Deadlock-Freeness Verification of Business Process Configuration Using SOG

Souha Boubaker[1,2], Kais Klai[3(✉)], Katia Schmitz[3], Mohamed Graiet[4], and Walid Gaaloul[1]

[1] Telecom SudParis, UMR 5157 Samovar, Universite Paris-Saclay, Paris, France
souha.boubaker@telecom-sudparis.eu
[2] ENIT, UR-OASIS, University of Tunis El Manar, Tunis, Tunisia
[3] LIPN, CNRS UMR 7030, University of Paris 13, Villetaneuse, France
kais.klai@lipn.univ-paris13.fr
[4] ISIMM, Monastir University, Monastir, Tunisia

Abstract. Configurable process models are increasingly used in many industries as reference processes shared between different process tenants. These processes are configured and adapted according to their specific needs through *configurable elements* (i.e. the variation points). Since configuration decisions are taken prior to execution, incorrect ones may lead to critical behavioral issues such as deadlocks. In this work, we propose a formal behavioral model based on the Symbolic Observation Graph (SOG) allowing to find the set of correct configuration choices while avoiding the state-space explosion problem. This set of configuration choices, jointly provided with the configurable process, will support and help business analysts in deriving deadlock-free variants.

Keywords: Business process management · Configurable process model · Process variants · Formal verification

1 Introduction

A configurable business process model [11,17] represents a family of a large number of related process models. Such a process model is reused and configured according to a given application context by selecting one design option for each configurable element (i.e. a *variation point*). The non-configurable elements represent the commonalities in the configurable model. The configuration decisions of a configurable element are made at design-time [17] leading to configured processes called *variants*. For instance, In Fig. 1, a simplified example of a configurable process model designed by a process provider for a hotel booking agency is presented. The process is modeled using the Configurable Business Process Model and Notation (C-BPMN) [5,14], a configurable extension to BPMN. The travel agency has a number of branches in different countries. Depending on specific needs of a country, each branch performs a different variant of this process model in terms of structure and behavior. For instance, a process tenant may need an exclusive execution of the connector *S1*'s outputs (configurable connectors are modeled with a thicker border). This refers to configuring *S1* to an *XOR-split*. Another tenant may choose to execute them concurrently by configuring *S1* to an *AND-split*.

As the configuration decisions of the configurable elements are applied at design-time [17], any design mistake (e.g. configuring *S1* to *OR-split* and *j3* to *AND-join* leads to a deadlock) should be avoided in order to avert execution errors in the derived variants. Furthermore, configurable processes may be large with complex inter-dependencies between the different possible configurations. Consequently, the configuration can not be done manually and a correctness verification phase is essential. So far, a number of approaches have addressed the verification of the process configuration correctness. Some of them have only discussed the syntactical correctness (e.g. [11,17]), others have attempted to verify behavioral correctness but have faced the exponential number of state-space problems (e.g. [13]). Very few have addressed the configuration behavior verification while trying to reduce state explosion problem (e.g. [1,4]) but still suffer from the exponential complexity of generating their reachability graph.

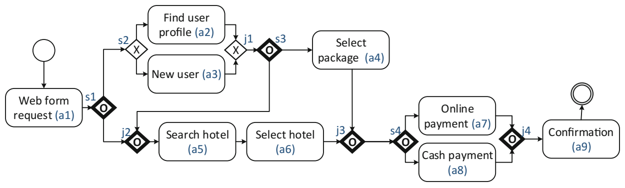


Fig. 1. A configurable hotel booking process model

The aim of this paper is to address this state space problem while verifying one of the most important behavioral correctness properties a process execution should hold, the deadlock-freeness. We propose an abstraction of a configurable process model using the Symbolic Observation Graph (SOG for short) [12,15] based on its configurable elements. This abstraction offers a two-fold advantage: (1) the analysis and the verification of the corresponding configurable process can be reduced to the analysis of its abstraction, and (2) the set of possible combinations of elements configurations that result in deadlock-free variants are obtained prior to configuration time. Once found, these combinations are used to assist the business analyst in deriving deadlock-free variants.

The SOG is a versatile symbolic representation formalism that allows to build an abstraction of the reachability state graph of a formally modeled system (e.g. using Petri net). In our case, this abstraction is achieved by observing the configurable elements of the process (that label the SOG arcs) and by hiding non configurable elements inside the aggregates (the SOG nodes). Moreover, without limiting the generality of our approach, we propose to use C-BPMN as input notation. BPMN is highly adopted by stakeholders of different roles (e.g. IT architects, business analysts, etc.) since it is considered as the internationally recognized industry standard notation for business process description. Also, since the large majority of modeling languages can be mapped into it, we use Petri-net as a pivot formalism to represent C-BPMN process model and its corresponding semantics. This semantics depicts the generic behavior of configurable connectors and thus all possible behavior.

Figure 2 depicts the milestones followed in order to obtain deadlock-free process variants using our SOG-based approach. First of all, as depicted on the left-hand side of the figure, C-BPMN is used as input process. Then, we map this process to a Petri net-based model; and we define new semantics to take into account configurable connectors (step 1, see Sect. 3). Afterward, we extend the algorithm of SOG graph construction by three main points (step 2): (i) by observing and highlighting configurable connectors in the graph arcs; (ii) by hiding non-configurable elements' states in aggregates (see Sect. 4.1); and (iii) by restricting the graph nodes to the ones leading to deadlock-free configurations (see Sect. 4.2). As a result, we obtain a reduced SOG graph that groups the behavior of all correct configurations. The set of correct configurations combinations is then extracted (step 3). The last three steps are performed on-the-fly during the SOG construction. The correct configurations are finally supplied to the business analyst in order to derive deadlock-free variants, with no need to verify correctness at each intermediate configuration step.

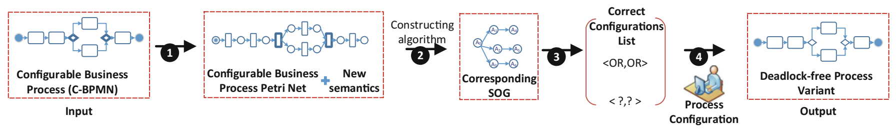

Configurable Business Process (C-BPMN)
Input
1
Configurable Business Process Petri Net
+
New semantics
Constructing algorithm
2
Corresponding SOG
3
Correct Configurations List
<OR,OR>
< ?,? >
4
Process Configuration
Deadlock-free Process Variant
Output

Fig. 2. Our approach overview

The remainder of the paper is organized as follows. In Sect. 2, some preliminary concepts on Petri nets are described. New Petri net-based models for business process and then for configurable process models as well as their semantics are defined in Sect. 3. Then, in Sect. 4.1, we define a new Symbolic Observation Graph associated with the configurable Petri net-based model and we explain our approach based on the SOG construction algorithm. Our approach is evaluated in Sect. 5. We present the related work in Section 6. Finally, we conclude and provide insights for future work.

2 Preliminaries and Notations

In this work, we use Petri nets, which offer a formal model for concurrent systems. Note that our approach does not rely on specific Petri net properties but can be applied to any formal model as soon as states and transition relation are well defined.

Definition 1 (Petri Nets). *A Petri net is a tuple $N = \langle P, T, F, W \rangle$ s.t.:*

- *P is a finite set of places and T a finite set of transitions with $(P \cup T) \neq \emptyset$ and $P \cap T = \emptyset$,*
- *A flow relation $F \subseteq (P \times T) \cup (T \times P)$,*
- *$W : F \to \mathbb{N}^+$ is a mapping assigning a positive weight to arcs.*

Each node $x \in P \cup T$ of the net has a pre-set and a post-set defined respectively as follows: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. For a transition t, $W^-(t) \in \mathbb{N}^{|P|}$ (resp. $W^+(t) \in \mathbb{N}^{|P|}$) denotes the vector where, $\forall p \in P$, $W^-(t)(p) = W(p, t)$ (resp. $W^+(t)(p) = W(t, p)$). A marking of a Petri net N is a function $m : P \to \mathbb{N}$.

Semantics: Let m be a marking of $t \in T$, a transition t is said to be enabled by m, denoted by $m \xrightarrow{t}$, iff $W^-(t) \leq m$. When t is enabled by m, its firing leads to a new marking m', denoted by $m \xrightarrow{t} m'$, s.t. $m' = m - W^-(t) + W^+(t)$.

For a finite sequence $\sigma = t_1 \ldots t_n$, $m_i \xrightarrow{\sigma} m_n$ denotes the fact that σ is enabled by m_i, and that its firing leads to m_n. Given a set of markings S, we denote by $Enable(S)$ the set of transitions enabled by elements of S. The set of markings reachable from a marking m in N is denoted by $R(N, m)$. The reachability graph of a Petri net N, denoted by $G(N, m_i)$ (m_i is the initial marking), is the graph where nodes are elements of $R(N, m_i)$ and an arc from m to m', labeled with t, exists iff $m \xrightarrow{t} m'$. The set of markings reachable from a marking m, by firing the transitions of a subset T' only is denoted by $Sat(m, T')$. By extension, given a set of markings S and a set of transitions T', $Sat(S, T') = \bigcup_{m \in S} Sat(m, T')$. For a marking m, $m \not\rightarrow$ denotes that m is a dead marking (i.e., there is no transition s.t. $m \xrightarrow{t}$ which means $Enable(\{m\}) = \emptyset$).

Definition 2 (WF-Nets). *Let $N = \langle P, T, F, W \rangle$ be a Petri net and F^* is the reflexive transitive closure of F. N is a Workflow net (WF-net) iff:*

- *there exists exactly one input place $i \in P$, s.t. $|{}^\bullet i| = 0$,*
- *there exists exactly one output place $o \in P$, s.t. $|o^\bullet| = 0$,*
- *each node is on a directed path from the input place to the output place, i.e. $\forall n \in P \cup T, (i, n) \in F^* and (n, o) \in F^*$.*

Definition 3 (Deadlock-free WF-Net). *Let $N = \langle P, T, F, W \rangle$ be a WF-net and m_i, m_f be the initial (i.e. only i is marked) and final (i.e. only o is marked) markings respectively. N is said to be deadlock-free iff $\nexists m \in (R(N, m_i) \setminus \{m_f\})$ s.t. $m \not\rightarrow$.*

3 Formal Model for Configurable Business Processes

In order to obtain an abstract formal definition of a business process model, we formally map a process in BPMN notation to *Petri nets*, specifically into a new model called Business Process Petri Nets (*BP2PN*). Then, we extend the

BP2PN to take into account configurable connectors, leading to a new model, namely the Configurable Business Process Petri Nets (*CBP2PN*). Authors in [10] have established a mapping from well-formed BPMN models to Petri nets. In this work, we extend this mapping by preserving blocks as transitions allowing to define configurable transitions.

3.1 Business Process Petri Nets (BP2PN)

Definition 4 (BP2PN). *A BP2PN is a tuple* $B = \langle P, T \cup OP, F, W, O\rangle$ *where:*

- $\langle P, T \cup OP, F, W\rangle$ *is a WF-Net,*
- $F \subseteq (P \times T \cup OP) \cup (T \cup OP \times P)$ *is the flow relation,*
- $O : OP \rightarrow \{OR^-, OR^+, XOR^-, XOR^+, AND^-, AND^+\}$ *is a mapping that assigns a type to each operator,*

BP2PN is a *Workflow net* such that, the set of places P corresponds to the set of conditions determining the enabling of a task or a connector; and the set of transitions $T \cup OP$ corresponds to the set of tasks and connectors. These nodes are interconnected through a set of arcs (using F). Each connector must either be a join (the $-$ right exponent) or a split (the $+$ exponent) while having a type: OR, XOR or AND.

Semantics: In the previous notation, we retain the connectors blocks and we define new execution semantics inspired from the original semantics of Petri nets.

Given a marking m of a *BP2PN* B, the fireability and the firing of any transition in $T \cup \{t \in OP \mid O(t) \in \{AND^-, AND^+\}\}$ follows the original semantics of Petri nets. However, transition t s.t. $O(t) \in \{OR^-, OR^+, XOR^-, XOR^+\}$ follows a new semantics:

Let m be a marking and t be a transition of OP, the fact that t is enabled by m is denoted by $m\xrightarrow{t}$, and $m\xrightarrow{t}m'$ denotes that m' is reached by firing t from m:

- $O(t) = OR^-$
  - m enables t iff $\exists S \subseteq {}^\bullet t$ s.t. $m_{|S} \geq W^-(t)_{|S}$
  - when m enables t, the firing of t from m leads to a marking m' iff $m' = m - W^-(t)_{|S} + W^+(t)$ where S is the biggest subset of ${}^\bullet t$ satisfying $m_{|S} \geq W^-(t)_{|S}$.
- $O(t) = XOR^-$
  - m enables t iff $\exists p \in {}^\bullet t$ s.t. $m(p) \geq W^-(t)(p) \wedge \forall q \in {}^\bullet t$, $m(q) < W^-(t)(q)$
  - when m enables t, the firing of t from m leads to a marking m' iff $m' = m - W^-(t))_{|\{p\}} + W^+(t)$ where p is the sole place satisfying the firability condition.
- $O(t) = OR^+$ **(resp.** $O(t) = XOR^+$**)**
  - when m enables t, the firing of t from m leads to a marking m' iff $\exists S \subseteq t^\bullet$ **(resp.** $\exists p \in t^\bullet$**)** s.t. $m' = m - W^-(t) + W^+(t)_{|S}$ **(resp.** $m' = m - W^-(t) + W^+(t)_{|\{p\}}$**)**.

Note that only the firing of transitions t s.t. $O(t) \in \{OR^+, XOR^+\}$ is defined because the fireability follows original semantics. It is worth mentioning that the previous semantics of OR^+ and XOR^+ leads to non-deterministic firing. For instance, having a split transition OR^+ with 2 output places p_1 and p_2, its firing leads to 3 possible reachable markings m_1 (only p_1 is marked), m_2 (only p_2 is marked), and m_{1_2} (both places are marked). Also, we emphasize that the semantics of a join transition OR^- is inline with the well defined pattern 8 in [3] (*Multi merge*), that expressly allows the firing of the join as soon as it condition is satisfied (without synchronizing the different flows).

3.2 Configurable Business Process Petri Nets (CBP2PN)

Definition 5 (CBP2PN). *A CBP2PN is a tuple* $CB = \langle P, T \cup OP, F, W, O, C\rangle$ *where:*

- $\langle P, T \cup OP, F, W, O\rangle$ *is a BP2PN;*
- $C : OP \rightarrow \{true, false\}$ *is a function determining the configurable operators (i.e. any* $t \in OP$ *s.t.* $C(t) = true$*).*

Back to our example, our *C-BPMN* process is mapped onto *CBP2PN* in Fig. 3. In this notation, according to Definition 5, activities and connectors are modeled by transitions and their ordering is modeled by places connecting the different transitions. Configurable transitions are also highlighted with a thick border. This example includes 6 configurable transitions: s_1, s_3, s_4, j_2, j_3 and j_4. We denote by OP^c the set of configurable operators s.t. $OP^c = \{o \in OP \mid C(o) = true\}$. A configurable operator $c^c \in OP^c$ includes a generic behavior which is restricted using the configuration phase. It is configured by changing its type (e.g. from OR to AND) w.r.t. the set of configuration constraints [17] defined in Table 1. Each row corresponds to a configurable connector that can be configured to one or more of the connectors in columns. Thus, these constraints allow to specify which regular connector's type may be used in the derived process variant. For example, a configurable *OR* can be configured to any connector's type while a configurable *AND* can only be configured to an *AND*. In the following, we define a configuration of a connector $t^c \in OP^c$ by $Conf(t^c) \in \{OR^-, OR^+, XOR^-, XOR^+, AND^-, AND^+\}$ and the set of all possible configurations of t^c by $AllConf(t^c)$.

Note that, when configuring all configurable connectors of a *CBP2PN*, we obtain a *BP2PN*, as a configurable connector is changed into regular connector after configuration. One possible configuration of the process net of Fig. 3 can be done by selecting the following configuration choices: (i) s_1, s_3 and s_4 are configured to regular XOR^+, (ii) j_2 is configured to a regular AND^-; and (iii) j_3 and j_4 are configured to regular XOR^-.

Table 1. Constraints for the configuration of connectors [17], $x \in \{+, -\}$.

	OR^x	XOR^x	AND^x
OR^x	✓	✓	✓
XOR^x		✓	
AND^x			✓

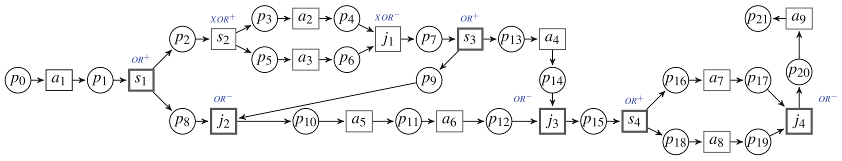


Fig. 3. The CBP2PN of the configurable process in Fig. 1

Semantics: The semantics of *CBP2PN* is described in the following, on the one hand, by inheriting the dynamics of *BP2PN* for non configurable connectors, on the other hand, by adding new semantics for configurable ones. This semantics is defined such that any reachable marking by any possible instance of a configuration is represented. In the following, we consider a configurable transition as the union of all possible configurations. That way, we can define its enabling and firing rules as if it is the union of all executable configured transitions. Since a configuration of AND^-, AND^+, XOR^- and XOR^+ do not change type, its semantics remains the same as previously defined. Regarding configurable OR^- and OR^+ transitions, the fireability and the firing rules follow the new semantics as follows. Let m be a marking and t^c be a transition of OP^c, s.t. $O(t^c) \in \{OR^-, OR^+\}$:

- m enables t^c, denoted by $m\xrightarrow{t^c}$ iff $\exists x \in AllConf(t^c)$ s.t. $m\xrightarrow{x}$
- when m enables t^c, for some configuration x of t^c, the firing of t^c from m, under configuration x, leads to a marking m', denoted by $m\xrightarrow{t^c,x}m'$ iff $m\xrightarrow{x}m'$

Using this semantics, the reachability marking graph associated with a *CBP2PN* covers the behavior of all the possible configurations. For instance, having the *CBP2PN* of Fig. 3, the configurable transition $s1$ could be configured either to: (i) AND^+, with all of its output places marked, (ii) XOR^+, with only one of the output places marked, or (iii) OR^+ with one or more output places marked.

Definition 6 (Deadlock-free CBP2PN). *Let* $\underline{CB}$ *be a CBP2PN. CB is said to be deadlock-free if at least one deadlock-free BP2PN could be configured from CB.*

Our *CBP2PN* of Fig. 3 is considered correct since one can configure at least one correct variant by choosing XOR type as configuration choice for all its configurable connectors (the correctness of such a variant is proven in Sect. 4.2). However, incorrect variants could be derived from this process as well. For instance, one can choose the alternatives presented earlier that leads to a deadlock caused by an exclusive choice XOR^+ (i.e. s_1) followed by a synchronizing join AND^- (i.e. j_2). In this situation, in order to be enabled, the transition AND^- will be waiting for both places p_8 and p_9 to be marked, however only one could be marked. So, the resulting variant could never terminate properly and the corresponding reachability graph contains a dead marking. In the following, we propose to use the SOG in order to abstract the reachability graph of a *CBP2PN*, and to extract the correct configurations (leading to deadlock-free *BP2PN*).

4 Symbolic Observation Graph for Process Configuration

In this paper, we check the behavior correctness of all possible configurations of a configurable model. This refers to verifying the reachability graph that covers them all. In order to reduce the underlying state space explosion problem, we propose to use the Symbolic Observation Graph (SOG). The SOG-based abstraction technique was introduced for model checking of concurrent systems [12] and then applied on the verification of inter-enterprise business processes [15].

4.1 Symbolic Observation Graph

Given a *CBP2PN*, the set of observed transitions, denoted by *Obs* is the set of configurable connectors i.e. $Obs = OP^c$, while any other transition belongs to the set of unobserved transitions, denoted by *UnObs*, i.e., $UnObs = (T \cup OP) \setminus Obs$. In such a way, we construct the Symbolic Observation Graph (SOG) as a graph where each node is a set of states linked by unobserved transitions and each arc is labeled by an observed transition. Nodes of the SOG are called aggregates and are represented and managed efficiently using Binary Decision Diagrams (BDDs). As a result, by highlighting observable transitions, the SOG represents the global behavior of a process configuration in only one reduced graph. In the following, we first formally define *an aggregate*, and then the SOG associated with a *CBP2PN*.

Definition 7 (Aggregate). *Let* $N = \langle P, T \cup OP, F, W, O, C\rangle$ *be a CBP2PN having* m_i *and* m_f *as initial and final markings respectively. An aggregate of* N *is a triplet* $\langle S, d, f\rangle$ *s.t.:*

- $S \subseteq R(N, m_i)$ *is a set of reachable markings, where* $\forall s \in S$*:*
  - $(\exists (s', u) \in R(N, m_i) \times UnObs \mid s \xrightarrow{u} s') \Leftrightarrow s' \in S$;
  - $(\exists (s', o) \in R(N, m_i) \times Obs \mid s \xrightarrow{o} s') \wedge (\nexists (s'', u) \in S \times UnObs) \mid s'' \xrightarrow{u} s') \Leftrightarrow s' \notin S$.
- $d \in \{true, false\}$; $d = true$ *iff* S *contains a dead state.*
- $f \in \{true, false\}$; $f = true$ *iff* S *contains a final state.*

In addition to the d and f attributes of an aggregate, the above definition specifies the states that must belong to an aggregate (the aggregation criterium) and those that must be excluded: For any state s in the aggregate, any state s' being reachable from s by the occurrence of an unobserved transition, belongs necessarily to the same aggregate. (2) For any state s in the aggregate, any state s' which is reachable from s by the occurrence of an observed transition is necessarily outside the aggregate, unless s' is reachable from a state s' in the aggregate by an unobserved transition.

Before defining the SOG, let us introduce the following operation: $Out(a,t)$: returns, for an aggregate a and an observed transition t, the set of states outside a that are reachable from some state in a by firing t, i.e., $Out(a,t) = \{s' \in R(N, m_i) \mid \exists s \in a.S, s \xrightarrow{t} s'\}$

Definition 8 (Deterministic SOG). *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN having m_i and m_f as initial and final markings respectively. The Deterministic Symbolic Observation Graph (SOG) associated with N is a graph $\mathscr{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ where:*

(1) A is a non empty finite set of aggregates satisfying :
  - *$\forall a \in A,\ \forall t \in Obs,\ Out(a,t) \neq \emptyset \implies \exists a' \in A$ s.t. $a' = Sat(Out(a,t), UnObs)$*

(2) $\rightarrow \subseteq A \times Act \times A$ is the transition relation where:
  - *$((a,t,a') \in \rightarrow') \Leftrightarrow ((t \in Obs) \wedge Out(a,t) \neq \emptyset \wedge a' = Sat(Out(a,t), UnObs))$*

(3) A_0 is the initial aggregate s.t. $A_0.S = Sat(m_i, UnObs)$.

(4) $\Omega = \{a \in \mathscr{A} \mid m_f \in a.S\}$.

The nodes of the symbolic observation graph are aggregates (1). The finite set of aggregates A of a SOG is defined in a complete manner so that the necessary aggregates are represented. Point (2) defines the transitions relation: there exists an arc, labeled with an observed transition t, from a to a' iff a' is obtained by saturation on the set of reached states ($Out(a,t)$) by the firing of t from $a.S$. The last two points of Definition 8 characterize the initial aggregate and the set of final aggregates respectively. Starting from the initial marking, the original SOG construction algorithm introduced in [12] follows a classical depth first search based traversal of the built aggregates. Each aggregate is built by a transitive closure application on unobserved transitions. The successor a' of an aggregate a is built by, first, firing an observed transition from states of a, then by adding all the reachable states by unobserved transition.

At this stage, the correctness of the *SOG* can be characterized as follows.

Definition 9 (Correct SOG). *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN. Let $\mathscr{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ the SOG associated with N.*

*$\mathscr{G}$ is correct **iff** there exists a configuration c of N ($c = \{\langle t, Conf(t)\rangle : t \in OP^c\}$) s.t. for every path $\pi = A_0 \xrightarrow{t_1, conf(t_1)} A_1 \ldots A_{n-1} \xrightarrow{t_n, conf(t_n)} A_n$, with $A_n \in \Omega$; if $\{\langle t_i, Conf(t_i)\rangle : 0 \leq i \leq n\}$ $=c$ then $\forall 0 \leq i \leq n, A_i.d = false$.*

Based on Definition 6, characterizing a deadlock-free *CBP2PN*, and Definition 9, characterizing a correct *SOG* associated with a *CBP2PN*, the following result naturally links these two characterizations.

Proposition 1. *Let $N = \langle P, T \cup OP, F, W, O, C\rangle$ be a CBP2PN. Let $\mathscr{G} = \langle A, Obs, \rightarrow, A_0, \Omega\rangle$ the SOG associated with N. Then, N is deadlock-free **iff** $\mathscr{G}$ is correct.*

Proof. Let N be a *CBP2PN* and $\mathscr{G}$ its corresponding *SOG*. First, according to Definition 9, if $\mathscr{G}$ is correct then there exists a configuration c s.t. for every path π in the *SOG* having $\pi = A_0 \xrightarrow{t_1, conf(t_1)} A_1 \ldots A_{n-1} \xrightarrow{t_n, conf(t_n)} A_n$, with $A_n \in \Omega$; if it's configurations set $\{\langle t_i, Conf(t_i)\rangle : 0 \leq i \leq n\}$ is equal to c, then all aggregates are deadlock-free, i.e. $A_i.d = false, 0 \leq i \leq n$. Since the *SOG* preserves by construction all possible configurations of N, then each path from the initial to the final aggregate represents one configuration allowing to derive one variant. Hence, there exist at least a deadlock-free variant of N. Consequently, according to Definition 6, N is correct.

In the following, we propose to adapt the original SOG construction algorithm [12], associated with a *CBP2PN*, in three ways. First, by adopting the new semantics. Second, the deadlock-freeness property is checked on the fly, such that any aggregate containing a deadlock state is not inserted in the graph and so are all the underlying paths. Finally, the set of correct configurations is extracted on-the-fly.

4.2 Extracting Correct Configurations Using the SOG

In this section, we present the core contribution of this paper: A construction algorithm of the SOG associated with a *CBP2PN*. Regarding to the original SOG construction algorithm [12], Algorithm 1 allows to reduce the SOG, by removing, on-the-fly, the paths involved in incorrect configurations, and by saving, within the initial aggregate the correct configurations. To reach this goal, two new attributes are added to an aggregate: (1) c, which is the set of correct (possibly partial) configurations, starting from this aggregate (and leading to a final aggregate). (2) nc, which is the set of incorrect (possibly partial) configurations, starting from this aggregate (leading to a dead one). Once the *SOG* is built, the set of correct configurations will be saved within the initial aggregate.

In the following, we go through Algorithm 1 to explain the main steps while using our running example, and the corresponding (reduced) SOG, in Fig. 4a for illustration. Note that the main novelties of this algorithm w.r.t. the algorithm of [12], are underlined.

Two main data are used: The SOG graph $\mathscr{G}$, containing aggregates and edges, and a stack containing the to-be-treated aggregates associated with the set of fireable observed transitions F_{obs}.

The first step of Algorithm 1 (lines 5–10) allows to build the initial aggregate and to push it onto the stack. Then, the main loop (lines 11–49) processes the set of to-be-treated aggregates as follows: a stack item (line 12) and the corresponding current observed transition in F_{obs} (line 14) are picked, and the successor of the current aggregate by that transition, if any, is calculated using the semantics of Subsect. 3.2 (line 15–20). This includes the computation of the dead (line 19) and final (line 20) attributes of the obtained successor aggregate. If the latter is deadlock-free aggregate, and if it has not already been explored, then it is pushed onto the stack with its set if fireable observed transitions (lines 21–24). For instance, following the path at the top of Fig. 4a, new aggregates A_0 until the final one A_6 are consecutively pushed onto the stack. Since A_6 is a final aggregate (does not enable any observable transition), it will be popped from the stack (line 38), and we start the loop again by picking A_5 to consider its remaining observed transitions (in this case the transition $\langle j4, OR\rangle$ leads again to A_6), and so on.

If the newly built successor aggregate a' has already been treated (lines 25–30), then the current aggregate a inherits from a' its correct and incorrect configuration (to which the transition linking a to a' is added). This is ensured by functions *UpdateC* and *UpdateNC* (lines 26–29). The function *UpdateC* also verifies that, starting from the same aggregate a, a correct configuration do not include an existing (or to-be-treated) incorrect one, as in this case it leads to a deadlock in a different transitions' firing order. This way, correct and incorrect configurations are computed backwards starting from the final aggregate to the initial one. For instance, in Fig. 4a, consider the aggregate A_{10} obtained through A_8 and A_9, the firing of $\langle j3, AND\rangle$ leads to the existing aggregate A_4. As A_4 was already dealt with earlier through the path on top of the graph, this means that 3 correct *partial* configurations are added to this aggregate, namely $\{\langle s4, XOR\rangle,\langle j4, XOR\rangle\}$, $\{\langle s4, XOR\rangle, \langle j4, OR\rangle\}$ and $\{\langle s4, AND\rangle,\langle j4, AND\rangle\}$. Hence, A_{10} inherits these configurations while being concatenated to the current fired transition $\langle j3, AND\rangle$. Similarly, going backwards to A_0 after entirely processing A_8 and A_9, we obtain the complete correct configurations $13 - 15$ depicted in Fig. 3(b).

Regarding an aggregate a' holding a dead state, firstly, the corresponding fired observed transition is concatenated to the incorrect configurations of its predecessor a (line 33). Obviously, a' is not pushed onto the stack and no edge is created. Then, we recursively verify its predecessors starting from a using the function *recRemoveAggregate(*a, t*)* (line 34). Using this function, each predecessor aggregate is removed only if the states enabling the current one becomes dead (i.e. there is no other enabled transition from that state). In this case, its successors are also recursively eliminated in case they do not have other predecessors. As an example, the red path in Fig. 4(a) refers to firing $\langle s_1, AND\rangle$, $\langle s_3, XOR\rangle$ then $\langle j_2, OR\rangle$. According to our semantics, $\langle j_3, OR\rangle$ may be fired by 4 possible markings in the aggregate A_{12}, namely m_{12}, m_{10_14}, m_{11_14} and m_{12_14}. However, in case of firing by either m_{10_14} or m_{11_14}, the obtained aggregate will allow a second firing of the same transition (i.e. using the remaining token in p_{10} or p_{11}). This leads to a final state holding two tokens, which is a dead state in our approach. Hence, according to Algorithm 1 the obtained aggregate is eliminated as well as its predecessors A_{12} and A_{11} (following the blue dashed line). And yet, since it enables $\langle S_3, AND\rangle$, A_{10} is not deleted.

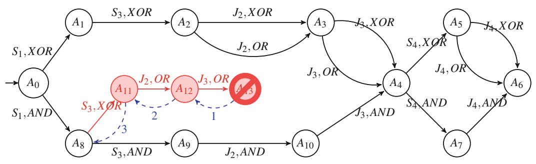
It is worth noting that before popping an aggregate from the stack and storing it in the graph (lines 38–39), a final check is carried out on its correct configurations by the function *CompareCorrect* (line 37). Actually, many observed transitions may be fired from the same aggregate, so some of the corresponding correct configurations may refer to the same one. Hence, a correct sequence is preserved if, for every first fired observed transition *op*, (i) it is fireable by

nofillcomment 1. Deadlock-free Symbolic Observation Graph

```
Require: $N\langle P, T \cup OP, F, W, O, C\rangle$, $Obs$, $m_i$, $m_f$
Ensure: $\mathscr{G}\langle A, Obs, \rightarrow, A_0, \Omega\rangle$ , $C$
Vertices $A$=$\emptyset$; vertex $a, a'$;                              # Aggregates
Vertices $C$=$\emptyset$;                                              # Correct configurations
set $S, S'$, $UnObs = (T \cup OP) \setminus Obs$, $F_{obs}$, $F'_{obs}$;
stack st; Edges E= $\emptyset$;
$S = Sat(\{m_i\}, UnObs)$;                                             # first Aggregate
$a.S = S$;
$a.d = DetectDead(a.S)$;
$a.f = IsFinal(a)$;
$F_{obs} = fireableObs(a)$;                                            # fireable observed transitions of a
$st.Push(\langle a, F_{obs}\rangle)$;
while $st == \emptyset$ do
    $\langle a, F_{obs}\rangle = st.Top()$;
    if $(F_{obs} \neq \emptyset)$ then
        $t = F_{obs}.next()$;
        $S' = Out(a.S, t)$
        if $(S' \neq \emptyset)$ then
            $S' = Sat(S', UnObs)$;
            $a'.S = S'$;
            $a'.d = DetectDead(a'.S)$;
            $a'.f = IsFinal(a')$;
            if $\underline{(\neg a'.d)}$ then                          # there is no dead state in a'
                if $(\nexists x \in A$ s.t. $x == a')$ then            # a' found for the first time
                    $F'_{obs} = fireableObs(a')$;
                    $st.Push(\langle a', F'_{obs}\rangle)$;
                else                                                   # a' is an existing aggregate
                    free $a'$;
                    Let $a'$ be the already existing aggregate;
                    $\underline{UpdateC(a, a', t)}$;
                    $\underline{UpdateNC(a, a', t)}$;
                end if
                E = E $\cup\{a, \langle t, Conf(t)\rangle, a'\}$;
            else                                                       # there is a dead state in a'
                $\underline{a.nc = a.nc \cup \{\langle t, Conf(t)\rangle\}}$;
                $\underline{recRemoveAggregate(a, t)}$
            end if
        end if
        $\underline{CompareCorrect(a)}$;
        $st.Pop()$;
        $\underline{A = A \cup \{a\}}$ ;
        if $\underline{(m_i \in a.S)}$ then
            $\underline{C = a.c}$;
        end if
    end if
end while
```

the states that have fired another sequence starting by *op* (i.e. different configurations), or (ii) if their common operators have the same configured type (i.e. the same configurations but in a different order). Otherwise, the sequence is considered as incorrect and is eliminated.

Finally, the set of correct configurations is obtained from the initial aggregate, the last one popped from the stack. As a result, each path of the obtained SOG starting from the initial aggregate and leading to a final aggregate, represents one possible configuration and belongs to the set of configurations C. In this case, this configuration leads to a deadlock-free *BP2PN*. Note that, different paths could represent a configuration (e.g. two concurrent configurable connectors).



(a) The reduced SOG

	S1	*S2*	*J2*	*J3*	*S4*	*J3*
1	XOR^+	XOR^+	XOR^-	XOR^-	XOR^+	XOR^+
2	XOR^+	XOR^+	XOR^-	XOR^-	XOR^+	OR^+
3	XOR^+	XOR^+	XOR^-	XOR^-	AND^+	AND^+
4	XOR^+	XOR^+	XOR^-	OR^-	XOR^+	XOR^+
5	XOR^+	XOR^+	XOR^-	OR^-	XOR^+	OR^+
6	XOR^+	XOR^+	XOR^-	OR^-	AND^+	AND^+
7	XOR^+	XOR^+	OR^-	XOR^-	XOR^+	XOR^+
8	XOR^+	XOR^+	OR^-	XOR^-	XOR^+	OR^+
9	XOR^+	XOR^+	OR^-	XOR^-	AND^+	AND^+
10	XOR^+	XOR^+	OR^-	OR^-	XOR^+	XOR^+
11	XOR^+	XOR^+	OR^-	OR^-	XOR^+	OR^+
12	XOR^+	XOR^+	OR^-	OR^-	AND^+	AND^+
13	AND^+	AND^+	AND^-	AND^-	XOR^+	XOR^+
14	AND^+	AND^+	AND^-	AND^-	XOR^+	OR^+
15	AND^+	AND^+	AND^-	AND^-	AND^-	AND^-

(b) Deadlock-free configurations

Fig. 4. Reduced SOG and extracted configurations for the CBP2PN in Fig. 3

The reduced SOG of our example contains 8 nodes and 10 arcs, and all correct configurations are summarized in Fig. 3(b). Hence, the analyst may be helped on-the-fly during the configuration process by confronting his/her configurations with the correct configurations in this table.

For instance, we can evaluate the correctness of the *BP2PN* variant discussed in Sect. 3.2. After applying $\langle s_1, XOR \rangle$, the control-flow is either propagated through the place p_2 or p_8. In this case, it is clear that the connector j_2 (i.e. after applying $\langle j_2, AND \rangle$) could never be enabled, which causes a deadlock. Relying on Fig. 3(b), we can notice that there is no configuration starting with $\{\langle s_1, XOR \rangle, \langle j_2, AND \rangle\}$.

Using the SOG, the state space is greatly reduced in three fashions: (i) only configurable transitions are observed, and the remaining transitions are hided in aggregates; (ii) the graph is deterministic since it groups, for each configuration, all reachable markings in one aggregate; and (iii) the different process variants share common markings in one common SOG graph, instead of constructing graphs as much as the number of possible configurations. In the following section, we conduct experiments to demonstrate such mitigation of the state explosion problem as well as the feasibility of our approach.

5 Experiments and Evaluation

To prove its feasibility, we have implemented and deployed our approach as an extension of an existing tool that initially computes the SOG of a petri-net model w.r.t. a set of observed transitions. As explained previously, this extension takes into account the new semantics presented in this paper for *CBP2PN* models. It also allows to symbolically detect on-the-fly deadlocks within aggregates and to reduce the SOG accordingly. The developed tool takes as input a GrML (Graph Markup Language) file [8] describing the *CBP2PN* model (i.e. transitions, operators annotated as configurable, and arcs) and returns the reduced SOG and the correct configurations.

In order to evaluate its performances and to demonstrate the opportunities offered by our approach, we performed experiments to show (i) the reduction of the space explosion problem and (ii) the impact of the input model structure on the size of the obtained SOG. Firstly, we propose to explore the size of the constructed SOG using our tool against a naive approach, where each variant of a *CBP2PN* is built and analyzed separately. Secondly, we propose to analyse the impact of the variation of the structure complexity and the number of observed transitions of a *CBP2PN*, on the size of the corresponding SOG. Taking our running example model (Fig. 3), this variation leads to 86 different process models. We basically evaluate the structure complexity using the well known metric CFC (Control Flow Complexity)[9] which is defined as: $\sum_{c \in AND^+} 1 + \sum_{c \in XOR^+} |c^\bullet| + \sum_{c \in OR^+} (2^{|^\bullet c|} - 1)$.

Table 2 contains three multi-columns. The first one varies the considered parameters of the *CBP2PN* model (i.e. CFC and observed transitions (Obs)) and gives the number of possible configurations for each variation. Then, the size of the obtained SOG is evaluated in terms of number of correct configurations (Nb correct confs), aggregates (A), edges (E) and execution time. This graph is finally compared against the naive approach. However, since the naive approach is very fastidious, we built only the reachability graphs corresponding to the correct configurations. The three first columns give the average number if states, arcs and execution time over these correct configurations. The last column, gives the worst execution time in case all the configuration have been analyzed to extract correct ones. The construction of the reachability graph has been performed with our SOG-based tool as well, by observing all the transitions of the model (in this case, the SOG coincides with the reachability graph).

In this evaluation, as we can observe from the Table 2, we took into account three levels of complexity (depending on the number of OR+). The higher the value of CFC, the more complex is a process's configuration, since the number of

Table 2. Checking deadlock-freeness on SOG vs RG

CBP2PN			*SOG*				*Naive approach (RG)*			
CFC (avg)	Obs	Nb possible confs(avg)	Nb correct confs(avg)	A(avg)	E(avg)	Exec time (sec)	Sates (sum)	Arcs (sum)	Exec time correct (sec)	Overall Exec time(sec)
21	6	729	15	13	26	1.580	283.50	331.95	0.051	2.478
	5	243	5.66	8.66	16	0.693	104.14	133.57	0.017	0.729
$3OR_+$	4	81	2.33	5.66	8.66	0.353	42.17	49.62	0.007	0.243
	3	27	1	4	4	0.044	18	21	0.003	0.070
15.5	6	243	11.33	11	21	0.093	208.47	243.25	0.037	0.802
	5	81	5	7.77	13.77	0.051	93	106.30	0.017	0.267
$2OR_+$	4	57.85	3.66	6.09	10.33	0.030	66.72	77.81	0.012	0.191
	3	22.50	2	4.33	5.83	0.018	36.20	42.20	0.006	0.068
10	6	81	8	9.50	17.50	0.015	144	168	0.024	0.243
	5	54	4	7	11.83	0.010	72	84	0.014	0.184
$1OR_+$	4	18	4.25	5.75	9.87	0.008	76.71	89.46	0.014	0.058
	3	13.24	2.58	4.23	6.29	0.006	46.44	54.18	0.008	0.040

possible configurations increases with the number of configurable OR connectors. For example, the CFC value 21 regards the process with only OR connectors, we can observe that the number of possible configurations as well as the extracted correct ones are relatively high compared to those having CFC 10. Moreover, the more transitions are observed, the less reduced is the SOG comparing to the reachability graph.

Comparing to the naive approach, the obtained results in Table 2 show that the SOG is always significantly smaller in terms of number of states and arcs. For example, in case of a model having 6 configurable operators with OR type (i.e. the first row), we can observe that the obtained SOG includes only 13 aggregates and 26 arcs which is very reduced comparing to the size of the original graph of 729 possible configurations. Indeed, after applying a naive approach on only correct configurations (i.e. extracted from the SOG), the obtained graph has almost 283 states and 331 arcs resulting from the sum of 15 reachability graphs. Consequently, our work not only helps finding correct configurations but also further minimize the memory usage and the computing time, since only one reduced graph is constructed. To ensure the reproducibility of our experiments, please refer to our web page[1].

6 Related Work

In order to facilitate the design of configurable process models, a range of process modeling languages have been recently extended with variable elements such as Event-driven Process Chain (EPC) (e.g. [17,18]), Business Process Model Notation (BPMN) (e.g. [5,14]) and Yet Another Workflow Language (YAWL) (e.g. [11]). Based on some of them, a number of approaches have attempted to reach correct process configuration either syntactically [11,17] or behaviorally. Traditionally, behavioral correctness related to process configuration can be handled by verifying every single possible configuration using existing work on verification of business processes and workflows [2] and some existing tools such as Woflan [19]. However, these methods are too time-consuming and lead to the state space explosion problem. Authors in [13] discuss the Provop approach [14] for ensuring soundness of process variants derived by options. However, this approach is not feasible in large processes and runs into the state space problem. In [1], Petri net was used to formalize and verify correctness and soundness properties of Configurable EPC (C-EPC) processes. They derive propositional logic constraints that guarantee the behavioral correctness of the configured model. However, in these approaches authors achieve correctness by checking constraints at each configuration step. Also, authors impose that the C-EPC process model should be syntactically correct. In our work, we propose a model that finds all possible correct configurations at design time instead of configuration time without any restriction on the input C-BPMN process. This allows the process analyst to derive correct processes without intermediate computing. In [4], based on partner synthesis, the approach characterize all weakly terminating configurations

[1] http://www-inf.it-sudparis.eu/SIMBAD/tools/SOGImplementation.

using configuration guidelines. This technique was applied on C-YAWL and the configuration is built by hiding and blocking transitions while our approach configures C-BPMN process by changing configurable connectors behavior.

[16], which is applied on C-EPC using questionnaire models, and [6], which is applied on C-BPMN using configuration guidelines, have attempted to provide guidance to analysts for process configuration, however, these approaches especially ensure domain compliant variant and they do not consider any correctness criterion.

In our previous work [7], a formal approach for deriving correct process variants from a C-BPMN was proposed. It models the process using Event-B language and verifies the different constraints and properties using predicates. These predicates must be satisfied by each configuration step. This work contributes essentially to prevent structural correctness issues in process models configuration using a systematic design. However, structural correctness may not be sufficient. To the best of our knowledge, our previous work is the first one attempting to achieve correctness for specifically C-BPMN configurations. In the current work, we aim to especially achieve the behavioral correctness capturing the dynamics of the executable configured process model. Thus, for all possible instances of an executable configured process model, deadlocks should never occur. Our approach can be easily adapted to obtain *sound* [2] process variants, due to the lack of space, we focus in this work on the deadlock-freeness property.

7 Conclusion and Further Work

In this work, we propose an approach to assist business analyst to configure configurable processes correctly. In this paper, the correction criterion is characterized by the deadlock freeness of the obtained variant. We use a SOG-based abstraction model to find all correct configurations, i.e. leading to deadlock-free process variants. Such anomalies are excluded on-the-fly during the construction of the SOG. As a result, we obtain a reduced graph as well as a set of correct configurations. Then, this set will serve to support analysts during configuration. Our approach was implemented as an extension to an existing tool. And preliminary experiments show that our approach outperform naive approaches in terms of size of the explored configurable model.

As future work, we plan to first take into account other types of process configurations such as, activity and resource configuration as well as other patterns of OR-join, i.e. *Synchronizing merge* and *Discriminator* [3]. Then, we aim to entirely automate our approach procedure (depicted by Fig. 2). Finally, we aim to adapt the SOG construction algorithm in order to integrate other correctness constraints: generic properties, e.g. *soundness*, and specific properties, e.g. domain constraints.

References

1. Aalst, W.V.D., et al.: Preserving correctness during business process model configuration. Formal Asp. Comput. **22**(3–4), 459–482 (2008)
2. Aalst, W.V.D., et al.: Soundness of workflow nets: classification, decidability, and analysis. Formal Asp. Comput. **23**(3), 333–363 (2011)
3. Aalst, W.V.D., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14**(1), 5–51 (2003)
4. Aalst, W.V.D., Lohmann, N., Rosa, M.L.: Ensuring correctness during process configuration via partner synthesis. Inf. Syst. **37**(6), 574–592 (2012)
5. Assy, N.: Automated support of the variability in configurable process models. Ph.D. thesis, University of Paris-Saclay, France (2015)
6. Assy, N., Gaaloul, W.: Extracting Configuration Guidance Models from Business Process Repositories. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 198–206. Springer, Cham (2015). doi:10.1007/978-3-319-23063-4_14
7. Boubaker, S., et al.: A formal guidance approach for correct process configuration. In: Service-Oriented Computing - 14th International Conference, pp. 483–498 (2016)
8. Brandes, U., et al.: GraphML Progress Report Structural Layer Proposal, pp. 501–512 (2002)
9. Cardoso, J.S.: Business process control-flow complexity: Metric, evaluation, and validation. Int. J. Web Serv. Res. **5**(2), 49–76 (2008)
10. Dijkman, R., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008)
11. Gottschalk, F., et al.: Configurable workflow models. Int. J. Coop. Inf. Syst. **17**(02), 177–221 (2008)
12. Haddad, S., Ilié, J.-M., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 196–210. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30476-0_19
13. Hallerbach, A., et al.: Guaranteeing soundness of configurable process variants in provop. In: IEEE Conference on Commerce and Enterprise Computing, CEC, pp. 98–105 (2009)
14. Hallerbach, A., et al.: Capturing variability in business process models: the provop approach. J. Softw. Maintenance **22**(6–7), 519–546 (2010)
15. Klai, K., Tata, S., Desel, J.: Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes. Data Knowl. Eng. **70**(5), 467–482 (2011)
16. La Rosa, M., et al.: Questionnaire-based variability modeling for system configuration. Softw. Syst. Model. **8**(2), 251–274 (2008)
17. Rosemann, M., Aalst, W.V.D.: A configurable reference modelling language. Inf. Syst. **32**(1), 1–23 (2007)
18. Van Der Aalst, W., et al.: Configurable Process Models as a Basis for Reference Modeling, pp. 512–518. Springer, Berlin (2006)
19. Verbeek, H., Basten, T., Aalst, W.V.D.: Diagnosing workflow processes using woflan. Comput. J. **44**(4), 246–279 (2001)