Abstract—A configurable process model captures a family of similar processes. Such models can be configured to obtain a process variant according to specific requirements. With this aim, several approaches have been proposed for the configuration of process models. Nevertheless, an increasing attention is being paid to achieve this in a sound manner due to the complex inter-dependencies between the configuration decisions. In this work, we aim to guide the process analyst to easily configure process models while preserving soundness. To do so, we propose a formal approach for ensuring correctness of business process configurations while considering structural constraints they have to obey. Specifically, using the Event-B language, we formally define a configurable process model, its correctness-preserving conditions and its configuration constraints.

Index Terms—Business process management; Configurable process model; Formal verification; Event-B;

I. THE EVENT-B METHOD

Event-B [2] is a state-based formal method for modeling and analyzing systems. It is based on classical logic and set theory. A formal model in Event-B uses two types of components to describe a system: machines and contexts. A machine contains dynamic elements that describe the state of the system, which are variables $v$ and events $E$. Variables are constrained by invariants $I(v)$, which are supposed to hold whenever the state of the system change. Whereas a context represents the static part of the model, it consists of sets $x$, constants $c$, and axioms $A$ which specifies their properties. To have access to its elements, a context is seen by a machine. A context may be also extended by another to introduce more elements.

In the Event-B model, we use events to describe the behavior of a system. In general, an event takes the form:

- $\text{evt } \triangleq \text{any } x \text{ where } G \text{ then } \text{Act end}$

Where $x$ is the list of event parameters, $G$ denotes a conjoined list of predicates defining the guard of the event which represents the necessary conditions for the event to occur. An action $\text{Act}$ is a simple assignment to a state variable to describe the consequence of the event occurrence. In this paper, we restrict ourselves to the deterministic assignment, the becomes equal substitution, denoted by $(x := e)$.

An Event B model encodes a state transition system where variables represent the state, and events represent the transitions from one state to another. A system is characterized by the set of licit traces corresponding to the fired events of the model which respects the described properties. The traces define a sequence of states that may be observed by properties. Formal modeling of BPMN using Event-B is motivated by the observation that BPMN is a transition-based language where (i) sequence flows are considered as transitions and (ii) states are linked by transition lines representing control flows. These transitions are represented in Event-B by the events of the EVENTS clause allowing the system state change.

Event-B is supported by the eclipse-based RODIN platform [3] on which different external tools (provers, animators, model-checkers) can be plugged in order to animate/validate a formal development.

II. VERIFYING BUSINESS PROCESS MODEL SOUNDNESS

In this section, we introduce our Event-B formalism of the business process model and its correctness constraints. The consistency of the model is ensured by formal proofs (see Section IV-A).

A. Business Process Model

We start by presenting the context of the model $C0$ which holds the following finite sets: (i) $BPS$, which defines the set of possible processes, (ii) $NODES$, which contains three values denoting types of nodes: activities (i.e. ACTS), split connectors (i.e. CON_S), and join connectors (i.e. CON_J), and (iii) $TYPES$, which defines three types of connectors: OR, XOR and AND.

Then, we define the machine $M0$ which sees the context $C0$ described above. The variables of $M0$ and their typing invariants (i.e. the properties of the machine) are given in Listing 1. We define a variable $BP$ to store the created processes. Obviously, $BP$ is a subset of $BPS$ (Inv1, Listing. 1). To map each process to its nodes, we introduced the relation $BP\_Nodes$ from set $BP$ to $NODES$ (Inv2, Listing. 1). We define the start and the end events as activities, respectively the initial and the final activities, using respectively the total function $Initial$ (Inv3, Listing. 1) and the relation $Final$ (Inv4, Listing. 1); since a business process has exactly one initial activity but may have several final ones. In BPMN, each connector, either a split or a join, has a type. This is modeled using the total function $CON\_Type$ (Inv5, Listing. 1).

Listing 1: M0’s variables and typing invariants
The control flow perspective describes activities and their execution ordering through different constructors, which permit the control flow execution [10]. This execution order is modeled using the total function \( SEQ \) (Inv6, Listing 1).

### B. Control flow Constraints

In order to ensure consistent and syntactically correct process control flow, we define a set of constraints. We illustrate some of them in Listing 2:

- Each activity (except the initial and the final ones) have exactly one incoming (Inv11) and outgoing arc (Inv12).
- A split connector has exactly one incoming (Inv13) and at least two outgoing arcs (Inv14).
- A join connector has at least two incoming arcs (Inv15) and exactly one outgoing (Inv16).

### C. Soundness Constraints

A process is considered to be sound if and only if it fulfills the following conditions:

- All nodes of the process can be activated (i.e., every node can be reached by the initial activity), as depicted by Inv20 in Listing 3 (where \( cls^3 \) is the transitive closure of the relation \( SEQ(bp) \)).
- For each activity in the process, there is at least one possible sequence leading from this activity to a final activity (i.e., the termination is always possible). This condition is captured by Inv21 of Listing 3.

1. A \( \triangleq f \) denotes a domain restriction: \( A \triangleq f = \{ x \rightarrow y : x \rightarrow y \in f \land x \in A \} \).
2. The inverse of a function \( f, (f^{-1}) \), is denoted in Event-B as \( (f \sim) \).
3. \( cl(r) \) denotes the closure of the relation \( r \) defined, for each relation \( (r \in S \leftrightarrow S) \), by:

\[
\begin{align*}
(1) & \quad cl(r) = \bigcup_{i=1}^{\infty} r^i \\
(2) & \quad r^1 = r \\
(3) & \quad \text{for each } n \geq 2r^n = (r;r^{n-1})
\end{align*}
\]

The transitive closure formulations were expressed as machine theorems.

### Listing 3: Soundness Constraints invariants

| Inv20 | ∀bp.node.(bp ⇒ node ∈ BP_Nodes ∧ node ≠ Initial(bp) ⇒ node ∈ (cls(SEQ(bp)))([Initial(bp)])) |
| Inv21 | ∀bp.node.(bp ⇒ node ∈ BP_Nodes ∧ node ≠ Final(bp) ⇒ (cls(SEQ(bp)))([node]) ∩ Final([bp]) ≠ ∅) |

In the following section, we tackle the configuration procedure. Hence, we define configuration operations to apply on a business process as well as a set of configuration constraints to be respected.

### III. Formalizing Configurable Process Models using Event-B

In this section, we introduce the formalization of a configurable process model as well as the configurable elements: activity configuration, and Connector configuration. In this formalization each configuration step is performed by an appropriate event. In order to derive correct variants, we define a set of constraints using invariants and event guards. Once preserved by all events, our approach avoid erroneous situations such as deadlock and lack of synchronization.

### A. Configurable Process Model

We define a total function \( Configurable_Nodes \) (Inv24, Listing 4) returning a Boolean value to state whether a given node is configurable or not in each process in which it appears.

### Listing 4: Configuration invariants

| Inv24 | ∀bp.Configurable_Nodes ∈ BP_Nodes ⇒ BOOL |
| Inv25 | Is_Configuration_Of ∈ BP ⇒ BP |
| Inv26 | Is_Configuration_Of = Is_Configuration_OFF.Act ∪ Is_Configuration_ON.Act ∪ Is_Configuration.OR_S ∪ Is_Configuration.OR_J ∪ ... ∪ Is_Configuration_ToSeq |

We assume that a configurable process could be changed by applying a sequence of operations on it (e.g., excluding an activity or changing a connector type).

So, at each configuration step, we define the process change using the partial function \( Is_Configuration_Of \) (Inv25, Listing 4). For instance, a couple \( (pb2, bp1) \) belongs to Is_Configuration_Of if and only if at least one potential configuration has been applied on \( bp1 \) in order to obtain a configured process \( bp2 \). Inv26 asserts that these configurations could affect either (i) activity configuration, by excluding (i.e. Is_Configuration_OFFAct) or including it (i.e. Is_Configuration_ONAct), or (ii) connector configuration, by restricting splits outgoing branches (i.e. Is_Configuration.OR_S, Is_Configuration.XOR_S and Is_Configuration.AND_S), by restricting joins incoming branches (i.e. Is_Configuration.OR_J, Is_Configuration.XOR_J and Is_Configuration.AND_J) or by keeping only one branch (i.e. Is_Configuration.ToSeq). These configuration types are further detailed in the following sections.

The configuration of a business process model may affect the soundness by two types of potential errors: lack of synchronization and deadlocks [18]. These situations result from a mismatch between splits and joins. Therefore we define two
invariants for each error. These invariants should be preserved by all the events that we define in the following subsections (i.e. the events capture the configuration operations). For instance, the lack of synchronization is captured by joining with an exclusive choice operator, a control-flow that was split by an AND operator, which leads to an improper termination. Thanks to Inv22 (Listing 5), this situation is not allowed in our model. Specifically, having a AND-split operator (line 2), for each couple of outgoing nodes n1 and n2 (lines 3 and 4), the first common node⁴ opj (lines 4 to 7) should be an AND or a not yet configured OR connector (lines 9 and 10).

Listing 5: Synchronization invariant

\[
\begin{align*}
1. \text{Inv22} & : \forall bp, node. (bp \rightarrow opj \in BP_Nodes \rightarrow CON_S \\
2. \quad \wedge \text{CON_Type}(bp \rightarrow opj) = AND \\
3. \quad \wedge n1 \in \text{SEQ}(bp) \wedge n2 \in \text{SEQ}(bp) \quad \wedge n1 \neq n2 \\
4. \quad \Rightarrow \exists (\text{opj}, opj) \in \text{(seq}(bp)) \cap (n1 \cup (n1)) \\
5. \quad \cap (\text{seq}(bp)) \cap (n2) \wedge \quad \text{SEQ}(bp) = \{\{1\} \cap (\text{seq}(bp)) \cap (n1) \cup (n2)) \wedge \\
6. \quad \Rightarrow SEQ(bp) = \{\{1\} \cap (\text{seq}(bp)) \cap (n1) \cup (n2)) \\
7. \quad \Rightarrow (\text{SEQ}(bp) \cap (n2) \cap (n2)) \neq \emptyset \mid \{1\}) \\
8. \quad \Rightarrow (\text{CON_Type}(bp \rightarrow opj) = AND \wedge (\text{CON_Type}(bp \rightarrow opj) = OR \\
9. \quad \wedge (\text{CON_Type}(bp \rightarrow opj) = TRUE))
\end{align*}
\]

Furthermore, ensuring a deadlock-free control flow is an important criterion for a configuration while preserving soundness. We aim to guarantee the absence of situations where a node can never be activated. We model such situations using Inv23 in Listing 6. This invariant asserts that an OR-split or an XOR-split should not be followed by an AND-join. Basically, having a split type different from AND (AND line 2), we check that the first common node (line 4 to 7) of every couple of outgoing nodes n1 and n2 (line 3) is not an AND-join (line 9).

Listing 6: Deadlock-freeness invariant

\[
\begin{align*}
1. \text{Inv23} & : \forall bp, node. (bp \rightarrow opj \in BP_Nodes \rightarrow CON_S \\
2. \quad \wedge \text{CON_Type}(bp \rightarrow opj) = \neq AND \\
3. \quad \wedge n1 \in \text{SEQ}(bp) \wedge n2 \in \text{SEQ}(bp) \quad \wedge n1 \neq n2 \\
4. \quad \Rightarrow \exists (\text{opj}, opj) \in \text{SEQ}(bp) \cap (n1 \cup (n1)) \\
5. \quad \cap (\text{SEQ}(bp)) \cap (n2) \wedge \quad \text{SEQ}(bp) = \{\{1\} \cap (\text{SEQ}(bp)) \cap (n1) \cup (n2)) \\
6. \quad \Rightarrow (\text{SEQ}(bp) \cap (n2) \cap (n2)) \neq \emptyset \mid \{1\}) \\
7. \quad \Rightarrow (\text{CON_Type}(bp \rightarrow opj) = \neq AND )
\end{align*}
\]

**B. Activity Configuration**

The configurable activities could be included or excluded in process variants according to process analyst choice. To define this activity configuration, two invariants (i.e. to define configuration constraints) and two events (i.e. the operations to allow the process change) are introduced.

For instance, in case of excluding activity, the invariant inv24 insures that for each couple (bp2, bp1) belonging to Is_Configuration_OFFAct (line 2) there exists an activity act such that: (i) act is configurable (line 5), (ii) act is the only difference between bp1 nodes and bp2 nodes (line 6), (iii) act and its dependencies are removed and a new one is created in bp2 (lines 7 and 8).

Listing 7: OFF Configuration invariant

\[
\begin{align*}
1. \text{Inv24} & : \forall bp1, bp2. (bp1 \in BP \wedge bp2 \in BP) \\
2. \quad \Rightarrow bp2 \rightarrow bp1 \in \text{Is_Configuration_OFFAct} \\
3. \quad \Rightarrow ( \exists \text{act}. (\text{act} \in \text{ACTS} \wedge \text{act} \in \text{BP_Nodes} (\{bp1\}) \\
4. \quad \Rightarrow \text{bp} \rightarrow \text{act} \rightarrow \text{TRUE} \in \text{Configurable_Nodes} \wedge \\
5. \quad \Rightarrow \text{BP_Nodes} (\{bp1\}) \cap \text{BP_Nodes} (\{bp2\}) = \{\text{act}\} \\
6. \quad \Rightarrow \text{SEQ}(\text{bp2}) = \{\{\text{act}\} \neq \text{SEQ}(\text{bp1}) \neq \{\text{act}\} \} \\
7. \quad \Rightarrow \{\text{SEQ}(\text{bp1}) \sim \{\text{act}\} \times (\text{SEQ}(\text{bp1}))\{\text{act}\}\}
\end{align*}
\]

Activity configuration is performed through either: (i) ConfigureACTON event which keeps the activity; or (ii) ConfigureACTOFF event which excludes it. For instance, we present in Listing 8 the event ConfigureACTOFF. Based on a configurable process bp1, a configured process bp2 is a result of excluding an activity act. As pre-condition, act must be configurable (grd3). The triggering of this event allows the configured process bp2 to inherit from the configurable process bp1: (i) its nodes whilst removing act (act2), (ii) its initial (act3) and final activities (act4), (iii) all its nodes relations (i.e. SEQ(bp1) while removing act dependencies and creating a new one connecting act successor and predecessor (act5), (iv) its configurable nodes (act6), and (iv) types of its connectors. Finally, we define bp2 as a configuration of bp1 whilst excluding act using act8. Similarly, the event ConfigureACTON allows to maintain the same process by keeping the configurable activity. The only change applied on the resulting process is the conversion of the activity from a configurable node to a non configurable one.

Listing 8: Excluding activity event

\[
\begin{align*}
1. \text{ConfigureACTOFF} & \wedge \text{ANY bp1 bp2 act} \\
2. \quad \text{WHERE} \\
3. \quad \text{grd1} : \text{bp1} \in BP \wedge \text{act} \in \text{ACTS} \wedge \text{bp1} \rightarrow \text{act} \in \text{BP_Nodes} \\
4. \quad \text{grd2} : \text{bp2} \in \text{BP} \wedge \text{BP} \\
5. \quad \text{grd3} : \text{Configurable_Nodes}(\text{bp1} \rightarrow \text{act}) = \text{TRUE} \\
6. \quad \text{...} \\
7. \quad \text{then} \\
8. \quad \text{act1} : \text{BP} \equiv \text{BP} \wedge \text{act2} \\
9. \quad \text{BP_Nodes} := \text{BP_Nodes} \cup (\{bp2\} \times (\text{BP_Nodes}(\{bp1\}) \setminus (\{act\})) \\
10. \text{act3} : \text{Initial}(bp2) \equiv \text{Initial}(bp1) \\
11. \text{act4} : \text{Final}(bp2) \equiv \text{Final}(bp1) \\
12. \text{act5} : \text{SEQ}(bp2) := (\{\text{act}\} \neq \text{SEQ}(bp1) \neq \{\text{act}\}) \cup \\
13. \quad (\text{SEQ}(bp1) \sim \{\text{act}\} \times (\text{SEQ}(bp1))\{\text{act}\}) \\
14. \text{act6} : \text{Configurable_Nodes} := \text{Configurable_Nodes} \cup \\
15. \quad (\{\text{node} \in \text{BP_Nodes}(\{bp1\}) \setminus \{\text{act}\})) \\
16. \text{act7} : \text{CON_Type} := \text{CON_Type} \cup \\
17. \quad (\{\text{node} \in \text{BP_Nodes}(\{bp1\}) \setminus \{\text{act}\}) \cup \\
18. \text{act8} : \text{Is_Configuration_OFFAct} := \text{Is_Configuration_OFFAct} \cup \\
19. \quad \{\text{cbp} \rightarrow \text{bp}\}
\end{align*}
\]

**C. Connector Configuration**

A connector configuration represents a decision point that is of relevance during the process configuration life cycle. Each decision has to consider the following requirements: (1) the configuration constraints for each type of connector (e.g. ⁵)

⁴Having two nodes n1 and n2, the first common node is the first node which belongs to the transitive closure of both nodes n1 and n2

⁵A \( f \notin A \) denotes a domain subjection: \( A \ni f = \{ x \mapsto y | x \in f \wedge x \notin A \} \)
To model a connector configuration, we define for each type two events (operations). A first event models the split configuration, ConfigureORSplit, ConfigureXORSplit and ConfigureANDSplit. A second event models the join configuration, ConfigureORJoin, ConfigureXORJoin and ConfigureANDJoin.

For instance, Listing 10 illustrates the event ConfigureORSplit. This event allows configuring a configurable split connector ops (grd3) from OR type (grd4) to any type (grd5) (as OR could be configured to any type) while preserving the branches starting by nodes in nodes. Obviously, the number of remaining branches should be greater than two (grd6). However, each branch can be removed only if all its nodes are configurable (grd8). Furthermore, our model avoids connecters types mismatching by considering corresponding join connectors. For example, using grd9, for each two outgoing branches if the corresponding join is an AND, then the split should be configured to an AND as well.

### Listing 10: Or Split configuration event

```plaintext
ConfigureORSplit ≜ ANY bp1 bp2 ops nodes to_deletedNodes subgraph conToSeq
WHERE
grd1: bp1 ∈ BP ∧ ops ∈ CON_S ∧ bp1 → ops ∈ BP_Nodes
grd2: bp2 ∈ BPS ∪ BP

grd3: Configure_Nodes(bp1) → ops ⇒ TRUE
grd4: CON_Type(bp1) → ops ⇒ OR
grd5: to ∈ TYPES
grd6: nodes ⊆ SEQ(bp1)[ops] ∧ card(nodes) ≥ 2
grd7: deletedNodes = (∪zz.zz ∈ BPS \ BP \ Initial[bp1]) \ ((Initial[bp1]) \ {zz})
grd8: deletedNodes ⊆ Configurable_Nodes ⇒ (TRUE) \ bp1
grd9: subgraph = (deletedNodes ∈ SEQ(bp1) \ ((ops) × SEQ(bp1) \ nodes))) \ ∈ deletedNodes
```

Now, we examine the join operator configuration. Obviously, a join operator depends of one or more splits. Then, we take into account the corresponding split connector, previously configured, by checking its configured type. For instance, in the event dealing with configuring OR-join, we add the grd9 (see Listing 11) to verify that if there exists two distinct nodes n1 and n2 of op incomings (line 2) having the first common previous node (lines 3 to 6) configured to a type not equal to AND (line 7), then op should not be configured to an AND (line 8). Hence, this condition guarantees a deadlock-free configuration of the join operator. Similarly, we add another guard to avoid the lack of synchronization situation.

### Listing 11: Or join guard

```plaintext
grd9 : ANY n1 n2.
  n1 ∈ SEQ(bp1) ∪ SEQ(bp2) ∧ n2 ∈ SEQ(bp1) ∪ SEQ(bp2) ∧ n1 ≠ n2
  ∧ (∃opi.opi ∈ InitialOp \ ((ops) × (SEQ(bp1) ∪ SEQ(bp2))) \ (n1))
  ⇒ ((SEQ(bp1) \ nodes) ∪ (SEQ(bp2) \ nodes)) \ ∈ deletedNodes
  ⇒ two AND
```

Besides the configuration of a connector from one type to another, it is possible to configure it to a sequence of only one branch. This is defined using the event Configure_Thread. As mentioned previously, only OR and XOR types could be configured to a sequence. This constraints is ensured by a guard (CON_Type(bps → ops) ≠ AND). This event checks also whether the corresponding join should be derived to a sequence or not. As a result, the branch to retain is linked to the predecessor and the successor of the deleted operators. For reasons of space, we do not illustrate this event in detail.

### IV. Verification and Validation

#### A. Verification using Proof Obligations

In order to demonstrate that the formal specification of configurable process models is correct, a number of generated proof obligations (POs) should be discharged. Using the Rodin tool [3], our model generated 358 proof obligations; most of them (272 POs ≈ 76%) were automatically discharged; more complex ones (86 POs ≈ 24%) required the interaction with the provers to help them find the right rules to apply but also to define additional rules that may lack in the rule base of the prover.

These POs ensure that the invariants which model the different constraints on the configurable business processes and the derived variants, are always satisfied (i.e. they hold
initially; and each event preserves them). For each event of the form (WHEN G THEN Act) with G and Act representing the guard and the action respectively, the following proof obligation is generated to verify that the execution of the action Act under the guard G permits to preserve the invariant [2]:

\[(Inv \land G) \Rightarrow [Act]Inv\]

An example of the proofs, we have established, concerns the event ConfigureACTOFF correctness with respect to the invariant \(inv20\): we have to prove that even if an activity \(act\) is set to OFF, it remains possible to reach each node from the initial one. This holds since we have added a control from linking the predecessor of \(act\) to its successor. To discharge this proof that refers to the closure of a relation, we have added the rule defining the closure of the union of two relation \(s\) and \(r\):

\[
r \in t \leftrightarrow t \land s \in t \Rightarrow cls(r \cup s) = cls(r)((id(t) \cup cls(r));s)+(id(t) \cup cls(r))
\]

B. Validation by animation

In this section, we discuss the validation of our developed formal model for business process configuration using the ProB [12] plugin. Hence, we use animation and model checking to execute this specification. Concretely, the configurable process specified using our event-B model is executed by triggering the configuration events presented earlier. Then, we could play and observe different scenarios and check the behavior of our model by showing at each step the values of each variable, which events are enabled and which are not. The process of animating our Event-B model involves three steps:

- **Step1**: We first give values to the constants, carrier sets and the variables of the model (through AXIOMS and INITIALISATION clauses), to specify the configurable process model elements values.

- **Step2**: We start the animation by firing the INITIALISATION event to set the system in its initial state; at this step all defined invariants should be respected in order to ensure the correctness of our constraints, and thus the correctness of our configurable process model.

- **Step3**: Then, we proceed the configuration steps we need to follow, and for each step: (i) the animator computes all guards of all events, enables the ones with true guards, which refer to the possible configurations with respect to all constraints, and shows values make these guards true; thus, (ii) we pick one value, if any, which allows to fire the enabled event and, in consequence, the substitutions are computed. Finally, (iii) the animator checks if the invariants still hold.

To illustrate these steps, we proceed the animation of a scenario as follows. After initializing the model using the configurable process model (Step1). We ensure that it is correct (Step2). Next, we follow these configuration steps (Step3): (1) we trigger the ConfigureORSplit event to configure the split operator \(ops1\) from OR to an AND (to = AND) while maintaining the same branches, (2) \(ops3\) and \(opj1\) are configured (using ConfigureToSeq event) to a sequence starting from \(a3\) (\(a3\) is set to ON as well), (3) the activity \(a7\) is set the to ON (using ConfigureACTIONS) since \(a3\) in included in the previous step (the mapping of \(a7\) to OFF in not allowed in accordance with guideline (1)), (4) \(ops5\) and \(opj3\) are also configured to a sequence starting from \(a7\) (only this branch could be preserved, since the second branch nodes are configurable), next, (5) when configuring the join operator \(opj2\), the only allowed alternative is to fire the event ConfigureORJoin with the connector type parameter AND (see Fig. 1). By restricting configuration choices, we guaranteed that the resulted variant have not improper termination caused by the lack of synchronization.

![Fig. 1: the scenario animation using ProB](image)

V. RELATED WORK

To cope with business process variability [14], processes need to support flexibility. On the basis of this observation, our work is based on configurable process models [15]. Several approaches have been proposed to model variability in configurable process models [15], [6], [9]. Our approach stems from the fundamental method of configuration through configurable nodes [15].

Many approaches have been proposed to assist process analysts in selecting desirable configuration choices according to specific requirements. In [13], authors propose a questionnaire-based approach for configuring process models. They use a set of questions defined by domain experts and answered by designers. Authors in [11] introduce the use the configuration rules in order to configure a reference process template. In [4], [7], authors propose a feature-oriented approach based on feature models to represent variability. Authors in [5] use configuration guidelines for assisting analysts in configuration based on business needs without preserving correctness. In our work, we assist analyst to derive correct variants using a step-based Event-B model while formally preserving correctness.

One objective is to ensure soundness of the configured process variant. For each process variant its soundness should be guaranteed at design time. For instance, authors in [8] discuss ensuring soundness of variant models in the Provop
framework. In [16], authors define the requirements for a correctness preserving configurable process modeling. They highlight the soundness property and they derive propositional logic constraints that, if satisfied guarantee the behavioral correctness of the configured model. Then, the approach in [17] used partner synthesis. However, existing proposals are often applied to Workflow nets, which means that other modeling languages such as BPMN, which is one of the most suitable languages for business analyst) should be mapped with equivalent induced Petri Net to check their correctness [19]. In our work, we provide an Event-B formal approach to check the configuration procedure correctness of a BPMN process based through different steps.

VI. CONCLUSION

Based on a formal framework, we propose in this paper to check and analyze correctness of not only configurable process model but also configured variants using Event-B. To do so, we introduce a step-based configuration approach to guide the analyst by providing at each step the potential configuration choices which will guarantee a correct result. We have succeeded to verify structural constraints concerning configurable process model soundness (e.g. each node reachable from the initial activity, has always the option to complete). We have also reached our goal in guaranteeing configured variants correctness by avoiding improper completion situations (i.e. cased by the deadlock or the lack of synchronization).

We intend to extend this work by adding a refinement level in which we take into account configuration requirements and guidelines. We aim also to include the resource allocation verification in a configurable process model.

REFERENCES


