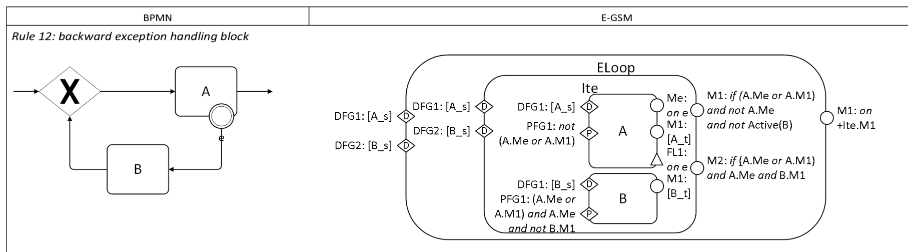

Figure 14: Transformation rule for backward exception handling blocks.

In BPMN, boundary events could also be non interrupting: i.e., 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 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*, and a new transformation rule.

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.

Rule 13 *A non interrupting exception handling block is translated by applying Rule 8 to that block.*

- `Sx.PFG1`*, being* **`x`** *the Stage that represents the exceptional control flow, checks that the Stage derived by Rule 3 from the Activity to which the Boundary Event is attached is opened.*

- `EInc.M1` *requires that, for the Stage that represents the normal control flow, one of its* `Sy.Mj` *be achieved, as long as the Stage representing the exceptional control flow is not opened.*

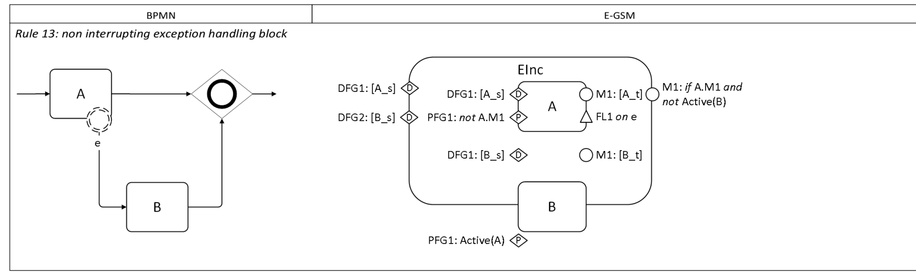

Figure 15: Transformation rule for non-interrupting exception handling blocks.

After identifying all possible blocks and defining the corresponding transformation rules, we can now use nested **Stages** to translate any well-structured process model. Note that, since we only admit one start event and one end event per process, we can identify a single sequence block that covers the entire process definition. This sequence block corresponds to the control flow that initiates with the start event, ends with the end event, and traverses any other activity, intermediate event, or internal block, if present. To manage the nested **Stages**, the following transformation rule is defined.

Rule 14 *Each BPMN Subprocess Activity is mapped to the sequence Stage that encloses all the Stages that correspond to elements that belong to the Subprocess.*

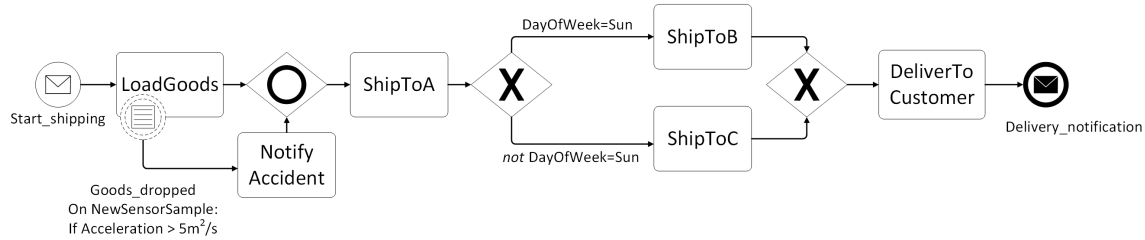

Figure 16: BPMN of the example shipping process.

4 Validation

The transformation rules presented 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[4], where the transformation rules are implemented in ATL (ATLAS Transformation Language [4]), and validated —and refined— the proposed rules against several BPMN business processes with different levels of complexity.

In this document we report an example taken from the logistics domain to better explain how E-GSM models can be used to monitor the execution of complex (distributed) processes. A manufacturing company M has to ship its goods to one of its customers N and, to do so, it relies on a shipping company S for rail and sea-cargo transportation, and on a shipping company T for truck transportation. The shipping process comprises four main phases: (i) loading goods into a shipping container; (ii) shipping such a container to an intermediate site A by truck; (iii) depending on the day of the week, shipping the goods to either site B by rail, or to site C by sea; (iv) delivering the goods to the customer's site by truck. Furthermore, if part of the goods drops during the loading phase, that activity should stop, all involved actors should be notified of such an accident, and then the loading phase redone. Figure 17 shows the BPMN definition of this process in the upper part, and the derived E-GSM process, which is produced by our translator, in the lower part.

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

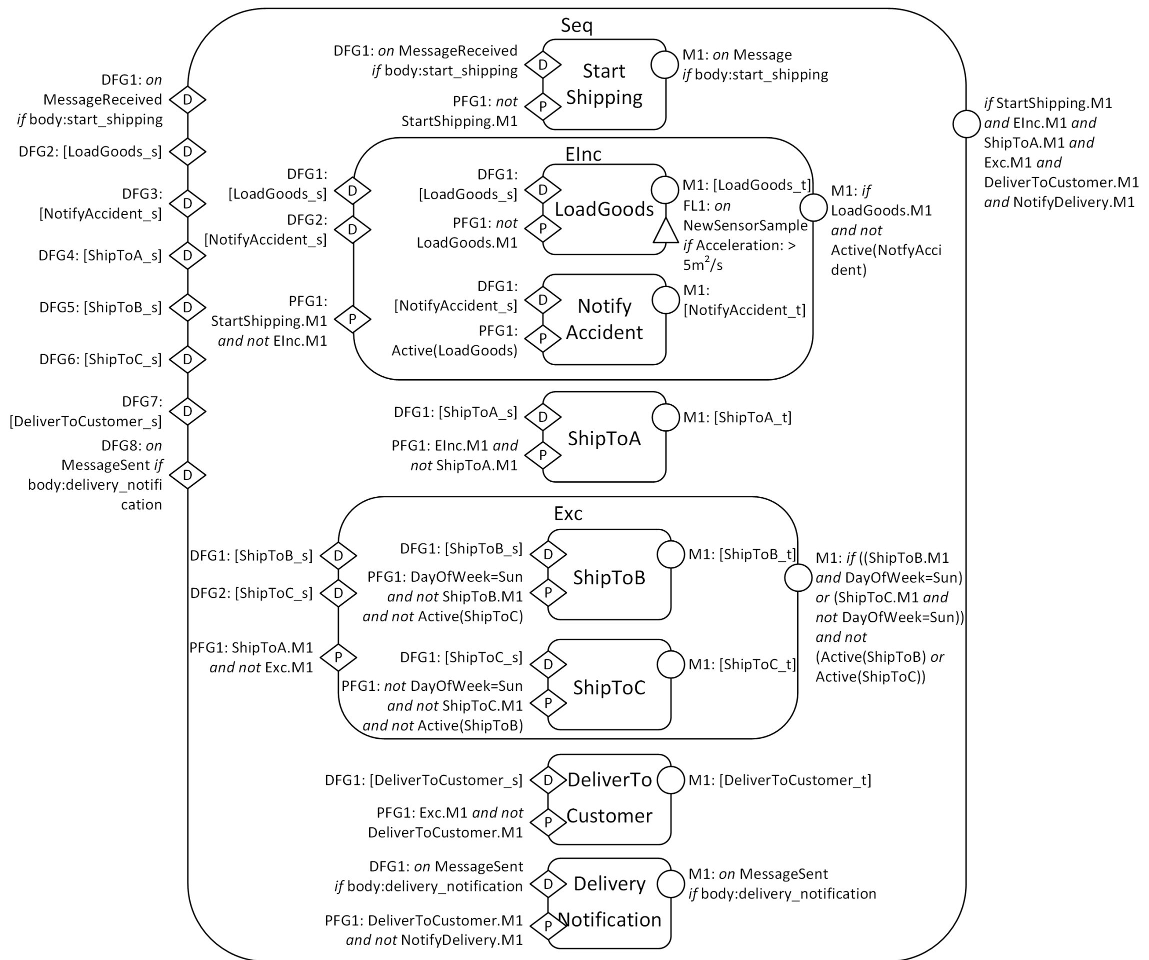

Figure 17: Corresponding E-GSM model of the example shipping process.

5 Conclusions

This paper extends the Guard-Stage-Milestone (GSM) notation to embed control flow information in the process model definition, and presents a solution for transforming BPMN models into equivalent E-GSM ones. These process models exploit **Milestones** and **Data Flow Guards** to detect the start and end of all the activities in the process no matter the execution flow. **Process Flow Guards** help keep the information on the expected execution flow and, thus, identify deviations in the actual execution. Finally, **Fault Loggers** provide means to deal with foreseen exceptions.

A tool implementing the transformation rules presented in this paper has been developed and it can be used to automatically transform well-structured BPMN into E-GSM.

References

[1] L. Baresi, G. Meroni, and P. Plebani, "A gsm-based approach for monitoring cross-organization business processes using smart objects." Accepted for publication, 2015.

[2] C. Cabanillas, A. Baumgrass, J. Mendling, P. Rogetzer, and B. Bellovoda, "Towards the enhancement of business process monitoring for complex logistics chains," in *Business Process Management Workshops*, pp. 305–317, Springer, 2014.

[3] R. Hull, E. Damaggio, F. Fournier, M. Gupta, I. Heath, Fenno(Terry), S. Hobson, M. Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculin, "Introducing the guard-stage-milestone approach for specifying business entity lifecycles," in *Web Services and Formal Methods*, vol. 6551 of *Lecture Notes in Computer Science*, pp. 1–24, Springer, 2011.

[4] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of computer programming*, vol. 72, no. 1, pp. 31–39, 2008.

[5] M. Reichert and B. Weber, *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media, 2012.

[6] N. Russell, A. H. M. T. Hofstede, and N. Mulyar, "Workflow controlflow patterns: A revised view," Tech. Rep. BPM-06-22, BPM Center Report, BPMcenter.org, 2006.

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