Using model checking to control the structural errors in BPMN models

Oussama Mohammed Kherbouche, Adeel Ahmad, Henri Basson

Using model checking to control the structural errors in BPMN models

Oussama Mohammed Kherbouche, Adeel Ahmad, Henri Basson
Université Lille Nord de France
Laboratoire d'Informatique, Signal et Image de la Côte d'Opale
BP-719 62228 CALAIS Cedex FRANCE
Email: {kherbouche, ahmad, basson}@lisic.univ-littoral.fr

***Abstract*—The emergence of BPMN as a standard notation to express the business processes is based on its simplicity of notations and its exhaustive expressiveness. Nevertheless the lack of formal semantics in the BPMN can cause syntactic and structural errors. The former requires less effort to be checked, while the later usually requires attention to prove some properties, like deadlock-freedom and livelock-freedom. In this paper, we address the issue of detecting the structural errors with an approach based on model checking. It verifies the soundness of business process model and helps the business modelers to avoid the deadlocks, livelocks, and multiple terminations errors.**

***Keywords*—BPMN process models; Kripke structure; LTL; Model checking; SPIN.**

I. INTRODUCTION

Business Process Management (BPM) (1, 2) is a managerial approach which empowers organizations to ensure that its processes are implemented both effectively and efficiently to fulfill expectations of stakeholders. It is therefore during the last decade a lot of literary work focused business process modeling. The Business Process Modeling Notations (BPMN) (3) has emerged as standard notation to express the business processes. It has been also used as a tool for expert analysis for decision making. This success is based on its simplicity of notations (4) and its exhaustive expressiveness. Nevertheless, the modeling of these business processes relies generally on the human expertise and lack the formal semantics (5). It characterizes the BPMN to cause undesirable errors which can be classified into two categories, either syntactical or structural.

The syntactical errors may occur by mistaking the use of modeling elements i.e. an *AND-join*, *OR/XOR-join* or an *event* when it does not allow more than one outgoing arc, etc. The valid or invalid combinations to be used are usually prescribed by the corresponding standard. The syntactical correctness of models, such as invalid construct or flow, can usually be found within reasonable time by simply parsing through the process model by using some modeling tools such as BizAgi[1], Intalio[2], or Bonita[3], etc.

1 http://www.bizagi.com/
2 http://www.intalio.com/
3 http://fr.bonitasoft.com/

Unlike the syntactical errors which can be found during the design-time (by using modeling tools), the structural errors are mostly found during the run-time, since a syntactically correct process can exhibit unexpected behavior during its run-time, as a result of poorly controlled data or structural errors. The structural errors, such as wrong combination of the sequence of elements or misaligned splits and joins, are difficult to be detected during the design-time due to lack of formal semantics of BPMN process models. Subsequently, the run-time behavior of a process should be analyzed before execution to achieve the complete verification, showing whether the process model fulfills important structural criteria. These can be either deadlock-freedom or livelock-freedom to avoid the improper functioning of the process, which can cost financially expensive damages.

To respond the issues raised above, we propose in this paper an approach to automate the checking of some structural errors such as deadlocks, livelocks, and multiple terminations in BPMN process models based on model checking. The approach has two major advantages. First, we assume a computable polynomial time, i.e. most of the structural errors are actually detectable. Second, if an error is found, it provides a direct graphical path leading to the error. The main objective is to map the BPMN process model to Kripke structure, and then check the validity of major properties (e.g. absence of deadlocks, livelocks and multiple terminations) expressed in Linear Temporal Logic (LTL) (6, 7) formulae. This ensures to provide the soundness of business process model and avoid any structural errors.

The rest of the article is structured as follows. Section II, summarizes the preliminaries used to illustrate our approach. In the section III, we describe the structural errors. Section IV discusses, in detail, the proposed approach. Section V describes the implementation details of the approach. The section VI briefly narrates the closely related work. Finally in Section VII, we conclude our contribution.

II. BASIC DEFINITIONS

The following sections, briefly explain the concepts and technical terms used in the proposed approach.

A. *Business Process Modeling Notation*

The Business Process Modeling Notation (BPMN) is graphical notation and a language for modeling business processes. It was adopted by OMG and it has been specified since February 2006 (3). The primary goal of BPMN is to provide the notations which are readily understandable by all business users. BPMN creates a standardized bridge for the gap between the business process design and process implementation.

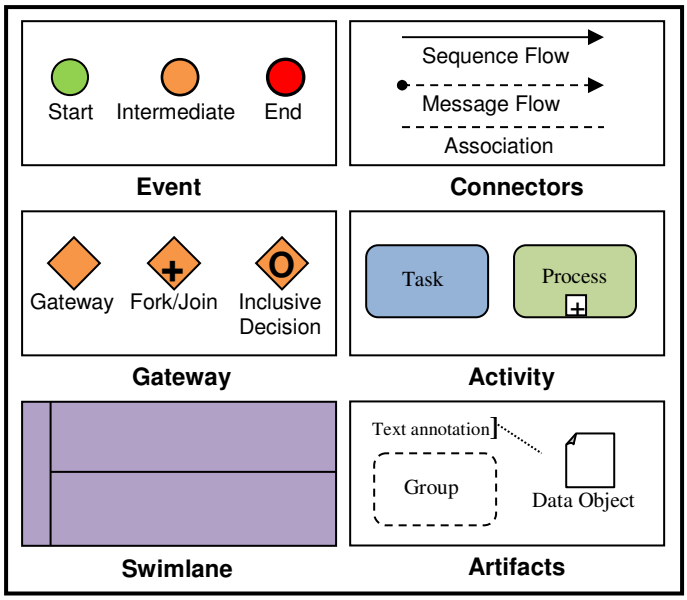


Figure 1. Some core concepts of BPMN elements

As shown in Fig. 1, BPMN process diagrams provide some graphical notations for business processes. These can be categorized as below:

- *Flow Objects*: are the main graphical elements to define the behavior of a business process. There are three kinds of flow objects, which are event, activity, and gateway.
- *Connectors*: are the graphical elements to connect the Flow Objects to each other. There are three kinds of Connecting Objects, which are Sequence Flow, Message Flow and Association.
- *Swimlanes*: are the graphical elements to group the modeling elements. There are two ways of grouping the primary modeling elements, which are pools and lanes.
- *Artifacts*: are used to provide additional information about the Process. There are two standardized Artifacts, which are Group and Text Annotation.

B. *Model Checking*

The model checking (8) is a tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior. Schematically, a model checking algorithm takes as input an abstraction of the behavior of the reactive system (a transition system). It includes the Kripke structures or other models as petri nets, finite automata, timed automata, etc and a formula of some temporal logic (LTL, PLTL, CTL, CTL*, TCTL, etc.), and meets if the abstraction satisfied or not the formula as shown in Fig.2. The term model checking refers to the transition system as a model of the formula. The major advantage of model checking is, for it, to be completely automatic in most of cases and it returns a counterexample when the properties are not verified.

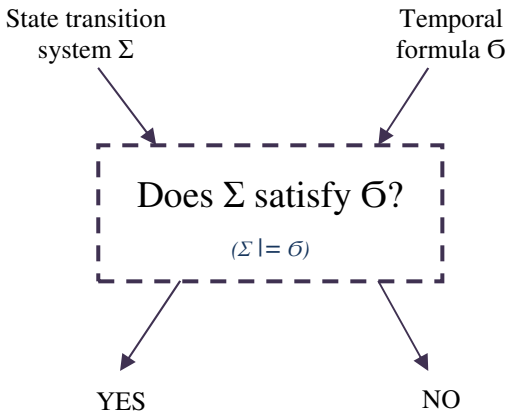


Figure 2. Model Checking

Among the possible models to describe a system and a given property, the choice is often a compromise between expressiveness and ease of analysis. There exist many widely used tools such as SPIN (9) and NuSMV (10) to achieve this goal.

C. *Kripke structure*

A Kripke structure (11) is a variation of nondeterministic automaton used in model checking to represent the behavior of a system. It is a graph where nodes represent the reachable states of the system and the edges represent state transitions. A labeling function maps each node to a set of properties (atomic proposition) that hold in the corresponding state. The semantics are based on temporal logics for most of the widely used specification languages for reactive systems.

Let us assume, AP as a set of atomic proposition i.e., a set of labels over the system.

A Kripke structure is a 4-tuple M = (S, I, R, $\mathcal{L}$) where:

- S is a finite non-empty set of states
- I ⊆ S is a set of initial states
- R ⊆ S × S is a transition relation which associates with each state s ∈ S its possible successors are, $\forall s \in S, \exists s' \in S$ such that $(s, s') \in R$
- $\mathcal{L} : S \rightarrow 2^{AP}$, is a labeling function which associates with each state s ∈ S the set of atomic propositions $\mathcal{L}$(s) holds in s.

D. *Linear Temporal Logic (LTL)*

LTL is the most commonly used language for specifying temporal properties of software or hardware designs. It is able to discuss about the future of paths.

LTL is build up from proposition variables *p, q, r* …, the usual logic connectives ⊤ (true), ⊥ (false), ¬ (not), ⋁ (or), ⋀

(and), → (imply), ↔ (one-to-one) and four temporal connectives X, F, G, and U (as shown in Fig.3). The temporal connectives are explained as follows:

- G ('always'): is read always in the future (in all future states of path). Graphically, it can be denoted as : □
- X ('next time'): is read at the next time (in the next state of path), and denoted as: ○
- F ('eventually'): is read eventually (in some future state of path, and denoted as: ◊
- U ('until'): is read until, which can be denoted as: *U*

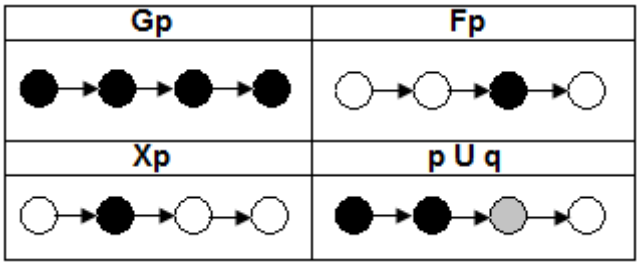


Figure 3. LTL Temporal connectives representation

LTL formulas are generally evaluated over paths and a position on that path. A LTL formula as such is satisfied if and only if it is satisfied for initial position on that path.

We briefly narrate the semantics of LTL as bellow:

Let $\pi = s_0, s_1, s_2 \ldots s_n$ be a sequence of states and L such as: $\forall i \geq 0, L(q_i) \subseteq AP$.

The sequence π satisfies Ϭ. It is denoted by $\pi \models Ϭ$. This relation can be defined inductively and gives semantics of LTL formulas as below:

- $p \in AP, \pi \models p \leftrightarrow p \in L(q_0)$
- $\pi \models p \vee q \leftrightarrow \pi \models p$ or $\pi \models q$
- $\pi \models \neg p \leftrightarrow \pi \not\models p$
- $\pi \models Xp \leftrightarrow \pi_1 \models p$
- $\pi \models p \, U \, q \leftrightarrow \exists j \geq 0, \pi_j \models q \wedge (\forall k < j, \pi_k \models p)$.

E. SPIN model checker

SPIN model checker (12) is a widely used tool for the specification and simulation of concurrent systems, which are designed primarily for the verification of communication protocols. It is developed by G. J. Holzmann (2003) and it uses a high level language to specify system descriptions, called PROMELA (PROcess MEta LAnguage) (12). This language allows dynamic creation of processes with both synchronous and asynchronous communication, through communication channels. It is an executable model. A PROMELA language consists of variables, channels and processes. Processes are global objects, while variables and channels may be declared either as global or local to a process.

A PROMELA model can be analyzed by Spin either through:

- Simulation: the model is run step by step, which makes it easier to be familiar with its behavior
- Verification: The states of the model are explored exhaustively to verify that the model satisfies properties (eg, mutual exclusion) specified in LTL.

The resulting model is written in PROMELA and it is translated to Kripke structures by SPIN model checker.

III. STRUCTURAL ERRORS

This section, presents some structural errors which can occur during run-time of BPMN process models. We intend by the term structural errors as the deadlocks, livelocks, or multiple terminations. These are used to illustrate the proposed approach to ensure the soundness of the business process models.

A. Deadlock patterns

Generally, a deadlock is defined as a system which reaches a dead-end state. For the process models, it is a case for which certain instances of a process model cannot continue working, while they have not reached the process end (i.e. deadlock is a condition used to describe a process that cannot be completed).

According to Onada *et al.* (13) there are two complementary concepts. The first is reachability which is symbolized by the existence of at least one path from node A to node B in a process graph and the second is absolute transferability, which means, it is a much stronger concept to state that a token can always be transferred from node A to all input points of node B. The deadlock occurs, whenever there is reachability without absolute transferability.

In (13), the authors have also identified several potential causes of deadlocks, as follows:

- Loop deadlock: as shown in Fig.4.a, occurs when there is an execution path from the output of an *AND-join* back to its input points. If this path does not contain *XOR-splits,* deadlock occurrence is certain; otherwise, it can occur if the branch leading to the loop is chosen.
- Multiple sources: as shown in Fig.4.b, the multiple sources occur when two different sources lead at the input points of *AND-join* gateway. Assuming that none of the source nodes is the *AND-Join* itself, it can be observed that the multiple source patterns can occur when one of the process structure is as follows:
    - Any of the two sources is an *XOR-split* gateway.
    - The process has multiple start points that will be synchronized later. In case of models specified in BPMN, multiple starts are permissible. Actually, multiple start points resemble an *AND-split* gateway between the start events; hence we can deduce that there is reachability between two or more sources (start events) to the *AND-join* node.

- Improper structuring: as shown in Fig.4.c, occurs when an *AND-join* gateway receives input that started earlier from an *XOR-split*.

B. Livelock patterns

Livelock can be defined as a state from which it is possible to proceed, but it may be impossible to reach the desired final state (the system is locked into a small subset of states and makes no progress).

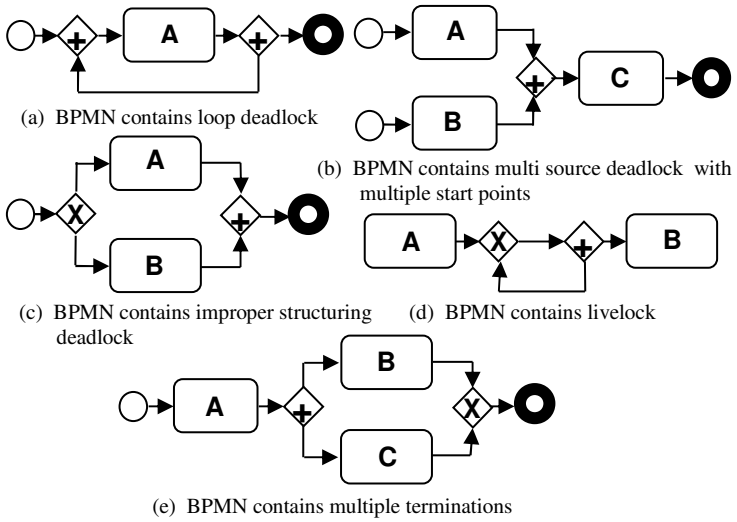
As shown in Fig.4.d, the livelock can result an infinite execution of process. In this case some of the processes may run successfully but some may trap in an endless loop of execution.

This can happen when an *AND-split* is used instead of an *XOR-split* for modeling an existing loop.

C. Multiple terminations patterns

The multiple terminations correspond to the situations where exists an *AND-split* before an *XOR-join* gateway, as shown in Fig.4.e.

In this case, only one sequence is traversed when the exclusive gateway is executed. This case leads to the violation of soundness criterion. Thus, the BPMN process model does not terminate the predefined (expected) terminate processes.



(a) BPMN contains loop deadlock

(b) BPMN contains multi source deadlock with multiple start points

(c) BPMN contains improper structuring deadlock

(d) BPMN contains livelock

(e) BPMN contains multiple terminations

Figure 4. Structural errors in BPMN

IV. MODEL CHECKING TO DETECT STRUCTURAL ERRORS

In this section, we discuss the detection of prominent structural errors which can occur in a BPMN process. Our approach is broadly described in Fig. 5. The main idea is to map BPMN process model into a finite-states model (Kripke structure) for specifying the system behavior and provide some LTL formulae that may be used by model checker to verify the absence of structural errors and ensure the soundness of process model. Otherwise, it returns a counterexample. Several LTL properties can be defined simultaneously for a Kripke structure. The verification steps are detailed, as follows:

A. Finite state generator

The first step is to map BPMN process models into Kripke structure to express the behavior of the process models. The states of a Kripke structure represent the behavior of the process model. This translation facilitates the better verification of the desired temporal properties such as: $M \models \phi$ iff $M, \pi \models \phi$ for all paths π in a Kripke Structure M.

The finite non-empty set of states *S* of the Kripke structure represents the nodes *N* of the process model. *N* is a finite set of flow objects in BPMN process which can be partitioned into events *E*, activities *A* and gateways *G*. The transition relations *R* represent the edge relations *T* (where, Transition $T \subseteq F \times F$ is a finite set of sequence flows connecting objects).

TABLE I. MAPPING OF BPMN OBJECTS TO KRIPKE STRUCTURE

BPMN Object	Kripke Structure
Start s	Start
End e	Final
Message M	E_M, M, exM
Task T	E_T, T, exT
T, Ex, T_x	E_T, T, exT; E_T_x, T_x, exT_x
A, X, B, C	exA; E_B, B, exB; E_C, C, exC
A, B, X, C	E_A, A, exA; E_B, B, exB; E_C
A, B, +, C	E_A, A, exA; E_B, B, exB, exC; E_C, C
A, B, +, C	E_A, A; E_B, B; exA, exB; E_C, C, exC

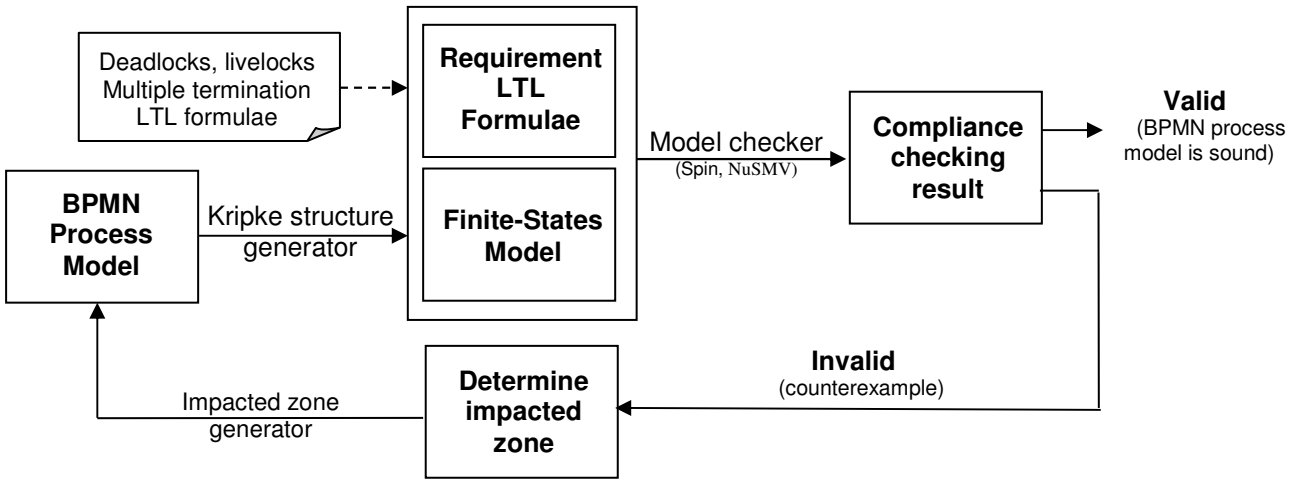


Figure 5. Global schema of model checking approach for detecting structural errors

To obtain a Kripke structure, we define *AP* as the set of atomic propositions and associate with each state $s \in S$ such as $\mathcal{L}(s)$ holds in s ($\mathcal{L}$ is the labeling function of Kripke structure *M*). It expresses all properties of a given state.

The initial state $I \subseteq S$ is the start point E^S (start event) of the process model. Each state *s* is labeled with *enabled* and *executed* transitions. Where, *E_A* signifies that the transition *A* is enabled and *ex_A* signifies that the transition *A* is executed (completed). A brief description of the mapping from a set of BPMN tasks, events, and gateways to Kripke structure is given in Tab.I.

Once the Kripke structure is obtained, we then proceed to define the desired correctness temporal formulae.

B. LTL formulae generator

The soundness of BPMN process model to avoid structural errors can be ensured by satisfying the following temporal properties:

1) Detect absence of deadlocks

A Kripke structure is said to be deadlock-free, if it does not contain any computation that can lead to a deadlock. The deadlock freedom is considered as safety property (i.e. something bad never happens). Let us assume a temporal formula (Final), which represents the set of final states. In such a case, we can express deadlock-freedom by the following LTL formula:

$$\Box\,(\circ \bot \rightarrow Final) \tag{1}$$

This formula must be satisfied as valid on every path. The formula $\circ\bot$ (means that "there is no next state") is easy to deduce, i.e. no transition is possible. Likewise, we can express reachability of a given deadlock state as the existence of a state with the dual property.

$$\Diamond\,(\circ \bot \rightarrow \neg\, Final) \tag{2}$$

2) Detect absence of livelocks

As previously described, the livelock is a state from which it is impossible to reach the desired final state.

A property which expresses the non-existence of livelock is a liveness property (i.e. something good eventually happens). A typical LTL formula is shown below:

$$\Diamond\Box\phi \rightarrow \Box\Diamond\psi \tag{3}$$

If a task tries to run infinitely, then it will be always in the execution state. This simplifies to $\neg\,\Diamond\Box\phi$ (i.e. it will not succeed; 'at run' forever). In a counterexample of these properties is an infinite execution according to which any of the expected behavior does not happens (i.e. the process does not terminate). Detection of a livelock can be expressed in the LTL formula, as shown below:

$$\Diamond\Box exA \rightarrow \Box\Diamond Final \tag{4}$$

3) Detect absence of multiple terminations

The multiple terminations is a situation where exists an *AND-split* before an *XOR-join* gateway. Detection of this case is based on checking the safety property of LTL (i.e. something bad never happens). It can be verified by the following formula:

$$\Box\,\neg\,(\Diamond\,(\phi \bigwedge \circ\psi) \rightarrow \text{Final}) \tag{5}$$

A counterexample of these properties is a finite execution which leads to unexpected behavior.

C. Model Checking

The finite state machines and the temporal logic formulae are presented as input to a model checker. The model checker verifies whether σ temporal formula holds for that finite state machines *M* or not. As a result, it confirms the soundness of the process models. Otherwise, it returns a counterexample in cases of structural errors.

D. Determine impacted zone

The model checking has the capability of providing counterexamples when the temporal properties to be checked are not satisfied by the process model (14, 15). Mostly, these counterexamples are given in terms of internal state transitions rather than in terms of process models that are difficult to understand by a non-technical user. To benefit

from these counterexamples, the output of the model checker should be translated in the visual notation, which is easier for the user to understand.

The mapping of counterexample to the source BPMN process model supports the better determination of the impacted zone (by structural errors). We use model checker dependency, it contain a tool chain that translates the output of the model checker back to the process model notations. This can allow the mapping of each state to the elements of original BPMN process model, which can highlight (notify) through a change of color or assignment of a particular label, to better visualize the impacted zone. This may help to find the actual causes of errors in the business model and also to correct them.

V. TOOL DESIGN AND IMPLEMENTATION

Currently, we implement a prototype tool to validate the presented approach called BPMN2SPIN. It is developed as a set of Eclipse[4] IDE plug-ins. We use Eclipse BPMN 2.0 Modeler[5] plug-in as a tool to modeling business process. As model checker, we opted for EpiSpin[6] plug-in.

The BPMN process model can be transformed into PROMELA language, which maps the processes, sub-processes and activities into PROMELA processes and connector paths into PROMELA channels. The messages between processes are represented, without loss of generality using integers in PROMELA.

The main concern is to transform automatically the BPMN process model to PROMELA model by providing input PROMELA model file (*.pml) (i.e. EpiSpin translates the PROMELA model into Kripke structure) and provides some pre-defined LTL formulae to the EpiSpin model checker to detect the structural errors. If the temporal properties are not satisfied by the PROMELA model, then the returned counterexample is mapped to the source BPMN process model.

VI. CASE STUDY

We further illustrate the translation of BPMN process model to PROMELA model with the help of an example. It uses the notation `c` to represent a channel and `m` to represent a message (sent or received). `cS` denotes an array of channels and `mS` denotes an array of messages to be sent or received in each channel in `cS`. The functions `inline send(q, msg){q!msg;}` and `inline receive(q,msg){ q?msg;}` are used to exchange messages between processes.

In Tab. II, we represent the translation of some principal elements which are *Sequence, AND-Split, AND-Join, XOR-Split, XOR-join, OR-Split, OR-Join* to PROMELA Language.

The given case study consists of a simple car salesman process, as shown in Fig.6. The seller gets pay bills at the end of each month. It gets a bonus when it sells more than twenty cars (in addition to his regular paycheck). The deadlock in this process can occur (when the salesman sells less than twenty cars i.e. the salesman does not get a bonus) because of the parallel gateway still requires both paths before completion, excluding the case when the bonus-path is never started. Otherwise, the deadlock cannot occur, when the salesman sells more than twenty cars, both paths are selected from the *OR* gateway, and then both paths are combined in the parallel gateway.

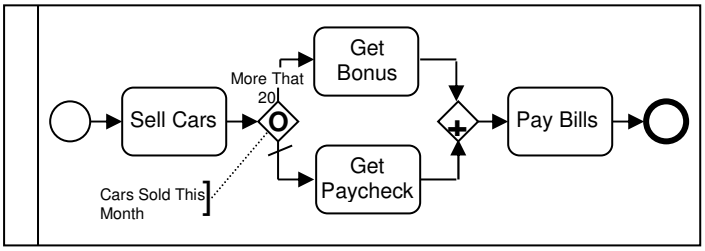


Figure 6. Car Salesman process

The absence of deadlocks and livelocks in EpiSpin are detected by the *invalid endstates* and the absence of *non-progress cycle* features, respectively. The generated PROMELA model corresponding to the car salesman process is described as follows:

We define six global channels denoted as `cS` to establish the communication between activities. We define also `cS1` and `cS2` as auxiliary array channels to simplify the exchange of messages in given activity. The PROMELA model of involved activities is detailed in below:

Sell Cars –This activity is initiated by receiving a request from the start event, it choose one or more activities in non-deterministic manner without loss of generality. It is translated by the following PROMELA process.

```
proctype sellCars(){
  chan cS1[2]=[1] of {int};
  int x, choice, msgs[2];
  cS1[0]= cS[1]; /*send to Get Bonus*/
  cS1[1]= cS[2]; /*send to Get Paycheck*/
  R: receive(cS[0],x);/*receive from start
                        event*/
  /* Choice of a non-deterministic manner */
     if
          ::choice=0  /* Less that 20 cars */
          ::choice=1  /* More that 20 cars */
     fi;
     msgs[0]=1;
     msgs[1]= choice;
     ORSplit(cS1,msgs,cS1, msgs);
     goto R
}
```

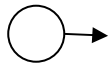
Bonus Activity - As mentioned above, this activity is chosen in non-deterministic manner without loss of generality. It is translated by the following PROMELA process.

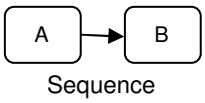
```
proctype GetBonus(){
  int x;
  receive(cS[1],x); /* To receive from
                       Sell Cars. */
  send(cS[3],1); /* To send to Pay Bills. */}
```

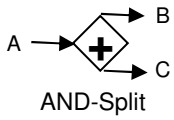
4 www.eclipse.org/
5 svn+ssh://svn.java.net/bpmn-modeler~source-code-repository
6 http://epispin.ewi.tudelft.nl/

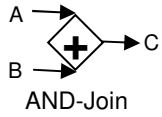
TABLE II. MAPPING OF BPMN OBJECTS TO PROMELA LANGUAGE

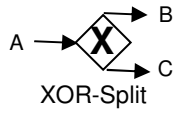
BPMN Object	PROMELA Language	Observations

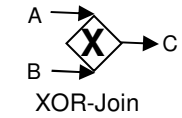
	`chan c = [1] of {int};`
	`active proctype start(){`
	`  /* To start process */`
	`  send(c,1);`
	`}`
		The *start event* in PROMELA use the *send* statement and the next process use the *receive* statement.

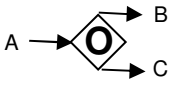
	`proctype A(){`
	`  /* details of activity */`
	`  send(c,1); /* Activate process B */`
	`}`
	`proctype B(){`
	`  int x;`
	`  receive(c,x); /*Waiting token to run*/`
	`  /* details of activity.*/`
	`}`
		An Activity 'B' is enabled after the Activity 'A' is completed in the same process. The process 'A' in PROMELA use the *send* definition and the process 'B' use the *receive* definition.

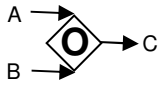
	`inline ANDSplit(cS, mS){`
	`  int ind, length;`
	`  ind = 0; length = len(cS);`
	`  atomic {`
	`  do`
	`  ::ind <length->send(cS[ind],mS[ind]);`
	`  ind++;`
	`  ::ind >=length->break;`
	`  od;`
	`  }`
	`}`
		The *AND-Split* is translated to inline definition, where each channel in the array *cS* is used to communicate with each activity. The process in PROMELA which represents the activity that splits the process use the *ANDSplit* definition and the processes which represent the activities to be initiated use the *receive* definition.

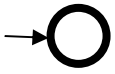
	`inline ANDJoin(cS, mS){`
	`int ind, length;`
	`ind = 0; length = len(cS);`
	`skip;`
	`S: if`
	`  ::full(cS) ->`
	`  do`
	`  ::ind <length && nempty(cS[ind])->`
	`  receive(cS[ind],mS[ind]);ind++;`
	`  ::ind >=length-> break; goto E`
	`  od;`
	`  :: nfull(cS) -> ind=0; timeout; goto S`
	`  fi;`
	`E: skip;`
	`}`
		The *AND-Join* represents the case in which two or more parallel execution flow branches merge into a single flow, after all branches are completed. This *AND-Join* is translated to the inline definition. The process in PROMELA that receives the input branches use the *ANDJoin* definition and the processes to be syncronized use the *send* definition. The keyword *timeout* is used to avoid a starvation of process and allow other processes to be executed.

	`inline XORSplit(cS,choice,m) {`
	`  int length;`
	`  length = len(cS);`
	`  if`
	`  ::(choice<length && choice>=0) ->`
	`  send(cS[choice],m);`
	`  ::else -> skip;`
	`  fi;`
	`}`
		The *XOR-Split* represents the case in which the execution flow is spawn in two or more branches, thus enabling the execution of one and only one activity among the available set. The exclusive split is translated to the inline definition. The process in PROMELA which represents the activity which makes the choice to use *XORSplit* definition, and the alternative processes use the *receive* definition.


XOR-Join
	inline XORJoin(cS, m){
	int ind,length;
	ind=0;
	length=len(cS);
	skip;
	B: **if**
	::nempty(cS[ind])-> receive(cS[ind],m);
	goto E
	::empty(cS[ind])-> ind++; goto I
	fi;
	I: **if**
	::ind==length -> ind=0; timeout; goto B
	::ind<length -> goto B
	fi;
	E: skip;
	}
		The *XOR-Join* represents the case in which two (or more) mutually exclusive execution branches merge into a single flow. An exclusive join is translated to the inline definition. The process in PROMELA which merges the input branches use the *XORJoin* definition and the merged activities use the *send* definition.


OR-Split
	inline ORSplit(cS, mS, aCs, choices){
	int ind, length;
	ind=0; length=len(cS);
	skip;
	S: **if**
	::choices[ind]==1 -> send(aCs[ind],1);
	send(cS[ind],mS[ind]);
	::choices[ind]==0 -> send(aCs[ind],0);
	fi;
	ind++;
	if
	::ind < length -> goto S
	::ind >= length -> skip
	fi;
	}
		The *OR-Split* represents the case in which the execution flow is spawn in two or more parallel branches, thus enabling possible parallel execution of two (or more) activities. The process that represents the activity which makes the choice to use *ORSplit* definition and the alternative processes use the *receive* definition. *aCs* provides information about the activated channels.


OR-Join
	inline ORJoin(cS, aCs, mS){
	int ind, length, x;
	ind =0;length = len(aCs);
	do
	::ind <length -> timeout; ind++;
	::ind ==length -> ind=0; break;
	od;
	skip;
	B:**if**
	::length > 0 && ind <length &&
	receive(aCs[ind],x) == 1->
	receive(cS[ind],mS[ind]); ind++;
	::ind >= length -> goto E
	fi;
	E: skip;
	}
		The *OR-Join* represents the case in which two (or more) parallel execution flow branches merge into a single flow. The process that represents the merging activity use the *ORJoin* and the processes that represent merged activities use the *send* definition.

	**proctype** End(cS, mS){
	int ind, length;
	ind = 0; length = len(cS);
	end:
	do
	::ind <length->receive(cS[ind],mS[ind]);
	ind++;
	::ind >=length->break;
	od;
	}
	}
		The *end event* indicates where a process will end. The *end event* in PROMELA should use the *receive* statement. To specify that a state is not a deadlock, but rather a proper end state, we should use the end label instruction to specify where the process stops.

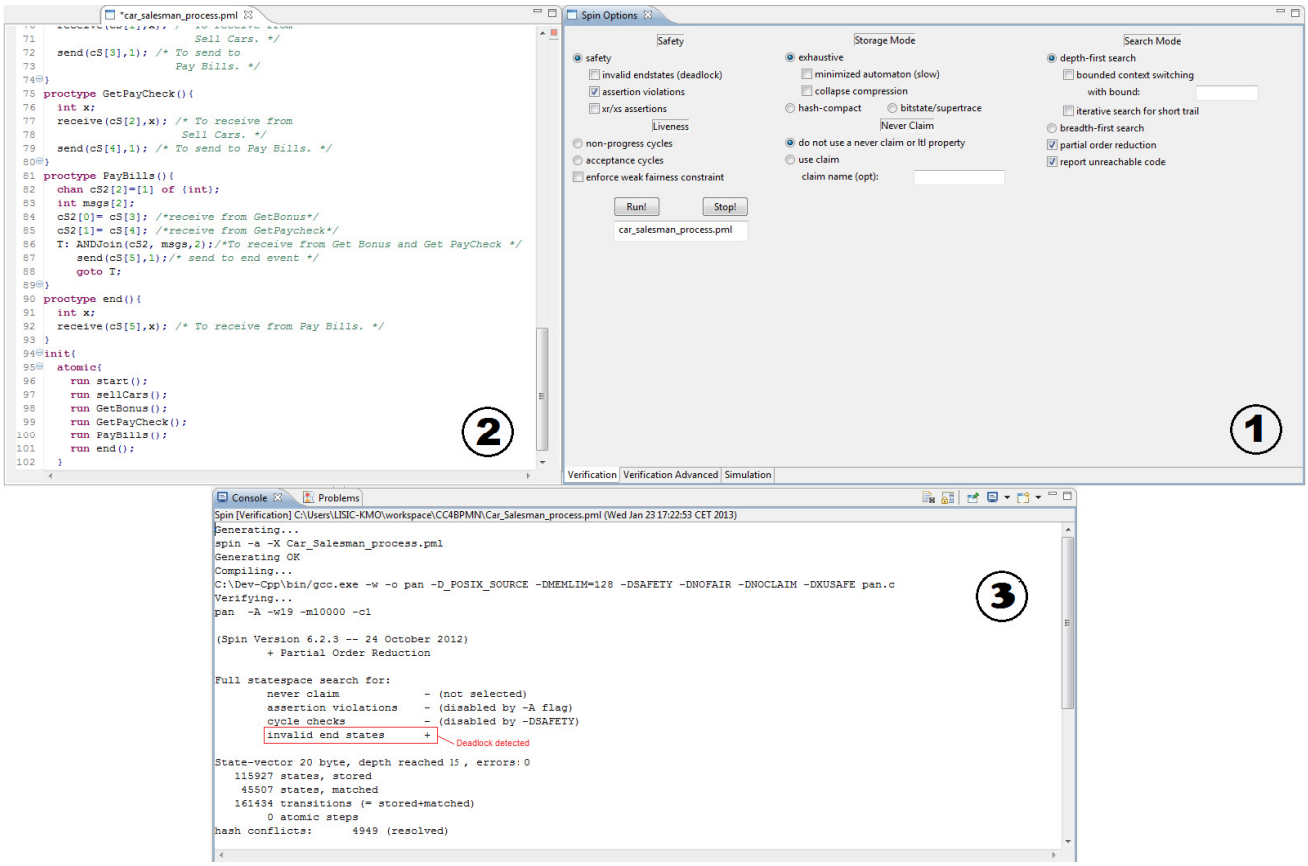


Figure 7. Detection of deadlock using EpiSpin model checker

Get PayCheck - This activity is translated by the following PROMELA process.

```
proctype GetPayCheck(){
  int x;
  receive(cS[2],x); /* To receive from
                       Sell Cars. */
  send(cS[4],1); /* To send to Pay Bills. */
}
```

Pay Bills - This activity receives a bonus when it sells more than twenty cars (in addition to the regular paycheck of seller). It is translated by the following PROMELA process.

```
proctype PayBills(){
  chan cS2[2]=[1] of {int};
  int msgs[2];
  cS2[0]= cS[3]; /*receive from GetBonus*/
  cS2[1]= cS[4]; /*receive from GetPaycheck*/
  T: ANDJoin(cS2, msgs);/*To receive from
           Get Bonus and Get PayCheck */
     send(cS[5],1);/* send to end event */
     goto T;
}
```

Following is the description corresponding to the car salesman process from *'Car_Salesman_process.pml'* input file.

```
chan cS[6] = [1] of {int};
proctype Start(){....}
proctype sellCars(){....}
proctype getBonus(){....}
proctype getPayCheck(){....}
proctype PayBills(){....}
proctype End(){....}
init {
  atomic{run start(); run sellCars();
         run getBonus();run getPayCheck();
         run PayBills();run End();
  }
}
```

The Fig. 7 shows the screenshots of the input PROMELA model file "*Car_Salesman_process.pml*" provided to EpiSpin and the result of verification of the absence of deadlock (i.e. invalid endstates). It can be observed that the EpiSpin detects an error because the paths are selected in non-deterministic manner. In this case, the path through "*Get PayCheck*" activity is chosen from *OR-split* gateway (without the path containing "*Get Bonus*" activity). Therefore, the *AND-join* gateway receives only one input, which causes a deadlock.

It is important to note that for better understanding of the different results, we can obtain the finite-states model (Kripke structure) with the help of EpiSpin from PROMELA model, as shown in Fig 8. It proposes also to generate a DOT code from a *.pml input file, which is generally saved in *.dot file. We use the Graphiz[7] to compile the DOT code into an image for its visualization.

7 http://www.graphviz.org/

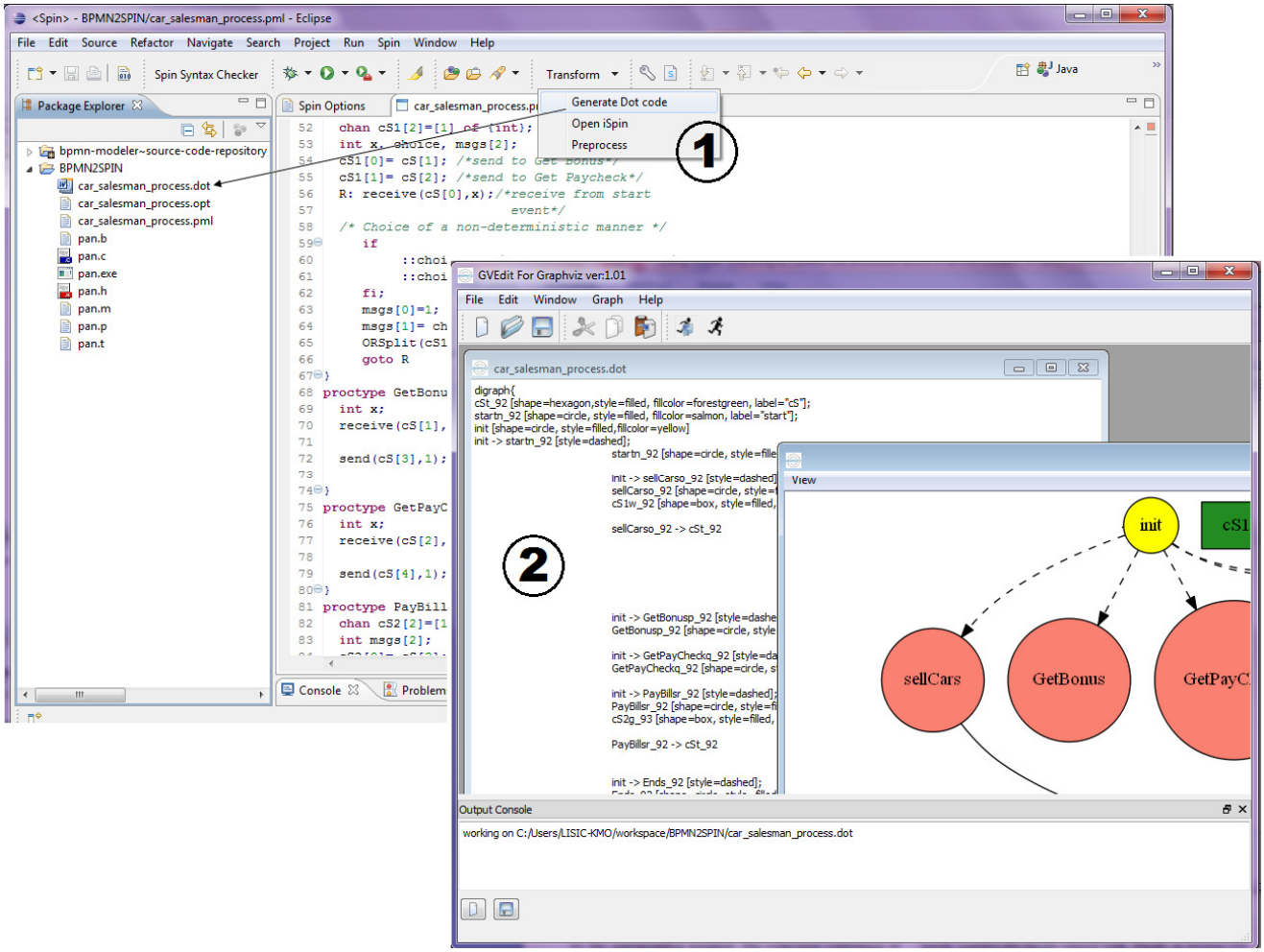


Figure 8. EpiSpin DOT code generation and Graphiz visualization

VII. RELATED WORK

The problematic regarding lack of formal semantics in the modeling languages has been addressed in many literary studies during last decade. As a result, numerous approaches and techniques emerged to verify the process models (in particular the soundness property). Sadiq and Orlowska (16) are among the pioneers to identify the structural errors, like livelock and deadlock, in business processes modeling. Their work primarily targets the syntactical errors. It relatively lacks a support to handle the semantic errors of business processes.

Literature survey reveals that many techniques and methods are focused to treat the semantic issues in business process modeling. Furthermore, several authors developed notations based on a process modeling language to express the allowed executions of a BPM. Such approaches have been presented for Event-Driven Process Chains (17), UML Activity Diagrams (18), BPEL specifications (19, 14) and for BPMN (20). The most prominent are based on either formal approaches or design approaches.

The design approaches are verification methods based on a design model given in a specific language. Awad *et al.* (21, 22) present an approach to detect deadlocks using a method concerning BPMN-Q (23). The approach is constrained to detect the deadlock errors. Another design approach (24) focuses on the graphical structure of the model. It is based on BPMN VQL (24) query language. Its main purpose is to find crosscutting concerns in BPM. However, modeling processes using this notation necessitates the advanced technical skills and the resulting model is usually complex and far from intuitive.

A vast variety of formal approaches is also in practice to detect structural errors. One of the techniques, in this category, is based on Petri nets (25, 26). W.M.P. van der Aalst (27, 28) proposes soundness criterion, in this regard, to guide the modelers for the specification of Event-Process chain and to detect the livelock and deadlock errors in the control-flow (29) using Petri nets. Also, the authors in (30) propose to handle deadlock and multiple termination patterns in SAP reference model. It is particularly intended to be applied on two popular modeling languages i.e. Event-Process Chain (EPC) and Petri nets. Another approach, proposed by Dijkman *et al.* (31), uses Petri nets based method to verify the BPMN process models. In this approach, BPMN model is first transformed to Petri nets and later, ProM is utilized to verify them.

However, it is important to notice that certain components in BPMN such as multiple instances of model, exception handling, and message flows cannot be changed into Petri nets (31). It reveals difficulties to define the correspondence of these objects to Petri nets.

Another approach based on π- calculus is used to represent workflow patterns (32). However, the verification through π- calculus involves the checking of bi-simulation equivalence. It consumes more time to obtain results, even to prove the simple correctness requirements. In (33), authors propose to use the finite-state automata to detect deadlocks and multiple terminations. But their propositions lack the detection of livelock errors.

The model-checking is also one of the techniques used in formal approaches to verify whether the business process models satisfy some properties formalized in LTL or CTL (e.g. checking compliance rules). In this approach, the business process models are transformed into states and transitions between the states. Furthermore, the business processes are transformed into Petri nets in order to detect in first step the deadlocks, and livelocks errors, followed by a transformation into Kripke structures which are used in addition to the temporal logic formulae as input to a model-checker which verifies whether the temporal logic formulae are respected by the given finite state machines or not (14, 34, 35, 36). The prominent focus, in these studies, remained on the expression of different proprieties in LTL formulae rather than the transformations of process model to finite-states. Hence, they lack implementation details.

The existing research in the literature is focused not only to verify the control-flow but also interested in the verification of data-flow. In fact, the importance of data-flow verification in workflow processes was first mentioned in (37). They identified several possible errors in the data-flow e.g., the redundant data error, reading from an uninitialized element type of errors, but no means for checking these errors are provided. In (38), a model called dual workflow nets is proposed, that can describe both the data-flow and the control-flow. The notion of soundness is extended to support the case when data-flow can influence control-flow.

The research work, presented in this paper, has two major objectives. The first, and the foremost, is to provide an automated assistance to verify the absence of most structural errors in the business process models. We use model-checking technique to achieve this goal. The second objective is to resolve the ambiguity regarding the transformation of BPMN process model into a finite-state model. We transform the BPMN process model directly to Kripke structure without going through the intermediate step which is generally petri-nets. The automated transformation of BPMN process model to Kripke structure is both faster and easy to maintain. The soundness of the BPMN process model can be verified through the compliance checking verification. In case of the error, the interpretation of the result along with the mapping of returned counterexample, in BPMN model, can lead to the cause of the error.

VIII. CONCLUSION

The model checking can help to better detect structural errors in business process models. The automated checking of such errors allows both to compute the polynomial time and the error traceability. In this paper, we have discussed, in detail, the structural errors which can occur during run-time of business process models and the properties to be checked to avoid these errors. The objective is to provide assistance for the process modelers in the better detection of errors and their correction.

The approach proposes to map the BPMN process model directly to Kripke structures to express the behavior of the process models. The generated finite states (Kripke structures) are used to satisfy the temporal properties (e.g. absence of deadlocks, livelocks and multiple terminations), which are expressed using the Linear Temporal Logic (LTL) formulae. The approach is supported by the implementation of a tool that integrates EpiSpin plug-in in eclipse IDE as a model checker and translations of BPMN models to PROMELA models as expressed in case study which represent an input of EpiSpin model checker. The result of model checking can verify the soundness of the process model, otherwise it return a counterexample. We are continuing the development of plug-in for the counterexamples to further facilitate the determination of the impacted zone. In the future, we intend to continue this approach and particularly to focus the compliance checking rules for the BPMN post change scenarios.

REFERENCES

[1] R.Lu, "Constraint-Based Flexible Business Process Management," in School of Information Technology and Electrical Engineering, University of Queensland, 2008.

[2] W.M.P. van der Aalst, *et al.* , "Business Process Management: A Survey," in Proceedings of Conference on Business Process Management (BPM 2003), Eindhoven, Netherlands 2003.

[3] Object Management Group. BPMN 2.0: OMG final adopted specification DOI= http://www.omg.org/spec/BPMN/2.0/PDF. January 2011

[4] I. Kitzmann, C. Konig, D. Lubke, and L. Singer, "A simple algorithm for automatic layout of bpmn processes," in CEC '09: Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing. Washington, DC, USA: IEEE Computer Society, 2009, pp. 391–398.

[5] S. Kühne, H. Kern, V. Gruhn, and R. Laue, "Business process modelling with continuous validation," in MDE4BPM 2008 – 1st International Workshop on Model-Driven Engineering for Business Process Management, C. Pautasso and J. Koehler, Eds., Milan, Italy, September 2008.

[6] Y. Kristin Rozier. , "Linear Temporal Logic Symbolic Model Checking," NASA Ames Research Center,Moffett Field,CA 94035,USA. 2010.

[7] S. Hornus, P. Schnoebelen , "On solving temporal logic queries," In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 163–177. Springer, Heidelberg. 2002

[8] E.M. Clarke Jr., O. Grumberg, and D.A. Peled. , "Model Checking," The MIT Press,Cambridge, Massachusetts and London, UK, 1999.

[9] J. Gerard Holzmann. "The Model Checker SPIN," In Proceedings of IEEE Transactions on software Engineering – Special issue on formal methods in software practice IEEE Press Piscataway, NJ, USA, 1997.

[10] A. Cimatti *et al.*, "NUSMV: a new symbolic model checker ," International Journal on Software Tools for Technology Transfer, 2000.

[11] M. C. Browne *et al.*,."Characterizing finite Kripke structures in propositional temporal logic," Theoretical Computer Science - International Joint Conference on Theory and Practice of Software Development. Elsevier Science Publishers Ltd. Essex, UK.1988.

[12] G. Holzmann,. "The SPIN MODEL CHECKER. Primer and Reference Manual," Addison-Wesley. Pearson Education , 2003.

[13] S. Onoda, Y. Ikkai, T. Kobayashi, and N. Komoda,."Definition of deadlock patterns for business processes workflow models". In HICSS '99: Proceedings of the Thirtysecond Annual Hawaii International Conference on System Sciences-Volume 5, pages 50–65, Washington, DC, USA, IEEE Computer Society 1999.

[14] Y. Lui, S. Müller, and K. Xu.: "A static compliance-checking framework for business process models," IBM SYSTEMS JOURNAL, 46(2) pp. 335-362, 2007.

[15] A. Foerster, G. Engels, and T. Schattkowsky.: "Activity diagram patterns for modeling quality constraints in business processes," In MoDELS, pages 2{16}, 2005.

[16] W. Sadiq and M.E. Orlowska., "Modeling and verification of workflow graphs," Technical Report No. 386, Department of Computer Science, The University of Queensland, Australia, 1996.

[17] S.Feja, D.Fötsch, "Model checking with graphical validation rules," In Proceedings of 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems,2008,pp.117–125.

[18] A.Forster, G.Engels, T.Schattkowsky, R.V.D.Straeten, "Verification of business process quality constraints based on visual process patterns," in Symposium on Theoretical Aspects of Software Engineering, 2007,pp.197–208.

[19] R.Wörzberger, T.Kurpick, T.Heer, "Checking correctness and compliance of integrated process models," in Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC2008), 2008.

[20] M.Brambilla, "LTL formalization of BPML semantics and visual notation for linear temporal logic," Tech. Rep.,Politecnico di Milano,2005.

[21] A. Awad and F. Puhlmann, "Structural detection of deadlocks in business process models," in BIS'08, pp. 239–250 , 2008.

[22] Ralf Laue, A. Awad, "Visual suggestions for improvements in business process diagrams," J. Vis. Lang. Comput. 22 (5): 385-399 2011.

[23] A. Awad, "BPMN-Q: A Language to Query Business Processes," In EMISA, pp.115–128 ,2007.

[24] C.D. Francescomarino, P. Tonella, "Crosscutting concern documentation by visual query of business processes," in proceedings of the International Workshop on Business Process Design, 2008.

[25] J. Desel and J. Esparza., "Free Choice Petri Nets," volume 40 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK,1995.

[26] T. Murata., "Petri Nets: Properties, Analysis and Applications," Proceedings of the IEEE, 77(4):541{580, April 1989.

[27] W.M.P. van der Aalst, "Formalization and verification of event driven process chains," Information and Software Technology, vol. 41, no. 10, pp. 639–650, July 1999.

[28] W.M.P. van der Aalst, "Workow Verification: Finding Control-Flow Errors using Petri-net-based Techniques," In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of Lecture Notes in Computer Science, pages 161{183. Springer-Verlag, Berlin, 2000.

[29] W.M.P. van der Aalst, K.M. van Hee, A.H.M. ter Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn, "Soundness of Workflow Nets: Classification, Decidability, and Analysis," BPM Center Report BPM-08-02, BPMcenter.org, 2008.

[30] B. F. van Dongen, M. H. Jansen-Vullers, H. M. W. Verbeek,and W. M. P. van der Aalst, "Verification of the SAP reference models using epc reduction, state-space analysis, and invariants," Comput. Ind., vol. 58, no. 6, pp. 578–601, 2007.

[31] R. M. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and automated analysis of bpmn process models," Tech. Rep, 2007.

[32] F. Puhlmann, M. Weske, "Using the pi-calculus for formalizing workflow patterns " Proceedings of the 3rd International Conference on BPM, volume 3649 of LNCS, Berlin, Springer-Verlag (2005) 153–168

[33] N. Tantitharanukul *et al.*, "Detecting deadlock and multiple termination in BPMN model using process automata," In electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON), (2010).

[34] A. Ghose, G. Koliadis, "Auditing business process compliance," In Kramer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 169–180. Springer, Heidelberg (2007).

[35] S. Goedertier, J. Vanthienen,"Compliant and Flexible Business Processes with Business Rules," in 7th Workshop on Business Process Modeling (2006).

[36] W.M.P. van der Aalst., H.T de Beer, B.F. van Dongen, "Process mining and verification of properties: An approach based on temporal logic," in Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, pp. 130–147. Springer, Heidelberg (2005).

[37] S.W. Sadiq, M.E. Orlowska, W.Sadiq, and C. Foulger, "Data Flow and Validation in Workflow Modelling," In Fifteenth Australasian Database Conference (ADC), Dunedin, New Zealand, volume 27 of CRPIT, pages 207{214. Australian Computer Society, 2004.

[38] S. Fan, W.C. Dou, and J. Chen, "Dual Workflow Nets: Mixed Control/Data-Flow Representation for Workflow Modeling and Verification," In Advances in Web and Network Technologies, and Information Management (APWeb/WAIM 2007 Workshops), volume 4537 of Lecture Notes in Computer Science, pages 433{444. Springer-Verlag, Berlin, 2007.