DataFlow Analysis in BPMN Models

Anass Rachdi, Abdeslam En-Nouaary and Mohamed Dahchour
Institut National des Postes et Télécommunications, 2, av ALLal EL Fassi Madinat AL Irfane, Rabat, Morocco

Keywords: BPMN, Business Process Modeling, Formal Verification, Dataflow Anti-patterns, Information Systems.

Abstract: Business Process Management and Notation (BPMN) is the defacto standard used in enterprises for modeling business processes.However, this standard was not provided with a formal semantics, which makes the possibility of analysis limited to informal approaches such as observation. While most of the existing formal approaches for BPMN models verification focus on the control-flow, only few has treated the data-flow angle. The latter is important since the correct execution of activities in BPMN models is based on data's availability and correctness. In this paper, we present a new approach that uses the DataRecord concept, adapted for the BPMN standard. The main advantage of our approach is that it locates the stage where the data flow anomaly has taken place as well as the source of data flow problem. Therefore the designer can easily correct the data flow anomaly.The model's data flow problems are detected using an algorithm specific for the BPMN standard.

1 INTRODUCTION

Business Process Management (BPM), is a managerial approach that enables an organization to ensure that its processes are implemented effectively and efficiently. Therefore, it brings an additional value to organizations by improving their performance, productivity and customer services quality. One of the most important phases that constitutes the life cycle of Business Process Management is Business Process Modeling. The latter is considered essential for designing and analyzing business process models that compose information systems. It involves the use of simple and intuitive modeling languages that makes models understandable by all business actors (Business analysts, Technical developers, final users…). One of the latest languages that verifies these criteria is Business Process Modeling and Notation (BPMN 2.0) (OMG, 2011). It is an adopted standard in both academia and industry that was designed to provide a graphical notation for XML-based business process languages, like Business Process Execution Language (BPEL) (OASIS, 2007). However, BPMN defines the execution semantics of flow elements with their data needs and data results only informally, in a textual representation (Stackelberg et al., 2014), which limits verification to using solely informal techniques such observation and inspection. Formal methods help us avoid flow control anomalies as well as data flow errors. Since several approaches have addressed the control flow problems (deadlock, livelock …) (Dijkman et al., 2007),(PYH. Wong, 2008),(J.Ye et al., 2008),(Rachdi et al., 2016), we will focus in this paper on formal methods that deal with dataflow anomalies (Missing, lost, redundant and inconsistent data…). In order to analyze formally dataflow in BPMN, we usually define a mapping from the graphical notation to a formal language such as Petri Nets (PN). If we adopt the Petri net approach explained in (Stackelberg et al., 2014), we would have to go through several and complex steps (Definition of mapping & Unfolding rules, BPMN to PN transformation, Process-specific anti-patterns generation, Model checking …) before we get the dataflow errors made in the BPMN model. Therefore, we have taken a different approach that can detect the anti-patterns representing dataflow anomalies using data record concept (Kabbaj et al., 2015). The latter is direct and simple and explains to the designer the origin of the anomaly so he/she can fix it easily in remodeling phase.In addition, this concept does not need several operations to complete the desired analysis.

The remainder of this paper is structured as follows: The next section proposes the work related to our approach. Section 3 introduces the technical background needed for the rest of the paper; it is divided into three major parts. The first one presents BPMN elements as well as their main properties. The second one presents the dataflow anti-patterns that have to be avoided during the design phase and the last one introduces the datarecord concept (Kabbaj et al., 2015). Section 4 presents our contribution for the

Rachdi, A., En-Nouaary, A. and Dahchour, M.
DataFlow Analysis in BPMN Models.
DOI: 10.5220/0006271202290237
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 229-237
ISBN: 978-989-758-248-6

analysis and the verification of BPMN models by introducing a formalization of the dataflow anti-patterns as well as proposing an algorithm that detects these dataflow errors.Section 5 concludes the paper and presents future work. An example is used throughout the paper to illustrate the proposed method.

2 RELATED WORK

Most approaches that deals with business process analysis focus on the control flow perspective and ignore the equally important data flow angle. Few work has focused on the data flow verification in business processes in general and in BPMN in particular.

Sadiq et al. (Sadiq et al., 2008) have highlighted the importance of dataflow analysis in business processes and identified seven types of data flow anomalies: redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data. However, they have proposed no formal solution to detect these anomalies.

In (Trcka et al., 2009), Trcka et al. formalized the data flow errors using computational temporal logic CTL*. The latter helps us detect dataflow errors in WFD-nets (a variante of Petri-Net) by using some standard model-checking techniques (Clarke et al., 1999). However these techniques were not elaborated explicity in this proposal.

Another approach was proposed in (Kabbaj et al., 2015), where Kabbaj et al. have tried to anticipate the data-flow errors during the modeling phase. This is accomplished by providing a tool for real-time analysis that triggered the verification process whenever a model fragment is added. Unfortunately, the proposition has only covered the exclusive paths (i.e, XOR branches) and not parallel paths (i.e, And-Split), which leaves the inconsistent data anomaly uncovered during the verification process.

To the best of our knowledge, (Stackelberg et al., 2014) is the only approach that has treated the data flow angle in BPMN Process models, in which Stackelberg et al. have used an extension of the BPMN-Petri-net mapping to include the data dimension in their analysis. The data flow errors were formalized using CTL and detected using a specific model checker. However, this proposal is complex and has to go through several steps (Definition of mapping & Unfolding rules, BPMN to PN transformation, Process-specific anti-patterns generation, Model checking ...) in order to detect the data flow errors.

In our paper, we take a different approach that can detect the anti-patterns representing dataflow anomalies using data record concept (Kabbaj et al., 2015). The latter is clear and simple and proposes some solutions to the designer so he can avoid these data flow errors. In addition, this concept is used conjointly with a specific algorithm (one Procedure) to detect the maximum of data flow errors while exploring the exclusive and parallel paths of a business process.

3 BACKGROUND

This section presents the different concepts and specifications used in this paper namely, BPMN, Dataflow anti-patterns and DataRecord.

3.1 Business Process Management & Notation (BPMN)

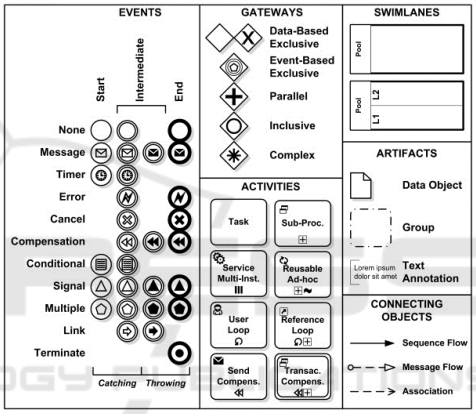

Figure 1: BPMN elements.

BPMN is a recent notation standardized by OMG and BPMI (OMG, 2011). It is considered user-friendly to all company stakeholders (business analysts, technical developers, final users ...). It has received a lot of interest and support from academia, industry. BPMN provides users with a range of diverse components. They are divided into four sets: flow objects (activities, events and gateways), connection objects (control flow, message flow and associations), artifact objects (data stores, data objects, data input and data output) and swim lanes (pools and lanes within pools) as illustrated by figure 1.

Events can be partitioned into disjoint sets of start events, intermediate events (intermediate message, time and error events) and end events. A start event indicates the start of a process while an end event represents the end of a process. An intermediate event is used to indicate that something might happen during the execution of a process. An activity is either a task or a subprocess that can be used to provide some business service, wait for a message from another partici-

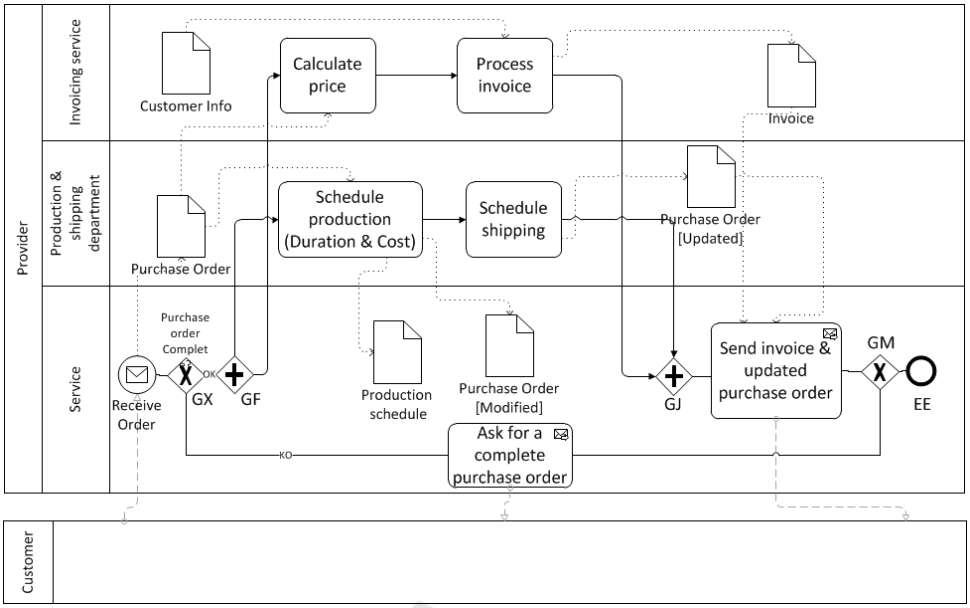

Figure 2: A Simple Order processing modeled in BPMN.

pant, or send a message to another participant. A gateway is a connector used to control sequence flows. We distinguish between multiple types of gateways: an AND-split gateway is used to create parallel flows and an AND-join gateway is used to synchronize incoming parallel flows (Prandi et al., 2008). A XOR data-based gateway defines a set of alternative paths; each of them is associated with a conditional expression. Based on this condition, only one path can be taken during the execution of the process.Conditions can be based either on data-base entries or on external events. An exclusive merge gateway is used as a merge for alternative sequence flows (Prandi et al., 2008).

In BPMN, we use artifacts objects (especially data objects) to model documents (OMG, 2011), data, and other objects that are read, created or updated during the process flow. They can be mandatory or optional (Stackelberg et al., 2014) but in case of they are mandatory, they do interfere in the correct execution of tasks or events. Therefore, in the rest of the paper, we focus only on the mandatory data objects and not on the optional ones.

An example of BP is shown in Figure 2. It represents a simple order processing, with four participants namely: the Customer, the Front-office Service, The Production department and the Invoicing. This model was elaborated based on the following scope statements:

- The provider service receives a purchase order from the customer
- If the purchase order is not complete, the service asks for a complete one and the process is completed.
- If it is complete, the provider's production department generates the production schedule (cost & duration) based on which we update the purchase order.
- In parallel, the invoicing service starts calculating the total price that will be included in the invoice in addition to the customer infos.
- Once the production schedule is produced, the production's department starts preparing the shipping schedule.
- The service can not send the invoice and the updated purchase order (containing informations about the shipping and production schedule) to the customer unless the invoice and the shipping schedule are both generated by their related activities.

Hereafter, we give BPMN a formal definition that takes into consideration the data and control flow components that are highlighted in this paper;

Definition 1 (Core BPMN Process): A core BPMN process is a tuple P=(O_P, F_P, $Data_P$, $Indata_P$, $Outdata_P$) where :

- O_P is a set of flow objects, which can be partitioned into disjoint sets of activities A_P, events E_P, and gateways G_P,(Dijkman et al., 2007)
- E_P can be partitioned into disjoint sets of start events E_P^S, intermediate events E_P^I, and end events

E_P^E(Dijkman et al., 2007). Intermediate events E_P^I may be partitioned into catch events (e.g., receive message event) and throw events (e.g., send message event) (OMG, 2011).

- G_P can be partitioned into disjoint sets of parallel fork gateways G_P^F , parallel joint gateways G_P^J , data-based XOR decision gateways G_P^X , event-based XOR decision gateways G_P^V , and XOR merge gateways G_P^M (Dijkman et al., 2007),
- $Data_P$ is a set of Data types which can be partitioned into disjoint sets of Data objects DO_P, Messages $Mess_P$ and Flow objects properties PR_P.
- $F_P \subseteq O_P \times O_P$ is a set of sequence flows.
- $Indata_P : A_P \cup E_P \rightarrow \mathcal{P}(DO_P)$ is the function that assigns to each activity or event a set of input data objects that is defined in the activity's InputOutputspecification or in the End and Intermediate throw events properties.(OMG, 2011).($\mathcal{P}(DO_P)$ is the powerset of DO_P)
- $Outdata_P : A_P \cup E_P \rightarrow \mathcal{P}(DO_P)$ is the function that assigns to each activity or event a set of output data objects defined in the activity's InputOutputspecification or in the Start and Intermediate Catch events properties.(OMG, 2011).

F_P^* is a reflexive transitive closure of F_P, i.e. xF_P^*y if there is a path from x to y and by definition xF_P^*x (Dijkman et al., 2007).

A core BPMN process P is a directed graph with nodes (objects) O_P and arcs (sequence flows) F_P. Output nodes of x are given by out(x) = { y $\in O_P$ | xF_Py }(Dijkman et al., 2007).

In the rest of the paper, we assume the BPMN process has only one start event e_s and one end event e_e. On one hand, processes with multiple start events are not considered a good practice since their semantics are ambiguous and not clear in the BPMN specification. On the other hand we can always transform BPMN processes with multiple end events to a one with single one end event (Vanhatalo et al., 2008).

We define Pa=$(e_s, SO, e_e) \in (E_P^S \times \mathcal{P}(O_P \backslash (E_P^S \cup E_P^E)) \times E_P^E)$ as a path in P if :

- $SO = \emptyset$ and $e_s F_P e_e$ OR
- $\exists n \in \mathbb{N}^*$, $\exists (o_1, \ldots, o_n) \in (O_P \backslash (E_P^S \cup E_P^E))^n$ such as $SO = \{o_1, \ldots, o_n\}$ and $e_s F_P o_1$, …,$o_n F_P e_e$. A start event is linked to an end event through flow objects that are connected to each other by means of directed sequence flows.

  e.g., Pa_1=(Receive Order,GX,GF,Calculate price,Process invoice,GJ,Send invoice & updated purchase order,GM,EE), Pa_2=(Receive Order,GX,Ask for a complete purchase order,GM,EE) are two paths in the Order Processing example presented in Figure 2.

A trace Tr is an ordered finite set of flow objects that exist on the same path Pa.We call $SPa_P \subseteq (E_P^S \times \mathcal{P}(O_P \backslash (E_P^S \cup E_P^E)) \times E_P^E))$ the set of paths in process P.

We define the operation "And" between paths as follows : $\forall i \in [1,n]$ $Pa_i = (e_s, SO_i, e_e) \in SPa_P$: $(Pa_1 And \ldots And Pa_n) = (e_s, \cup_{i=1}^{i=n} SO_i, e_e)$

An instance Γ in a Business process "P" is a set of activities and events that are executed from the start event e_s to the end event e_e. These activities and events are not necessarily connected to each other via sequence flows. This is due to the existence of parallel activities and events. These latter are not linked to each other via sequence flows.

An instance Γ of a process "P" can be a simple path Pa or a set of parallel paths $(Pa_1, Pa_2, \ldots Pa_n)$ linked to each other via the operation "And". This proposition states that all activities and events that exist on the path Pa or on the paths $(Pa_1, Pa_2, \ldots Pa_n)$ have to be executed in order to reach the completion state of the BPMN process.Formally, we write:

$$\begin{cases} \Gamma = Pa \quad with \quad Pa \in SPa_P \quad OR \\ \Gamma = (Pa_1 And Pa_2, \ldots And Pa_n) = (e_s, \cup_{i=1}^{i=n} SO_i, e_e) \\ with \quad (Pa_1, \ldots, Pa_n) \in SPa_P^n \end{cases}$$

with $Pa = (e_s, SO, e_e)$ and $\forall i \in [1,n]$ $Pa_i = (e_s, SO_i, e_e)$.

e.g., Γ_1=(Receive Order,GX,GF,Calculate price,Process invoice,Schedule production,schedule shipping,GJ,Send invoice & updated purchase order,GM,EE) and Γ_2=(Receive Order,GX,Ask for a complete purchase order,GM,EE) are both instances in the order processing example.However, Γ_2 is a path and Γ_1 is not.

We call Γ_P the set of instances of a Process "P". We define A_P^Γ (resp E_P^Γ) the activities (resp the events) that take place during an instance Γ of a process "P" .

We present below a DFS (Depth-First-Search) algorithm, adapted for the BPMN standard to calculate all instances of a process "P". The result of the algorithm will be as the following :Γ_1 XOR Γ_2 … XOR Γ_n. Γ_i are exclusive instances of P, i.e, $\forall (i,j) \in [1,n]^2$ $A_P^{\Gamma_i} \cup E_P^{\Gamma_i} \neq A_P^{\Gamma_j} \cup E_P^{\Gamma_j}$ It remains to be noted that in instance Γ of a process "P", the flow object that are connected via sequence flows, are ordered, however, the order of the activities that are parallel is not known until runtime.

Algorithm 1: Algorithm for the calculation of BPMN Model instances.

Data: A_P,E_P,G_P,F_P
Result: Γ_1,$\Gamma_2 \ldots \Gamma_n$

1 Tracehistory $\leftarrow \emptyset$ // keeps track of all traces,it is used to avoid loops and store different traces that took place in the algorithm
2 Add e_s to Tr and Tr to Tracehistory // e_s : start event of P
3 endeventreached=false // a boolean variable to indicate that we have reached the end of a path
4 CalculateBPMNModelPaths(e_s, Tr)// Main function that generates an expression ''Ex'' contaning a different combinations of paths using the operators ''XOR'' and ''AND'' : e.g, Ex=(Pa_1 XOR Pa_2) AND (Pa_3 XOR Pa_4)
5 Transform_paths_into_instances // develop the generated expression Ex into the form Γ_1 XOR Γ_2 ...XOR Γ_n

Function: CalculateBPMNModelPaths(flow object F_O,trace Tr).

1 **forall the** $i \in Out(F_O)$ **do**
2 **if** *endeventreached* **then**
3 Save the path (tr) to an array then add the ”XOR” or ”AND” operator according to the F_O's type ($F_O \in G_P$ if this IF clause is verified))
4 Tr' = Gettrace (F_O, Tr) // returns a new trace Tr′ = Tr− all flowobjects that came after F_O if the latter exists in the trace, otherwise it returns Tr
  b = contains (Gettrace(i, Tr' ∪ i),Tracehistory) // verifies if the trace ''Gettrace(i, Tr′ ∪ i)'' has taken place in the algorithm
  if b **then**
5 go to the next iteration in for loop
6 add i to Tr' and Tr' to Tracehistory
  if $i \in E_P^E$ **then**
7 endeventreached $\leftarrow$ true
8 **else**
9 endeventreached $\leftarrow$ false
10 CalculateBPMNModelinstances(i,Tr')

3.2 Dataflow Anti-patterns

In this subsection, we present data-flow anti-patterns (DAP).

- DAP 1 (Missing Data) : This anti-pattern takes place when we try to access, (i.e. read) some data object d that has never been created (i.e written) before by any activity or event in the process (Trcka et al., 2009).
- DAP 2 (Inconsistent Data) : A data object d is considered inconsistent if an activity T1 (or an event E1) is using d while some other activity T2 (event E2) is writing to d in parallel (Trcka et al., 2009).
- DAP 3 (Weakly (resp Strongly) Redundant Data) : A data object d is considered weakly (resp strongly) redundant if **there exists a certain instance of ”P” in which (resp, for each instance of ”P”)** d is written by an activity T or an event E and never gets read by any activity that follows T or E in this instance (Trcka et al., 2009).
- DAP 4 (Weakly (resp Strongly) Lost Data) : A data object d is considered weakly (resp strongly) lost if **there exists a certain instance of ”P” in which (resp, for each instance of ”P”)** d is written by an activity T1 (or event E1) and is overwritten by an activity T2 (or an event E2) and never gets read between these two activities (two events) (Trcka et al., 2009)..

3.3 Data Record

The concept we adopted to detect dataflow anti-patterns in BPMN process models is “Data Record”. It is explained as follows : For each data object (Data object references are not included) we specify its state which is a couple (x, y) where x, y denote respectively the activity (event) reading, creating/updating the data object. This state is updated after each completion of an activity or an event that exists on a certain path. If x, or y is 0 it respectively means that the data object is not read nor written by any activity otherwise we put the activity (event) that has respectively read, written the data object.In BPMN example shown in Figure 2, the invoice‘state after completion of the process is (Send invoice,Process invoice).

4 OUR APPROACH FOR BPMN ANALYSIS

As mentioned so far, BPMN has a great success in the industrial world; However it has not been pro-

vided with a formal semantics, which limits business process verification to using informal techniques such as observation. Therefore, we need to define semantics for BPMN in order to analyze business processes properly. Our analysis will be focused on the dataflow anomalies mentioned above. We will use the data record concept to formalize the four dataflow anti-patterns mentioned in the previous section as well as to detect the data flow errors using a specific algorithm adapted for the BPMN standard .

4.1 Dataflow Anti-patterns Formalization

In order to express formally the dataflow anti-patterns, we will need to define the following functions (Γ is an instance of the process "P"):

$$DOstate_P^\Gamma : DO_P \times O_P \cup \{begin\} \mapsto (A_P^\Gamma \cup E_P^\Gamma \cup \{0\})^2 \\ : (d, F_O) \mapsto (x, y).$$

where couple (x,y) reflects the state of a data object d after passing through the flow object F_O in an instance Γ.

We note the event "begin" before the begining of the process (before start event) such as : $\forall \Gamma \in \Gamma_P$, $\forall d \in DO_P \; DOstate_P^\Gamma(d, \{begin\}) = (0,0)$

$$Parallel^\Gamma : A_P^\Gamma \cup E_P^\Gamma \mapsto \mathcal{P}(A_P^\Gamma \cup E_P^\Gamma) \\ : a_p \mapsto S.$$

where S is a set of activities and events that are executed in parallel to a_p in instance Γ.

- if $\exists Pa \in SPa_P$ such as $\Gamma = Pa$, $\forall a_p \in A_P^\Gamma \cup E_P^\Gamma$ $Parallel^\Gamma(a_p) = \emptyset$
- if $\exists (Pa_1, \ldots, Pa_n) \in SPa_P^n$ such as $\Gamma = (Pa_1 And Pa_2 \ldots And Pa_n) = (e_s, \cup_{k=1}^{k=n} SO_k, e_e)$ $\exists i \in [1,n]$ such as $a_p \in SO_i$ and $Parallel^\Gamma(a_p) = \cup_{j=1, j\neq i}^{j=n} (SO_j \backslash Onpath(a_p))$ with $Onpath(a_p) = \{F_O \in O_P \mid F_O F_P^* a_p$ OR $a_p F_P^* F_O\}$

$$RE : (A_P \cup E_P \cup \{0\})^2 \mapsto (A_P \cup E_P \cup \{0\}) \\ : (x, y) \mapsto x.$$

$$WR : (A_P \cup E_P \cup \{0\})^2 \mapsto (A_P \cup E_P \cup \{0\}) \\ : (x, y) \mapsto y.$$

where x (resp y) is the activity or the event that read (resp created or updated) the data object whose state equals to (x,y).

The Data errors formalization is expressed regardless of the execution order existing between the parallel actvities of a certain instance Γ

- DAP 1 (Missing Data) : A data object d is considered missing if $\exists \Gamma \in \Gamma_P$, $\exists (F_{O_1}, F_{O_2}) \in O_P \times (A_P^\Gamma \cup E_P^\Gamma)$ that satisfy these conditions :
  - $DOstate_P^\Gamma(d, F_{O_2}) = (F_{O_2}, 0)$ $DOstate_P^\Gamma(d, F_{O_1})) = (0,0)$ and $(F_{O_1}, F_{O_2}) \in F_P$
  - $\forall a_p \in Parallel^\Gamma(F_{O_2}) \quad d \notin Outdata_P(a_p)$

  The first condition guarantees that no preceding flow object has initialized the data object d while the second condition verifies that no other parallel activity or event creates the object d.

- DAP 2 (Inconsistent Data) : A data object d is considered inconsistent if $\exists \Gamma \in \Gamma_P$,$\exists F_O \in (A_P^\Gamma \cup E_P^\Gamma)$ $\exists a_p \in Parallel^\Gamma(F_O)$ that satisfy one of these conditions :
  - $d \in Indata_P(a_p) \cap Outdata_P(F_O)$ OR
  - $d \in Outdata_P(a_p) \cap Indata_P(F_O)$ OR
  - $d \in Outdata_P(a_p) \cap Outdata_P(F_O)$

- DAP 3 (Weakly (resp Strongly) Redundant Data) : A data object d is considered weakly (resp strongly) redundant if $\exists (resp \;\; \forall) \Gamma \in \Gamma_P$, $\exists F_O \in (A_P^\Gamma \cup E_P^\Gamma)$ such as :
  - $DOstate_P^\Gamma(d, F_O) = (x, F_O)$ and $DOstate_P^\Gamma(d, e_e) = (x, F_O)$
  - $\forall a_p \in Parallel^\Gamma(F_O) \quad d \notin Indata_P(a_p) \cup Outdata_P(a_p)$

  The first condition guarantees that no activity or event will read the data object d after going through F_O and its parallel activities. While the second condition verifies the consistency of d, i.e, no other parallel activity of F_O reads or writes to d while it is being written by F_O

- DAP 4 (Weakly (resp Strongly) Lost Data) : A data object d is considered weakly (resp strongly) lost if $\exists (resp \; \forall) \Gamma \in \Gamma_P, \exists (F_{O_1}, F_{O_2}) \in (A_P^\Gamma \cup E_P^\Gamma)^2$ such as :
  - $DOstate_P^\Gamma(d, F_{O_1}) = (x, F_{O_1})$, $DOstate_P^\Gamma(d, F_{O_2}) = (x, F_{O_2})$ and $F_{O_1} F_P^* F_{O_2}$
  - $\forall a_p \in Parallel^\Gamma(F_{O_1}) \cup Parallel^\Gamma(F_{O_2}) \quad d \notin Indata_P(a_p) \cup Outdata_P(a_p)$

  The first condition guarantees that no activity or event reads the data object d after being written by F_{O_1} and before being updated by F_{O_2}.While the second condition verifies the consistency of d, i.e, no other parallel activity of F_{O_1} (resp F_{O_2}) reads or writes to d while it is being written by F_{O_1} (resp F_{O_2}).

4.2 Running Dataflow Analysis Algorithm

The algorithm used to detect dataflow anomalies is shown below. We start executing the algorithm by initializing all data objects state to the couple (0,0). Then, for each instance Γ in Γ_P (already calculated by Algorithm 1), and for each new flow object encountered in this instance, the algorithm updates the data objects states, then checks if the DAPs have taken place or not.The "Verifyinconsistency" subfunction verifies the consistency of a dataobject d before we checked it against the other three DAP.

Algorithm 2: Dataflow analysis of BPMN models.

Data: A_P,E_P,G_P,F_P,DO_P,Γ_P
Result: Dataflow errors

1 Data anomalies $\leftarrow \emptyset$ // Data anomaly=(DataObject d ,Flow Object F_O , Trace Tr, Data anomaly)
2 **forall the** $\Gamma \in \Gamma_P$ **do**
3 Tr $\leftarrow \emptyset$
4 **forall the** $d \in DO_P$ **do**
5 $(x_d, y_d) \leftarrow (0,0)$
6 **forall the** $F_O \in \Gamma$ **do**
7 add F_O to Tr
  UpdateSaveDatastate(F_O,Tr) // It updates the data objects state after passing throw F_O in a trace Tr ($x_d \leftarrow F_O$ if d $\in$ Indata(F_O) , $y_d \leftarrow F_O$ if d $\in$ Outdata(F_O))) then save it in a DO state store
  Verifydataanomalies(F_O,Tr)

Function: Verifyinconsistency(Dataobject d, flow object F_O , trace Tr).

1 **forall the** $a_p \in Parallel(F_O)$ **do**
2 **if** $(d \in Indata_P(a_p)\&\&Outdata_P(F_O)) \;||\; (d \in Outdata_P(a_p)\&\&Indata_P(F_O)) \;||\; (d \in Outdata_P(a_p)\&\&Outdata_P(F_O))$ **then**
3 we add (d,F_O,Tr,"Inconsistent data") to Data anomalies
4 Problem : d is written and read simultaneously by F_O and a_p

Function: Verifydataanomalies(flow object F_O , trace Tr).

1 **forall the** $d \in DO_P$ **do**
2 **if** *Verifyinconsistency(d,F_O,Tr)* **then**
3 go to another iteration
  // Missing Data
4 **if** *DOstate(d,F_O,Tr))=(F_O,0) && DOstate(d,in(F_O,Tr),Gettrace(in(F_O,Tr),Tr))*
5 *=(0,0)* // In(F_O,Tr) returns the input flow object F'_O of F_O in a trace Tr
  // DOstate(d,F_O,Tr) returns the state of d (x_d, y_d) after passing F_O in a trace Tr
6 **then**
7 we add (d,F_O,Tr,"Missing data") to Data anomalies
8 Problem : d is never created before F_O
  // Lost Data
9 **if** *WR(DOstate(d,F_O,Tr)))=F_O* **then**
10 **forall the** $i \in Tr\text{-}\{F_O\}$ **do**
11 **if** *WR(DOstate(d,i,Gettrace(i,Tr))))= i && RE(DOstate(d,i,Gettrace(i,Tr))))= RE(DOstate(d,F_O,Tr)) && (!Verifyinconsistency(d,i,*
12 *Gettrace(i,Tr)))* **then**
13 we add (d,F_O,Tr,"Lost data") to Data anomalies.
  Problem : d is never read or accessed between F_O and i
  // Redundant Data
14 **if** $F_O \in E_P^E$ **then**
15 WR(DOstate(d,F_O,Tr))) = y **if** *RE(DOstate(d,y,Gettrace(y,Tr)))= && RE(DOstate(d,F_O,Tr))) && (!Verifyinconsistency(d,y,*
16 *Gettrace(y,Tr)))* **then**
17 we add (d,F_O,Tr,"Redundant data") to Data anomalies.
  Problem : d is never read after the execution of F_O

To illustrate the advantages of our approach, we consider again the example of Figure 2.We consider only the instance Γ_1 (purchase order complete), Γ_2 is irrelevant to our case.A simple investigation of the model makes it easy to notice that several dataobjects were subject to dataflow anomalies (Section 3.2).

To make this analysis formal, we will base our study on the aformentioned Data record concept, we

Table 1: Data flow errors occured in the Order processing model.

Flowobject	Trace Tr	DOstates	Data Anomalies
begin	$\emptyset$	Purchase order={0,0}, CustomerInfo={0,0}, Invoice={0,0}, Production schedule={0,0}	$\emptyset$
Receive order	Tr $\cup${Receive order}	Purchase order={0,Receive order}, CustomerInfo={0,0}, Invoice={0,0}, Production schedule={0,0}	$\emptyset$
GX	Tr $\cup$ {GX}	Purchase order={0,Receive order}, CustomerInfo={0,0}, Invoice={0,0}, Production schedule={0,0}	$\emptyset$
GF	Tr $\cup$ {GF}	Purchase order={0,Receive order}, CustomerInfo={0,0}, Invoice={0,0}, Production schedule={0,0}	$\emptyset$
Calculate price	Tr $\cup$ {Calculate price}	Purchase order={Calculate price,Receive order},CustomerInfo={0,0}, Invoice={0,0}, Production schedule={0,0}	{(Purchase order, ”Inconsistent data”)}
Process invoice	Tr $\cup$ {Process invoice}	Purchase order={Calculate price,Receive order}, CustomerInfo={Process invoice,0}, Invoice={0,Process invoice}, Production schedule={0,0}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
Schedule production	Tr $\cup$ {Schedule production}	Purchase order={Schedule production,Schedule production}, CustomerInfo={Process invoice,0}, Invoice={0,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
Schedule shipping	Tr $\cup$ {Schedule shipping}	Purchase order={Schedule production,Schedule shipping}, CustomerInfo={Process invoice,0}, Invoice={0,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
GJ	Tr $\cup$ {GJ}	Purchase order={Schedule production,Schedule shipping}, CustomerInfo={Process invoice,0}, Invoice={0,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
Send invoice & updated purchase order	Tr $\cup$ {Send invoice & updated purchase order}	Purchase order={Send invoice & updated purchase order,Schedule shipping}, CustomerInfo={Process invoice,0}, Invoice={Send invoice & updated purchase order,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
GM	Tr $\cup$ {GM}	Purchase order={Send invoice & updated purchase order,Schedule shipping}, CustomerInfo={Process invoice,0}, Invoice={Send invoice & updated purchase order,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”)}
End event	Tr $\cup$ {End event}	Purchase order={Send invoice & updated purchase order,Schedule shipping}, CustomerInfo={Process invoice,0}, Invoice={Send invoice & updated purchase order,Process invoice},Production schedule={0,Schedule production}	{(Purchase order, ”Inconsistent data”),(Customer Info, ”Missing data”),(Production schedule, ”Redundant data”)}

can verify the made errors by executing the Algorithm detailed above. In the previous table we have the obtained results. "Purchase order", "Customer Info", "Production schedule", were subject to many errors at different stages of the analysis. Consequently, some modifications related these dataobjects have to be made in order to guarantee a safe dataflow through the process:

- "Customer Info" : It has to be created by an activity that preceeds the "process invoice" task, (e.g, the start event after receiving a message from the customer).
- "Purchase order" : It should not be used by tasks (or events) that are in parallel. we would rather establish an sequential order (instead between activities that write and read simultaneously this data object.
- "Production schedule" : It should not figure in the "schedule production" task if it is not used in the process nor send to another participant.

5 CONCLUSION

In this paper, we proposed a formal dataflow analysis of BPMN models based on Data Record concept. The suggested approach allows us to detect four dataflow anti-patterns. The Order processing was taken as an example to illustrate the advantages brought by this method. In our future work, we intend to include the OR-join and OR-split semantics in our analysis, which will extend the area of covered business processes.We are working also on the implementation of this method into an Eclipse Plugin called the "BPMN process Analysis".

REFERENCES

Clarke et al. (1999). *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK.

Dijkman et al. (2007). Formal semantics and analysis of BPMN process models using petri nets. Technical Report 7115, Tech Univ QLD, Brisbane.

J.Ye et al. (2008). Transformation of BPMN to yawl. In *International Conference on Computer Science and Software Engineering*, pages 354–359, Wuhan, China.

Kabbaj et al. (2015). Towards an active help on detecting data flow errors in business process models. *International Journal of Computer Science and Applications*, 12:16–25.

OASIS (2007). *Web Services Business Process Execution Language*. Burlington, USA.

OMG (2011). *Business Process Management and Notation (BPMN 2.0)*. Needham,USA.

Prandi et al. (2008). Formal analysis of bpmn via a translation into cows. In *10th COORDINATION Intenrnational Conference ICFEM*, pages 249–263, Oslo, Norway.

PYH. Wong, J. G. (2008). A process semantics for BPMN. In *10th Formal Engineering Methods ICFEM*, pages 355–374, Kitakyushu, Japan.

Rachdi et al. (2016). Liveness and reachability analysis of BPMN process models. *CIT. Journal of Computing and Information Technology*, 24:195–207.

Sadiq et al. (2008). Data flow and validation in workflow modelling. In *15th Australasian database conference*, pages 207–214, Dunedin, New Zealand.

Stackelberg et al. (2014). Detecting data-flow errors in BPMN 2.0. *Open Journal of Information Systems*, 1:1–19.

Trcka et al. (2009). Data-flow anti-patterns: Discovering data-flow errors in workflows. In *Advanced Information Systems Engineering (21st International Conference, CAiSE)*, pages 425–439, Amsterdam, The Netherlands.

Vanhatalo et al. (2008). Automatic workflow graph refactoring and completion. In *6th International Conference on Service-Oriented Computing*, pages 100–115, Sydney, Australia.