Checking Business Process Evolution

Ajay Krishna, Pascal Poizat, Gwen Salaün

Checking Business Process Evolution

Ajay Krishna[a], Pascal Poizat[b,c], Gwen Salaün[*,d]

[a] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France*
[b] *Université Paris Lumières, Université Paris Nanterre, F-92000, Nanterre, France*
[c] *Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005, Paris, France*
[d] *Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France*

Abstract

Business processes support the design and implementation of software as workflows of local and inter-organization activities. Tools provide the business process designer with modelling and execution facilities, but they barely provide formal analysis techniques. When one makes a process evolve, for example by refactoring it or by adding new features in it, it is important to be able to check whether, and how, this process has changed, and possibly correct evolution flaws. To reach this objective, we first present a model transformation from the BPMN standard notation to the LNT process algebra and LTS formal models. We then propose a set of relations for comparing business processes at the formal model level. With reference to related work, we propose a richer set of comparison primitives supporting renaming, refinement, property and context-awareness. We also support BPMN processes containing unbalanced structures among gateways. In order to make the checking of evolution convenient for business process designers, we have implemented tool support for our approach as a web application.

Key words: Business processes, evolution, model transformation, automated verification, tool, BPMN, LNT, LTS.

*Corresponding author
Email addresses: ajay.muroor-nadumane@inria.fr (Ajay Krishna), pascal.poizat@lip6.fr (Pascal Poizat), gwen.salaun@inria.fr (Gwen Salaün)

1. Introduction

Business processes describe the production of goods or services as a set of local tasks and inter-organization exchanges. The main business process modelling notations, BPMN 2.0, UML Activity Diagrams, and Event-driven Process Chains, have a workflow perspective of business processes. BPMN 2.0 (BPMN for short in the sequel) is an ISO standardized notation for modelling business processes. Numerous tools support the design or execution of BPMN models, *e.g.*, Activity, Bonita BPMN, or the Eclipse BPMN Designer. They can be used to set up the first version of a process model, and then to make it evolve by refactoring parts of it (to optimize it or better suit partner organizations) or by adding new features in it.

Motivations. The BPMN modelling tools support basic activities on the models but performing formal analyses on them is barely found in these. Further, evolution needs a special treatment. It can be supported by a form of non-regression verification where the whole set of formal verifications (*e.g.* relative to the descriptions of expected behaviours) that has been checked on a version of a process is checked again on its evolution. It would also be interesting to have a more global vision of the process behaviour by being able to specify what evolution should / should not be in terms of observable behaviours. Our objectives are to propose to process designers a variety of formally grounded (behavioural) evolution relations between process models, and, given two process models, to support this designer with automated techniques for checking these evolution relations and more generally behavioural properties. These automated techniques should enable the designer to understand the impact of evolution and, if necessary, support the refinement of an incorrect evolution into a correct one.

Approach. To reach these objectives we develop an approach, as shown in Figure 1, based on model transformation, and on property and behavioural equivalence checking. We start with BPMN models that may have been defined in any business process IDE that can output BPMN files that conform to the standard. The process designer uses our Web application, VBPMN, to input the (one or two depending on the verification to perform) process models and the verification parameters. We then have to transform these models into formal models that can support formal analysis. This is achieved in three steps. First we transform the business processes into an intermediate process meta model and format that we propose, the Process Intermediate

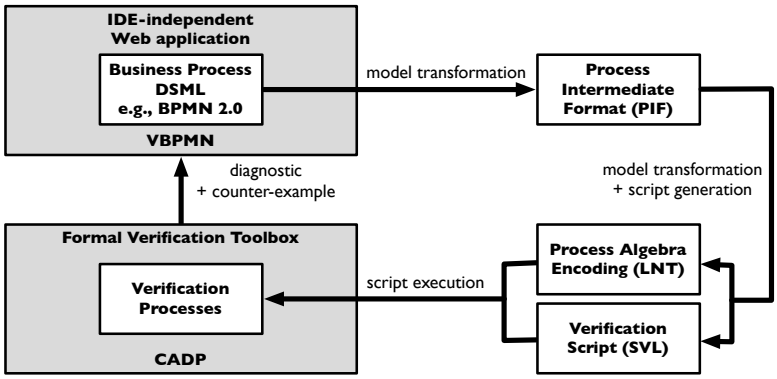


Figure 1: Framework Overview.

Format (PIF). PIF gathers a subset of the main common concepts found in business process Domain Specific Modelling Languages (DSMLs). Using PIF as an intermediate model enables one to accept in the future UML Activity Diagrams or Event-driven Process Chains as inputs as soon as model transformations from these DSMLs to PIF are defined. Once we have PIF models, we perform a second transformation into the LNT process algebra. This is an expressive formal language that is at the core of the CADP verification toolbox and that has an operational semantics defined in terms of Labelled Transition Systems (LTSs). We also generate automatically verification scripts in the CADP SVL script language [1]. Their content depends on the verification to be performed (checking for the absence of deadlock, for the satisfaction of a temporal property, or for the correctness of evolution). The CADP toolbox is then finally used to perform the verifications on the LTS semantics of the LNT processes and in case there is an error, a counter-example is returned to the user.

Contributions. The contributions of the work presented here are as follows:

- We present a model transformation from business processes defined in the BPMN standard to LTS formal models. This transformation supports the main gateways found in BPMN and is able to deal with unbalanced workflow models.

- We define a set of evolution relations for business processes that are grounded on formal behavioural relations.

- We propose an abstract intermediate meta model and format for busi-

ness processes, PIF. It contains common process workflow concepts found in different business process DSMLs, thus opening the possibility to apply our approach to several of these DSMLs.

- We introduce VBPMN, a freely available tool [2] that implements the model transformation and that enables, through a Web application, to check for the evolution of business process and get informative feedback in case of an error.

Outline. Section 2 introduces the BPMN business process modelling language and the running example we will use for illustration purposes. The transformation from BPMN models to the formal models we use for checking evolution is presented in Section 3, together with PIF, a business process meta model that plays an intermediate role between business process modelling languages (such as BPMN) and verification models (such as LTS). In Section 4 we then formally define several behavioural relations that can be used to compare business processes and, accordingly, to check business process evolution. Section 5 addresses the implementation of the outcomes of the previous sections. We present there VBPMN, our Web application for business process verification and some experimental data on the use of its core verification module. Finally, Section 6 reviews related work and Section 7 concludes the article.

2. BPMN

In this section, we give a short introduction on BPMN. We then present the running example we will use for illustration purposes in the rest of this paper.

BPMN is a workflow-based graphical notation (Fig. 2) for modeling business processes that can be made executable either using process engines (*e.g.*, Activiti, Bonita BPM, or jBPM) or using model transformations into executable languages (*e.g.*, BPEL). BPMN is an ISO/IEC standard since 2013 but its semantics is only described informally in official documents [3, 4]. Therefore, several attempts have been made for providing BPMN with a formal semantics, *e.g.*, [5, 6, 7, 8, 9]. In this paper, we abstract the features of BPMN related to data and we focus on the core features of BPMN, *i.e.*, its control flow constructs, which is the subset of interest with respect to the properties we propose to formally analyse in this paper. More precisely,

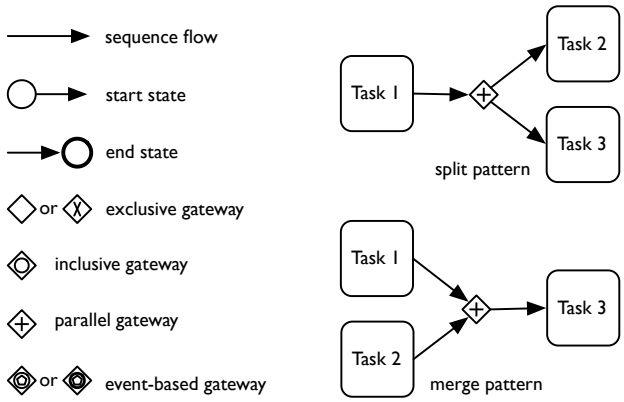


Figure 2: BPMN Notation (part of).

we consider the following categories of workflow nodes: *start* and *end event*, *tasks*, and *gateways*.

Start and end events are used to denote respectively the starting and the ending point of a process. A task is an abstraction of some activity and corresponds in practice, *e.g.*, to manual tasks, scripted tasks, or inter-process message-based communication. In our context, we use a unique general concept of task for all these possibilities. Start (end, resp.) events must have only one outgoing (incoming, resp.) flow, and tasks must have exactly one incoming and one outgoing flow.

Gateways are used, along with sequence flows, to represent the control flow of the whole process and in particular the task execution ordering. There are five types of gateways in BPMN: *exclusive, inclusive, parallel, event-based* and *complex gateways*. An exclusive gateway is used to choose one out of a set of mutually exclusive alternative incoming or outgoing branches. It can also be used to represent looping behaviours. For an inclusive gateway, any number of branches among all its incoming or outgoing branches may be taken. A parallel gateway creates concurrent flows for all its outgoing branches or synchronizes concurrent flows for all its incoming branches. For an event-based gateway, it takes one of its outgoing branches based on events (message reception). Finally, complex gateways are used to model complex synchronization behaviours especially based on data control. If a gateway has one incoming branch and multiple outgoing branches, it is called a *split* (gateway). Otherwise, it should have one outgoing branch and multi-

Table 1: Analysis of the BPMN elements found in the BIT process library, release 2009.

category	element	occurences	present in files
flow	sequence flow	35.082	825/825
gateway	parallel gateway	11.175	715/825
task	task	7.759	825/825
event	end event	3.533	825/825
event	start event	3.027	825/825
gateway	exclusive gateway	1.956	478/825
structure	*sub-process*	*883*	*58/825*
structure	definitions	825	825/825
structure	process	825	825/825
gateway	inclusive gateway	135	47/825

ple incoming branches, and it is called a *merge* (gateway).

We support workflows that exhibit an unbalanced structure between split-merge gateways. More precisely, this means that any merge gateway does not have necessarily a corresponding split gateway, that is, with the same type and with the same number of branches. We require that BPMN processes are syntactically correct, which is checked by ensuring that the BPMN model conforms to BPMN 2.0 specification. Moreover, although specific processes are syntactically correct, they may be semantically flawed. Those erroneous models are usually refered as anti-patterns [10]. This is the case for instance when a process exhibits a looping behaviour coming back in-between a parallel/inclusive split and merge gateway. In that case, the semantical model will be infinite. This problem is well-known in process algebra, which usually forbids recursive agent calls through parallel composition operators (referred as *finite control property* [11]). In our work, this case is detected by applying a pre-processing check and then discarded before model transformation and analysis. The pre-processing traverses the process and, for each parallel or inclusive merge gateway involved in an unbalanced structure, checks whether that gateway is inside a cycle.

Limitations. BPMN has three main kinds of models: processes, collaborations, and choreographies. In this work we deal with the first kind, and the other two are perspectives for a future release of VBPMN (see Section 7). Different subsets of the notation, called process modeling conformance sub-

classes, are defined in the BPMN standard. In order to select a sufficient one, we did an analysis of the 825 BPMN processes available in the BIT process library, release 2009 [12]. These processes are industrial process models taken from different business domains such as finance or telecommunications. Table 1 presents the outcomes of this analysis, with the number of occurrences for each BPMN element (in the whole set of processes) and the number of processes in which at least one occurrence of the element is found. In our approach we are able to deal with all the BPMN elements we found in this analysis, but for sub-processes. This subset of BPMN that we support corresponds to what is defined as the descriptive conformance sub-class in the BPMN standard, without sub-processes and data, but with inclusive gateways. Sub-processes are a structuring mechanism in BPMN and they could possibly be removed by flattening the processes. Supporting data is of real interest, however, in the descriptive conformance sub-class, its role is unclear. There are no conditions on sequence flows going out of exclusive gateways in this sub-class for example. In the presence of conditional constructs and assignment activities, the support for data would require either to bound the data domains (which could be done using our approach) or to rely on symbolic approaches like in [13].

Example. We use the online shopping system depicted in Figure 3 as a running example. This process starts by searching items, logging in, and initiating payment (exclusive gateways, top part). Then, once the payment is completed, the availability of the ordered items is tackled (inclusive gateways, bottom part). Finally, items are shipped and delivered (exclusive gateways, bottom right part). If some item is unavailable or if the delivery fails (*e.g.*, nobody is present to receive the parcel), a refund is processed provided the payment is made using voucher or card. We can see that this workflow is unbalanced (top right and bottom part) and includes a loop (bottom right part).

3. Models and Transformations

3.1. Process Intermediate Format

Business processes may be modelled using different DSMLs. BPMN is possibly the main one but one may also consider the use of UML Activity Diagrams (UML AD) or Event-driven Process Chains (EPC) for this. The formal approach for the verification of business process evolution advocated by our framework, could indeed apply not only to BPMN, but also to

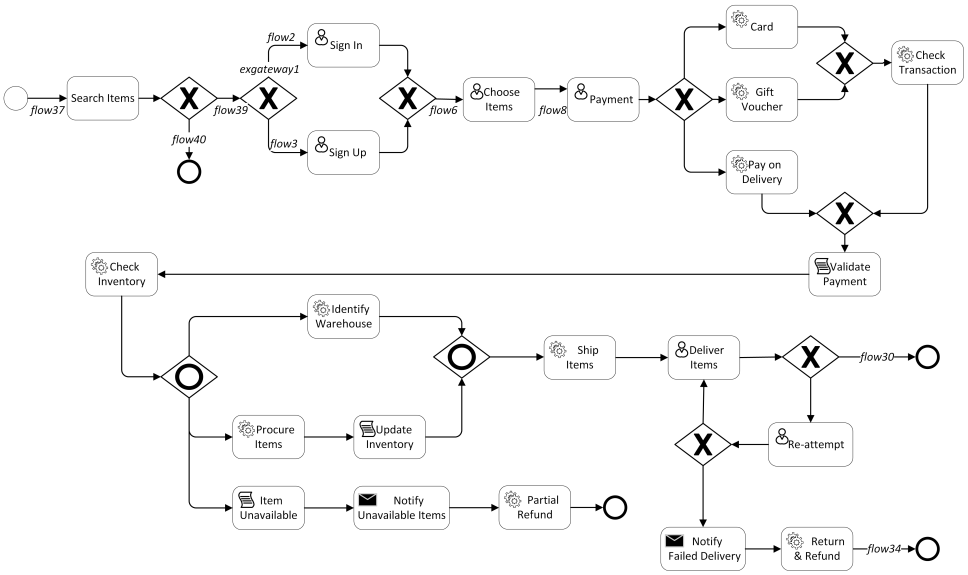


Figure 3: Online Shopping Process in BPMN (element identifiers in *italics*).

UML AD and EPC, and possibly others, as soon as one is able to retrieve two operational semantic models for the two processes to be compared defined in terms of LTSs. Provided one LTS gives the semantics of an EPC business process, and another LTS gives the semantics of a refactoring of this process in BPMN, our approach would apply.

LTS is a classic model to perform formal analysis. However, they are also quite low-level and do not make explicit the common concepts of business process DSMLs such as the workflow structure and split-join gateway patterns. Therefore, we propose to rely on a pivot meta model, the Process Intermediate Format (PIF), as an intermediate between business process DSMLs and LTS. As presented in Figure 4, PIF contains a subset of concepts found in existing workflow-based DSMLs. The Workflow Patterns Initiative, see *e.g.*, [14], has shown that the whole set of concepts in workflow-based DSMLs is larger[1] but we believe that the subset supported by PIF in its cur-

[1]One may find a correspondence at `http://www.workflowpatterns.com/evaluations/standard/` between abstract workflow concepts and their counter-

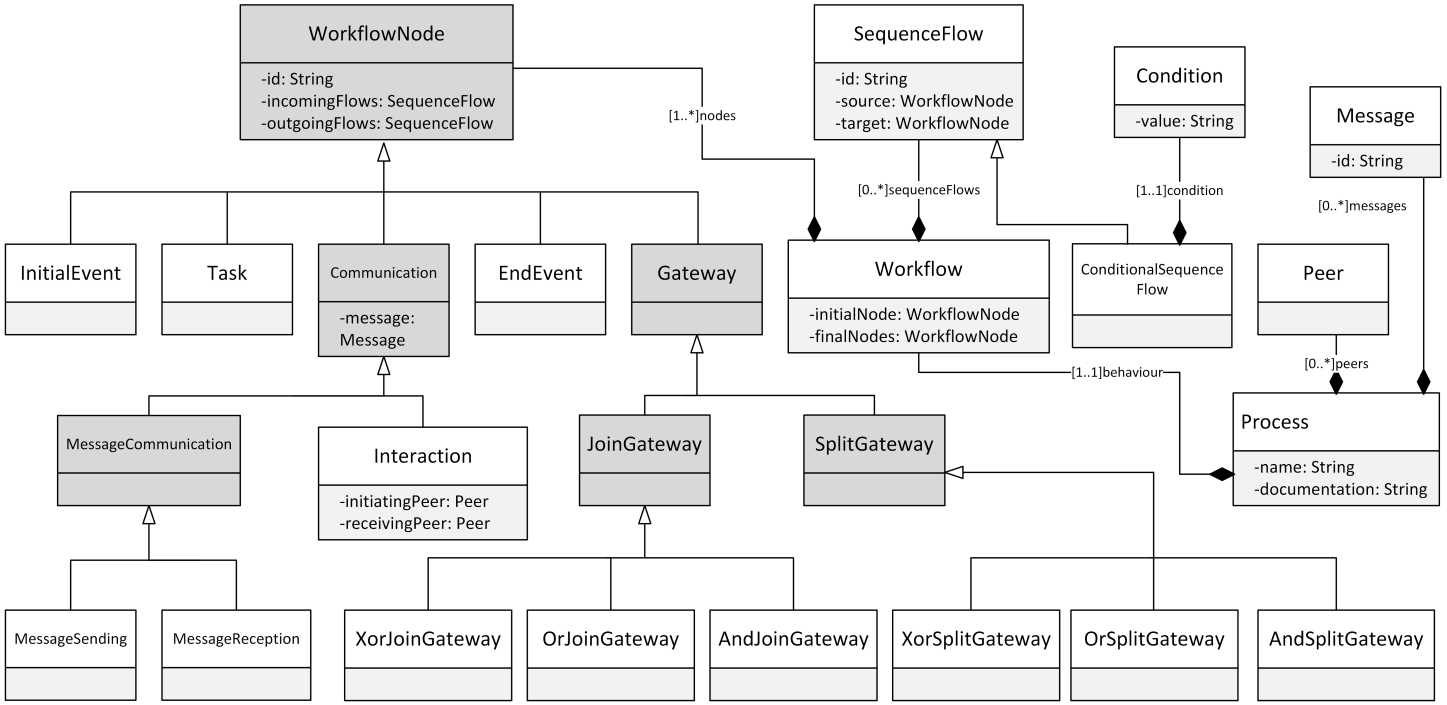


Figure 4: PIF Meta Model (abstract concepts are shown in grey).

Table 2: Mapping from BPMN to PIF.

BPMN	PIF
Start event	InitialEvent
End event	EndEvent
Tasks (several)	Task
Sequence flow	SequenceFlow
sourceRef (Seq flow)	source
targetRef (Seq flow)	target
sourceRef (Flow node)	incomingFlows
targetRef (Flow node)	outgoingFlows

BPMN gateway	PIF gateway
Parallel split	AndSplit
Parallel merge	AndJoin
Exclusive split	XorSplit
Exclusive merge	XorJoin
Inclusive split	OrSplit
Inclusive merge	OrJoin

rent version is sufficient for most process models. It is worth observing that PIF was used recently for validation of Mangrove models [15] and that a new transformation to Maude was developed for verification of timed business processes [16].

3.2. From BPMN to PIF

BPMN process models are specified using XML documents conforming to the BPMN schema. A typical BPMN model contains BPMN Diagram Definition (DD) information in addition to BPMN Model information. However, from the verification point of view, DD information is not useful. Thus, model transformation from BPMN to PIF, extracts relevant data from a BPMN model and transforms it into PIF. The Workflow element in PIF is composed of a process, WorkflowNodes and SequenceFlows. WorkflowNodes can be of five different types: InitialEvent, Task, Communication, Gateway and EndEvent. PIF assumes that there is exactly one InitialEvent and one or more EndEvent in the Workflow. Gateway element is further classified into SplitGateway and JoinGateway. Gateways can follow Or, Xor or And pattern to control the process flows. BPMN to PIF transformation patterns are described in Table 2.

As one may have noticed in Table 2, PIF has elements similar to BPMN. So, the transformation is more like a mapping of BPMN model elements to

parts in DSMLs, and the BPMN 2.0 standard refers to these workflow patterns when presenting the elements of its modelling notation.

PIF elements.

3.3. From PIF to LTS

We present here our transformation from PIF to LTS, obtained through a transformation from PIF to the LNT process algebra, LNT having an LTS semantics. We gave a semantics to each PIF construct based on the informal semantics given in the standard for the corresponding BPMN construct. Hence, using our BPMN to PIF mapping and our PIF to LNT transformation, one gets an LTS semantics for BPMN business processes. It is worth noting that the generated LTS is finite (the number of states is finite) because the BPMN/PIF process is syntactically correct and free of flawed patterns that may generate infinite models.

LNT. LNT [17] is an extension of LOTOS [18], an ISO standardized process algebra, which allows the definition of data types, functions, and processes. Table 3 provides an overview of the behavioural fragment of LNT syntax and semantics. B stands for a LNT term, G for a gate or action, E for a Boolean expression, T for a type, and P for a process name. The syntax fragment presented in this table contains the termination construct (**stop**) and gates or actions (G) that may come with offers (send an expression E or receive in a variable x). LNT processes are then built using several operators: sequential composition (**;**), conditional statement (**if**), assignment (**:=**) where the variable should be defined beforehand, hiding (**hide**) that hides some action in a behaviour, nondeterministic choice (**select**), parallel composition (**par**) where the communication between the process participants is carried out by rendezvous on a list of synchronized actions, looping behaviours described using process calls or explicit operators (**loop**, **while**).

We also present in Table 3 some examples of rules defining the operational semantics of LNT (action prefix, sequential composition, conditional construct, variable assignment, hiding operator, and choice). An action or gate G can come with offers (send offer '!' or receive offer '?'). The value received in the variable x must be of the type of x (checked with the 'type' function). Note that the received value v′ substitutes x in B. The conditional statement **if** executes when the corresponding condition is evaluated to true (expressed using $[\![.]\!]$). If the condition expression evaluates to false, it results in termination of the block without executing the condition block. Sequential composition consists of two rules. The first one corresponds to normal evolution of the B_1 behaviour. The second rule executes when the first behaviour terminates correctly (δ action). In that case, the second behaviour

(B_2) starts executing. The operational rule for variable assignment shows how the fresh variable x is substituted by the evaluated expression E in the whole behaviour B. There are two rules for the hiding operator. The first one results in a normal execution if the gate does not belong to the set of gates to be hidden. The second rule, contrarily, involves a gate that is part of the gates to be hidden and the gate transforms into an unobservable τ action. Finally, the **select** operator triggers nondeterministically one of the choice branches. The reader interested in more details about the syntax and semantics of LNT should refer to [17].

The use of LNT is preferred over the direct use of LTS since this yields a simpler, high-level and more declarative transformation. Thanks to the LTS semantics of LNT, one can use thereafter a rich set of existing tools for LTS-based verification. The choice of LNT over other process algebras has been guided by the existence of the CADP toolbox [19], which comes with a very comprehensive set of verification tools, including ones supporting the implementation of the various checks presented in the sequel.

Overview. The idea is to encode as LNT processes all PIF elements involved in a process definition, that is, the nodes (tasks, gateways), which correspond to the behaviour of the business process, initial/end events, and sequence flows, which encode the execution semantics of this process. Finally, all these independent LNT processes are composed in parallel and synchronized in order to respect the business process execution semantics. For instance, after execution of a node, the corresponding LNT process synchronizes with the process encoding the outgoing flow, which then synchronizes with the process encoding the node appearing at the end of this flow, and so on.

The encoding patterns for the main PIF constructs are described in Table 4 except for Or gateways, which are tackled separately in Tables 5 and 6. The actions corresponding to the flows (incf, outf, etc.) will be used as synchronization points between the different workflow elements. The begin and finish actions in the initial/end events are just used to trigger and terminate, respectively, these events. The actions used in task constructs (*e.g.*, task) will be the only ones to appear in the final LTS. All other synchronizations actions will be hidden because they do not make sense from an observational point of view. We do not present the encoding of communication/interaction messages in Table 4 because they are translated similarly to tasks. And gateways are encoded using the **par** LNT operator, which corresponds in this case to an interleaving of all flows. Xor gateways are encoded using the

Table 3: LNT Syntax and Semantics (Behaviour Part) [17].

$$
\begin{array}{rcl}
B & ::= & \textbf{stop} \\
& | & G(!E, ?x) \\
& | & B_1; B_2 \\
& | & \textbf{if } E \textbf{ then } B \textbf{ end if} \\
& | & \textbf{var } x{:}T \textbf{ in } x := E; B \textbf{ end var} \\
& | & \textbf{hide } G_1, ..., G_m \textbf{ in } B \textbf{ end hide} \\
& | & \textbf{select } B_1 \,[]\, ... \,[]\, B_n \textbf{ end select} \\
& | & \textbf{par } G_1, ..., G_m \textbf{ in } B_1 || ... || B_n \textbf{ end par} \\
& | & P[G_1, ..., G_m](E_1, ..., E_n) \\
& | & \textbf{loop } B \textbf{ end loop} \\
& | & \textbf{while } E \textbf{ loop } B \textbf{ end loop}
\end{array}
$$

$$
\textsc{Act}\ \frac{v' \in \mathit{type}(x)}{G(!E, ?x)\ ; B \xrightarrow{G\ ![\![E]\!]\ !v'} B\{v'/x\}}
$$

$$
\textsc{Seq-1}\ \frac{B_1 \xrightarrow{\beta} B_1'}{B_1; B_2 \xrightarrow{\beta} B_1'; B_2} \qquad \textsc{Seq-2}\ \frac{B_1 \xrightarrow{\delta} B_1' \quad B_2 \xrightarrow{\beta} B_2'}{B_1; B_2 \xrightarrow{\beta} B_2'}
$$

$$
\textsc{If-1}\ \frac{[\![E]\!] = \text{true} \quad B \xrightarrow{\beta} B'}{\textbf{if } E \textbf{ then } B \textbf{ end if} \xrightarrow{\beta} B'} \qquad \textsc{If-2}\ \frac{[\![E]\!] = \text{false}}{\textbf{if } E \textbf{ then } B \textbf{ end if} \xrightarrow{\delta} \textbf{stop}}
$$

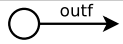
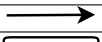
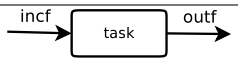
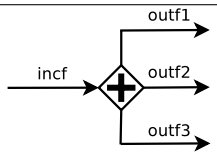
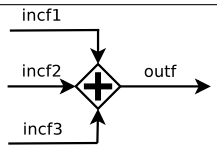
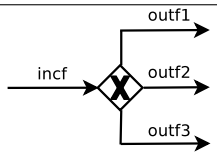
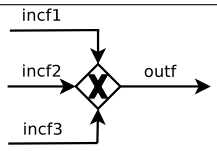
$$
\textsc{Var}\ \frac{B\{[\![E]\!]/x\} \xrightarrow{\beta} B'}{\textbf{var } x{:}T \textbf{ in } x := E; B \textbf{ end var} \xrightarrow{\beta} B'}
$$

$$
\textsc{Hid-1}\ \frac{B \xrightarrow{\beta} B' \quad \mathit{gate}(\beta) \notin \{G_1, ..., G_m\}}{\textbf{hide } G_1, ..., G_m \textbf{ in } B \textbf{ end hide} \xrightarrow{\beta} \textbf{hide } G_1, ..., G_m \textbf{ in } B' \textbf{ end hide}}
$$

$$
\textsc{Hid-2}\ \frac{B \xrightarrow{\beta} B' \quad \mathit{gate}(\beta) \in \{G_1, ..., G_m\}}{\textbf{hide } G_1, ..., G_m \textbf{ in } B \textbf{ end hide} \xrightarrow{\tau} \textbf{hide } G_1, ..., G_m \textbf{ in } B' \textbf{ end hide}}
$$

$$
\textsc{Sel}\ \frac{i \in [1, n] \quad B_i \xrightarrow{\beta} B_i'}{\textbf{select } B_1 \,[]\, ... \,[]\, B_n \textbf{ end select} \xrightarrow{\beta} B_i'}
$$

Table 4: Encoding PIF into LNT (part of, continued below).

PIF Construct	BPMN Notation	LNT Encoding
InitialEvent		begin ; outf
EndEvent		incf ; finish
SequenceFlow		**loop** begin ; finish **end loop**
Task		**loop** incf ; task ; outf **end loop**
AndSplit		**loop** incf ; **par** outf1 || outf2 || outf3 **end par end loop**
AndJoin		**loop par** incf1 || incf2 || incf3 **end par ;** outf **end loop**
XorSplit		**loop** incf ; **select** outf1 [] outf2 [] outf3 **end select end loop**
XorJoin		**loop select** incf1 [] incf2 [] incf3 **end select ;** outf **end loop**

select LNT operator, which corresponds to a nondeterministic choice among all flows. All constructs (sequence flows, tasks, gateways) are enclosed within an LNT **loop** operator since these elements can be repeated several times if the business process exhibits looping behaviours. However, the decision to repeat is not taken at this local level, but it depends of the overall workflow structure whose encoding will be presented in the rest of this section.

Composition. Once all workflow elements are encoded into LNT, the next step is to compose them in order to obtain the behaviour of the whole business process. To do so, we compose in parallel all the flows with all the other constructs. All flows are interleaved because they do not interact one with another. All events and nodes (start/end events, tasks, gateways) are interleaved as well for the same reason. Then both sets are synchronized on

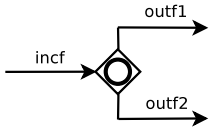
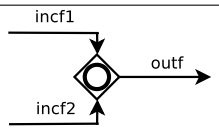
flow sequences (actions denoted by sequence flow IDs). These additional actions are finally hidden because they should not appear as observable actions and will be transformed into internal transitions in the resulting LTS. Each process call is accompanied with its alphabet, that is, the list of actions used in that process. For instance, each call of a flow process comes with a couple of actions corresponding to the initiation and termination of the flow.

Balanced Or gateways. The Or gateways in PIF correspond to the BPMN inclusive gateways. The semantics of the inclusive gateways is quite intricate in the original version of BPMN [20]. BPMN 2.0 simplifies the semantics of the inclusive merge gateway by allowing the merge to execute whenever a token arrives at the merge level. From the process algebra point of view, this can be simply encoded as an XorJoin gateway as already presented beforehand in this section. In the rest of this section, we assume that the OrJoin gateways correspond to a real synchronization point as described in the execution semantics of the first version of BPMN. We support both semantics in the tool support we will present in Section 5.

As a first step, we also assume here that each OrJoin gateway has a corresponding OrSplit gateway with the same number of branches. This is what we call *balanced* gateway that can be generalized to balanced workflows / processes if all gateways in the process exhibit a balanced structure. The encoding of the Or gateways use the **select** and **par** operators to allow all possible combinations of the outgoing branches. Table 5 describes the translation patterns in LNT. Note the introduction of synchronization points (s_i), which are necessary to indicate to the OrJoin gateway the behaviour that was executed at the OrSplit level. Without such synchronization points, an OrJoin gateway does not know whether it is supposed to wait for one or several branches (and which branches in this second case).

Unbalanced Or gateways. In case of an unbalanced structure of the workflow, we cannot use the synchronization points solution we used for balanced Or gateways. Therefore, the only solution we have is to use a global process in charge of keeping track of all active tokens and of deciding if a certain OrJoin gateway can be triggered or not. To do so, we use an additional process called *scheduler*. The scheduler runs in parallel with all the other LNT processes encoding the aforementioned workflow constructs and synchronizes with them to keep track of active flows in the whole PIF process. Each time a synchronization occurs between the scheduler and a flow process, the scheduler updates a local set of flow identifiers corresponding to

Table 5: Encoding PIF into LNT (balanced Or gateways).

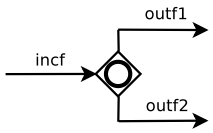
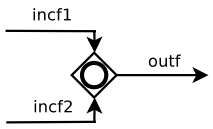
PIF Gateway	BPMN Notation	LNT Encoding
OrSplit		**loop** incf ; **select** (*s_i *if one matching merge**) outf1 ; s1 [] outf2 ; s2 [] par outf1 || outf2 **end par** ; s3 **end select end loop**
OrJoin		**loop select** (* s_i *if one matching split* *) s1 ; incf1 [] s2 ; incf2 [] s3 ; par incf1 || incf2 **end par** **end select** ; outf **end loop**

the set of active flows. This means that at any moment during the process exploration, the scheduler exactly knows all tokens currently active.

Table 6 describes the translation patterns in LNT for unbalanced Or gateways. Note that synchronization points (s_i) are still necessary at the OrSplit level to indicate to the scheduler how many branches have been fired and to enable it to capture the exact number of active tokens. One can also see that each flow action has as parameter the identifier of the flow, which is required at the scheduler level to keep track of active tokens. The process encoding the OrJoin gateway is different in case of unbalanced structure. First of all, the process is activated when a first token arrives at this level of the workflow. Concretely speaking, this corresponds to a synchronization between the OrJoin process and one of the incoming flows. Then, the OrJoin process can either accumulate tokens or can receive a message from the scheduler indicating that the merge is possible (no more tokens are to be awaiting). In that case, the scheduler moves on and synchronizes with the outgoing flow, which means that a token is generated for that flow. Note that the synchronization between the OrJoin process and the scheduler is effective only when the scheduler decides so, we will see how it works in practice in the next paragraph.

The scheduler runs in parallel to all other LNT processes and can synchronize (Sync) on all actions as described in Figure 5. The scheduler is passive in the sense that it does not impact the execution of the workflow, but for OrJoin gateways. In that latter case, the scheduler decides whether to trigger a specific OrJoin gateway. For all other workflow elements, the scheduler synchronizes on flows and keeps track of all active tokens in a local set. To do so, the scheduler is encoded in LNT using a **select** construct. It

Table 6: Encoding PIF into LNT (unbalanced Or gateways).

PIF Gateway	BPMN Notation	LNT Encoding
OrSplit		**loop** incf (?ident of ID) ; **select** (* *s_i if one matching merge* *) s1 ; outf1 (?ident of ID) [] s2 ; outf2 (?ident of ID) [] s3 ; **par** outf1 (?id1 of ID) || outf2 (?id2 of ID) **end par** **end select end loop**
OrJoin		mergestatus := False ; **while** mergestatus == False **loop** **select** incf1 (?ident of ID) [] incf2 (?ident of ID) [] MoveOn(!mergeid) ; mergestatus := True **end select** **end loop ;** outf (?ident of ID)

is worth noting that in order to catch tokens without "losing" them during the execution, the scheduler has to reproduce faithfully the business process execution. As an example, when a token arrives at a task, the scheduler synchronizes in sequence with the flow incoming that task followed by the flow outgoing the task. This behaviour appears in the same choice of the **select** mentioned before. If it were to be encoded differently, the token would disappear during the task execution, generating erroneous computation of the scheduler at the OrJoin gateway level. If we put it differently, the scheduler implements task execution in an atomic manner.

Figure 6 shows an excerpt of the scheduler process for our running example. Here, we can note that the scheduler keeps track of the tokens by adding and removing the identifiers in the activeflows set. Consider flow37_begin which refers to the initial event, in this case the scheduler does not remove any identifier from the active flow set as there is no incoming flow. Similarly, for flow30_finish which is an end event, the scheduler does not add any identifier to the active flow set. As mentioned earlier, the scheduler mimics the workflow behaviour. This is illustrated in flow39_finish, which refers to an Xor gateway. Since Xor is a choice, we see the **select** operator with two choices corresponding to the outgoing flows (flow2, flow3). It is also worth noting that the scheduler removes ident1 from the active flows and adds the identifier corresponding to the begin flow (ident2 or ident3) in order to keep track of the tokens.

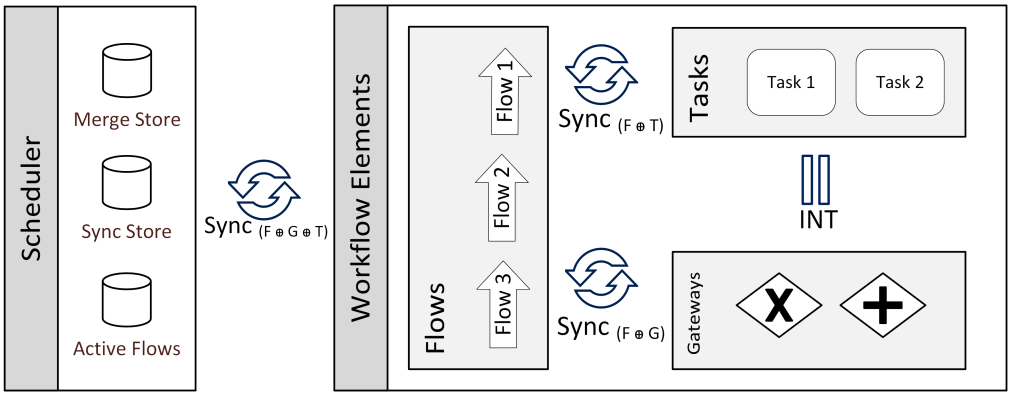


Figure 5: Scheduler Composition with PIF Elements (Sync denotes synchronization and INT denotes interleaving. F, G and T denote flows, gateways and tasks respectively).

Let us now focus on the active part of the scheduler, that is, the part of its behaviour where it decides whether an activated OrJoin gateway can be triggered or not. Note that the scheduler process is equipped with four parameters corresponding to: (i) the set of active tokens (IDS is a set of identifiers), (ii) the structure of the business process, (iii) the set of tokens accumulated at the OrJoin level, and (iv) the set of currently activated OrJoin gateways. The scheduler triggers a merge behaviour if there is an active merge (an OrJoin gateway) waiting to happen. If there are several activated OrJoin gateways, there is an enumeration on all their identifiers. Given an OrJoin gateway identifier, we check whether the merge is possible (is_merge_possible). In case of no token being present upstream of the OrJoin gateway, all tokens have arrived, hence the merge is possible. But we also have to check if the scheduler has consumed all arrived tokens (is_sync_done). If both conditions are satisfied, we synchronize with the process encoding the concerned OrJoin gateway indicating it that the merge can be triggered (MoveOn). The entire LNT code for those functions (is_merge_possible and is_sync_done) is given in appendix. Finally, we synchronize on the flow outgoing of that OrJoin gateway and we recursively call the scheduler process updating all parameters. If the merge is not possible, we have a simple recursive call to the scheduler process without changing any parameter.

Example. The translation of the online shopping process in LNT results

```
type NODE is
 i ( initial: INITIAL ),
 f ( finals: FINALS ),
 g ( gateways: GATEWAYS ),
 t ( tasks: TASKS )
end type

type NODES is
 set of NODE
end type

type BPROCESS is
 proc ( name: ID, nodes: NODES, flows: FLOWS )
end type

process scheduler [...] (activeflows: IDS, bpmn: BPROCESS, syncstore: IDS,
    mergestore: IDS) is
 select
  flow37_begin(?ident1 of ID); scheduler [...] (union ({ident1},
      remove_ids_from_set({}, activeflows)), bpmn, syncstore, mergestore)
 []
  flow39_finish(?ident1 of ID);
  select
   flow2_begin(?ident2 of ID); scheduler [...] (union ({ident2},
       remove_ids_from_set({ident1}, activeflows)), bpmn, syncstore,
       mergestore)
  []
   flow3_begin(?ident3 of ID); scheduler...
  end select
 []
  flow30_finish(?ident1 of ID); scheduler[...] (union ({},
      remove_ids_from_set({ident1}, activeflows)), bpmn, syncstore,
      mergestore)
 []
  mergeid := any ID where member(mergeid, mergestore);
  if(is_merge_possible(bpmn, activeflows, mergeid) and is_sync_done(bpmn,
      activeflows, syncstore, mergeid)) then
   MoveOn(!mergeid);
   outf(?ident1 of ID);
   scheduler[...] (union ({ident1}, remove_incf(bpmn, activeflows,
       mergeid)), bpmn, remove_sync(bpmn, syncstore, mergeid),
       remove(mergeid, mergestore))
  else
   scheduler [...] (activeflows, bpmn, syncstore, mergestore)
  end if
 end select
end process
```

Figure 6: Excerpt of the LNT Scheduler Process for the Online Shopping Process.

in several processes. The main process is given in Figure 7. This excerpt of specification shows first how the scheduler is in parallel with the process,

```
process main [signIn:any, signUp:any, chooseItems:any, ...] is
 hide begin:any, finish:any, flow2_begin:any, flow2_finish:any, ... in
  par MoveOn, flow2_begin, flow2_finish, flow3_begin, flow3_finish, ... in
   scheduler [flow3_begin, ..., flow67_finish, MoveOn] (nil, p1(), nil, nil)
   ||
    par flow2_begin, flow2_finish, flow3_begin, flow3_finish, ... in
     par
      flow [flow2_begin, flow2_finish] ||...|| flow [flow40_begin,
           flow40_finish]
     end par
    ||
    par
        init [begin, flow37_begin] || final [flow30_finish, finish]
     || final [flow34_finish, finish] ||...||
     || xorsplit_exclusivegateway1 [flow39_finish, flow2_begin, flow3_begin]
     || task_1_1 [flow6_finish, chooseItems, flow8_begin] || ...
    end par
   end par
  end par
 end hide
end process
```

Figure 7: Main LNT Process for the Online Shopping Process.

synchronizes on flow actions as well as on the MoveOn action which is used by the scheduler to submit messages to the processes encoding OrJoin gateways. The process is encoded in two parts, flows on the one hand and nodes on the other hand. The LNT processes for flows and nodes synchronize on flow actions. Last but not least, from an external point of view, all flow actions are hidden to let visible only task names. One can see the corresponding LTS (56 states and 119 transitions) shown in Figure 8 where we removed all internal transitions for readability reasons. The top part of the LTS is quite dense because the use of inclusive gateways in BPMN (Or gateways in PIF) induce many possible combinations in the order of task executions.

4. Evolution Notions

In this section, we formally define several kinds of comparisons between BPMN processes. Their analysis allows one to ensure that the evolution of one process into another one is satisfactory.

Notation. LNT processes are denoted in italics, *e.g.*, p, and BPMN processes are denoted using a bold font, *e.g.*, **b**. In the sequel, we denote with $||p||$ the semantic model of an LNT process p, that is the LTS for p. Further,

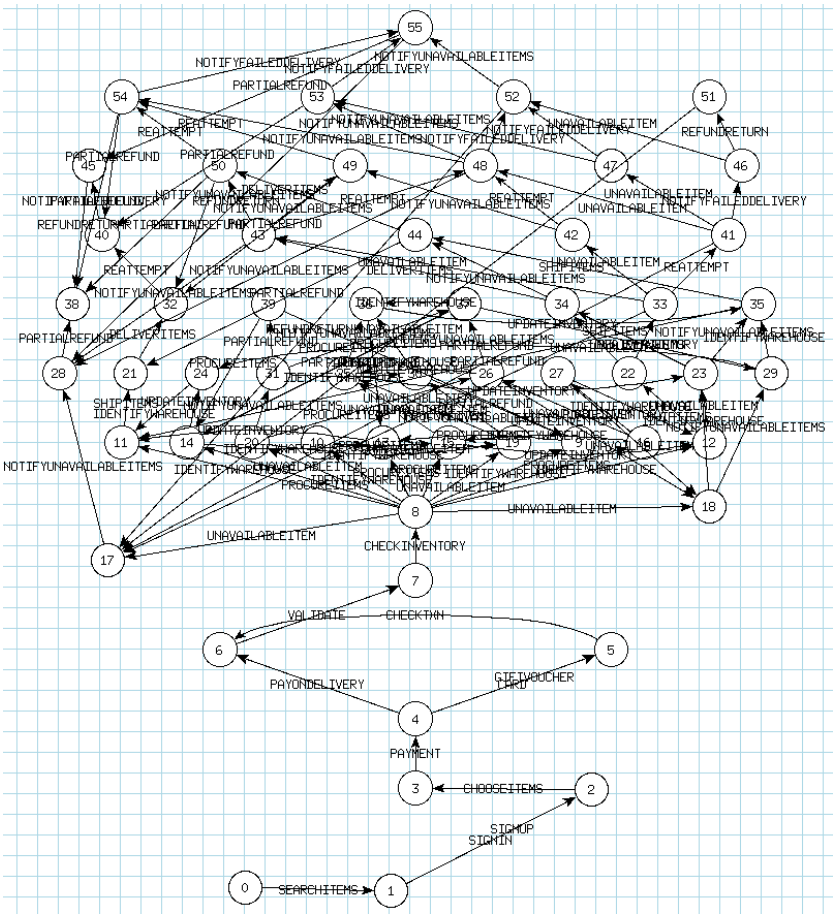


Figure 8: LTS Formal Model for the Online Shopping Process.

we denote the BPMN to LNT transformation introduced in the previous section using Θ, and the application of it to a BPMN process $\mathbf{b}$ using $\Theta(\mathbf{b})$. Accordingly, $||\ \Theta(\mathbf{b})\ ||$ denotes the LTS for this process. As far as the comparisons are concerned, we suppose we are in the context of the evolution of a BPMN process $\mathbf{b}$ into a BPMN process $\mathbf{b}'$, denoted by $\mathbf{b} \dashrightarrow \mathbf{b}'$.

4.1. Preliminaries

The following definitions are taken from [21]. They are relative to the LTS-level relations that we ground on to define our evolution relations.

Definition 1 (Branching bisimulation)**.** *Two graphs g and h are branch-*

ing bisimilar if there exists a symmetric relation R [...] (called branching bisimulation) between the nodes of g and h such that:

(i) The roots are related by R;

(ii) If $R(r, s)$ and $r \rightarrow^a r'$, then either $a = \tau$ and $R(r', s)$ or there exists a path $s \Rightarrow s_1 \rightarrow^a s_2 \Rightarrow s'$ such that $R(r, s_1)$, $R(r', s_2)$ and $R(r', s')$.

The original definition (given above) refers to (process) graphs that are "*connected, rooted, edge-labelled and direct graphs*" [21]. LTSs are such graphs, with the roots corresponding to initial states. In the definition, $r \rightarrow^a r'$ denotes an edge (= LTS transition) from node (= LTS state) r to node r' labelled by a. Further, $s \Rightarrow s_1$ denotes a sequence (path) of zero or more silent (τ) transitions from s to s_1 (s can be s_1). In the sequel we also refer to branching bisimulation as branching equivalence, and when g and h are branching bisimilar we may use $g \overset{\text{br}}{\equiv} h$.

If we remove the symmetric constraint on R in the previous definition we get a *branching simulation* or *branching preorder* between g and h, denoted with $g \overset{\text{br}}{<} h$, and we say that h simulates g.

4.2. Conservative Evolution

Our first comparison criterion is strong. Given an evolution $\mathbf{b} \dashrightarrow \mathbf{b}'$, it ensures that the observable behaviour of $\mathbf{b}$ is exactly preserved in $\mathbf{b}'$. It supports very constrained refactorings of BPMN processes such as grouping or splitting parallel or exclusive branches (*e.g.*, $\diamondsuit(\diamondsuit(\mathbf{a}, \mathbf{b}), \mathbf{c}) \overset{\dashrightarrow}{\dashleftarrow} \diamondsuit(\mathbf{a}, \mathbf{b}, \mathbf{c})$ where $\diamondsuit(x_1, \ldots, x_n)$ denotes a balanced exclusive split-merge). At the semantic level, several behavioural equivalences could be used. We have to deal with internal transitions introduced by hiding (see Section 3). Hence, we chose to use branching equivalence, introduced in Section 4.1, since it is the finest equivalence notion in presence of such internal transitions.

Definition 1. *(Conservative Evolution) Let* $\mathbf{b}$ *and* $\mathbf{b}'$ *be two processes,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is a conservative evolution iff* $|| \Theta(\mathbf{b}) || \overset{\text{br}}{\equiv} || \Theta(\mathbf{b}') ||$.

4.3. Inclusive and Exclusive Evolution

In most cases, one does not want to replace a business process by another one having exactly the same behaviour. Rather, one wants to be able to add new functionalities in the process, without interfering with the existing ones.

A typical example is adding new paths, *e.g.*, $\diamondsuit_{+}(\mathbf{a},\mathbf{b}) \dashrightarrow \diamondsuit_{+}(\mathbf{a},\mathbf{b},\mathbf{c})$, or evolving an existing one, *e.g.*, $\diamondsuit_{+}(\mathbf{a},\mathbf{b}) \dashrightarrow \diamondsuit_{+}(\mathbf{a},\diamondsuit_{\times}(\mathbf{b},\mathbf{c}))$. So here, we ground on a preorder relation rather than on an equivalence one, ensuring that, given $\mathbf{b} \dashrightarrow \mathbf{b}'$, all observable behaviours that were in $\mathbf{b}$ are still in $\mathbf{b}'$. For this we rely on the branching preorder, introduced in Section 4.1.

Definition 2. *(Inclusive Evolution) Let* $\mathbf{b}$ *and* $\mathbf{b}'$ *be two processes,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is an inclusive evolution iff* $|| \Theta(\mathbf{b}) || \overset{\text{br}}{<} || \Theta(\mathbf{b}') ||$.

Similarly, one may refine a process by implementing only a part of it. Here, in $\mathbf{b} \dashrightarrow \mathbf{b}'$, one does not want that $\mathbf{b}'$ exposes any additional behaviour that is outside what is specified in $\mathbf{b}$. This is a reversed form of inclusive evolution.

Definition 3. *(Exclusive Evolution) Let* $\mathbf{b}$ *and* $\mathbf{b}'$ *be two processes,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is an exclusive evolution iff* $|| \Theta(\mathbf{b}') || \overset{\text{br}}{<} || \Theta(\mathbf{b}) ||$.

The duality between inclusive and exclusive evolution is usual when one formalizes the fact that some abstract specification $\mathbf{a}$ is correctly implemented into a more concrete system $\mathbf{c}$. For some people, this means that at least all the behaviours expected from $\mathbf{a}$ should be available in $\mathbf{c}$. Taking the well-known "coffee machine" example, if a specification requires that the machine is able to deliver coffee, an implementation delivering either coffee or tea (depending on the people interacting with it) is correct. For others, *e.g.*, in the testing community, an implementation should not expose more behaviours than what was specified.

4.4. Selective Evolution

Up to now, we have supposed that all tasks in the original process were of interest. Still, one could choose to focus on a subset of them, called tasks of interest. This gives freedom to change parts of the processes as soon as the behaviours stay the same for the tasks of interest. For this, we define selective evolution up to a set of tasks T. Tasks that are not in this set will be hidden in the comparison process. Formally, this is achieved with an operation $[T]$ on LTSs, which, given an LTS l, hides any transition whose label is not in T by changing this label to τ (it becomes an *internal* transition). Again, here we can rely on branching equivalence to deal with these internal transitions.

Definition 4. *(Selective Conservative Evolution) Let* $\mathbf{b}$ *and* $\mathbf{b}'$ *be two processes, and* T *be a set of tasks,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is a selective conservative evolution with reference to* T *iff* $|| \Theta(\mathbf{b}') || [T] \overset{\text{br}}{\equiv} || \Theta(\mathbf{b}) || [T]$.

A specific interesting case of selective evolution is when the set of tasks of interest corresponds exactly to the tasks of the original process. This lets the designer add new behaviours not only in a separate way (as with inclusive evolution) but also within the behaviours of the original process. For example, $\mathbf{a}; \mathbf{b} \dashrightarrow \diamondplus(\mathbf{a}, \mathbf{log}); \mathbf{b}$, that is a way to log some information each time $\mathbf{a}$ is done, is not an inclusive evolution but is a selective conservative evolution with reference to $\{\mathbf{a}, \mathbf{b}\}$. Accordingly to selective conservative evolution, we can define selective inclusive evolution (respectively selective exclusive evolution) by using the branching preorder, $\overset{\text{br}}{<}$, instead of $\overset{\text{br}}{\equiv}$.

4.5. Renaming and Refinement

One may also want to take into account renaming when checking an evolution $\mathbf{b} \dashrightarrow \mathbf{b}'$. For this we use a relabelling relation $R \subseteq T_{\mathbf{b}} \times T_{\mathbf{b}'}$, where $T_{\mathbf{b}}$ (respectively $T_{\mathbf{b}'}$) denotes the set of tasks in $\mathbf{b}$ (respectively $\mathbf{b}'$). We require that for every t in $T_{\mathbf{b}}$, if we have (t, t'_1) and (t, t'_2) in R then $t'_1 = t'_2$, *i.e.,* R is a (possibly partial) function. Applying a relabelling relation R to an LTS l, which is denoted by $l \lhd R$, consists in replacing in l any transition labelled by some t in the domain of R by a transition labelled by $R(t)$.

To take into account task renaming in any of the above-mentioned evolutions, we just have to perform the equivalence (or preorder) checking up to relabelling in the formal model for $\mathbf{b}$. For example, $\mathbf{b} \dashrightarrow \mathbf{b}'$ is a conservative evolution up to a relabelling relation R for $\mathbf{b}$ and $\mathbf{b}'$ iff $|| \Theta(\mathbf{b}) || \lhd R \overset{\text{br}}{\equiv} || \Theta(\mathbf{b}') ||$.

Sometimes renaming is not sufficient, *e.g.,* when evolution corresponds to the refinement of a task by a workflow. We define a refinement rule as a couple (t, W), noted $t \dashrightarrow W$[2], where t is a task and W a workflow. A set of refinement rules, or refinement set, $\mathcal{R} = \bigcup_{i \in 1 \ldots n} t_i \dashrightarrow W_i$ is valid if there are no multiple refinements of the same task ($\forall i, j \in 1 \ldots n,\ i \neq j \Rightarrow t_i \neq t_j$) and if no refinement rule has in its right-hand part a task that has to be refined ($\forall i, j \in 1 \ldots n,\ t_i \notin W_j$). These constraints enforce that refinements

[2]The $\dashrightarrow$ symbol is overloaded since a refinement rule is an evolution at the task level.

do not depend on the application ordering of refinement rules, *i.e.*, they are deterministic.

To take into account refinement in evolution, a pre-processing has to be performed on the source process. For example, given that $\mathbf{b} \blacktriangleleft \mathcal{R}$ denotes the replacement in $\mathbf{b}$ of t_i by W_i for each $t_i \dashrightarrow W_i$ in $\mathcal{R}$, $\mathbf{b} \dashrightarrow \mathbf{b}'$ is a conservative evolution up to a refinement set $\mathcal{R}$ iff $|| \Theta(\mathbf{b} \blacktriangleleft \mathcal{R}) || \overset{\text{br}}{\equiv} || \Theta(\mathbf{b}') ||$.

4.6. Property-Aware Evolution

A desirable feature when checking evolution is to be able to focus on properties of interest and avoid in-depth analysis of the workflows. This gives the freedom to perform changes (including some not possible with the previous evolution relations) as long as the properties of interest are preserved. Typical properties are deadlock freedom or safety and liveness properties defined over the alphabet of process tasks and focusing on the functionalities expected from the process under analysis. Such properties are written in a temporal logic supporting actions and, to make the property writing easier, the developer can rely on well-known patterns as those presented in [22].

Definition 5. *(Property-Aware Evolution) Let* $\mathbf{b}$ *and* $\mathbf{b}'$ *be two processes,* T *be a set of tasks, and* ϕ *be a formula defined over* T*,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is a property-aware evolution with respect to* ϕ *iff* $|| \Theta(\mathbf{b}) || \models \phi \Rightarrow || \Theta(\mathbf{b}') || \models \phi$.

4.7. Context-Aware Evolution

A process is often used in the context of a collaboration, which in BPMN takes the form of a set of processes ("pool lanes") communicating via messages. When evolving a process $\mathbf{b}$, one may safely make changes as soon as they do not have an impact on the overall system made up of $\mathbf{b}$ and these processes. To ensure this, we have to compute the semantics of $\mathbf{b}$ communicating on a set of interactions I (a subset of its tasks) with the other processes that constitute the context of $\mathbf{b}$. We support two communication modes: synchronous or asynchronous. For each mode m we have an operation $\overset{m}{\times}_I$, where $|| \Theta(\mathbf{b}) || \overset{m}{\times}_I || \Theta(\mathbf{c}) ||$ denotes the LTS representing the communication on a set of interactions I between $\mathbf{b}$ and $\mathbf{c}$. For synchronous communication, $\overset{\text{m}}{\times}_I$ is the LTS synchronous product [23]. For asynchronous communication, $\overset{\text{m}}{\times}_I$ is achieved by adding a buffer to each process [24]. Here, to keep things simple, we will suppose without loss of generality, that a context is a single process $\mathbf{c}$.

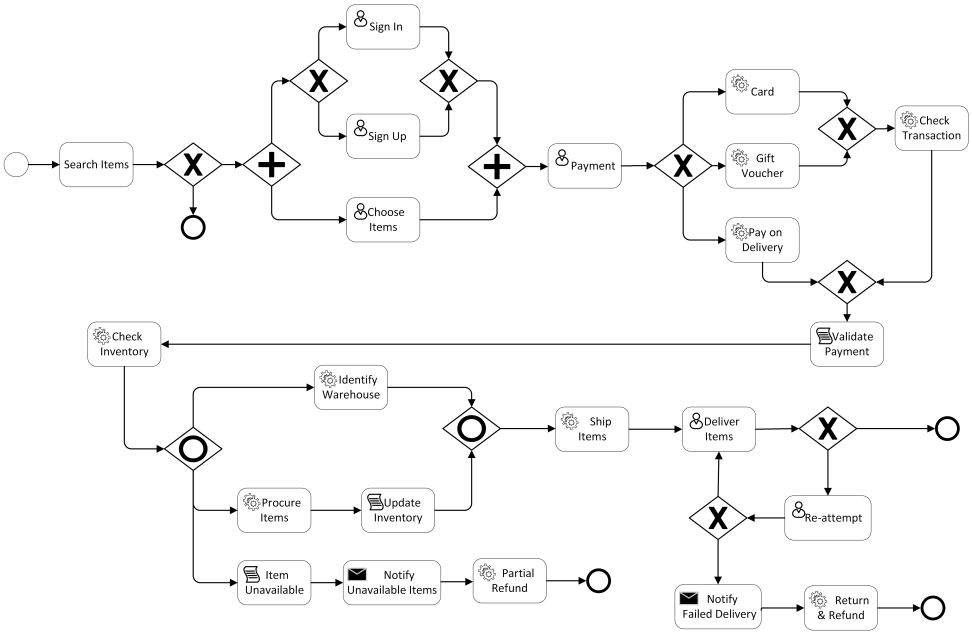


Figure 9: Online Shopping Process in BPMN (v2).

Definition 6. *(Context-Aware Conservative Evolution) Let* $\mathbf{b}$*,* $\mathbf{b}'$*, and* $\mathbf{c}$ *be three processes,* $\mathbf{c}$ *being the context for* $\mathbf{b}$ *and* $\mathbf{b}'$*,* m *be a communication mode (*$m \in \{\text{sync}, \text{async}\}$*), and* I *be the set of interactions taking place between* $\mathbf{b}$ *and* $\mathbf{c}$*,* $\mathbf{b} \dashrightarrow \mathbf{b}'$ *is a context-aware conservative evolution with reference to* $\mathbf{c}$*,* m*, and* I *iff* $|| \Theta(\mathbf{b}) || \overset{m}{\times}_I || \Theta(\mathbf{c}) || \overset{\text{br}}{\equiv} || \Theta(\mathbf{b}') || \overset{m}{\times}_I || \Theta(\mathbf{c}) ||$.

Accordingly, we may define context-aware inclusive and exclusive evolution, or combine them with renaming and refinement.

Example. We introduce in Figure 9 a revised version of the online shopping process presented in Figure 3. In this new process, when the client decides to choose and buy items, (s)he can sign in/up in parallel with its decision to purchase specific items. In the new process, it is achieved by adding an additional parallel gateway.

The two versions of the online shopping process are not conservative because the Choose Items task can appear before Sign In/Up tasks in the new version of the process. However, both versions are related with respect to the

inclusive/exclusive evolution notions. The new version includes all possible executions of the former one (the opposite is false) while incorporating new traces (those including Choose Items and Sign In/Up in sequence for instance).

As far as property-aware evolution is concerned, one can check for instance whether any Choose Items task eventually leads to a Deliver Items task. Following [22], this corresponds to the pattern:

REQUIREMENT: The choice of items will eventually lead to items being delivered.
PATTERN: Response, between Choose Items and Deliver Items
SCOPE: Global

In [22], the patterns are related to the LTL and CTL temporal logics, and to quantified regular expressions. Since we target the CADP verification toolbox, we use MCL [25] instead. The pattern can be formalized in MCL using box modalities ([..]) and fix points (mu X . (..)) as follows:

```
[true* . "CHOOSEITEMS"] mu X . (⟨ true ⟩ true and [not ("DELIVERITEMS")] X)
```

The property specifies that each Choose Items transition in the LTS denoted by [true* . "CHOOSEITEMS"], would inevitably lead to the delivery of items, which is specified by the formula: mu X . (⟨ true ⟩ true and [not ("DELIVERITEMS")] X). This property is actually not satisfied for any version of the process because some item may be unavailable or because the delivery may abort after several attempts. If using the same pattern we check that the Choose Items task eventually leads to the Check Inventory task, this property is satisfied by both versions of the online shopping process.

5. Tool Support

In this section, we focus on the implementation of our approach. We will present first the Web application that can be used to access and use the VBPMN analysis functionalities. Then, we will focus on the verification of BPMN processes using the CADP toolbox, and we will particularly present some experimental results to show how our approach scales.

5.1. Web Application

Business processes are usually designed by business analysts that may not be familiar with formal verification techniques and tools. Our goal is to enable one to benefit from formal verification without having to deal with a

steep learning curve. The VBPMN Web Application has been developed in this direction. It hides the underlying transformation and verification process, it provides the users with simple interaction mechanisms, and it generates analysis results that are easily relatable to the input process model(s). There are numerous IDEs supporting the modelling of business processes. Extending a specific one, *e.g.* the Eclipse BPMN designer, would limit the community that could use VBPMN. Hence, we have decided to architect it as a Web application. However, the integration as a plug-in of a platform for business processes that goes beyond modelling, such as ProM [26], would be a relevant and complementary approach as far as broadening the audience of our verification techniques is concerned.

The VBPMN Web application is hosted on a Tomcat application server. Its responsive UI invokes a RESTful API to trigger the transformation from BPMN to PIF and the verification of the process models. The use of such an API makes the platform more extensible – other people could build custom UIs using them. Internally, the API is built using the Jersey JAX-RS implementation. The model-to-model transformation from BPMN to PIF is realized at the XML level (both BPMN and PIF have XML representations) using a combination of JAXB and of the Woodstox Streaming XML API (StAX), which implements a pull parsing technique and offers better performance for large XML models. The model-to-text transformation from PIF to LNT and SVL [1] is achieved using a Python script that can also be used independently from the Web application as a command-line interface tool.

As far as the user interface is concerned, one can choose either to verify some property or to check process evolution correctness. In the first case (Fig. 10), one has to upload the BPMN process model and specify the temporal logic formula for the property.

In the later case (Fig. 11), one has to upload two BPMN processes, specify the evolution relation, and optionally give tasks to hide or to rename in the comparison. As a result one can visualize the LTS models that have been generated for the BPMN processes. Further, in case the verification fails, *i.e.*, either the property does not yield or the evolution is not correct, one gets a counter-example model.

5.2. Analysis with CADP

The operational semantics of the LNT process algebra enables us to generate LTSs corresponding to the BPMN process model given in the VBPMN UI. These LTSs may then be analyzed using CADP. VBPMN currently provides

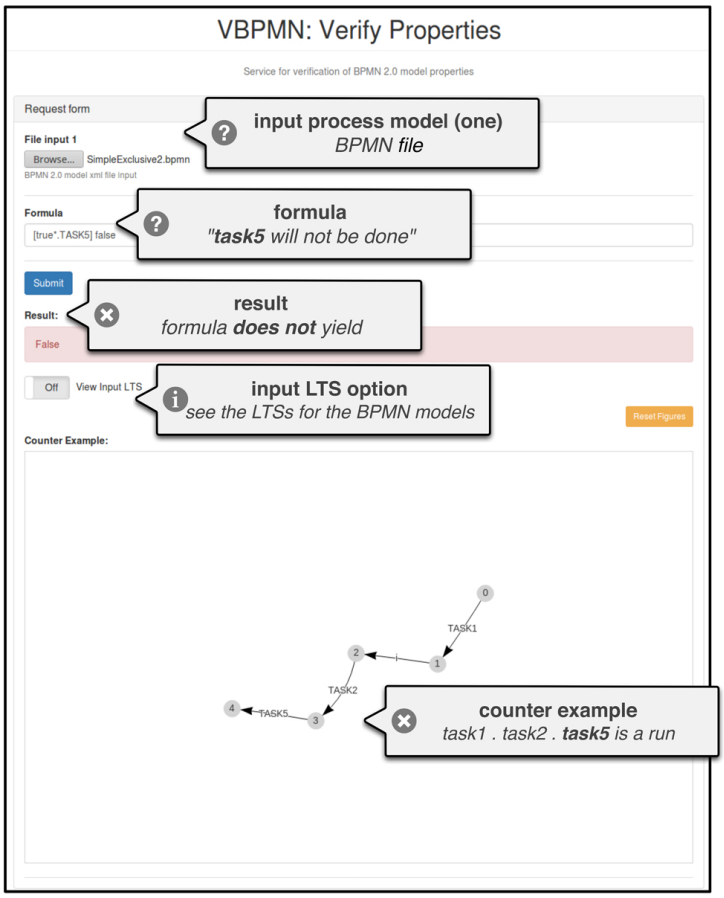


Figure 10: VBPMN Web Application in Use: Property Verification.

two kinds of formal analysis: functional verification using model checking and process comparison using equivalence checking.

Functional verification. One can for example use model checking techniques to search for deadlocks or livelocks. Another option is to use the CADP model checker for verifying the satisfaction of process-specific safety and liveness properties. In this case, since the properties depend on the process, they have to be provided by the analyst. The use of patterns for

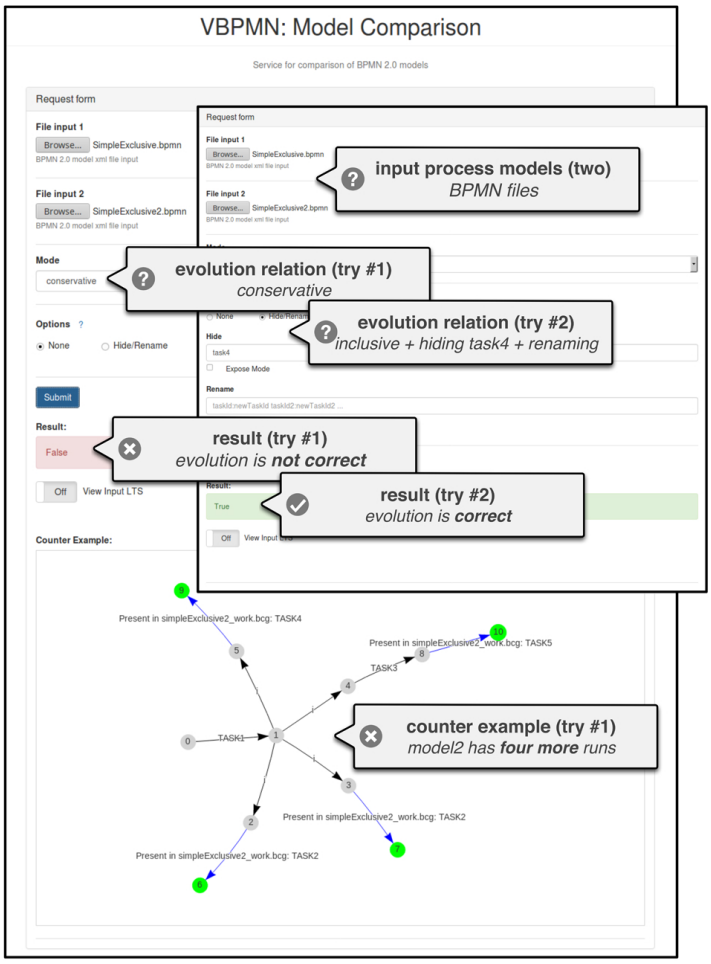


Figure 11: VBPMN Web Application in Use: Model Comparison.

properties is a perspective (see Section 7).

One can also be interested in checking soundness of the process. Soundness as defined in [27] requires three conditions:

- C1, from the initial state it is always possible to reach the final state,
- C2, from the moment the final state is reached, there are no remaining tokens in the process,

- C3, there is no dead transition from the initial state.

All three conditions can be checked using MCL formulas and model checking techniques. C3 can be checked using, for any task t, the following formula:

$$\langle\ \mathsf{true}^*\ .\ \text{"t"}\ \rangle\ \mathsf{true}$$

C1 requires to let visible (it is hidden for now, see Fig. 7) the finish action used to encode end events in LNT (Tab. 4) and to check the reachability on all paths of those actions:

$$\mathsf{mu\ X\ .\ (\langle\ true\ \rangle\ true\ and\ [not\ ("finish")]\ X)}$$

C2 requires in addition to make appear as parameter to this finish action the number of current tokens remaining in the process, and then to check that when such actions are reached, the number of tokens is zero.

Process comparison. It takes as input two process models, an evolution relation and possibly additional parameters for the relation. Several evolution relations are proposed. Conservative evolution ensures that the observational behaviour is strictly preserved. Inclusive evolution ensures that a subset of a process behaviour is preserved in a new version of it. Selective evolution (that is compatible with both conservative and inclusive evolution) allows one to focus on a subset of the process tasks. It is also possible to have VBPMN work up-to a renaming relation over tasks. If the two input process models do not fulfil the constraints of the chosen evolution relation, a counter-example that indicates the source of the violation is returned by VBPMN in the UI. This helps the process analyst in understanding the impact of evolution and supports the refinement into a correct evolved version of a process model. All the evolution relations are checked using the CADP equivalence checker and SVL scripts for hiding and renaming as demonstrated with the SVL patterns in Figure 12.

5.3. Experiments

We used a Mac OS laptop running on a 2.3 GHz Intel Core i7 processor with 16 GB of Memory. We carried out experiments on many examples taken from the literature or hand-crafted, and we present in Table 7 some of these results.

checking conservative evolution of $\mathbf{b} \dashrightarrow \mathbf{b}'$
$\rightarrow$ *in the tool: mode=conservative, options=none*

```
bcg_open "b.bcg" bisimulator -equal -branching -diag "b'.bcg"
```

checking inclusive evolution of $\mathbf{b} \dashrightarrow \mathbf{b}'$
$\rightarrow$ *in the tool: mode=inclusive, options=none*

```
bcg_open "b.bcg" bisimulator -smaller -branching -diag "b'.bcg"
```

checking exclusive evolution of $\mathbf{b} \dashrightarrow \mathbf{b}'$
$\rightarrow$ *in the tool: mode=exclusive, options=none*

```
bcg_open "b.bcg" bisimulator -greater -branching -diag "b'.bcg"
```

checking selective conservative evolution of $\mathbf{b} \dashrightarrow \mathbf{b}'$ *wrt. a set of tasks* $T=\{t_1, ...\}$
$\rightarrow$ *in the tool: mode=conservative, options=hide/rename + hide T with expose mode*
– expose mode is a shortcut to hide all tasks but for the ones given

```
"b.bcg" = total hide all but t1, ...  in "b.bcg"
"b'.bcg" = total hide all but t1, ...  in "b'.bcg"
bcg_open "b.bcg" bisimulator -equal -branching -diag "b'.bcg"
```

checking conservative evolution of $\mathbf{b} \dashrightarrow \mathbf{b}'$ *up to a relabelling* $R=\{(t_1, t_1'), \ldots\}$
$\rightarrow$ *in the tool: mode=conservative, options=hide/rename + rename R with rename first*
– rename first is an option to rename in the first process only
– it is also possible to rename in the second process or in both

```
"b.bcg" = total rename t1 -> t1', ...  in "b.bcg"
bcg_open "b.bcg" bisimulator -equal -branching -diag "b'.bcg"
```

Figure 12: SVL Patterns for the Verification of the Evolution Notions.

Each example consists of two versions of the process (original and revised). For each version, we first characterize the workflow by giving the number of tasks, sequence flows, and gateways. We also indicate whether the workflow exhibits balanced structure especially for inclusive gateways (B)[3] and if there is looping behaviour in the process (L). We show then the size (states and transitions) of the resulting LTS before and after minimization. Minimization

[3]The table shows a $\surd$ if there are no inclusive gateways or if there are balanced inclusive gateways, and a $\times$ if there are unbalanced inclusive gateways

Table 7: Experimental Results.

BPMN	Characteristics					LTS (states/transitions)		Evol.		
Proc.	Tasks	Flows	Gateways	B	L	Raw	Minimized	≡	<	>
1	6	11	2⟨×⟩	√	×	29/29	8/9	×	√	×
1'	7	15	2⟨×⟩+ 2⟨+⟩	√	×	78/118	11/14		15s	
2	4	7	1⟨○⟩	√	×	70/105	7/9	√	√	√
2'	8	14	2⟨×⟩	√	×	36/38	10/12		15s	
3	7	14	2⟨×⟩+ 2⟨+⟩	√	×	62/87	10/11	×	×	×
3'	8	16	4⟨○⟩	√	×	1,786/5,346	28/56		15s	
4	15	29	3⟨×⟩+ 2⟨+⟩+ 2⟨○⟩	×	×	469/1,002	24/34	×	√	×
4'	16	33	5⟨×⟩+ 2⟨+⟩+ 2⟨○⟩	×	√	479/1,013	26/37		15s	
5	12	32	7⟨×⟩+ 2⟨+⟩+ 2⟨○⟩	√	√	3,038/9,785	32/63	×	√	×
5'	12	33	7⟨×⟩+ 2⟨+⟩+ 2⟨○⟩	√	√	3,039/9,787	32/64		16s	
6	22	46	10⟨×⟩+ 4⟨+⟩	√	√	179/248	32/41	×	√	×
6'	23	52	10⟨×⟩+ 4⟨+⟩+ 2⟨○⟩	√	√	570/1,295	38/57		16s	
7	12	24	6⟨○⟩	√	×	742,234/3,937,158	148/574	×	×	√
7'	12	24	4⟨○⟩+ 2⟨×⟩	√	×	6,394/21,762	60/152		31s	
8	22	41	8⟨×⟩+ 2⟨○⟩	×	√	1,818/2,236	149/210	×	√	×
8'	22	44	8⟨×⟩+ 2⟨○⟩+ 2⟨+⟩	×	√	1,889/2,327	158/223		1m13s	
9	12	26	4⟨×⟩+ 2⟨+⟩+ 2⟨○⟩	×	×	11,990/17,949	289/453	×	×	√
9'	8	20	4⟨×⟩+ 2⟨○⟩	×	×	1,003/1,274	53/77		1m2s	
10	192	210	12⟨×⟩+ 6⟨+⟩	×	×	791/988	201/218	×	√	×
10'	193	214	14⟨×⟩+ 6⟨+⟩	×	×	799/997	203/221		2m48s	
11	20	43	6⟨○⟩+ 6⟨+⟩	√	×	4,488,843/26,533,828	347/1,450	×	√	×
11'	20	39	8⟨○⟩	√	×	4,504,775/26,586,197	348/1,481		9m31s	
12	16	34	3⟨×⟩+ 6⟨+⟩+ 2⟨○⟩	×	×	797,335/1,549,764	1,498/2,537	×	√	×
12'	18	40	5⟨×⟩+ 6⟨+⟩+ 2⟨○⟩	×	√	867,055/1,659,930	2,007/3,374		22m15s	

is useful for automatically removing unnecessary internal transitions, which were introduced during the process algebra encoding but do not make sense from an observational point of view. We use branching reduction [21], which is the finest equivalence notion in presence of internal transitions and removes most internal transitions in an efficient way. Finally, the last column gives the results when comparing the LTSs for the two versions of the process using conservative, inclusive, and exclusive evolution, resp., and the overall computation time.

Examples 8 and 8' correspond to the first and second versions of the online shopping process we used in this paper as running example. Medium-size

examples (*e.g.*, example 7) can result in quite huge LTSs involving millions of states and transitions. This is due to our choice to show processes in the table containing several parallel and inclusive gateways, in most cases nested, which result in many possible interleaved executions in the corresponding LTSs. In contrast, example 10/10' involves about 200 tasks but exhibits a quite sequential structure, and then the generated LTSs are very small (less than a thousand states and transitions). Another comment concerns the considerable drop in size of the LTSs before and after minimization. Example 3' for example goes from about 2,000 states/5,000 transitions to about 30 states/60 transitions. This drastic reduction is due to all sequence flow actions encoded in LNT for respecting the BPMN original semantics. They do not have any special meaning *per se*, and are therefore hidden and removed by reduction.

As far as computation times are concerned, we observe that the final column of Table 7 gives the overall time, that is, the time for generating both LTSs, minimizing and comparing them. The comparison time is negligible. It takes 568 seconds (9 minutes and 28 seconds) for instance for generating and minimizing both LTSs for examples 11 and 11', and only 3 seconds for comparing both LTSs *wrt.* the three evolution notions considered in the table. Note that for processes involving unbalanced inclusive gateways, the time importantly increases for generating corresponding LTSs compared to balanced workflows. This is the case between example 12/12' which takes more than 10 minutes for generating an LTS consisting of about 1 million of states/transitions whereas example 11/11' results in less than 10 minutes to obtain an LTS containing about 4 millions of states and 26 millions of transitions. This drastic fall-down in performance comes from the LNT encoding for unbalanced workflows, which induces extra-computations during the LTS generation process. On a wider scale, computation times remain reasonable (about half an hour for all the examples shown in Table 7) even for LTSs containing millions of states and transitions.

6. Related Work

The absence of a single accepted modelling notation for business processes, raised the need to find ways to relate models written using different notations. The Workflow Pattern Initiative, see, *e.g*, [14], has addressed this issue by caracterizing the atomic patterns one may find in business processes. A correspondence between these patterns and some workflow languages, in-

cluding BPMN, UML AD, and EPC, can be found on their Web site. Further, the BPMN standard itself refers to workflow patterns for its notational elements. Several general ontologies for business process modelling have also been proposed, a comprehensive one being BPMO [28][4]. PIF is simpler than BPMO. It originates from a previous intermediate format we had defined, CIF [9]. Its objective is neither to be a comprehensive set of all concepts found in workflows, nor to support automated mapping between notations in their whole. Rather, it is focused on a subset of common concepts found in the three main business process notations that support a formal treatment in order to achieve different formal analyses of business processes.

Several works have focused on providing formal semantics and verification techniques for business processes using Petri nets, process algebras, abstract state machines, or rewriting logic. There was a significant effort aimed at providing formal semantics and verification techniques for business processes using Petri nets, see, *e.g.*, [30, 31, 5, 32, 33]. To the best of our knowledge, none of these works focus on the formal comparison and evolution of processes. As far as rewriting logic is concerned, in [34], the authors propose a translation of BPMN into rewriting logic with a special focus on data objects and data-based decision gateways. They provide new mechanisms to avoid structural issues in workflows such as flow divergence by introducing the notion of well-formed BPMN process. Their approach aims at avoiding incorrect syntactic patterns whereas we propose automated analysis at the semantic level. Rewriting logic is also used in [16] for analyzing BPMN processes with time using simulation, reachability analysis, and model checking to evaluate timing properties such as degree of parallelism and minimum/-maximum processing times. We focus on behavioural and not on time analysis here.

Let us now concentrate on those using process algebras for formalizing and verifying BPMN processes, which are the most related to the approach presented in this paper. The authors of [6] present a formal semantics for BPMN by encoding it into the CSP process algebra. They show in [35] how this semantic model can be used to verify compatibility between business participants in a collaboration. This work was extended in [36] to propose a timed semantics of BPMN with delays. [37, 38] focus on the semantics

[4]Indeed (a quite old) one of these ontologies is named PIF [29]. We were not aware of it when we began working on (our) PIF.

proposed in [6, 36] and propose an automated transformation from BPMN to timed CSP. In a previous work [9], we have proposed a first transformation from BPMN to LNT, targeted at checking the realizability of a BPMN choreography. We followed a state machine pattern for representing workflows, while we here encode them in a way close to Petri net firing semantics, which favours compositionality and is more natural for a workflow-based language such as BPMN. In [33], the authors propose a new operational semantics of a subset of BPMN focusing on collaboration diagrams and message exchange. The BPMN subset is quite restricted (no support of the inclusive merge gateway for instance) and no tool support is provided yet. Compared to the approaches above, our encoding also gives a semantics to the considered BPMN subset by translation to LNT, although it was not our primary goal. The main difference with respect to these related works is our focus on the evolution of processes and its automated analysis.

In the rest of this section, we present existing approaches for comparing several BPMN processes (or workflows). In [39], the author proposes a theoretical framework for comparing BPMN processes. His main focus is substitutability and therefore he explores various sorts of behavioural equivalences in order to replace equals for equals. This work applies at the BPMN level and aims at detecting equivalent patterns in processes. In a related line of works, [40] studies BPMN behaviours from a semantic point of view. It presents several BPMN patterns and structures that are syntactically different but semantically equivalent. This work is not theoretically grounded and is not complete in the sense that only a few patterns are tackled. The notion of equivalence is similar to the one used in [39]. The authors of [40] also overview best practices that can be used as guidelines by modelers for avoiding syntactic discrepancies in equivalent process models. Compared to our approach, this work only studies strong notions of equivalence where the behaviour is preserved in an identical manner. We consider a similar notion here, but we also propose weaker notions because one can make deeper changes (*e.g.*, by introducing new tasks) and in these cases such strong equivalences cannot be preserved.

In Chapter 9 of [41], the authors study the evolution of processes from a migration point of view. They define several notions of evolution, migration, and refactoring. Our goal here is rather complementary since we have studied the impact of modifying a workflow *wrt.* a former version of this workflow on low-level formal models, but we do not propose any solutions for applying these changes on a running instance of that initial workflow. In [42], the

authors address the equivalence or alignment of two process models. To do so, they check whether correspondences exist between a set of activities in one model and a set of activities in the other model. They consider Petri net systems as input and process graphs as low-level formalism for analysis purposes. Their approach resides in the identification of regions (set of activities) in each graph that can coincide with respect to an equivalence notion. They particularly study two equivalence notions, namely trace and branching equivalences. The main limit of this approach is that it does not work in the presence of overlapping correspondences, meaning that in some cases, the input models cannot be analyzed. This work shares similarities with our approach, in particular the use of low-level graph models, hiding techniques and behavioural equivalences for comparing models. Still, our approach always provides a result and considers new notions of model correspondence such as property-aware evolution.

It is worth mentioning another line of work aiming at measuring the degree of similarity of business process models, see, *e.g.*, [43, 44]. As an example, [43] achieves this goal first using causal footprints as an abstract representation of the behaviour captured by a process model. Then, given two footprints, the similarity is computed using the vector space model approach from information filtering and retrieval. In this paper, we chose a different angle of this question by studying qualitative aspects of processes instead of quantitative aspects.

This article is an extended version of a conference paper published in [45]. The key additions of this journal version are as follows: (i) an improvement of our process algebra encoding to support unbalanced workflows, (ii) the extension of our tool support, and particularly of the Web application, to support comparison analysis as well as model checking of business processes, (iii) the extensive validation of our approach on a large set of case studies, most of them taken from the literature on this topic, and (iv) a refined review and comparison with related work.

7. Conclusion

We have introduced our approach for checking the evolution of BPMN processes. To promote its adoption by business process designers, we have implemented it in a tool, VBPMN [2], that can be used through a Web application. We have presented different kinds of atomic evolutions that can be combined and formally verified. We have defined a BPMN to LNT model

transformation, which, using the LTS operational semantics of LNT enables us to automate our approach using existing LTS-based verification tools. This translation to LNT supports exclusive, inclusive and parallel gateways (including different semantics of inclusive merge gateways), looping behaviours, and unbalanced structure of workflows. We have applied our approach to many examples for evaluation purposes. The experimental results shown in this paper confirm that our tool is rather efficient since it can handle quite large examples within a reasonable amount of time.

As far as perspectives are concerned, let us mention the main ones.

Larger subset of the BPMN standard. BPMN has three main kinds of models: processes, collaborations, and choreographies. In this work we support the first ones, at the descriptive conformance sub-class level, without sub-processes and data, but with inclusive gateways. We plan to extend our support to sub-processes (with boundary events to deal with errors and compensations) and data (to deal with conditional flows). As far as other kinds of models are concerned, collaborations where only one process evolves can be supported using context-aware evolution, one of the evolution relations we propose. The extension to full-fledged collaborations could be possible through the transformation of each process in the collaboration and by relating communication tasks (synchronously or using buffers for asynchronous communication). The extension to choreography is possible by adding choreography tasks in PIF and treating them as basic observable atoms in the behavioural equivalences and inclusion relations. However, based on our past experience on choreography verification [9], we believe that specific evolutions relations could be defined for choreographies.

Patterns of properties. When the analyst wants to check model-specific properties on the processes, they have to be input directly using the MCL temporal logic. The reuse of well-known patterns for properties such as those presented in [22], and the automation of a translation from such patterns to the MCL temporal logic would clearly help there.

Enhanced feed-back. Diagnoses are returned to the designers in the form of low-level counter-examples (LTSs). This could be enhanced by presenting this information directly on the BPMN models, *e.g.* using animation.

New front- and back-ends. In the implementation of our BPMN to LNT transformation, we rely on an intermediate format including the main workflow-based constructs. This paves the way for new front-end DSLs and other back-end verification techniques. Another perspective of this work is

to propose quantitative analysis for comparing business processes as studied in [43, 44]. Our goal is thus to consider non-functional requirements in BPMN processes, such as the throughput and latency of tasks, which can be modelled by extending LTSs with Markovian information and computed using steady-state analysis [46].

References

[1] H. Garavel, F. Lang, SVL: A Scripting Language for Compositional Verification, in: Proc. of FORTE'01, 2001, pp. 377–394.

[2] VBPMN Framework., https://pascalpoizat.github.io/vbpmn-web/.

[3] OMG, Business Process Model and Notation (BPMN) – Version 2.0, january 2011.

[4] ISO/IEC, International Standard 19510, Information technology – Business Process Model and Notation, 2013.

[5] R. Dijkman, M. Dumas, C. Ouyang, Semantics and Analysis of Business Process Models in BPMN, Inf. Softw. Technol. 50 (12) (2008) 1281–1294.

[6] P. Wong, J. Gibbons, A Process Semantics for BPMN, in: Proc. of ICFEM'08, 2008, pp. 355–374.

[7] F. Kossak, C. Illibauer, V. Geist, J. Kubovy, C. Natschläger, T. Ziebermayr, T. Kopetzky, B. Freudenthaler, K. Schewe, A Rigorous Semantics for BPMN 2.0 Process Diagrams, Springer, 2014.

[8] R. Mateescu, G. Salaün, L. Ye, Quantifying the Parallelism in BPMN Processes using Model Checking, in: Proc. of CBSE'14, 2014, pp. 159–168.

[9] M. Güdemann, P. Poizat, G. Salaün, L. Ye, VerChor: A Framework for the Design and Verification of Choreographies, IEEE Trans. Services Computing 9 (4) (2016) 647–660.

[10] J. Koehler, J. Vanhatalo, Process Anti-patterns: How to Avoid the Common Traps of Business Process Modeling, Tech. rep., iBM Research Report 3678 (2007).

[11] M. Dam, Model Checking Mobile Processes, in: Proc. of CONCUR'93, Vol. 715 of LNCS, Springer, 1993, pp. 22–36.

[12] D. Fahland, C. Favre, B. Jobstmann, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Instantaneous soundness checking of industrial business process models, in: Proc. of BPM'09, Vol. 5701 of LNCS, Springer, 2009, pp. 278–293.

[13] H. N. Nguyen, P. Poizat, F. Zaïdi, A Symbolic Framework for the Conformance Checking of Value-Passing Choreographies, in: Proc. of ICSOC'12, 2012, pp. 525–532.

[14] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros, Workflow Patterns, Distributed and Parallel Databases 14 (3) (2003) 5–51.

[15] M. C. Cornax, A. Krishna, A. Mos, G. Salaün, Automated Analysis of Industrial Workflow-based Models, in: Proc. of SAC'18, ACM Press, 2018.

[16] F. Durán, G. Salaün, Verifying Timed BPMN Processes using Maude, in: Proc. of COORDINATION, Vol. 10319 of LNCS, Springer, 2017, pp. 219–236.

[17] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, G. Smeding, Reference Manual of the LNT to LOTOS Translator, Version 6.7., Inria, 2018.

[18] ISO, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, Tech. Rep. 8807, ISO (1989).

[19] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes, STTT 2 (15) (2013) 89–107.

[20] D. R. Christiansen, M. Carbone, T. T. Hildebrandt, Formal Semantics and Implementation of BPMN 2.0 Inclusive Gateways, in: Proc. of WS-FM'10, Vol. 6551 of LNCS, Springer, 2011, pp. 146–160.

[21] R. J. van Glabbeek, W. P. Weijland, Branching Time and Abstraction in Bisimulation Semantics, J. ACM 43 (3) (1996) 555–600.

[22] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, Patterns in Property Specifications for Finite-State Verification, in: Proc. of ICSE'99, ACM, 1999, pp. 411–420.

[23] A. Arnold, Finite Transition Systems - Semantics of Communicating Systems, Prentice Hall, 1994.

[24] D. Brand, P. Zafiropulo, On Communicating Finite-State Machines, J. ACM 30 (2) (1983) 323–342.

[25] R. Mateescu, D. Thivolle, A Model Checking Language for Concurrent Value-Passing Systems, in: Proc. of FM'08, Vol. 5014 of LNCS, Springer, 2008, pp. 148–164.

[26] ProM Framework, http://www.processmining.org/prom.

[27] W. M. P. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits, Systems, and Computers 8 (1) (1998) 21–66.

[28] Z. Yan, E. Cimpian, M. Zaremba, M. Mazzara, Semantic Business Process Modeling and WSMO Extension, in: Proc. of ICWS'07, 2007, pp. 1185–1186.

[29] J. Lee, G. Yost, the PIF Working Group, The PIF process interchange format and framework, Tech. rep., MIT Center for Coordination Science (1994).

[30] A. Martens, Analyzing Web Service Based Business Processes, in: Proc. of FASE'05, Vol. 3442 of LNCS, Springer, 2005, pp. 19–33.

[31] G. Decker, M. Weske, Interaction-centric Modeling of Process Choreographies, Information Systems 36 (2) (2011) 292–312.

[32] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. van der Werf, J. F. Groote, L. Somers, Transformation of BPMN Models for Behaviour Analysis, in: Proc. of MSVVEIS'07, 2007, pp. 126–137.

[33] F. Corradini, A. Polini, B. Re, F. Tiezzi, An Operational Semantics of BPMN Collaboration, in: Proc. of FACS'15, Vol. 9539 of LNCS, Springer, 2015, pp. 161–180.

[34] N. El-Saber, A. Boronat, BPMN Formalization and Verification using Maude, in: Proc. of BM-FA, ACM, 2014, pp. 1–8.

[35] P. Wong, J. Gibbons, Verifying Business Process Compatibility, in: Proc. of QSIC'08, 2008, pp. 126–131.

[36] P. Y. H. Wong, J. Gibbons, A Relative Timed Semantics for BPMN, Electr. Notes Theor. Comput. Sci. 229 (2) (2009) 59–75.

[37] M. I. Capel, L. E. M. Morales, Automating the Transformation from BPMN Models to CSP+T Specifications, in: Proc. of SEW, IEEE, 2012, pp. 100–109.

[38] L. Mendoza-Morales, M. Capel, M. Pérez, Conceptual Framework for Business Processes Compositional Verification, Inf. & Sw. Techn. 54 (2) (2012) 149–161.

[39] V. Lam, Foundation for Equivalences of BPMN Models, Theoretical and Applied Informatics 24 (1) (2012) 33–66.

[40] K. Kluza, K. Kaczor, Overview of BPMN Model Equivalences. Towards Normalization of BPMN Diagrams, in: Proc. of KESE'12, 2012, pp. 38–45.

[41] M. Reichert, B. Weber, Enabling Flexibility in Process-Aware Information Systems - Challenges, Methods, Technologies, Springer, 2012.

[42] M. Weidlich, R. M. Dijkman, M. Weske, Behaviour Equivalence and Compatibility of Business Process Models with Complex Correspondences, Comput. J. 55 (11) (2012) 1398–1418.

[43] B. F. van Dongen, R. M. Dijkman, J. Mendling, Measuring Similarity between Business Process Models, in: Proc. of CAISE'08, 2008, pp. 450–464.

[44] A. K. A. de Medeiros, W. M. P. van der Aalst, A. J. M. M. Weijters, Quantifying Process Equivalence Based on Observed Behavior, Data Knowl. Eng. 64 (1) (2008) 55–74.

[45] P. Poizat, G. Salaün, A. Krishna, Checking Business Process Evolution, in: Proc. of FACS'16, Vol. 10231 of LNCS, Springer, 2016, pp. 36–53.

[46] N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, W. Serwe, Ten Years of Performance Evaluation for Concurrent Systems using CADP, in: Proc. of ISoLA'10, 2010, pp. 128–142.

A. LNT Types and Functions for the Scheduler Process

The following code block provides LNT specification of BPMN types and functions used in the scheduler process. It also shows how to check for the merge using BPMN 1.0 semantics. BPMN 2.0 merge semantics is simpler (which is not shown here) and it is checked in a similar manner.

```
module bpmntypes(id) with "get" is

(* set of identifiers *)
type IDS is
 set of ID
 with "==", "!=", "inter", "length", "empty", "member", "insert", "union",
      "remove", "diff"
end type

(* flow *)
type FLOW is
 flow ( ident: ID, source: ID, target: ID )
end type

(* set of flows *)
type FLOWS is
 set of FLOW
end type

(* task *)
type TASK is
 task ( ident: ID, incf: IDS, outf: IDS )
end type

(* set of tasks *)
type TASKS is
 set of TASK
end type

(* initial event *)
type INITIAL is
 initial ( ident: ID, outf: ID )
end type

(* final event *)
type FINAL is
 final ( ident: ID, incf: IDS )
end type

(* set of final events *)
type FINALS is
 set of FINAL
end type

(* type of gateway *)
type GSORT is
 xor, and, or
end type
```

```
(* gateway pattern *)
type GPATTERN is
 split, merge
end type

(* gateway *)
type GATEWAY is
 gateway ( ident: ID, pattern: GPATTERN, sort: GSORT, incf: IDS, outf: IDS )
end type

(* set of gateways *)
type GATEWAYS is
 set of GATEWAY
end type

(* node *)
type NODE is
 i ( initial: INITIAL ),
 f ( finals: FINALS ),
 g ( gateways: GATEWAYS ),
 t ( tasks: TASKS )
end type

(* set of nodes *)
type NODES is
 set of NODE
end type

(* bpmn-proc *)
type BPROCESS is
 proc ( name: ID, nodes: NODES, flows: FLOWS )
end type

(* Check for merge with BPMN 1.x semantics *)
function is_merge_possible(p: BPROCESS, activeflows:IDS, mergeid:ID): Bool
    is
 var incf:IDS, inactiveincf:IDS, active_merge:Nat, visited: IDS, result:
     Bool in
  visited := nil;
  (* just iterate through gateways *)
  incf := find_incf(p, mergeid);
  active_merge := find_active_tokens(activeflows, incf);
  (*--check if all the incf have tokens--*)
  if(active_merge == length(incf)) then
   return True
  else
   (* first remove incf with active tokens *)
   inactiveincf := remove_ids_from_set(activeflows, incf);
   (* then check upstream for remaining flows *)
   eval result := check_af_upstream(!?visited, p, activeflows,
       inactiveincf);
   return result
  end if
 end var
end function
```

```
function is_sync_done(p:BPROCESS, activeflows: IDS, syncstore: IDS,
    mergeid:ID): Bool is
 var incf:IDS, activesync: IDS in
  incf := find_incf(p, mergeid);
  activesync := inter(activeflows, incf);
  if (empty(activesync)) then
   return False
  elsif (inter(activesync, syncstore) == activesync) then
   return True
  else
   return False
  end if
 end var
end function

(* Merge check for parallel gateways *)
function is_merge_possible_par(p:BPROCESS, syncstore: IDS, mergeid:ID):
    Bool is
 var incf:IDS, activesync: IDS in
  incf := find_incf(p, mergeid);
  if (inter(incf, syncstore) == incf) then
   return True
  else
   return False
  end if
 end var
end function

(* finds all the upstream flows and checks for tokens *)
function check_af_upstream(in out visited:IDS, p:BPROCESS, activeflows:IDS,
    incf:IDS): Bool is
 var count:Nat, result:Bool, result2:Bool in
 case incf in
  var hd:ID, tl:IDS, upflow:IDS, source:ID in
  cons(hd, tl) ->
     source := find_flow_source(p, hd);
     if(source == DummyId) then
      return True
     elsif (member(source, visited)) then
      eval result := check_af_upstream(!?visited, p, activeflows, tl);
      return result
     else
      visited := insert(source, visited);
      upflow := get_incf_by_id(p, source);
      if (upflow == nil) then
       return True
      end if;
      count := find_active_tokens(activeflows, upflow);
      if(count == 0 of Nat) then
       eval result := check_af_upstream(!?visited, p, activeflows, upflow);
       eval result2 := check_af_upstream(!?visited, p, activeflows, tl);
       return result and result2
      else
       return False
      end if
     end if
  | nil -> return True
```

```
 end case
 end var
end function

function find_flow_source(bpmn: BPROCESS, flowid: ID): ID is
 case bpmn in
  var name: ID, nodes: NODES, flows: FLOWS in
  proc (name, nodes, flows) -> return traverse_flows(flows, flowid)
 end case
end function

function traverse_flows(flows: FLOWS, flowid:ID): ID is
 var dummySource:ID in
  dummySource := DummyId;
  case flows in
   var ident: ID, source: ID, target: ID, tl:FLOWS in
   cons(flow(ident, source, target), tl) ->
    if (ident==flowid) then
     return source
    else
     return traverse_flows(tl, flowid)
    end if
   | nil -> return dummySource
  end case
 end var
end function

(* given a node id, gets its incoming flows *)
function get_incf_by_id(p:BPROCESS, nodeid:ID): IDS is
 case p in
  var name: ID, nodes: NODES, flows: FLOWS in
  proc (name, nodes, flows) -> return traverse_nodes(nodes, nodeid)
 end case
end function

(* Traverse across all nodes in search of the node *)
function traverse_nodes(nodes: NODES, id:ID): IDS is
 case nodes in
  var gateways: GATEWAYS, initial: INITIAL, finals: FINALS, tasks: TASKS,
      tl: NODES, incf:IDS in
  cons(g(gateways), tl) ->
   incf := traverse_gateways(gateways, id);
   if (nil == incf) then
    return traverse_nodes(tl, id)
   else
    return incf
   end if
  | cons(i(initial), tl) -> return traverse_nodes(tl, id)
  | cons(f(finals), tl) ->
   incf := traverse_finals(finals, id);
   if (nil == incf) then
    return traverse_nodes(tl, id)
   else
    return incf
   end if
  | cons(t(tasks), tl) ->
   incf := traverse_tasks(tasks, id);
```

```
    if (nil == incf) then
     return traverse_nodes(tl, id)
    else
     return incf
    end if
   | nil -> return nil
 end case
end function

(* Find incf of gateways *)
function traverse_gateways(gateways: GATEWAYS, id: ID): IDS is
 case gateways in
  var ident: ID, pattern: GPATTERN, sort: GSORT, incf: IDS, outf: IDS, tl:
      GATEWAYS in
  cons(gateway(ident, pattern, sort, incf, outf), tl) ->
   if (ident==id) then
    return incf
   else
    return traverse_gateways(tl, id)
   end if
  | nil -> return nil
 end case
end function

(* Find incf of finals *)
function traverse_finals(finals: FINALS, id: ID): IDS is
 case finals in
  var ident: ID, incf: IDS, tl: FINALS in
  cons(final(ident, incf), tl) ->
   if (ident==id) then
    return incf
   else
    return traverse_finals(tl, id)
   end if
  | nil -> return nil
 end case
end function

(* Find incf of taks *)
function traverse_tasks(tasks: TASKS, id: ID): IDS is
 case tasks in
  var ident: ID, incf: IDS, outf: IDS, tl: TASKS in
  cons(task(ident, incf, outf), tl) ->
   if (ident==id) then
    return incf
   else
    return traverse_tasks(tl, id)
   end if
  | nil -> return nil
 end case
end function

(* Remove Incoming flows from activetokens *)
function remove_incf(bpmn:BPROCESS, activeflows:IDS, mergeid:ID): IDS is
 var incf:IDS in
  incf := get_incf_by_id(bpmn, mergeid);
  return remove_ids_from_set(incf, activeflows)
```

```
 end var
end function

function remove_sync(bpmn:BPROCESS, syncstore:IDS, mergeid:ID): IDS is
 return remove_incf(bpmn, syncstore, mergeid)
end function

function find_incf(p: BPROCESS, mergeid:ID): IDS is
 case p in
  var name: ID, nodes: NODES, flows: FLOWS in
  proc (name, nodes, flows) -> return find_incf_nodes(nodes, mergeid)
 end case
end function

function find_active_tokens(activeflows:IDS, incf:IDS): Nat is
 var tokens:IDS, count:Nat in
  tokens := inter(activeflows, incf);
  count := length(tokens);
  return count
 end var
end function

(* Helper function to remove a set of IDS *)
function remove_ids_from_set(toremove:IDS, inputset: IDS): IDS is
 return diff (inputset, toremove)
end function
```

Listing 1: LNT Types and Functions used in the Scheduler Process.