Analytic Evaluation of BWL Nets

Cristian IONITĂ
Academy of Economic Studies, Bucharest, Romania
crionita@ie.ase.ro

Workflow management systems play a central role in supporting the business operations of medium and large organizations. Because of this and the increasing complexity of the processes the properties of the languages used to describe those processes are becoming very important. This paper analyses the structural properties of the BWL process definition language. It defines a new class of Petri nets called BWL networks and uses it to prove that the control flow of BWL programs is structurally sound. The design of the language ensures that the modeled processes are inherently free from common structural problems.
Keywords: *BWL, Workflow, Petri Nets, Evaluation, Open System*

1 Introduction

All modern businesses depend on complex business processes in order to conduct their daily activities. These processes involve documents, people and internal or external information systems. Traditional task based systems support the user in performing specific tasks, but they fail to integrate all the aspects involved in a typical business process. In order to manage holistically the business processes running inside an organization a new class of information systems named Process Aware Information Systems (PAIS) are used. According to [3], process aware information systems are information systems that manage and execute operational processes involving people, applications, and / or information sources on the basis of process models.

In order to be executable inside a PAIS, a process must be described using a formal language. There are numerous process definition languages created by the industry or the academic community. The most important workflow languages are BPEL - Business Process Execution Language, BPMN - Business Process Modeling Notation and Yet Another Workflow Language - YAWL. One of the main problems identified using those languages is the complexity burden imposed on the process designer ([2]). When using those languages, the process designer is responsible for ensuring that the process models created are free from structural problems like livelocks, deadlocks, dangling tasks and other similar issues.

In [4], [5] a new process definition language called Business Workflow Language – BWL and the associated platform called DocuMentor was introduced. In this paper is proven that the language design of BWL guarantees that process models written in it are free of structural errors. The proof is based on translating the BWL programs into corresponding specialized Petri nets called BWL Networks and demonstrating that the equivalent net is sound.

2 Structural problems in existing workflow languages

In order to illustrate the structural problems that can arise in modeling processes using the existing workflow nets we consider a set of process descriptions using the YAWL (adapted from [1]) and BPMN languages.

The first example of a process considered is a simple insurance claim processing workflow. The process consists of five tasks:

- recording the receipt for the claim (*receipt*);
- the client's policy is checked to determine its validity and confirm that it covers what has been claimed for (*checkPolicy*);
- the claim is checked in to determine if it's a legitimate one and the amount payable to the customer (*checkClaim*);
- a rejection letter is sent to the customer if the verification tasks have found problems

with the policy or the claim (*sendLetter*);
- the amount determined in task *checkClaim* is paid to the customer if both the policy and the claim are found valid (*pay*).

Tasks checkClaim and checkPolicy must be performed after the claim is registered (*receipt*) and can be performed either sequentially or in parallel. The *sendLetter* task must be performed if the result of either *checkPolicy* or *checkClaim* is negative, otherwise the *pay* task must be executed. All process instances must execute exactly four tasks: *receipt*, *checkPolicy*, *checkClaim* and one of *sendLetter* and *pay*.

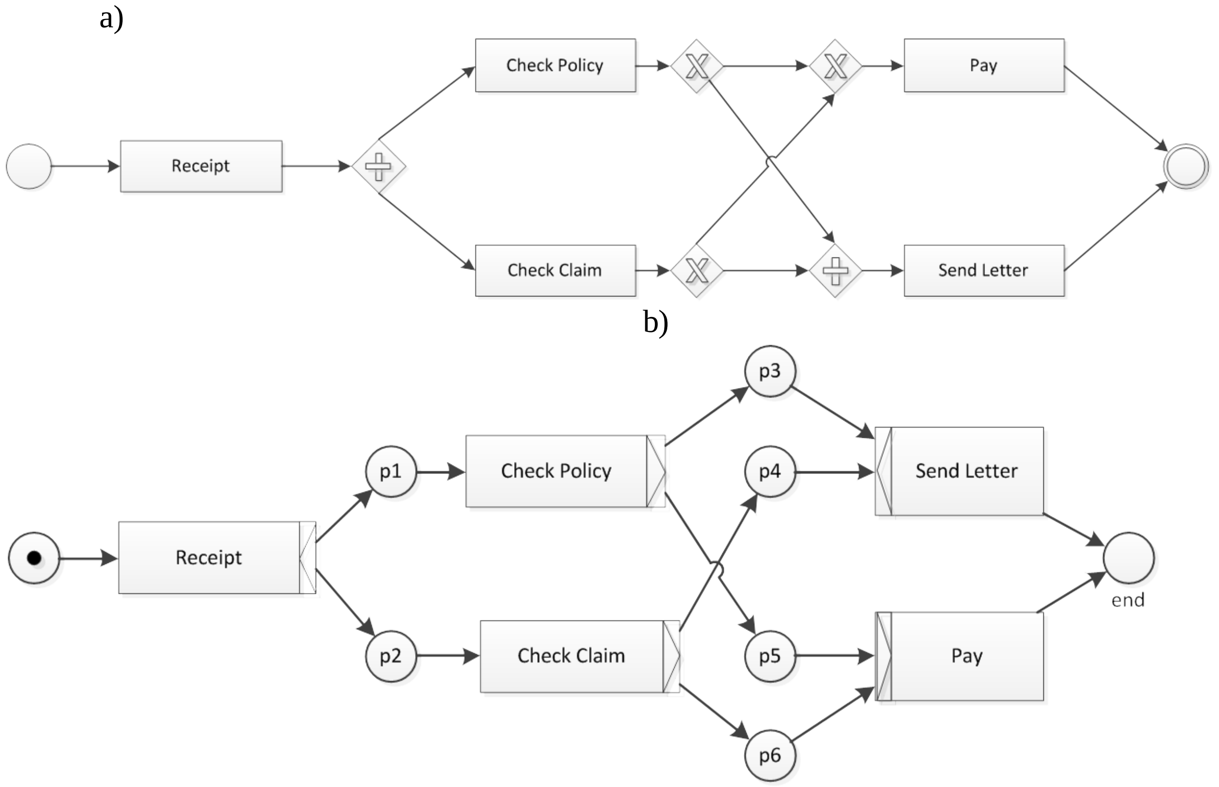

Fig. 1. BPMN (a) and YAWL (b) representation for the claim processing process

Figure 1 shows the claim processing sample described using the BPMN and YAWL languages. Although the syntax is different, the overall structure and semantics are exactly the same and programs are equivalent. Both languages use a PETRI net like structure that allows the user to combine the routing constructs (OR split, AND split, OR join and AND join) in any way, without any checks at syntax level. At first sight, the process definitions seem to accurately follow the process description. They execute the *receipt* task first and then the *checkPolicy* and *checkClaim* tasks are executed in parallel. After the checks are made, the claim is sent to payment or a rejection letter is sent and the process ends.

The combination of routing structures used in Figure 1 generates a series of problems. If both the *checkClaim* and *checkPolicy* tasks generate negative results then both *p3* and *p4* places will receive a token. Because the *sendLetter* contains an OR split the task will be executed twice and two letters will be sent to the customer. If the policy is not valid, but the claim is valid, then the places *p3* and *p6*will receive a token. The *sendLetter* will execute correctly, but the *pay* task, being guarded by an AND join, will remain indefinitely with a token in *p6* waiting for another token in *p5* to continue. A similar situation will arise if the policy is valid, but the claim is not valid. The only situation when the process is executed correctly is when both the claim and policy are valid.

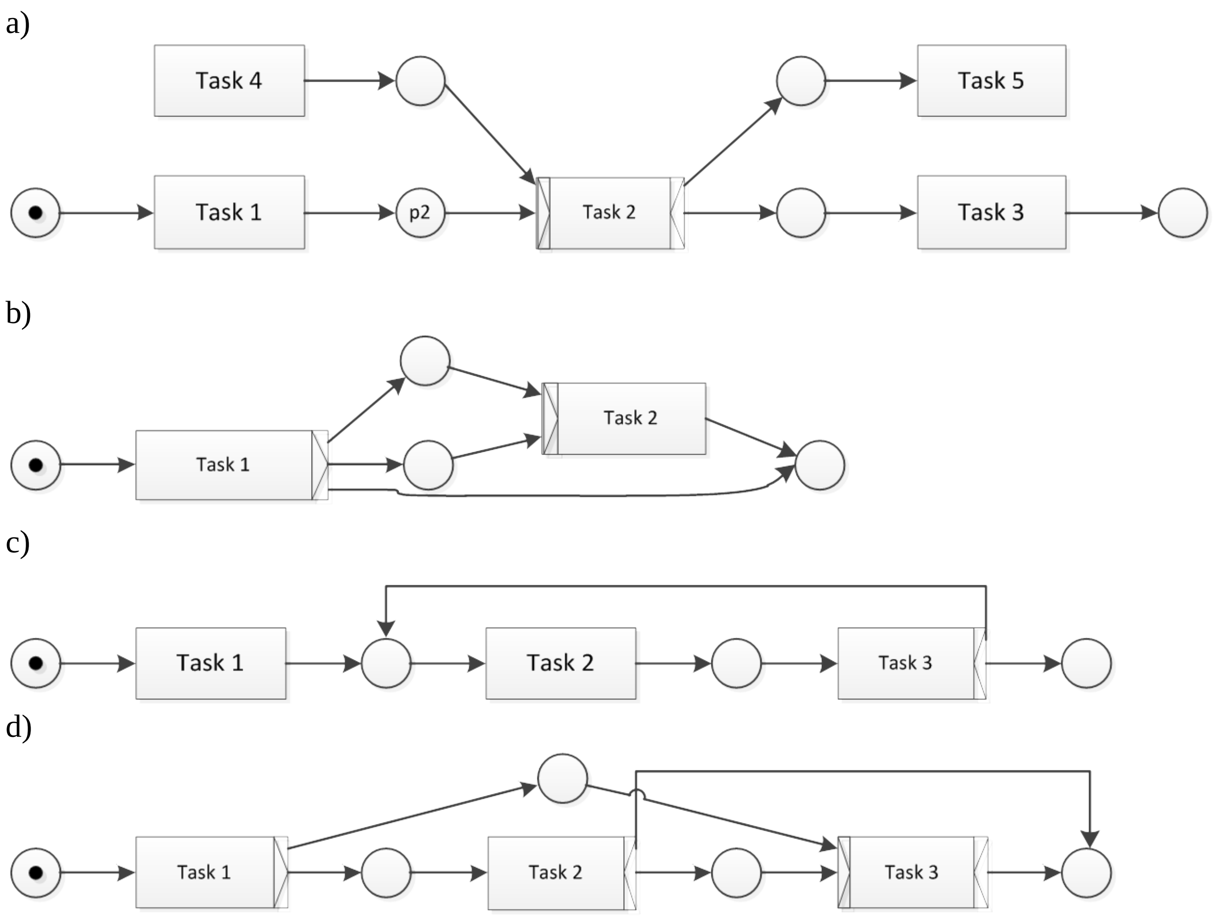

Fig. 2. Structural problems examples using the YAWL language [1]

The incorrect combination of routing structures in languages like BPMN and YAWL results in process definitions that contain evident or subtle structural problems. Mendling etal have shown in [6] that these errors are common and can be found frequently even in production code. In the paper is shown that from the 604 non-trivial process descriptions included in the SAP Reference Model at least 34 manifest structural control flow errors. Figure 2 shows four examples of process definitions that contain such errors. According to [1] these errors can be classified in six categories:

1. Tasks without input and/or output places

In Figure 2.a *Task 4* has no input places and, because of this, the moment of execution can't be determined. In the same example *Task 5* has no places and its execution is unnecessary for the completion of the process.

2. Dead tasks

Dead tasks are tasks that can never be completed because they can never accumulate the required tokens in their input places. *Task 2* from Figure 2.b can never be completed because the OR split from *Task 1* can place only a token in one of its output places. The same applies for *Task 3* in Figure 2.d.

3. Deadlock

A deadlock appears when the process can never be completed because is stuck waiting for tokens in some places. In Figure 2.b if the *Task 1* places a token in one of its top two output places then the process will wait forever for the execution of *Task 2* to begin. The process can be finished successfully only if *Task 1* places a token directly in the end place.

4. Livelock

A livelock occurs when the process can't be completed because is trapped into an endless cycle. This can happen in processes that contain iterative sections (like *Tasks 2* and *Task 3* in Figure 2.c) that always place tokens inside or before the loop before finishing.

5. Activities still take place after the condition "end" is reached

A correct process definition should complete when its end place is reached. This means that there should be exactly one token in the end place and no tokens elsewhere. In figure 2.c *Task 2* and *Task 3* will have tokens in their input places and will be able to execute after the end place of the process is reached.

6. Tokens remain in the process definition after the case has been completed

If the process is completed as a result of the firing of *Task 1* in Figure 2.d then there will remain a token in one of the places before *Task 3*. In this case it's unclear when the process execution ends.

These control flow errors are a direct result of the fact that the syntaxes of the analyzed process description languages don't have any means to restrict the user to use only correct combinations of routing constructs.

3. BWL Programs and Networks

The BWL language ([4], [5]) avoids these types of errors by grouping the routing constructs in block structured control flow instructions. Unlike the graph-like structure used by BPMN and YAWL that allows any combination of routing constructs, BWL allows only correct combinations. This is accomplished by grouping the routing constructs inside the control flow instructions like *sequence*, *if*, *while*, *parallel* and *firstOf*. The user selects only the high level control flow instructions, and the execution engine is responsible of generating the correct control routing constructs at the moment of execution.

To illustrate the use of the control structures of the proposed language the claim insurance process from section 1 was rewritten using BWL.

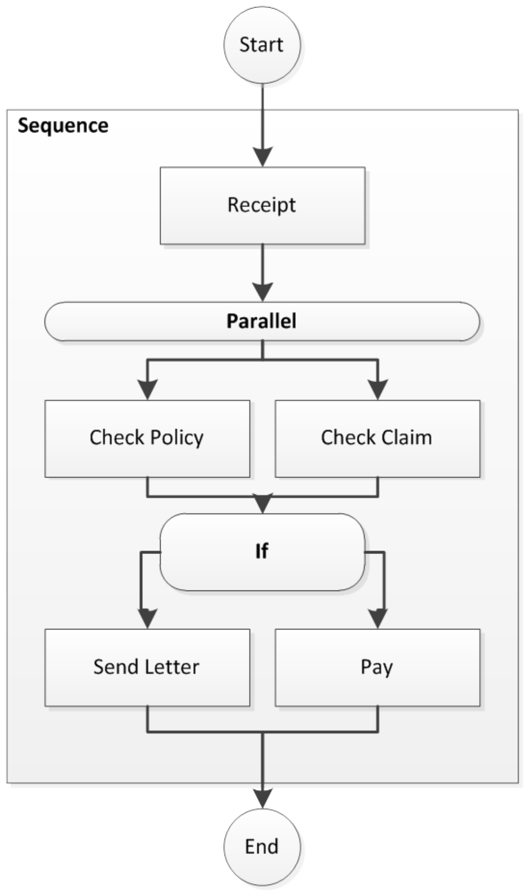

Fig. 3. BWL representation for the claim processing process

The BWL implementation of the process respects all the requirements presented in the process description. The *sequence* instruction ensures that the *Receipt* task is executed first. The *parallel* instruction executes its child tasks *checkPolicy* and *checkClaim* in parallel and continues only after both are completed. The *if* instruction executes exactly one of the tasks *pay* and *sendLetter* depending on the outcome of the *checkPolicy* and *checkClaim* tasks. In the BWL implementation the problems found in the processes from Figure 1 are avoided.

In order to evaluate the properties of BWL programs, a new class of Petri net called BWL Network is proposed. The BPNT algorithm was created to translate the abstract syntax tree of a BWL program to an equivalent BWL network for analysis.

A **BWL Network** (BWL) is a touple $RB = (P, T, F, tt, te)$, where:

- $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of places;
- $T = \{t_1, t_2, \dots, t_n\}$is a finite set of transitions;
- $F \subseteq (P \times T) \cup (T \times P)$is the set of arcs;
- $tt: T \rightarrow BNT, BNT = \{Simple, SeqStart, SeqEnd, IfTrue, IfFalse, WhileBegin, WhileTrue, WhileFalse, ParallelStart, ParallelEnd, FirstOfStart, FirstOfEnd\}$is the function that maps a BWL type to each transition;
- $te: T \rightarrow EV, \quad EV = \{(e_1, e_2, \dots, e_m) | 0 \leq m < n, e_i \in \{i, e, c, f, t\}\}$is the function that maps an event chain to each transition;
- $in \in P$ is the start place, where$\cdot in = \emptyset$;
- $out \in P$ is the end place where $out \cdot = \emptyset$;
- all places and transitions are placed on a path from in to out.

Every activation of a transition $t \in T$ generates a sequence of events $e \in EV$, where $e = te(t)$. A sequence $t_1, t_2, \dots, t_{n-1} \in T$ for which $t_1 \in in \cdot$ și $t_{n-1} \in \cdot out$ generates an event sequence $e = \bigcup_{i=1}^{n-1} te(t_i)$ called the execution trace of the program.

In order to evaluate the structural properties of a BWL program using BWL nets, an algorithm named BPNT (BWL Program to Net Transformer) was created. It maps the abstract syntax tree of any BWL program to its equivalent BWLN. An equivalent BWLN for a BWL program is a net $RB = (P, T, F, tt, te)$with the property that any execution trace$e \in E_{RB}$follows the semantic rules of the BWL language.

The **BPNT algorithm** starts with a BWL program $PB = (A, type, c)$and builds an equivalent BWLN$RB = (P, T, F, tt, te)$ using the following steps:

Step 1:

Let $RB = (P, T, F, tt, te)$be a BWLNwith $P = \{in, out\}$, $T = \{t_0\}$, $F = \{(in, t_0),$ $t0, out,$ $type t0 = Simple$and $PB = (A, type, c)$the source BWL program.

Let$SAct = \emptyset$be an empty stack with the following element structure: $(A_i \in A, t \in T)$.

Step 2:

The initial element (A_1, t_0) is added to the $SAct$stack.

Step 3:

While $SAct \neq \emptyset$ steps 4 – 7 are executed.

Step 4:

We extract the top element $(A_{crt}, t_{crt})$from $SAct$. Transition $t_{crt}$is replaced with the subnet corresponding to the A_{crt} statement.

Step 5:

A temporary BWLN called RB_{tmp}composed ofP_{tmp}, T_{tmp}, F_{tmp} and $tt_{tmp}$is created for the activity A_{crt}.

Step 6:

The current transition from RB is replaced by the subnet generated at step 5 using the following operations: $P = P \cup (P_{tmp} \setminus \{in, out\})$, $T = T \setminus \{t_{tmp}\} \cup T_{tmp}$, $F = F \setminus \{(*, tcrt, (*, tcrt)\} \cup Ftmp$ and $tt = tt \cup tt\_tmp$.

Step 7:

For each activity $A_j \in \{A_i \in c(A_{crt}) | type(A_i) \neq Simple\}$a new element $(A_j, t_j)$is added to $SAct$, where$t_j \in T_{tmp}$ is the transition generated by step 5 corresponding to activity A_j.

Construction of the sets is step 5 is done according to the semantic of each BWL statement. Figure 4 shows the mapping used for the BWL control flow statements.

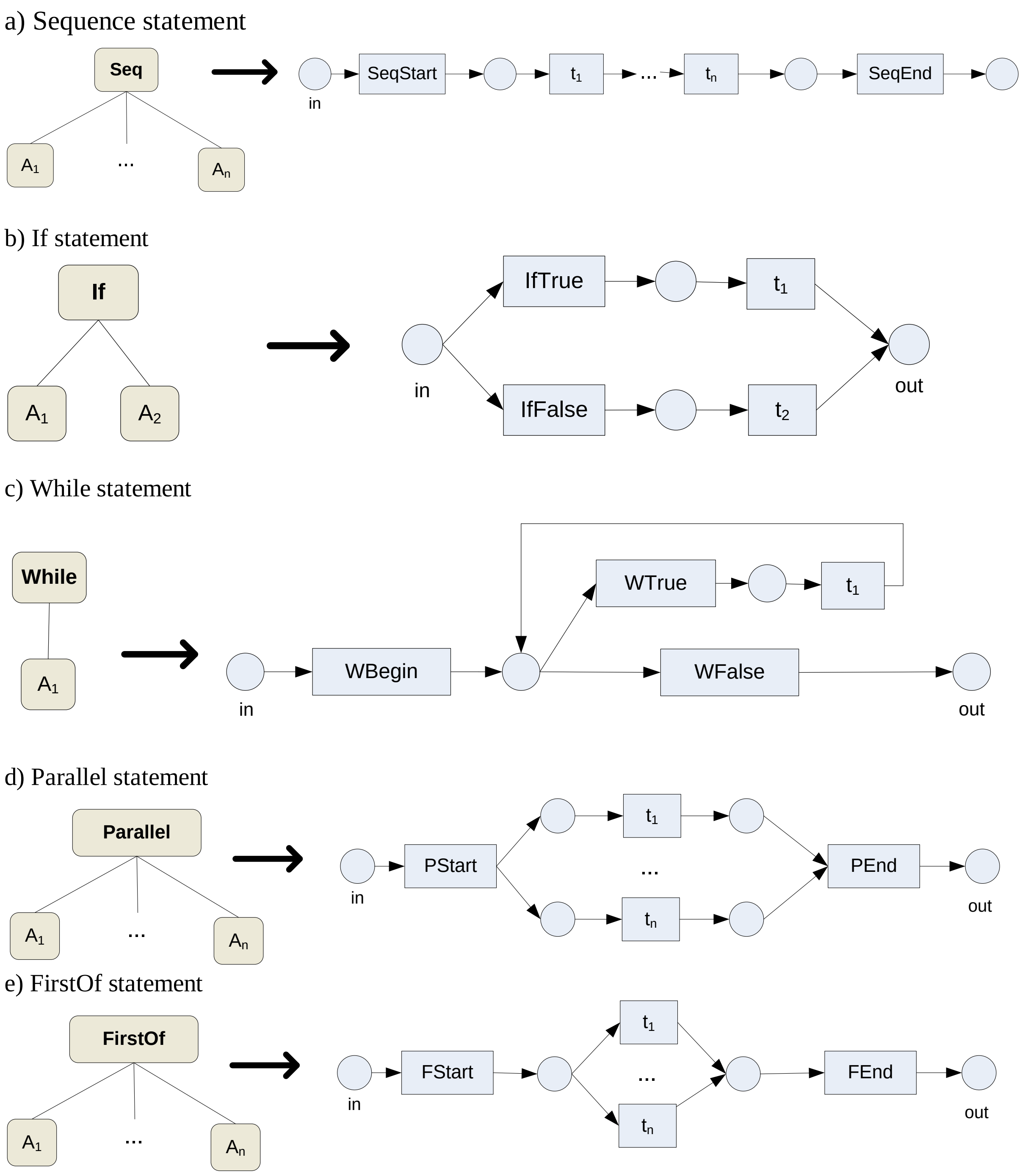

Fig. 4. Corresponding BWL networks for the *BWL* control flow statements

The BPNT algorithm works on the abstract syntax tree obtained by applying the BWL grammar [4] to the program. Figure 5 shows the abstract syntax tree obtained for the insurance claim process presented in Figure 3

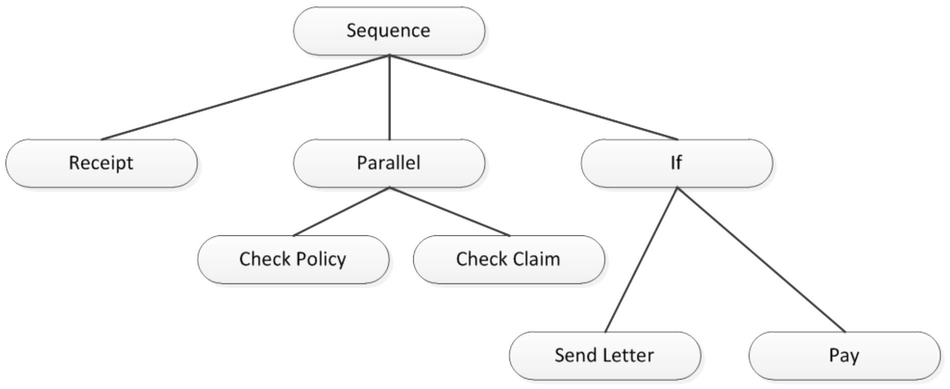

Fig. 5. Abstract syntax tree for the BWL claim processing process

The abstract syntax tree (figure 5) was converted to the equivalent BWL network using the BPNT algorithm.

Figure 6 shows the resulting BWL network and the program state after each algorithm iteration.

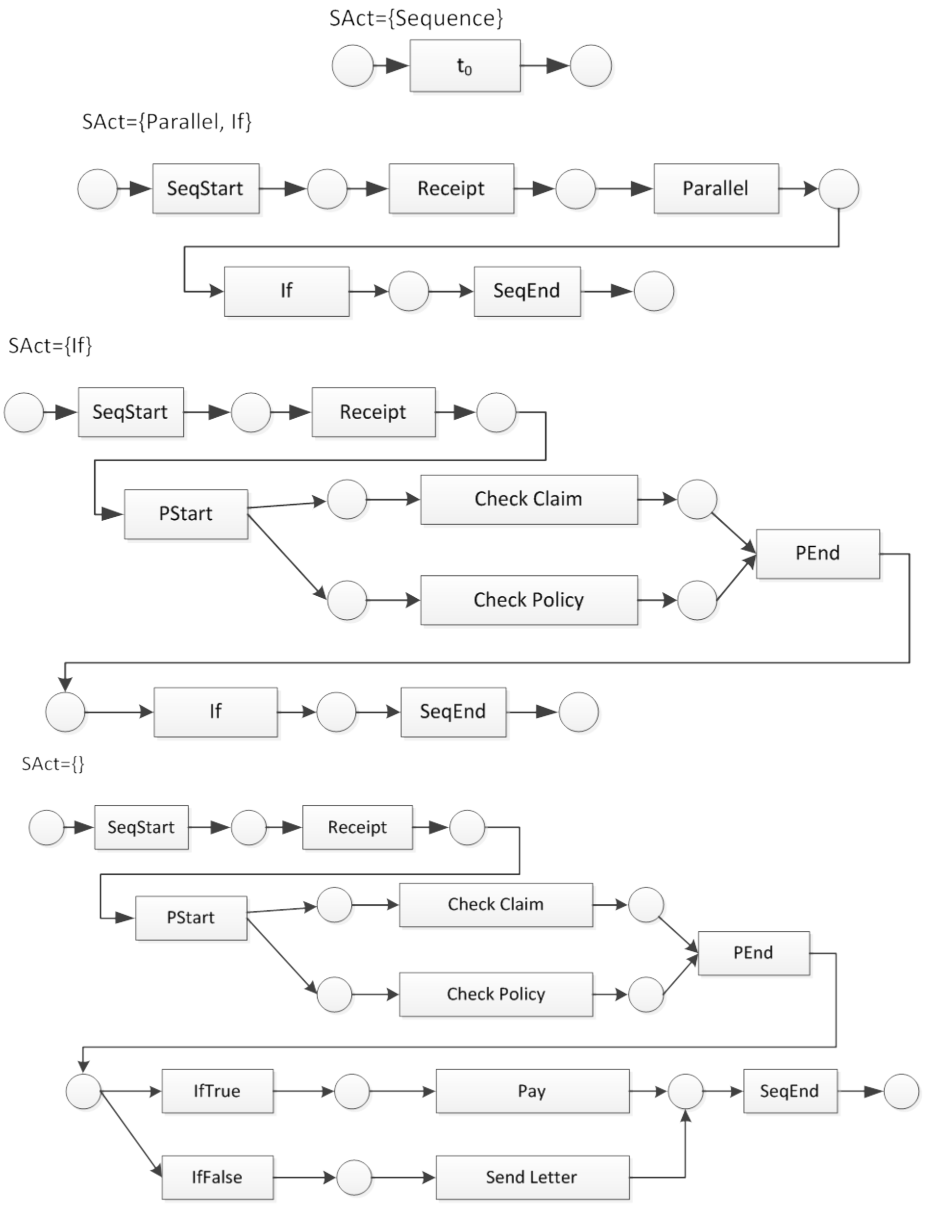

Fig. 6. BPNT algorithm application

The BWLN obtained after applying the algorithm to the abstract syntax tree represents accurately the behavior of the BWL program. The network can be used to analyze the structural properties of the program.

3 Soundness in BWL networks

In order to analyze the structural properties of BWL programs the notion of soundness was defined for BWL nets. According to [3], if the BWL network is sound then the equivalent BWL program is free of structural errors.

A BWL program with the equivalent BWL network $RB = (P, T, F, ttype)$is **sound** if and only if it meets the following three conditions:

- ***SC1:*** $\forall M(sin \overset{*}{\rightarrow} M) => (M \overset{*}{\rightarrow} sout)$ (for any place accessible from the initial state there is an activation sequence that leads the system to the final state $sout$);
- ***SC2:*** $\forall M(sin \overset{*}{\rightarrow} M \land M \geq sout) => M = sout)$ (the only state accessible from the initial state that contains a token in the final state is $sout$);
- ***SC3:*** $\forall t \in T there are M \ and \ M' with sin \overset{*}{\rightarrow} M \overset{t}{\rightarrow} M'$ (there are no dead transitions in the t state).

Mathias Weske has shown in [7] that the soundness problem for WF nets is decidable in polynomial time for free choice nets. A similar extension technique was used to prove that all BWLNs built using the BPNT algorithm are sound. It was shown the relationship between the liveness and boundness properties of the extended net and the soundness property and the fact that the networks built using BPNT respect those properties.

If $RB = (P, T, F, tt, te)$is a BWLN, then the net $RB_{ext} = (P, T_{ext}, F_{ext}, tt_{ext}, te)$, where $T_{ext} = T \cup \{t_{ext}\}$, $F_{ext} = F \cup \{(t_{ext}, in), (out, t_{ext})\}$ and $tt_{ext} = tt \cup \{(t_{ext}, Simple)\}$ is called the extended BWL network. The following propositions were proved for BWL networks:

Proposition 1: If the extended net $RB_{ext} = (P, T_{ext}, F_{ext}, tt_{ext}, te)$is live and bounded then the original net $RB = (P, T, F, tt, te)$is sound.

Proposition 2: If the $RB$is a sound net then the extended net $RB_{ext}$is bounded.

Proposition 3: If the RB is a sound net then the extended net RB_{ext} is live.

Theorem 1: A BWL net $RB = (P, T, F, tt, te)$is sound if and only if the extended net $RB_{ext} = (P, T', F', tt', te)$is live and bounded.

The proof for theorem 1 results directly from propositions 1-3. Using the theorem it was proved that all subnets used in the BPNT step 5 of the algorithm are sound. For exemplification we consider the case of the subnet corresponding to the *while* statement (Figure 2).

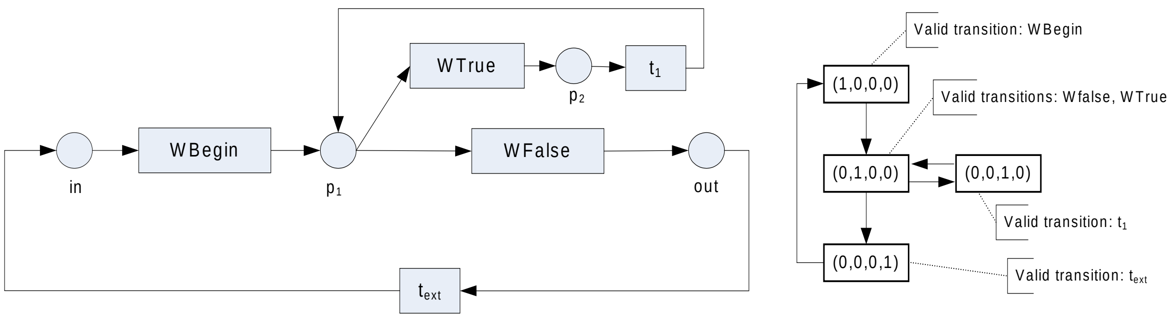

Fig. 7. Extended BWLN and reachability graph for the *while* statement

Let RB_{tmp} be the BWLN for the *while* statement and $RB_{ext}$the extended net.Each of $WBegin, WTrue, WFalse, t_1 \ and \ t_{ext}$transitions uses one token and generates exactly one token as output. The total number of tokens inside the net is constant. The initial state contains exactly one token, so the total number of tokens inside the net is 1 for any possible state. The extended net RB_{ext} is bounded with $k = 1$. From the reachability graph (figure 2) results that for every reachable state M and every transition t there is a state reachable from M that enables t, so the subnet is live. Because the RB_{tmp} is live and bounded it is also sound.

Theorem 2: If $RB = (P, T, F, tt, te)$ and $RB_{tmp} = \left(P_{tmp}, T_{tmp}, F_{tmp}, tt_{tmp}, te_{tmp}\right)$ are two sound BWL networks then the RB_{fin} network build by replacing the t_x transition from RB with the RB_{tmp} subnet following step 5 from the BPNT algorithm is a viable network.

Let in_{tmp} and out_{tmp} be the initial and final states for the RB_{tmp} net. From SC1 RB_{tmp} there is a firing sequence $\sigma_{tmp} = (t_1, t_2, \dots, t_n)$ with $t_i \in T$ for which $\text{sin}_{tmp} \xrightarrow{\sigma_{tmp}} sout_{tmp}$. From SC3 we know that firing the sequence σ_{tmp} in BT_{fin} is the same as firing transition t_x in BT. The operation move a token from location in_{tmp} to location out_{tmp}.

Let M be a state accessible from the initial state of RB_{fin}. If M doesn't contain tokens in one of the locations from P_{tmp}, then a sequence can be built that brings the system from state M in state $sout$ by replacing transaction t_x (if necessary) from the sequence obtained from SC1 applied on RB with the σ_{tmp} sequence. If M contains tokens id one of the locations from P_{tmp} then a sequence σ_{Mfin} to bring the system in state $sout$ can be built using the following rules:

- we build a state M_{tmp} for BT_{tmp} by taking the tokens corresponding to locations from P_{tmp} from state M; because BT_{tmp} is sound we have a sequence σ_{Mtmp} which brings the system in $sout_{tmp}$;
- using the definition we have a sequence σ_M that brings BT from state M to state $sout$;
- we build the sequence $\sigma_{Mfin} = \sigma_{Mtmp} \cup \sigma_M$ which brings BT_{fin} from state M to state out.

The condition SC1 is satisfied. The same rules can be used to prove condition SC3.

Let M be a state of BT_{fin} with $M \geq sout$ and σ_{fin} a sequence that brings the system from state sin to state M. If σ_{fin} doesn't contain transitions from T_{tmp} then $M = sout$ (using SC3). If σ_{fin} contains a sequence from T_{tmp} let M_0 be the state of BT_{fin} before the first transition of the sequence and M_1 the state after the last transition. Let M be a state of BT_{fin} with $M \geq sout$ and σ_{fin} a sequence that leads the system from state sin to state M. If σ_{fin} doesn't contain transitions from T_{tmp}, then $M = sout$ by SC3. If σ_{fin} contains a sequence from T_{tmp} then let M_0 be the state of BT_{fin} before the first transition of the sequence and M_1 the state after the last transition of the sequence. Using SC2 on BT_{tmp} we have $M_0\left(in_{tmp}\right) = 1, M_0\left(out_{tmp}\right) = 0$ and $M_1\left(in_{tmp}\right) = 0, M_1\left(out_{tmp}\right) = 1$. The sequence σ_{fin} for BT_{fin} is equivalent with cu sequence σ for BT obtained by replacing the transitions from T_{tmp} with the transition t_x. Using SC2 we have $M = out$,so SC2 is satisfied. Because all three conditions for the BT_{fin} are satisfied the net is sound.

Using theorem 2 and the properties of the subnets corresponding to the BWL statements it was proved that any BWT net obtained using the BPNT algorithm is sound. Because the algorithm is able to map any BWL program to the equivalent BWLN, all BWL programs are unaffected by structural problems.

4 Conclusions

Business processes modeled using workflow languages are increasingly complex. Because those process models are executed automatically by the system and have an immediate impact on the business it's critical to ensure that the models are correct. In [2] and [6] it's shown that process size, complexity and tools used have an important impact on process definition. Because BWL programs are proved to be sound, designers can use the language to express complex processes correctly, without worrying about the structural problems that might appear.

References

[1] W. M. P. van der Aalst, K. van Hee, *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*, The MIT Press, 2004, 368pp.

[2] D. Birkmeier, S. Kloeckner, S. Overhage, "An Empirical Comparison of the Usability of BPMN and UML Activity Diagrams for Business Users", *International Conference on the Quality of Software Architectures*, *Lecture Notes in Computer Science*, Prague 2010, pp 119-134.

[3] M. Dumas, W.M.P. van der Aalst, A.H.M. ter Hofstede, *Process Aware Information Systems. Bridging People and Software Through Process Technology*, Wiley Interscience, 2005.

[4] C. Ioniţă, "A Domain Specific Language for Secure Document Management", *Proceedings of the Eighth International Conference on Informatics in Economy*, Bucharest 2007.

[5] C. Ioniţă, "Collaborative Business Process Optimization Using Domain Specific Languages", *Proceedings of the ninth International Conference on Informatics in Economy*, Bucharest 2009.

[6] J. Mendling, M. Moser, G. Neumann, H. Verbeek, B. van Dongen, W. van der Aalst, "Faulty EPCs in the SAP Reference Model", *Proceedings of Business Process Management Conference*, *Lecture Notes in Computer Science*, Vienna 2006, pp 451-457.

[7] M. Weske, *Business Process Management: Concepts, Languages, Architectures*, Springer-Verlag, Berlin 2007.

Cristian IONITA has graduated the Faculty of Cybernetics, Statistics and Economic Informatics in 2004. He is a PhD student at the Academy of Economic Studies Bucharest since 2006 and a teaching assistant inside the Department of Computer Science in Economics at Faculty of Cybernetics, Statistics and Economic Informatics. Main interest topics are programming languages, data structures, distributed applications and enterprise systems. He is the author or coauthor of 16 journal articles and coauthor in 2 books on these topics.