TRANSLATING BPMN TO E-GSM: PROOF OF CORRECTNESS

Giovanni Meroni, Marco Montali, 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

May. 20, 2016

Technical Report

Acknowledgments

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

Giovanni Meroni, Luciano Baresi, Pierluigi Plebani are affiliated to the Dipartimento di Elettronica ed Informazione of Politecnico di Milano, Milan, Italy.

Marco Montali is affiliated to the Free University of Bolzano-Bozen, Bolzano, Italy.

Contents

Abstract

In this technical report, we prove the correctness of the BPMN to E-GSM translator described in [2].

1 Introduction

This report completes the discussion 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].

The specification and the main rules of the BPMN to E-GSM translator are defined in [2]. The goal of this paper is to prove that the translator operates correctly.

Section 2 introduces the formalism adopted along the paper. Section 3 formally defines the concept of conformance among traces of processes. Finally, Section 4 focuses on proving how translating a process model expressed using BPMN into an E-GSM model, according to the rules expressed in [2] does not affect the conformance.

2 Process Model

We provide a formal definition of (block-structured) BPMN process models manipulating artifact states. For simplicity, we consider only activities/tasks equipped with at most one boundary event.

2.1 Data Component

In our setting, the data component of a business process is constituted by a set of artifacts and their states. In particular, a data component is a set of pairs $\langle A, \Sigma \rangle$, where A is the name of an *artifact*, and Σ is the set of states in which that artifact can be. How an artifact moves from one state to another is implicitly determined by the process control-flow, and how atomic tasks operate over artifacts.

2.2 Blocks

Blocks account for the different units of work in the process, together with their control-flow dependencies. We therefore start by defining a generic notion of block, which then specializes depending on its type.

Definition 2.1 (Block)**.** *A* block *is a triple* $\langle BName, BType, BAttr \rangle$*, where:*

- *BName is the* block name*, used to uniquely identify the block.*
- $BType \in \{\texttt{event}, \texttt{proc}, \texttt{task}, \texttt{activity}, \texttt{seq}, \texttt{par}, \texttt{choice}, \texttt{or}, \texttt{loop}\}$ *is the* block name*, where: (i)* `event` *denotes event blocks; (ii)* `task` *denotes task blocks; (iii)* `activity` *denotes non-atomic activities specified in terms of a subprocess; (iv)* `seq` *denotes sequence blocks (indicating the acceptable ordering of execution for other blocks); (v)* `par` *denotes parallel blocks (whose multiple branches are executed concurrently); (vi)* `choice` *denotes choice blocks (whose multiple branches are mutually exclusive); (vii)* `or` *denotes inclusive or blocks (whose multiple branches are selectively executed in parallel); (viii)* `loop` *denotes loop blocks (where the flow may execute a block multiple times); (ix)* `proc` *denotes process blocks, which begin with a start event and finish with a termination event, provided that the process flow proceeds without exceptions in between.*
- *BAttr is a tuple of* type-dependent attributes*.*

In the following, we detail how *BAttr* is defined depending on the block type. If *BAttr* contains another block B', we say that B' is a *direct sub-block* of B.

2.3 Event Block

An event block represents a BPMN start, intermediate, or termination event. In the context of this paper, the specific kind of event is not important, nor it is whether it is an event triggered by the process itself, or caught by the process. Hence, we simply consider $BAttr = \langle\rangle$.

2.4 Task Block

A task block represents an atomic unit of work within the process. In the context of this work, we keep track of how a task relates to the state of relevant artifacts. Specifically, $BAttr = \langle IS, OS \rangle$, where IS and SO are two sets respectively expressing the precondition and effect of the task in terms artifact states required by the task to start, and new states in which artifacts are moved when the task completes. Each entry in $IS \cup OS$, in turn, is a pair $\langle A, S \rangle$, where A is an artifact and S is a state.

2.5 Process Block

A process block represents a BPMN process, triggered by a start event, consisting of a main block, and finally ending with a termination event. Hence, $BAttr = \langle E_s, B, E_t \rangle$, where:

- E_s is an event block, accounting for the start event;
- B is a generic block, accounting for the main execution block of the process;
- E_t is an event block, accounting for the termination event.

2.6 Activity Block

An activity is a generic unit of work within the process. While it is executed, it may spawn interrupting or non-interrupting exceptional flows, in the case where it is equipped with boundary events catching external events, together with corresponding handlers. For simplicity, and without loss of generality, we consider here the case where each activity block is equipped with at most one boundary event.

In this light, we have $BAttr = \langle B, [EH] \rangle$, where:

- B is either a task or a process block, respectively representing the case where the activity is an atomic unit of work, or a non-atomic unit of work captured by a (sub)process.
- EH is an optional triple $\langle E, H, f \rangle$, where E is a (boundary) event block, H is a generic block modeling the handler of the event, and f is a flag indicating how the exception handler is managed. We have in particular three cases:
  - ($f =$ `nonint`) the exception is managed in a *non-interrupting* way, i.e., H is run in parallel with the current block, synchronizing their termination;
  - ($f =$ `intfw`) the exception is managed by *interrupting* the normal execution, then continuing the execution in a *forward* way;
  - ($f =$ `intbw`) the exception is managed by *interrupting* the normal execution, then going *back* to re-execute the normal block again.

2.7 Sequence Block

Sequence blocks are made of 2 or more sub-blocks executed one after the other. We consequently have $BAttr = \langle B_1, \ldots, B_n \rangle$, where each B_i is a generic block, and $n \geq 2$. The ordering within $BAttr$ reflects the nature of the sequence. It is worth noting that

a process block can be seen as a sequence block constituted by three sub-blocks, the first being the start event of the process, the second being the main execution block, and the third being the termination event block.

2.8 Parallel Blocks

Parallel blocks are constituted by 2 or more sub-blocks running concurrently. In this case, we then have that $BAttr = \langle B_1, \ldots, B_n \rangle$, where each B_i is a generic block, and $n \geq 2$.

2.9 Decision Blocks

Decision blocks are those in which one or more sub-blocks are executed depending on the result of some decision. In particular, each sub-block is *conditional*, in the sense that it is associated to a guard, and it is executed if the guard evaluates to true. A conditional block is a pair $\langle \varphi, B \rangle$, where φ is a condition, and B is a generic block. In the case of a decision block, $BAttr = \langle C_1, \ldots, C_n \rangle$, where $n \geq 2$, each $C_i = \langle \varphi_i, B_i \rangle$ is a conditional block, and the guards are such that $\bigvee_{i \in \{1,\ldots,n\}} \varphi_i = \mathit{true}$ (i.e., at least one condition evaluates to true). We have two types of decision block:

- `choice`, representing an exclusive choice block. In this case, we have that guards are pairwise disjoint, i.e., for every $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $\varphi_i \wedge \varphi_j = \mathit{false}$. Combined with the assumption above, this means that exactly one guard evaluates to true.
- `or`, representing an inclusive or block. In this case, multiple guards may hold, leading to execute all the corresponding sub-blocks.

2.10 Loop Blocks

Loop blocks represent units of work that can be repeated multiple times. In this case, $BAttr = \langle FB, CB \rangle$, where FB is the block executed at least once when traversing the loop block, and $CB = \langle \varphi_{in}, B' \rangle$ is a conditional block executed when the loop condition φ_{in} evaluates to true. In this case, after executing CB the control-flow reiterates the execution of FB. If φ_{in} evaluates to false, then the loop is terminated.

2.11 Process Model

We are now in the position of defining what a process model is:

Definition 2.2 (Process Model)**.** *A process model $\mathcal{B}$ is a pair $\langle D, P \rangle$, where D is a data component (cf. Section 2.1), and P is a* process block *(defined in Section 2.5) that represents the top-level, end-to-end BPMN process of interest. We assume that P obeys to the following assumptions:*

- *P is constituted by finitely many (direct and indirect) sub-blocks;*
- *no two sub-blocks of P have the same name.*

The first requirement guarantees that the process model is finite. The second requirement simultaneously implies two properties: on the one hand, it makes the process model unambiguous; on the other hand, it makes the process model well-defined, in the sense that no block directly or indirectly embeds itself as a sub-block. Thanks to such assumptions, we get that the sub-block relation induces a tree-structure, rooted in the top-level, end-to-end process, and whose leaves are atomic tasks and events. More specifically, the sub-block relation forms a tree where P is the root, each other block has a unique parent block, and the leaves of the tree correspond to task or event blocks. We call this tree the *process tree* of $\mathcal{B}$ (or of P). Given a block B different than P, we use notation $B.Parent$ to identify its parent block. Furthermore, we respectively denote by $P.Blocks$, $P.Tasks$ and $P.Events$ the blocks present in the sub-tree rooted in P, the names of task blocks in $P.Tasks$, and the names of event blocks in $P.Tasks$.

The notion of parenthood is formally defined next.

Definition 2.3 (Block Parenthood)**.** *Let $\langle D, P\rangle$ be a process model, and B_p, B_c two blocks in P.Blocks different than P itself. We say that B_p* is the parent block of *B_c,* written *$B_p = B_c.Parent$, if one of the following conditions holds:*

- $B_p = \langle \mathtt{n_p}, \mathtt{activity}, \langle B, EH\rangle\rangle$*, with $B = B_c$, or with EH containing either a triple of the form $\langle B_c, H, f\rangle$ or of the form $\langle E, B_c, f\rangle$.*
- $B_p = \langle \mathtt{n_p}, Type, \langle B_1, \ldots, B_n\rangle\rangle$*, with $Type \in \{\mathtt{seq}, \mathtt{par}\}$ and some $B_i = B_c$.*
- $B_p = \langle \mathtt{n_p}, Type, \langle CB_1, \ldots, CB_n\rangle\rangle$*, with $Type \in \{\mathtt{choice}, \mathtt{or}\}$ and some $CB_i = \langle \varphi_i, B_c\rangle$.*
- $B_p = \langle \mathtt{n_p}, \mathtt{loop}, \langle B, CB\rangle\rangle$*, with either $B = B_c$, or $CB = \langle \varphi, B_c\rangle$.*

Specularly, given a block B_p we define the set of children blocks of B_p, written $B_p.Children$, as the set $\{B_c \mid B_c.Parent = B_p\}$.

3 Trace Conformance

In this section, we formally characterize when a trace over $\mathcal{B}$ *conforms to* $\mathcal{B}$, considering the control-flow semantics of the blocks contained in the process tree of $\mathcal{B}$.

Definition 3.1 (Trace)**.** *A* trace *over a process model $\mathcal{B} = \langle D, P\rangle$ is a finite sequence $\langle \mathtt{t_1} \cdots \mathtt{t_n}\rangle$ of* steps*, where each step $\mathtt{t_i}$ has one of the following forms:*

- *(event execution) $\mathtt{t_i} = E$, where $E \in P.Events$.*
- *(task start) $\mathtt{t_i} = \langle \mathtt{start}, \mathtt{n}\rangle$, where $\mathtt{n}$ is the name of a task in $P.Tasks$.*
- *(task end) $\mathtt{t_i} = \langle \mathtt{end}, \mathtt{n}\rangle$, where $\mathtt{n}$ is the name of a task in $P.Tasks$.*

In addition, $\mathtt{t_1} = E_s$, where E_s is the start event of P.

To define conformance, we first introduce a suitable notion of execution state, which in turn provides the basis for defining when an execution step is accepted, and what is the resulting execution state.

Definition 3.2 (Execution state)**.** *An* execution state *over $\mathcal{B}$ is a pair $\langle Curr, Next\rangle$, where Curr is the set of* enacted *blocks in $\mathcal{B}$, and Next is the set of* enactable *blocks in $\mathcal{B}$.*

Intuitively:

- Block B is *enacted* whenever there is an execution thread currently flowing through B.
- Block B is *enactable* if B becomes enacted in response to some execution step done in the current state of affairs.

Definition 3.3 (Initial execution state)**.** *Let $\mathcal{B} = \langle D, P \rangle$ be a process model, with $P = \langle E, A, F \rangle$. The* initial execution state *of $\mathcal{B}$ is $\langle \emptyset, \{P, E\} \rangle$.*

The initial execution state reflects the intuition that, at the beginning, no block is enacted, and the only enactable blocks are P itself together with its start event E.

Definition 3.4 (Executable step)**.** *Given an execution state $s = \langle Curr, Next \rangle$ over process model $\mathcal{B}$, we say that step* `t` is executable in *s if one of the following conditions holds:*

1. $\texttt{t} = \texttt{e}$ *and $E \in Next$, where E is the event block named* `e`*;*
2. $\texttt{t} = \langle \texttt{start}, \texttt{n} \rangle$ *and $T \in Next$, where T is the task block named* `n`*;*
3. $\texttt{t} = \langle \texttt{end}, \texttt{n} \rangle$ *and $T \in Curr$, where T is the task block named* `n`*.*

Obviously, the actual execution of an executable step leads to update the state of its corresponding block. However, depending on where that block is located in the process model, this update could recursively trigger state updates for other blocks. We classify such updates in three categories:

- disablement of a block, resulting in the removal of the block from the set of enacted/enactable blocks;
- enactability of a block, resulting in the insertion of the block in the set of enactable blocks;
- enactment of a block, leading to move the block from the set of enactable blocks to that of enacted blocks.

Definition 3.5 (Block disablement)**.** *Let $s = \langle Curr, Next \rangle$ be an execution state over $\mathcal{B}$,* `t` *a step executable in s, and B a block of $\mathcal{B}$. We say that B* is disabled by `t` in *s, or alternatively that* `t` disables *B* in *s, if one of the following conditions holds:*

1. *Task end execution step - base case:*
   a. *(i)* $\texttt{t} = \langle \texttt{end}, \texttt{n} \rangle$*, (ii)* $B = \langle \texttt{n}, \texttt{task}, \langle IS, OS \rangle \rangle$*, and (iii) $B \in Curr$. (A task end step disables its corresponding task block if such a block is currently in execution).*
2. *Event execution step - base cases:*
   a. *(i)* $\texttt{t} = \texttt{e}$*, (ii)* $B = \langle \texttt{e}, \texttt{event}, \langle \rangle \rangle$*, and (iii) $B \in Next$. (An event step disables its corresponding event block if it is enactable).*
   b. *(i)* $\texttt{t} = \texttt{f}$*, (ii)* $B = \langle \texttt{n}, \texttt{proc}, \langle E_s, B', E_t \rangle \rangle$ *with* $E_t = \langle \texttt{f}, \texttt{event}, \langle \rangle \rangle$*, and (iii) $E_t \in Next$. (An event step disables a process block if it corresponds to its termination event, and such termination event is enactable).*
   c. *(i)* $\texttt{t} = \texttt{e}$*, (ii) B is a descendant of, or corresponds to, block B', where B' is the inner block of an activity block that in turn has an interrupting boundary event named* `e`*, and (iii) $B' \in Curr$.*

(An event step disables all inner blocks of an enabled activity block having that event as interrupting exception).

3. *Activity block - inductive cases:*
   a. *(i)* $B = \langle \mathtt{n}, \mathtt{activity}, \langle B' \rangle \rangle$ *(ii)* B' *is disabled by* $\mathtt{t}$*. (An activity block without boundary events is disabled if its inner block is disabled by the given execution step).*
   b. *(i)* $B = \langle \mathtt{n}, \mathtt{activity}, \langle B', \langle E, B'', \mathtt{nonint} \rangle \rangle \rangle$ *(ii)* B' *is disabled by* $\mathtt{t}$*; (iii)* $B'' \notin Curr$*. (An activity block with non-interrupting boundary event is disabled if its inner block is disabled and its event handler is not in execution).*
   c. *(i)* $B = \langle \mathtt{n}, \mathtt{activity}, \langle B', \langle E, B'', \mathtt{nonint} \rangle \rangle \rangle$ *(ii)* B'' *is disabled by* $\mathtt{t}$*; (iii)* $B' \notin Curr$*. (An activity block with non-interupting boundary event is disabled if its event handler is disabled and its inner block is not in execution).*
   d. *(i)* $B = \langle \mathtt{n}, \mathtt{activity}, \langle B', \langle E, B'', f \rangle \rangle \rangle$*, with* $f \in \{\mathtt{intfw}, \mathtt{intbw}\}$*; (ii)* B' *is disabled by* $\mathtt{t}$*. (An activity block with interupting boundary event is disabled if its inner block is disabled - this guarantees that its handler is not in execution).*
   e. *(i)* $B = \langle \mathtt{n}, \mathtt{activity}, \langle B', \langle E, B'', f \rangle \rangle \rangle$*, with* $f \in \{\mathtt{intfw}, \mathtt{intbw}\}$*; (ii)* B'' *is disabled by* $\mathtt{t}$*. (An activity block with interupting boundary event is disabled if its event handler is disabled - this guarantees that its inner block is not in execution).*
4. *Sequence block - inductive case: (i)* $B = \langle \mathtt{n}, \mathtt{seq}, \langle B_1, \ldots, B_n \rangle \rangle$ *(ii)* B_n *is disabled by* $\mathtt{t}$*. (A sequence block is disabled if its last inner block is disabled).*
5. *Parallel/or block - inductive case: (i)* $B = \langle \mathtt{n}, type, \langle B_1, \ldots, B_n \rangle \rangle$*, with type* $\in \{\mathtt{par}, \mathtt{or}\}$ *(ii) there exists* $i \in \{1, \ldots, n\}$ *such that* B_i *is disabled by* $\mathtt{t}$ *and for each* $j \in \{1, \ldots, n\}$ *with* $j \neq i$*,* $B_j \notin Curr$*. (A parallel/or block is disabled as soon as one of its inner block is disabled, and there is no other inner blocks currently enacted).*
6. *Choice block - inductive case: (i)* $B = \langle \mathtt{n}, \mathtt{choice}, \langle B_1, \ldots, B_n \rangle \rangle$*, with type* $\in \{\mathtt{par}, \mathtt{or}\}$ *(ii) there exists* $i \in \{1, \ldots, n\}$ *such that* B_i *is disabled by* $\mathtt{t}$*. (A choice block is disabled when one of its inner blocks is disabled - that block is the one that was selected when taking the choice).*
7. *Loop block - inductive case: (i)* $B = \langle \mathtt{n}, \mathtt{loop}, \langle B_{fw}, \langle \varphi, B_{bw} \rangle \rangle \rangle$*, (ii)* B_{fw} *is disabled by* $\mathtt{t}$*, (iii)* φ *is false in* s*. (A loop block is disabled when its forward inner block is disabled, and the loop condition evaluates to false).*

We denote by $\mathcal{D}_s^{\mathtt{t}}$ *the set of blocks in* $\mathcal{B}$ *that are disabled by* $\mathtt{t}$ *in* s*.*

Definition 3.6 (Block enactment)**.** *Let* $s = \langle Curr, Next \rangle$ *be an execution state over process model* $\mathcal{B}$*,* $\mathtt{t}$ *a step executable in* s*, and* B *a block of* $\mathcal{B}$*. We say that* B is enacted by $\mathtt{t}$ in s*, or alternatively that* $\mathtt{t}$ enacts B in s*, if* $B \in Next$ *and one of the following conditions holds:*

1. *Task execution step - base case (i)* $\mathtt{t} = \langle \mathtt{start}, \mathtt{n} \rangle$*; (ii)* $B = \langle \mathtt{n}, \mathtt{task}, \langle \rangle \rangle$*. (An enactable task is enacted when it gets started).*
2. *Event execution step - base case: (i)* $\mathtt{t} = \mathtt{e}$ *(ii)* B *is the top process block, and its start event is* $\mathtt{e}$*. (The top process block is enacted when its start event*

occurs - this only applies to the top block, since for subprocesses the start event implicitly occurs whenever the flow reaches them).

3. *Activity block - inductive case: (i) B is an activity block of the form $\langle \mathtt{n}, \mathtt{activity}, \langle B', [EH] \rangle \rangle$; (ii) B' is enacted by $\mathtt{t}$ in s. (An activity block is enacted when its inner task/process block is enacted).*
4. *Sequence block - inductive case: (i) B is a sequence block of the form $\langle \mathtt{n}, \mathtt{seq}, \langle B_1, \ldots, B_n \rangle \rangle$; (ii) B_1 is enacted by $\mathtt{t}$ in s. (A sequence block is enacted when its first inner block is enacted).*
5. *Gateway blocks - inductive case: (i) B is a gateway block of the form $\langle \mathtt{n}, type, \langle B_1, \ldots, B_n \rangle \rangle$ with $type \in \{\mathtt{par}, \mathtt{choice}, \mathtt{or}\}$; (ii) there exists $i \in \{1, \ldots, n\}$ such that B_i is enacted by $\mathtt{t}$ in s. (A gateway block is enacted as soon as one of its inner, enactable blocks is actually enacted - for decision blocks, inner blocks are enactable only if their guard condition is true, see below).*
6. *Choice blocks - inductive case: (i) B is a loop block of the form $\langle \mathtt{n}, \mathtt{loop}, \langle B', CB \rangle \rangle$; (ii) B' is enacted by $\mathtt{t}$ in s. (A loop block is enacted when its inner forward block is enacted).*

We denote by $\mathcal{E}_s^{\mathtt{t}}$ the set of blocks in $\mathcal{B}$ that are enacted by $\mathtt{t}$ in s.

Definition 3.7 (Block enactability)**.** *Let $s = \langle Curr, Next \rangle$ be an execution state over process model $\mathcal{B}$, $\mathtt{t}$ a step executable in s, and B a block of $\mathcal{B}$. We say that B* is made enactable by *$\mathtt{t}$ in s, or alternatively that $\mathtt{t}$* makes *B* enactable in *s, if $B \notin Next \cup Curr$ and one of the following conditions holds:*

1. *Event execution step - base cases:*
   a. *(i) $\mathtt{t} = \mathtt{e}$; (ii) the (top) process block of $\mathcal{B}$ is $P = \langle \mathtt{n}, \mathtt{proc}, \langle E_s, B, E_t \rangle \rangle$, with $E_s = \langle \mathtt{e}, \mathtt{event}, \langle \rangle \rangle$; (iii) $P \in Next$. (The occurrence of the start event of the top process block makes its inner block enactable).*
   b. *(i) $\mathtt{t} = \mathtt{e}$; (ii) there exists block $B_p = \langle \mathtt{n}, \mathtt{activity}, \langle B', EH \rangle \rangle \in Curr$, where EH is of the form $\langle \langle \mathtt{e}, \mathtt{event}, \langle \rangle \rangle, B, f \rangle$. (The occurence of the boundary event of an enacted activity makes the corresponding handler enactable).*
2. *Activity block - base case: (i) B is an event block; (ii) $B.Parent$ is an activity block having B as boundary event, i.e., it has the form $\langle \mathtt{n}, \mathtt{activity}, \langle B_c, \langle B, H, f \rangle \rangle \rangle$ (iii) $B.Parent$ is enacted by $\mathtt{t}$ in s. (The enactment of an activity block makes its boundary event enactable).*
3. *Parent activity block - inductive case: (i) $B.Parent$ is an activity block having B as inner block, i.e., it has the form $\langle \mathtt{n}, \mathtt{activity}, \langle B, \langle E, H, f \rangle \rangle \rangle$; (ii) $B.Parent$ is made enactable by $\mathtt{t}$ in s. (As soon as an activity block becomes enactable, its inner block becomes enactable as well).*
4. *Parent sequence block - inductive cases:*
   a. *(i) $B.Parent$ is a sequence block having B as first inner block, i.e., it has the form $\langle \mathtt{n}, \mathtt{seq}, \langle B, \ldots \rangle \rangle$; (ii) $B.Parent$ is made enactable by $\mathtt{t}$ in s. (As soon as a sequence block becomes enactable, its first inner block becomes enactable as well).*
   b. *(i) $B.Parent$ is a sequence block having B as non-first inner block, i.e., it has the form $\langle \mathtt{n}, \mathtt{seq}, \langle \ldots, B_p, B, \ldots \rangle \rangle$; (ii) $B.Parent \in Curr$; (iii) B_p is disabled*

by $\mathtt{t}$ *in* s*. (The disablement of an inner block inside an enacted sequence makes the next inner block enactable).*

5. *Parent parallel block - inductive case: (i)* B*.Parent is a parallel block having* B *as one of its inner blocks, i.e., it has the form* $\langle \mathtt{n}, \mathtt{par}, \langle \ldots, B, \ldots \rangle \rangle$*; (ii)* B*.Parent is made enactable by* $\mathtt{t}$ *in* s*. (As soon as a parallel block becomes enactable, its inner blocks becomes all enactable as well).*
6. *Parent decision block - inductive case: (i)* B*.Parent is a decision block having* B *as one of its inner blocks, i.e., it has the form* $\langle \mathtt{n}, type, \langle \ldots, CB, \ldots \rangle \rangle$*, with* $type \in \{\mathtt{choice}, \mathtt{or}\}$ *and* $CB = \langle \varphi, B \rangle$*; (ii)* B*.Parent is made enactable by* $\mathtt{t}$ *in* s*; (iii)* φ *is true in* s*. (As soon as a decision block becomes enactable, its inner block(s) whose corresponding condition evaluates to true in the current state become(s) enactable as well).*
7. *Parent loop block - inductive cases:*
   a. *(i)* B*.Parent is a loop block having* B *as its forward inner block, i.e.,* B*.Parent is of the form* $\langle \mathtt{n}, \mathtt{loop}, \langle B, CB \rangle \rangle$*; (ii)* B*.Parent is made enactable by* $\mathtt{t}$ *in* s*. (As soon as a loop block becomes enactable, its forward inner block becomes enactable as well).*
   b. *(i)* B*.Parent is a loop block having* B *as its backward inner block, i.e.,* B*.Parent is of the form* $\langle \mathtt{n}, \mathtt{loop}, \langle FB, \langle \varphi, B \rangle \rangle \rangle$*; (ii) the forward block* FB *of the loop is disabled by* $\mathtt{t}$ *in* s*; (iii)* φ *is true in* s*. (The backward inner block of a loop becomes enactable when the forward block of that loop gets disabled, provided that the loop condition evaluates to true in the current state).*
   c. *(i)* B*.Parent is a loop block having* B *as its forward inner block, i.e.,* B*.Parent is of the form* $\langle \mathtt{n}, \mathtt{loop}, \langle B, \langle \varphi, B' \rangle \rangle \rangle$*; (ii) the backward block* B' *of the loop is disabled by* $\mathtt{t}$ *in* s*. (The forward inner block of a loop becomes enactable when the backward block of that loop gets disabled - this captures a new iteration of the loop).*

We denote by $\mathcal{N}_s^{\mathtt{t}}$ *the set of blocks in* $\mathcal{B}$ *that are made enactable by* $\mathtt{t}$ *in* s*.*

With the three notions of block disablement, enactment and enactability at hand, we are now able to define the key notions of state update and, in turn, conformance.

Definition 3.8 (State update)**.** *Let* $s = \langle Curr, Next \rangle$ *and* $s' = \langle Curr', Next' \rangle$ *be two execution states over process model* $\mathcal{B}$*,* $\mathtt{t}$ *a step executable in* s*, and* B *a block of* $\mathcal{B}$*. We say that* $\mathtt{t}$ updates s into s'*, or alternatively that* s is updated by $\mathtt{t}$ into s'*, if:*

1. $Curr' = (Curr \cup \mathcal{E}_s^{\mathtt{t}}) \setminus \mathcal{D}_s^{\mathtt{t}}$*;*
2. $Next = (Next \cup \mathcal{N}_s^{\mathtt{t}}) \setminus \mathcal{D}_s^{\mathtt{t}}$*;*

Definition 3.9 (Conformance)**.** *Given a process model* $\mathcal{B}$*, an execution trace* $\tau = \langle \mathtt{t}_1, \ldots, \mathtt{t}_n \rangle$ *over* $\mathcal{B}$ conforms to $\mathcal{B}$ *if there exists a sequence* $\langle s_0, \ldots, s_n \rangle$ *of execution states such that:*

1. s_0 *is the initial execution state of* $\mathcal{B}$*;*
2. *for every* $i \in \{1, \ldots, n\}$*, step* $\mathtt{t}_i$ *is executable in* s_{i-1}*;*
3. *for every* $i \in \{1, \ldots, n\}$*, step* $\mathtt{t}_i$ *updates* s_{i-1} *into* s_i*.*

We remark that this definition of conformance faithfully reconstructs the classical notion of control-flow conformance defined by translating the process model of interest into a workflow net. In particular, enactability corresponds to the enablement of transitions, enactment to the firing of a transition, and state update to the calculation of the marking resulting from executing a transition in a previous marking.

As an example, consider the BPMN well-structured process model in Figure 2 of the paper. At the beginning, no active block exists. The only block that can be activated is `Seq1` that, being a sequence, can be activated when the first element in the sequence is activated, which is `Start`. Therefore, the only acceptable execution step is `Start`. Suppose that it happens. This leads to activate the block `Seq1` and, in turn, the block `Start`, which is immediately deactivated. At the same time, block `ProvideContainer` can be activated next, since it directly follows `Start`. When task `ProvideContainer` is started, the corresponding block is activated and the only accepted execution step becomes the completion of that task. When `ProvideContainer` is completed, this causes the homonimous block to be deactivated, and the subsequent block `PickUpContainer` can be activated next. Once tasks `ProvideContainer`, `PickUpContainer`, `GoToProducer` and `VerifyId` are first started then terminated with no overlapping, the next block becomes `EExc`. Since it is a forward exception handling block, `EExc` is activated either when `PickUpFailure` is activated and the exception `Unauthorized` was risen, or when the exception was not risen and `Seq2` is activated, i.e., when tasks `LoadGoods` is started. Suppose that the exception was risen and `PickUpFailure` happens. This causes block `PickUpFailure` to be activated and immediately deactivated, which in turn deactivates `EExc` and makes `End`, which directly follows `EExc`, the only acceptable execution step. Once `End` happens, the homonimous block is activated, then deactivated, then `Seq1`, being `End` the last block in the sequence, is deactivated too and the process terminates.

4 Conformance Preservation of the BPMN to E-GSM Translation

We are now in the position of proving the main result of the paper. Recall that a process model $\mathcal{B}$ is transformed into a corresponding E-GSM model $\mathcal{G}^P_{\mathcal{B}}$ according to the transformation rules exhaustively defined in [1].[1] From now on, we formulate all our definitions and results using $\mathcal{B}$ to denote the BPMN process model of interest, and $\mathcal{G}^P_{\mathcal{B}}$ to refer to its corresponding E-GSM model.

As for E-GSM, we adopt the standard GSM execution semantics, with the extended lifecycle discussed in [1]. This gives raise to the following definition of conformance in the E-GSM sense.

Definition 4.1 (E-GSM conformance)**.** *Given an E-GSM model $\mathcal{M}$, an execution*

[1] See also the technical report available at `https://re.public.polimi.it/handle/11311/976678`.

trace τ *over* $\mathcal{M}$ conforms to $\mathcal{M}$ *if the state resulting from the execution of* τ *over* $\mathcal{M}$ *is such that no stage of* $\mathcal{M}$ *is* out-of-order.

We show that the transformation is correct, in the following precise sense.

Theorem 1. *Given a process model* $\mathcal{B}$ *and an execution trace* $\tau = \langle \mathtt{t}_1, \ldots, \mathtt{t}_n \rangle$ *over* $\mathcal{B}$*, the following two conditions hold:*

1. *If* τ *conforms to* $\mathcal{B}$*, then* τ *conforms to* $\mathcal{G}^P_{\mathcal{B}}$*.*
2. *If* τ *does not conform to* $\mathcal{B}$*, let* $\tau_p = \langle \mathtt{t}_1, \ldots, \mathtt{t}_k \rangle$ *(with* $k \leq n$*) be the minimum prefix of* τ *such that* τ_p *does not conform to* $\mathcal{B}$*; then* τ_p *does not conform to* $\mathcal{G}^P_{\mathcal{B}}$ *either.*

Notice that the theorem expresses that τ conforms to $\mathcal{B}$ if and only if it conforms to $\mathcal{G}^P_{\mathcal{B}}$, with the additional property that in the case of non-conformance, then $\mathcal{G}^P_{\mathcal{B}}$ is able to detect a deviation as soon as it occurs.

To prove Theorem 1, we show a stronger result. To formulate such a result, we first need to connect the execution state of a BPMN process model to that of the corresponding E-GSM model.

Definition 4.2 (Corresponding state)**.** *Let* $s = \langle Curr, Next \rangle$ *be an execution state over* $\mathcal{B}$*, and* σ *be an execution state over* $\mathcal{G}^P_{\mathcal{B}}$*. We say that* σ corresponds to *s if the following conditions hold:*

1. *a process block* B *of* $\mathcal{B}$ *belongs to Curr if and only if the corresponding stage* S_B *in* $\mathcal{G}^P_{\mathcal{B}}$ *is* regular *and* open *in* σ*;*
2. *a process block* B *of* $\mathcal{B}$ *belongs to Next if and only if the corresponding stage* S_B *is* regular *and* closed*, and becomes* regular *and* open *upon the execution of a single execution step in* σ*.*

With this notion at hand, we prove the following core result.

Lemma 1. *Let* s *be an execution state over* $\mathcal{B}$*,* σ *be its corresponding state over* $\mathcal{G}^P_{\mathcal{B}}$*, and* $\mathtt{t}$ *be an execution step over* $\mathcal{B}$*. Then:*

1. $\mathtt{t}$ *is executable for* $\mathcal{B}$ *in* s *if and only if the execution of* $\mathtt{t}$ *in* σ *does not lead any stage of* $\mathcal{G}^P_{\mathcal{B}}$ *to become* out-of-order.
2. *if* $\mathtt{t}$ *is executable for* $\mathcal{B}$ *in* s*, then the actual execution of* $\mathtt{t}$ *over* $\mathcal{B}$ *in* s *leads to a state* s' *that corresponds to the state* σ' *obtained by executing* $\mathtt{t}$ *over* $\mathcal{G}^P_{\mathcal{B}}$ *in* σ*.*

It is straightforward to prove that Lemma 1 implies Theorem 1 (through its inductive application over the input trace τ).

To prove Lemma 1, we proceed by induction on the execution state evolution, considering all the possible effects over the enacted and enactable blocks, depending on which blocks are affected by the current execution step. This, in turn, is captured by Definitions 3.4, 3.5, 3.6, and 3.7.

The base case is the initial state s_0. By definition, in this state there is no enacted block, and the only enactable blocks are the top process block together with its start event. Let $P = \langle \mathtt{n}, \mathtt{proc}, \langle E, A, F \rangle \rangle$ be the top process block of $\mathcal{B}$, and

$E = \langle \mathtt{e}, \mathtt{event}, \langle\rangle \rangle$ be its start event block. Then $s_0 = \langle \emptyset, \langle P, E \rangle \rangle$. By construction, the initial state σ_0 of $\mathcal{G}^P_{\mathcal{B}}$ is such that there is no open stage, and the only stage that can be regularly opened is the main stage S_P corresponding to the main process block P, together with its first sub-stage S_E, corresponding to E. Consequently, our induction hypothesis applies, since σ_0 corresponds to s_0.

In this initial state, the only executable step for $\mathcal{B}$ is $\mathtt{e}$, and the effect of its execution over s_0 is to obtain $s_1 = \langle \{P\}, \{A\} \rangle$. By considering Figure 7 of [1], this is also true for $\mathcal{G}^P_{\mathcal{B}}$. Assume that $\mathtt{t} = \mathtt{e}$. Then, $\mathtt{t}$ is executable in σ_0 according to $\mathcal{G}^P_{\mathcal{B}}$. In fact:

- $\mathtt{t}$ regularly opens stage S_P, because it makes true the data flow guard DFG1 of S_P, and no process-flow guard is present in S_P.
- $\mathtt{t}$ regularly opens and immediately closes the first inner stage S_E of S_P.

In addition, after executing $\mathtt{t}$, the sub-stage S_A of S_P that corresponds to A is the only stage that can be open upon the execution of an additional step. In fact, it is the only sub-stage of S_P whose process-flow guard PFG1 is actually true (due to the fact that S_E has been closed). This shows that the execution state obtained by applying $\mathtt{t}$ over σ_0 for $\mathcal{G}^P_{\mathcal{B}}$ indeed corresponds to s_1.

Now assume that $\mathtt{t} \neq \mathtt{e}$, which is not executable in s_0 according to $\mathcal{B}$. We show that also $\mathcal{G}^P_{\mathcal{B}}$ forbids the execution of $\mathtt{t}$. Since $\mathtt{t} \neq \mathtt{e}$, two cases may arise. Either $\mathtt{t}$ corresponds to the termination event of block F, or it corresponds to another task start/complete or event. We separately analyze the two cases.

If $\mathtt{t}$ corresponds to the termination event captured by F, then stage S_P regularly opens (since DFG3 becomes true), and S_F opens out-of-order, since its data-flow guard DFG1 is true, but its process flow guard PFG1 is false (due to the fact that stage S_A has not yet been opened, and hence has not yet achieved its milestone).

If $\mathtt{t}$ corresponds to the start/termination of a task, or to an event different from the start and termination events of P, $\mathtt{t}$ must trigger some sub-stage of S_A, In fact, S_A is the main stage of $\mathcal{G}^P_{\mathcal{B}}$, and hence directly or indirectly contains all events/tasks but the start and termination events of the top process. By the recursive definition of the transformation of $\mathcal{B}$ into $\mathcal{G}^P_{\mathcal{B}}$, this also means that the data-flow guard DFG1 of S_A becomes true. However, since the start event of P has not yet occurred, mileston M1 of stage S_E has not yet been achieved, and hence the process-flow guard PFG1 of S_A evaluates to false. This, in turn, means that S_A opens out-of-order.

This concludes the proof of the base case. As for the inductive case, let us consider an execution state $s = \langle \mathit{Curr}, \mathit{Next} \rangle$ over $\mathcal{B}$, the execution state σ over $\mathcal{G}^P_{\mathcal{B}}$ that corresponds to s, and an execution step $\mathtt{t}$. We show that $\mathtt{t}$ is executable in s over $\mathcal{B}$ if and only if its execution in σ over $\mathcal{G}^P_{\mathcal{B}}$ does not cause any stage to become out-of-order. Then, we show that every executable step over $\mathcal{B}$ in s leads to a state s' that corresponds to the state σ' obtained after executing that step over $\mathcal{G}^P_{\mathcal{B}}$ in σ. To exhaustively prove this, we have to consider all the three possible types of execution steps, and all possible transitions a block may be subject to when moving from state s to s', which depend in turn on two factors: the type of the block, and the kind of transition the block is subject to. There are four possibilities for this latter dimension.

Consider a block B. The four possibilites are: *(i)* the block is enactable in s and becomes enacted in s' (this is captured by Definition 3.6); *(ii)* the block is enactable or enacted in s and is disabled in s' - hence, it is not enacted nor enactable in s' (this is captured by Definition 3.5); *(iii)* the block is not enacted nor enactable in s, and it becomes enactable in s' (this is captured by Definition 3.7); *(iv)* the block is not affected by the transition.

We discuss one such combinations, the others can be proven similarly. Consider as execution step the start of a task, i.e., $\mathtt{t} = \langle \mathtt{start}, \mathtt{a} \rangle$, where $\mathtt{a}$ is the name of a task in $\mathcal{B}$, in turn identified as block $A = \langle \mathtt{a}, \mathtt{task}, \langle\rangle \rangle$ in $\mathcal{B}$. We use S_A to denote the stage of $\mathcal{G}^P_{\mathcal{B}}$ that correponds to A.

We prove that $\mathtt{t}$ is executable in s over $\mathcal{B}$ if and only if its execution in σ over $\mathcal{G}^P_{\mathcal{B}}$ does not lead any stage to become out-of-order.

($\Rightarrow$) According to Definition 3.4, $\mathtt{t}$ is executable in s over $\mathcal{B}$ if and only if A is enactable in s. By Definition 4.2, this implies that S_A can be regularly opened by means of a single execution step. By considering the transformation rule for tasks (cf. Figure 3 in [1]), such an execution step is indeed $\mathtt{t}$.

($\Leftarrow$) We prove the contrapositive, i.e., we show that if $\mathtt{t}$ is not executable in s over $\mathcal{B}$, then executing $\mathtt{t}$ in σ over $\mathcal{G}^P_{\mathcal{B}}$ causes at least one stage to open out-of-order. According to Definition 3.4, $\mathtt{t}$ is not executable in s over $\mathcal{B}$ if and only if A is not enactable in s. By Definition 4.2, this implies that S_A cannot be (regularly) opened by means of a single execution step. By inspecting the transformation rules in [1], one can immediately notice that the translation of every type of non-leaf block leads to a corresponding stage that reports, as data-flow guard, all the data-flow guards of its inner blocks. This, in turn, implies that there must be a stage S_P such that: *(i)* S_P has a data-flow guard that corresponds to the start of A, i.e., that matches with $\mathtt{t}$; *(ii)* the parent stage of S_P is already open (in the extreme case, the parent is the top stage of $\mathcal{G}^P_{\mathcal{B}}$). By considering all possible structures for S_P, and the fact that S_A cannot be opened by $\mathtt{t}$, one can show that there is always a sub-stage of S_P whose data-flow guard matches with $\mathtt{t}$, and that has a process-flow guard evaluating to false. This, in turn, implies that, upon the execution of $\mathtt{t}$, that stage will open out-of-order.

We now prove that, when $\mathtt{t}$ is executable in s, the actual execution of $\mathtt{t}$ in s and σ leads to consequent states s' and σ' that correspond to each other.

By inspecting Definition 3.5, we notive that $\mathcal{D}^{\mathtt{t}}_s = \emptyset$ (no block is disabled by the start of a task). Hence, we do not have to discuss the case of block disablement.

By inspecting Definition 3.6, we have to discuss the following cases:

- (Definition 3.6, Case 1) From the executability of $\mathtt{t}$, we get that $A \in \mathit{Next}$, hence the task block A is enacted by $\mathtt{t}$ in s. Since σ corresponds to s, S_A can be open by a single execution step. By considering Figure 3 in [1], such step is actually $\mathtt{t}$, since $\mathtt{t}$ matches the unique data-flow guard of S_A. Upon the execution of $\mathtt{t}$, S_A becomes open in σ'. Hence, as far as A and S_A are concerned, σ' indeed corresponds to s'.
- An enactable ancestor block B in $\mathcal{B}$ (which actually corresponds to a stage S_B

in $\mathcal{G}^P_{\mathcal{B}}$ containing the start of task `a` as one of its data-flow guards, hence opening when executing `t`) is enacted by `t` in s. We have to show, case-by-case, that for stage S_B all its process-flow guards evaluate to true, which in turn implies that S_B opens regularly.

- (Definition 3.6, Case 3) B is an activity block enacted by `t`. This case is trivial, since S_B in this case does not have any process-flow guard (cf. Figure 6 in [1]).
- (Definition 3.6, Case 4) B is an enactable sequence block whose first inner block B' is enacted by `t`. This case is tackled by Rule 5 in [1]. Such a rule guarantees that S_B is opened regularly, since it does not have any process-flow guard. At the same time, it also guarantees that $S_{B'}$ is opened regularly. In fact, being $S_{B'}$ the first substage of S_B, its process-flow guards only checks that $S_{B'}$ is not opened twice while its parent stage S_B is open. This is indeed true in this case, since S_B and $S_{B'}$ are being opened simultaneously.
- (Definition 3.6, Case 5) B is an enactable gateway block with one of its inner blocks B' enacted by `t`. By considering the rules in [1] that tackle the different gateway blocks, it is immediate to see that S_B is regularly enacted by `t` since it does not have any process-flow guard, whereas S'_B presents the same situation as the case discussed in the point above. The only subtle case is the one of `choice`. To show that this case is also properly handled, recall that, for a choice block, the fact that B' is enactable implies that the corresponding condition evaluated to true, and such a condition is also mentioned in the process-flow guard of S'_B.
- (Definition 3.6, Case 6) B is an enactable loop block whose inner forward block B' is enacted by `t`. This case is handled by Rule 9 in [1]. Again, also in this case S_B does not have any process-flow guard, and its substage $S_{B'}$ has a process-flow guard that simply checks that $S_{B'}$ has not yet been concluded.

By inspecting Definition 3.7, the only case that must be discussed given the fact that `t` corresponds to the start of a task, is that of an activity block B that is enacted by `t`. In this situation, Case 2 of Definition 3.7 applies, indicating that the boundary event of B becomes enactable. This situation is properly reconstructed by Rules 3 and 4 of [1]. Indeed, they guarantee that upon the opening of S_B, special (fault logging) milestones corresponding to the boundary event are indeed achievable.

This concludes the proof for the case where the considered execution step is the start of a task. Task termination and event occurrence are proven similarly.

References

[1] L. Baresi, G. Meroni, and P. Plebani, "A GSM-based Approach for Monitoring Cross-Organization Business Processes using Smart Objects." Accepted for publ., 2015.

[2] G. Meroni, L. Baresi, and P. Plebani, "Translating BPMN to E-GSM: specifications and rules," tech. rep., Politecnico di Milano, 2016.