Investigations on Soundness Regarding Lazy Activities

Frank Puhlmann and Mathias Weske

Business Process Technology Group
Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
{puhlmann,weske}@hpi.uni-potsdam.de

Abstract. Current approaches for proving the correctness of business processes focus on either soundness, weak soundness, or relaxed soundness. Soundness states that each activity should be on a path from the initial to the final activity, that after the final activity has been reached no other activities should become active, and that there are no unreachable activities. Relaxed soundness softens soundness by stating that each activity should be able to participate in the business process, whereas weak soundness allows unreachable activities. However, all these kinds of soundness are not satisfactory for processes containing discriminator, n-out-of-m-join or multiple instances without synchronization patterns that can leave running (lazy) activities behind. As these patterns occur in interacting business processes, we propose a solution based on lazy soundness. We utilize the π-calculus to discuss and implement reasoning on lazy soundness.

1 Introduction

Business Process Management (BPM) aims at designing, enacting, managing, analyzing, and adapting business processes [1]. A key technology for implementing BPM systems are service-oriented architectures (SOA). These aim at supporting business processes within and between companies [2]. However, they also increase the complexity to be modeled, especially regarding interacting business processes. Thus, special care has to be taken during the design phase to avoid errors leading to deadlocks or livelocks. The former leads to processes stopping execution and interaction with their environment, whereas the latter might continue working but the process is never finished. Three major approaches for analyzing the correctness of business processes have been published: *Soundness* [3], *Relaxed Soundness* [4], and *Weak Soundness* [5]. All three approaches operate on a special type of business processes, called workflow nets [6] but they can be adapted to graph-based approaches like BPMN, EPC, or UML Activity Diagrams.

However, soundness, relaxed soundness, and weak soundness are not satisfactory for processes containing discriminator, n-out-of-m-join or multiple instances without synchronization patterns. These patterns are required in interacting business processes for representing interaction patterns [7], as *Racing Incoming Messages* (discriminator), *One to many Send/Receive* (n-out-of-m-join) or execute secondary tasks (multiple instances without synchronization). All of these patterns can leave activities behind that are or can become active after the final activity has been reached. Thereby, all processes containing these patterns are not sound per definition (i.e. in terms of Petri nets they leave tokens in the net). One example is a business process where three experts are asked to write an expertise each. The process can continue after two expertises have been received. Only in certain cases a follow up activity has to wait for all three expertises to continue, e.g. if the first two expertises are very different. As the experts need different time for responding, the business process could have been already finished while the last expert is still writing her expertise. However, all three experts have to be paid after delivering their work. In this case, there is a *clean-up* or *lazy-activitiy* remaining (pay the last expert) that does in certain cases not directly contribute to the successful execution of the business process, but is an integral part of it.

Nevertheless, the given example might be relaxed sound. Relaxed sound processes, in turn, might contain deadlocks or livelocks that should be avoided. Weak soundness in contrast allows no activities to be active after the final activity has been reached. To overcome the limitations of soundness and weak soundness regarding these patterns, and to go beyond relaxed soundness by proving deadlock and livelock freedom, we propose a solution based on lazy soundness. Lazy soundness will be derived, discussed and implemented based on business processes formalized in the π-calculus, thus extending our prior work [8].

The paper is structured as follows. We first extend our motivation and discuss related work, followed by the preliminaries required for formal process representation and analysis. The main part introduces lazy soundness for formalized business processes, also including a running example. A tool support section shows how the theoretical results can be applied practically and also takes a look at performance. The paper concludes with an outlook of future work.

2 Motivation and Related Work

During our research on soundness for business processes defined in the π-calculus [9], a process algebra that can formally represent all Workflow patterns [8] as originally described in [10], we analyzed the soundness of discriminator, n-out-of-m-join and multiple instances without synchronization patterns (denoted as critical patterns in the remainder). These patterns can leave activities behind that are or can become active after the final activity has been reached. Soundness, in contrast, states that no activities are or can become active after a final activity has been reached. Thus, all processes containing these patterns are not sound per definition.

We investigated weak and relaxed soundness for supporting the critical patterns [4,5]. Relaxed soundness indeed supports the patterns but relaxed sound

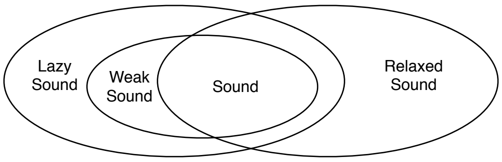

Fig. 1. A classification for different kinds of soundness.

processes might contain livelocks and deadlocks. Weak soundness proves processes to be free of locks, but also forces all activities to finish before the final one. Thus, it does not support the critical patterns. To overcome these limitations we propose lazy soundness, complementing relaxed soundness by covering livelocks and deadlocks, and extending weak soundness by allowing activities to become active after the final activity has been reached. Unreachable activities are not covered. However, by combining relaxed and lazy soundness we can prove processes to be free of deadlocks, livelocks, and dead activities.

Figure 1 gives a classification of the different kinds of soundness. Lazy soundness states that if an activity is reachable from the initial activity, then the final activity is always reachable from this activity. (guarantees deadlock and livelock freedom). Furthermore, the final activity will only be reached once to denote the successful execution of the business process. *Clean-up* or so called *lazy-activities* might still be or become active. Relaxed soundness states that all activities of a business process participate in it (dead activity freedom). A relaxed sound process might contain deadlocks or livelocks. Weak soundness is a subset of lazy soundness by prohibiting lazy-activities, but still permitting dead activities. The rules for soundness are fullfilled by the intersection of weak and relaxed soundness, representing deadlock, livelock, and dead activity free processes without lazy activities. The intersection of relaxed and lazy soundness without soundness will not be investigated in this paper. Nevertheless, it offers interesting properties.

An important piece of related work is YAWL [11]. YAWL claims to support all workflow patterns, but actually redefines some of them to fit the YAWL semantics. Actually, the semantics of the critical workflow patterns has been changed.[1] A YAWL discriminator cancels all other tasks before the discriminator. A YAWL n-out-of-m-join only joins instances of the same activity, using a multiple instance pattern. Finally, a multiple instances without synchronization task has to be joined by an OR-join. All three solutions contradict the original workflow patterns but allow a YAWL net to be sound. In [12], an approach of reasoning in YAWL focusing on relaxed soundness is introduced, i.e. it requires all activities of a business process to be on a path from the initial to the final activity. However, this kind of reasoning also allows deadlocks and livelocks in the process, which is too relaxed regarding formal analysis. Lazy soundness, in

[1] See `http://www.yawl.fit.qut.edu.au/about/patterns/` for details.

contrast, is based on π-calculus formalizations of the workflow pattern [8], that capture the original semantics of the critical patterns, as these is required for interacting business processes and even special cases of traditional ones.

3 Preliminaries

This section introduces the π-calculus and the representation of business processes in it. Our motivation on using the π-calculus rather than other formalisms like Petri nets is discussed in [13].

3.1 The π-calculus

The π-calculus is an algebra for the formal description and analysis of concurrent, interacting processes with support for link passing mobility. It is based on names and interactions used by processes defined according to [14].

Definition 1 (Pi Calculus). *The syntax of the π-calculus is given by:*

$$\begin{aligned} P &::= M \mid P|P' \mid \mathbf{v}zP \mid !P \\ M &::= \mathbf{0} \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle\tilde{y}\rangle \mid x(\tilde{z}) \mid \tau \mid [x = y]\pi \ . \end{aligned}$$

The informal semantics is as follows: $P|P'$ is the concurrent execution of P and P', $\mathbf{v}zP$ is the restriction of the scope of the name z to P, and $!P$ is an infinite number of copies of P. $\mathbf{0}$ is inaction, a process that can do nothing, $M + M'$ is the exclusive choice between M and M'. The output prefix $\overline{x}\langle\tilde{y}\rangle.P$ sends a sequence of names $\tilde{y}$ over the co-name $\overline{x}$ and then continues as P. The input prefix $x(\tilde{z})$ receives a sequence of names over the name x and then continues as P with $\tilde{z}$ replaced by the received names (written as $\{{}^{\tilde{name}}/_{\tilde{z}}\}$). Matching input and output prefixes might communicate, leading to an interaction. The unobservable prefix $\tau.P$ expresses an internal action of the process, and the match prefix $[x = y]\pi.P$ behaves as $\pi.P$, if x is equal to y.

Throughout this paper, upper case letters are used for process identifiers and lower case letters for names. Furthermore defined processes from the original paper on the π-calculus are used for parametric recursion, that is $A(y_1, ..., y_n)$ [9]. The formal semantics of the π-calculus is based on transition systems. We only give short definitions of the required concepts and refer to [14,15] for details.

Definition 2 (Transition Sequence). *A sequence of interactions on names or unobservable actions is denoted as $P \xrightarrow{\alpha} P'$, where α describes the sequence of actions required to transform a process P to P'.* □

Definition 3 (Context). A context *is a process term with a hole, denoted as $[\cdot]$. The hole can be filled with a process other than* $\mathbf{0}$. □

We write $C[P]$ for a context C with $[\cdot]$ replaced by P. The replacement is literal, which means that names free in P may be bound in $C[P]$. For example, let $C = \mathbf{v}x(\overline{x}a.\mathbf{0} \mid [\cdot])$, then $C[x(y).\mathbf{0}] = \mathbf{v}x(\overline{x}a.\mathbf{0} \mid x(y).\mathbf{0})$.

Definition 4 (Observability Predicate). Observability predicate $\downarrow_\mu$ *on names or co-names* μ *is defined by:*

1. $P \downarrow_x$ *if* P *can perform an input action with subject* x *and*
2. $P \downarrow_{\overline{x}}$ *if* P *can perform an output action with subject* x. □

The observables of a processes are then the free (unrestricted) names it can use for receiving and sending. For example, $P \overset{def}{=} \mathbf{v}z(!\overline{x}z.\mathbf{0} \mid \mathbf{v}z(\overline{w}a.\mathbf{0} \mid w(v).\mathbf{0} + y(u).\mathbf{0})))$, contains $P \downarrow_y$ and $P \downarrow_{\overline{x}}$ as the observables of P.

Definition 5 (Weak Open Bisimulation Equivalence). *Informally, two* π*-calculus processes* P *and* Q *are* weak open bisimulation equivalent*, denoted as* $\approx^o$*, if they have the same observable behavior regarding the observability predicates* $\downarrow_{\tilde{s}}$. □

Thus, regarding weak open bisimulation, we abstract from all other internal actions. Formal details can be found in [15].

3.2 Business Process Patterns in the Pi-Calculus

Business processes in the π-calculus have been introduced in [8], by giving a collection of all workflow patterns [10] in their respective π-calculus formalization. An additional pattern common in interacting business processes, called *Event-based Rerouting*, has been presented in [16]. All pattern representations are based on events rather than states. A π-calculus process representing an activity waits for its required events (preconditions), does some internal action (functional part), and thereafter generates new events (postconditions). A business process formalized in the π-calculus consists of π-calculus processes representing different workflow patterns and a set of names, used for representing events.

During our investigations on lazy soundness, some pattern formalizations from [8] had to be refined since their original definitions are erroneous under certain circumstances.

Deferred Choice. As the π-calculus supports no transactional transitions, we need to make the choices in the preceding process to support loop behavior:

$$A = \tau_A.(b_{env}.\overline{b}.\mathbf{0} + c_{env}.\overline{c}.\mathbf{0}) \qquad B = b.\tau_B.B' \qquad C = c.\tau_C.C' \ .$$

MI without Synchronization. B has to continue immediately, however instances of B may still be active. This formalization gives a more applicable semantics to the pattern while still corresponding to [10]:

$$A = \tau_A.\overline{b}.\mathbf{0} \qquad B = b.((\prod_{i=1}^{n} \tau_B.\mathbf{0}) \mid B') \ .$$

Cancel Activity. Cancel activity has to accept a cancel event even after the functional part τ has been executed to provide correct routing:

$$\begin{aligned} A =& a.env_A(test_1).[test_1 = \bot].\tau_A.env_A(test_2).[test_2 = \bot].A' \\ \mathcal{E}_\mathcal{A} =& \overline{env_A}\langle\bot\rangle.\mathcal{E}_A + \overline{env_A}\langle\top\rangle.\mathcal{E}_\mathcal{A}' \ . \end{aligned}$$

4 Process Graphs

This section defines how business processes are formalized in terms of set theory and process algebra. It grounds structural correctness, that in turn is required for behavioral analysis discussed later on.

4.1 Structure

We start with the definitions of a *Process Graph*, a data structure that represents the behavioral aspects of a business process. Process graphs provide us with a uniform semi–formal representation of business processes regardless of their actual notations.

Definition 6 (Process Graph). *A process graph is a four-tuple consisting of nodes, directed edges, types and attributes. Formally:* $P = (N, E, T, A)$ *with*

- N *is a finite, non-empty set of nodes.*
- $E \subseteq (N \times N)$ *is a set of directed edges.*
- $T : N \rightarrow 2^{TYPE}$ *is a function mapping nodes to types.*
- $A : N \rightarrow KEY \times VALUE$ *is a function mapping key/value pairs to nodes.*□

The nodes N of a process graph define the activities of a process, and the directed edges E define dependencies between activities. Each node can have none, one, or more types assigned by the function T. Furthermore, each node can hold optional attributes represented by key/values pairs assigned by the function A. Sub-Processes are represented by a node N of the special type $Reference$, that references another process graph, i.e. $T(N) = \{Reference\}$. As such composed process graphs can always be flattened, we only consider flat process graphs. Some additional functions for accessing the sets of a process graph are given by:

- $source : E \rightarrow N$ returns the source node of a directed edge.
- $target : E \rightarrow N$ returns the target node of a directed edge.
- $type : N \rightarrow T$ returns the types of a node (same as $T(N)$).

To show the coherence between a process graph and a graphical notation, we give an example of how to map the structurally relevant parts of a business process to a process graph. We consider business processes given as a *Business Process Diagram* (BPD) of the *Business Process Modeling Notation* (BPMN) [17]. Other graph-based notations like EPCs or UML2 Activity Diagrams can be mapped in a similar manner.

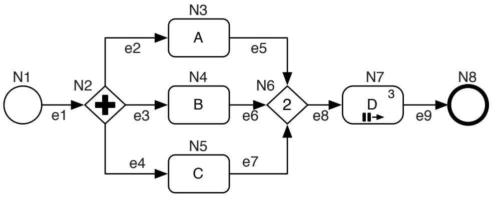

Fig. 2. A process containing a N-out-of-M-Join and a Multiple Instances without Synchronization pattern.

Example 1 (Partly Mapping of a BPD to a Process Graph). A BPD is exemplary mapped to a process graph $P = (N, E, T, A)$ by the following steps:

1. N is given by all flow object of the BPD.
2. E is given by all sequence flows of the BPD.
3. T is given by the corresponding types of the flow objects.
4. A is given by additional attributes of flow objects, e.g.:
   (a) The number of incoming sequence flows for an n-out-of-m-join node;
   (b) The number of instances to be created for an activity;
   (c) The nodes to be canceled for a cancel event. □

An actual example of a business process modeled in BPMN is given in Figure 2. The process contains a n-out-of-m-join pattern, modeled by a gateway with the number of required sequence flows inside, as well as a multiple instances without synchronization pattern, modeled by activity D. The activities A, B, and C can represent sub-processes for contacting three different experts for writing an expertise. After two of them are ready, the process continues. However, some cleanup work is left for the remaining activity, e.g. receiving the last expertise and paying the expert. Although this does not directly contribute to the process, it is still required. Activity D send the accepted expertises to three different involved persons. This is again a lazy activity, as the business process can actually finish, even while the documents are actually in delivery. The complete business process diagram is mapped to a process graph according to the mapping rules given in Example 1.

Example 2 (Expertise Process). The process graph $P = (N, E, T, A)$ of the example from Figure 2 is given by:

1. $N = \{N1, N2, N3, N4, N5, N6, N7, N8\}$
2. $E = \{ (N1, N2), (N2, N3), (N2, N4), (N2, N5), (N3, N6), (N4, N6), (N5, N6), (N6, N7), (N7, N8) \}$
3. $T = \{(N1, StartEvent), (N2, ANDGateway), (N3, Task), (N4, Task), (N5, Task), (N6, N\text{-}out\text{-}of\text{-}M\text{-}Join), (N7, MIwithoutSync), (N8, EndEvent)\}$
4. $A = \{(N6, (continue, 2)), (N7, (count, 3))\}$ □

4.2 Semantics

We now give formal semantics to a process graph by mapping it to π-calculus processes according to the following algorithm.

Algorithm 1 (Mapping Process Graphs to π-calculus Processes). A process graph $P = (P_N, P_E, P_T, P_A)$ is mapped to π-calculus processes as follows:

1. Assign all nodes of P an unique π-calculus process identifier $N1 \cdots N|P_N|$.
2. Assign all edges of P an unique π-calculus name $e1 \cdots e|P_E|$.
3. Define the π-calculus processes according to the behavioral patterns found in [8,16] as given by the type of the corresponding node. Take care of recursive definitions for supporting loop behavior, under the restrictions that:
   (a) All processes representing a node with no incoming edges do not support re-execution, and
   (b) All processes representing a node with no outgoing edges support re-execution by recursion.
4. Replace each functional part τ of the behavioral patterns mapped before with $[\cdot]$, thus constructing a context of each node.
5. Define a global process $N = (\mathbf{v}e1, \cdots, e|P_E|) \prod_{i=1}^{|P_N|} Ni$. This process can contain further components or restricted names according to the contained patterns. □

A node of a process graph is *executed* if the context of the corresponding π-calculus process is reached. We can now map the process graph from Example 2 to π-calculus processes.

Example 3 (π-calculus Process for Expertise Process).

$$
\begin{aligned}
Tasks:&\quad N3 = e2.[\cdot].(\overline{e5}.\mathbf{0} \mid N3)\ ,\quad N4 = e3.[\cdot].(\overline{e6}.\mathbf{0} \mid N4)\\
&\quad N5 = e4.[\cdot].(\overline{e7}.\mathbf{0} \mid N5)\\
ANDGateway:&\quad N2 = e1.[\cdot].(N2 \mid \overline{e2}.\mathbf{0} \mid \overline{e3}.\mathbf{0} \mid \overline{e4}.\mathbf{0})\\
N\text{-}out\text{-}of\text{-}M\text{-}Join:&\quad N6 = (\mathbf{v}h, run)(N6_1 \mid N6_2)\\
&\quad N6_1 = e5.\overline{h}.\mathbf{0} \mid e6.\overline{h}.\mathbf{0} \mid e7.\overline{h}.\mathbf{0}\\
&\quad N6_2 = h.h.\overline{run}.h.N6 \mid run.[\cdot].\overline{e8}.\mathbf{0}\\
MIwithoutSync:&\quad N7 = e8.([\cdot].\mathbf{0} \mid [\cdot].\mathbf{0} \mid [\cdot].\mathbf{0} \mid \overline{e9}.\mathbf{0} \mid N7)\\
StartEvent:&\quad N1 = [\cdot].\overline{e1}.\mathbf{0}\\
EndEvent:&\quad N8 = e9.[\cdot].N8\\
Global:&\quad N = (\mathbf{v}e1, \cdots, e9)\prod_{i=1}^{8} Ni
\end{aligned}
$$

A task waits for preconditions (the incoming edges), executes the functional perspective abstracted by a context, and generates postconditions (i.e. co-names). Although not required for the example, the processes use recursion to support loop behavior. Note that a BPMN AND Gateway combines two patterns, parallel split and synchronization, into one node. The process $N1$ representing a Start Event does not support re-execution by recursion. If a whole process should be executed another time, a new instance of it has to be created. □

5 Structural and Lazy Soundness

This section introduces correctness criteria for process graphs. We distinguish between structural and behavioral criteria. The former is denoted by *structural soundness*, whereas the latter is given by soundness, relaxed soundness, and lazy soundness. We focus on lazy soundness in this paper, although weak soundness can be defined and proved in a similar manner.

5.1 Structural Soundness

Structural soundness for process graphs is based on the concepts introduced in the following paragraphs.

Definition 7 (Path). *A path in a process graph $P = (N, E, T, A)$ is a sequence of directed edges leading from one node to another. Formally, a path ϵ from n_1 to n_2 is written as: $n_1 \xrightarrow{\epsilon} n_2$ with $n_1, n_2 \in N$ and $\epsilon \in E^*$, where we allow an empty sequence. An arbitrary path from n_1 to n_2 is denoted as $n_1 \xrightarrow{*} n_2$.* □

Definition 8 (Reachability). *A node of a process graph $P = (N, E, T, A)$ is reachable from another node if and only if there exist a path leading from the first to the second node. Formally: $n_2 \in N$ is reachable from $n_1 \in N$, iff $\exists \epsilon \in E^* : n_1 \xrightarrow{\epsilon} n_2$.* □

Definition 9 (Defined Process Graph). *A process graph $P = (N, E, T, A)$ is defined if and only if there is exactly one node of the type* Initial Node*, denoted as N_i, that is not the target of any edge and exactly one node of the type* Final Node*, denoted as N_o, that is not the source of any edge. Formally: $\exists n \in N : InitialNode \in type(n) \land \forall n_1, n_2 \in N : InitialNode \in type(n_1) \land InitialNode \in type(n_2) \Rightarrow n_1 = n_2$ and $\exists n \in N : FinalNode \in type(n) \land \forall n_1, n_2 \in N : FinalNode \in type(n_1) \land InitialNode \in type(n_2) \Rightarrow n_1 = n_2$. Furthermore: $\forall n \in N : InitialNode \in type(n) \Rightarrow \nexists e \in E : target(e) = n$ and $\forall n \in N : FinalNode \in type(n) \Rightarrow \nexists e \in E : source(e) = n$.* □

Definition 10 (Strongly Connected Process Graph). *A defined process graph $P = (N, E, T, A)$ is strongly connected, if and only if for all nodes exists a path from the initial to the final node. Formally: $\forall n \in N$ with $N_i \xrightarrow{*} n \Rightarrow n \xrightarrow{*} N_o$* □

This definition is in contrast to common definitions of a strongly connected directed graph, e.g. by Knuth [18]. We do not require a graph to be short circuited for analysis.

Lemma 1. *$P_{MIN}(N, E, T, A) = (\{N1\}, \emptyset, \{(N1, InitialNode), (N1, FinalNode)\}, \emptyset)$ is the smallest strongly connected process graph.*

Proof (Lemma 1). Direct proof. $P_{MIN}(N, E, T, A)$ is strongly connected as it is defined by exactly one initial and final node, and the only node lies on an (empty) path from the initial to the final node. Formally: $\exists n_1 \in N : InitialNode \in type(n_1) \wedge \exists n_2 \in N : FinalNode \in type(n_2)$. $\forall n_1, n_2 \in N : n_1 \overset{\emptyset}{\rightarrow} n_2$. All components of P_{MIN} have the lowest possible count of elements for a strongly connected process graph. Formally: $|P_{MIN}(N, E, T, A)| = (1, 0, 2, 0)$ following from Definition 6 and 9. □

Definition 11 (Structural Sound). *A process graph $P = (N, E, T, A)$ is structural sound if and only if:*

1. *There is exactly one initial node $N_i \in N$.*
2. *There is exactly one final node $N_o \in N$.*
3. *Every node is on a path from N_i to N_o.* □

Structural soundness for process graphs adapts the definition of a workflow net as a special kind of Petri net introduced in [6].

Lemma 2. *A strongly connected process graph is structural sound.*

Proof (Lemma2). Direct proof. Criterion 1 and 2 from Definition 11 are fulfilled, as a strongly connected process graph is defined. Criterion 3 follows directly from Definition 10. □

Lemma 3. *$P_{MIN}(N, E, T, A)$ is structural sound.*

Proof (Lemma 3). Follows directly from Lemma 1. □

Algorithm 2 (Deciding Structural Soundness). We describe an algorithm for deciding structural soundness of a process graph $P(N, E, T, A)$:

1. Check if P is defined, i.e. has exactly one initial and exactly one final node (see Definition 9).
2. Check if P is strongly connected, i.e. if every node is on a path from the initial to the final node (see Definition 10). □

5.2 Lazy Soundness

Lazy soundness extends structural soundness by taking the semantics of the process nodes into account. Therefore it considers the π-calculus representation of a process graph, which includes semantics for the types of the process nodes. Lazy soundness states that there are no livelocks or deadlocks in the process graph regarding the semantics of the nodes. Furthermore, the final node will be executed exactly once, while other nodes representing activities can still be or become executed. However, they must not trigger the final node again. To define lazy soundness, we need the definition of *semantic reachability*, i.e. if a node lies on a path from the initial to the final node according to the semantics of all nodes.

Definition 12 (Semantic Reachability). *A node of a process graph* $P = (N, E, T, A)$ *is semantically reachable from another node if and only if there exists a path leading from the first to the second node according to the semantics of all nodes.* □

Regarding the mapping of a π-calculus process from a process graph, a π-calculus process representing a node is semantically reachable from another π-calculus process representing a node, if and only if there exists a transition sequence from the functional abstraction τ of the first process to the functional abstraction τ of the second process. Lazy soundness is then defined as follows.

Definition 13 (Lazy Sound). *A structural sound process graph* $P = (N, E, T, A)$ *is lazy sound if it represents a business process that is deadlock free and livelock free, as long as the final node has not been reached. Once the final node has been reached, other nodes might still be executed, however the final node is not enacted again. Formally:*

1. *The final node* N_o *must be semantically reachable from every node* $n \in N$ *semantically reachable from the initial node* N_i *until* N_o *has been reached for the first time.*
2. *The final node* N_o *is reached exactly once.* □

To be able to trace the transition sequences required for semantics reachability, we annotate the π-calculus mapping of a process graph with two observability predicates $\downarrow_i$, and $\downarrow_{\bar{o}}$. Using these predicates, we can observe the execution of the initial activity by $\downarrow_i$, and the final activity by $\downarrow_{\bar{o}}$.

Algorithm 3 (Lazy Soundness Annotated π-calculus Process). To annotate a π-calculus process representing a process graph for reasoning on lazy soundness, we need to fill the holes, i.e. $[\cdot]$,of the process definitions with:

- τ, if the the corresponding process graph node has incoming and outgoing edges,
- $i.\tau$, if the corresponding process graph node has only outgoing edges,
- $\tau.\bar{o}$, if the corresponding process graph node has only incoming edges, and
- $i.\tau.\bar{o}$ if the corresponding process graph node has no incoming or outgoing edges. □

An example can be found in Example 4. Due to the fact of being able to observe the initial and the final activity, we can prove lazy soundness for process graphs. Thus, for every activity reachable after the initial activity has been observed, we must always be able to observe the final activity exactly once if the process graph if lazy sound. If we observe the final activity more than once or never at all, the process graph contains a deadlock or livelock. We derive this theorem by constructing the smallest lazy soundness annotated π-calculus mapping of a process graph and prove it to be lazy sound.

Lemma 4. $S_{LAZY} = i.\tau.\bar{o}.\mathbf{0}$ *with the observability predicates* $\downarrow_i$ *and* $\downarrow_{\bar{o}}$ *is the smallest lazy soundness annotated* π*-calculus mapping of a process graph satisfying lazy soundness.*

Proof (Lemma 4). The proof consists of two parts. We first show that S_{LAZY} is the smallest lazy soundness annotated π-calculus of P_{MIN}. Secondly, we prove that S_{LAZY} is lazy sound by constructing all transitions.

1. Direct proof. S_{LAZY} is the smallest lazy soundness annotated π-calculus mapping of P_{MIN}. It has exactly one node denoted by τ and no pre- or postconditions. The initial node is exactly the final node, denoted by i before and $\overline{o}$ after τ.
2. Direct proof. Lazy soundness for S_{LAZY} is proved by constructing all transitions: $i.\tau.\overline{o}.\mathbf{0} \xrightarrow{i} \tau.\overline{o}.\mathbf{0} \xrightarrow{\tau} \overline{o}.\mathbf{0} \xrightarrow{\overline{o}} \mathbf{0}$. The transition trace proves that the initial node is always executed once (observability predicate $\downarrow_i$), all possible transitions are executed thereafter (one τ-transition), and eventually the final node is executed (observability predicate $\downarrow_{\overline{o}}$) before S_{LAZY} reaches inaction. □

Now we are ready to introduce the theorem for proving lazy soundness on structural sound process graphs mapped to a lazy sound π-calculus representation.

Theorem 1. *Each structural sound process graph P more complex than P_{MIN} is mapped to a lazy soundness annotated π-calculus process D, so that $D \approx^o_{i,\overline{o}} S_{LAZY}$ if and only if P is lazy sound.* □

Proof (Theorem 1). Direct proof. Each structural sound process graph more complex than P_{MIN} is mapped to a lazy soundness annotated π-calculus process D with $\downarrow_i$ as the observability predicate of the initial node and $\downarrow_{\overline{o}}$ as the observability predicate of the final node. The observability predicates are thus the invariants of the π-calculus processes. If a lazy soundness annotated π-calculus process $D \sim^o_{i,o} S_{LAZY}$, the corresponding process graph P of D must then be lazy sound. □

Algorithm 4 (Deciding Lazy Soundness). We describe an algorithm for deciding lazy soundness of a structural sound process graph mapped to π-calculus processes.

1. Map the structural sound process graph to π-calculus processes, following Algorithm 1.
2. Annotate the π-calculus processes for lazy soundness, following Algorithm 3.
3. Check the annotated definition for weak open bisimulation equivalence with S_{LAZY} concerning $\downarrow_i$ and $\downarrow_{\overline{o}}$. □

This algorithm has already been implemented and will be discussed in the next section.

6 Tool Support and Discussion

This section evaluates how the theoretical results achieved can be applied and verified using existing tools such as Mobility Workbench (MWB), Advanced Bisimulation Checker (ABC), or Open Bisimulation Checker (OBC) for deciding weak open bisimulation equivalence on π-calculus processes [19,20,21].

6.1 Tool Integration

To be able to integrate these tools into our theoretical framework, we have created a tool chain consisting of several scripts. The first script is written in AppleScript and exports a graphical BPMN business process diagram from OmniGraffle[2] to a process graph. We had to use a slightly modified BPMN notation to support all workflow pattern, as can been see in Figure 2 were we introduced an n-out-of-m-gateway. We created Ruby scripts for deciding structural soundness of process graphs, as well as mapping process graphs to lazy and weak soundness annotated π-calculus processes. The generated π-calculus processes are then used as input to the tools MWB and ABC for deciding lazy or weak soundness. We illustrate lazy soundness by example in the corresponding input style for MWB or ABC:

Example 4 (Lazy Soundness annotated π-calculus process of Example 3 for Tool Analysis).

```
agent N8(e9,o)=e9.t.'o.N8(e9,o)
agent N7(e8,e9)=e8.(t.0 | t.0 | t.0 | 'e9.0 | N7(e8,e9))
agent N6(e5,e6,e7,e8)=(^h,run)(N6_1(e5,e6,e7,e8,h,run) | N6_2(e5,e6,e7,e8,h,run))
agent N6_1(e5,e6,e7,e8,h,run)=e5.'h.0 | e6.'h.0 | e7.'h.0
agent N6_2(e5,e6,e7,e8,h,run)=h.h.'run.h.N6(e5,e6,e7,e8) | run.t.'e8.0
agent N5(e4,e7)=e4.t.('e7.0 | N5(e4,e7))
agent N4(e3,e6)=e3.t.('e6.0 | N4(e3,e6))
agent N3(e2,e5)=e2.t.('e5.0 | N3(e2,e5))
agent N2(e1,e2,e3,e4)=e1.t.(N2(e1,e2,e3,e4) | 'e2.0 | 'e3.0 | 'e4.0)
agent N1(e1,i)=i.t.'e1.0
agent N(i,o)=(^e1,e2,e3,e4,e5,e6,e7,e8,e9)(N8(e9,o) | N7(e8,e9) | N6(e5,e6,e7,e8) |
      N5(e4,e7) | N4(e3,e6) | N3(e2,e5) | N2(e1,e2,e3,e4) | N1(e1,i))
agent S_LAZY(i,o)=i.t.'o.0
```

We can ask ABC for deciding weak open bisimulation equivalence on N and S_{LAZY}, thus deciding lazy soundness for the process graph from Example 2:

```
abc > weqd (i,o) N(i,o) S_LAZY(i,o)
The two agents are weakly related (315).
Do you want to see the core of the bisimulation (yes/no) ? no
```

Since $N(i, o)$ is weak open bisimulation equivalent to S_{LAZY}, the corresponding process graph is lazy sound. By simply modifying the AND Gateway of the example given in Figure 2 to an XOR Gateway in the corresponding lazy soundness annotated π-calculus process, we can prove the corresponding process graph to be not lazy sound:

```
abc > agent N2(e1,e2,e3,e4)=e1.t.(N2(e1,e2,e3,e4) | ('e2.0 + 'e3.0 + 'e4.0))
Agent N2 is defined.
abc > weqd (i,o) N(i,o) S_LAZY(i,o)
The two agents are not weakly related (9).
Do you want to see some traces (yes/no) ? no
```

Obviously, the modified process graph is not lazy sound as it contains a deadlock.

6.2 Supported Patterns and Performance

Tool support for reasoning on lazy soundness is still limited by the supported patterns as well as performance. Multi merge and simple merge patterns behave

[2] http://www.omnigroup.com/applications/omnigraffle/

	Fig. 2	**Fig. 2 mod.**	**Fig. A6 [6]**	**Fig. 2 [16]**	**Bookstore [24]**
Nodes	8 nodes	8 nodes	10 nodes	15 nodes	21 nodes
MWB	$10s$	$< 1s$	$< 1s$	$15s$	$6s$
ABC	$40s$	$2s$	$6s$	$275s$	$167s$
ABC.opt	$13s$	$< 1s$	$2s$	$55s$	$50s$
Lazy Sound?	Yes	No	Yes	Yes	Yes

Table 1. Performance results for deciding lazy soundness.

the same (indeed, same π-calculus representation). Since the π-calculus has a blocking semantics, parallel activation will be queued until the merge activity is ready again. The synchronizing merge pattern in the π-calculus has non local semantics and is thus only supported by workarounds ranging from introducing a local semantics (true/false token passing, corresponding split/choice and synchronizing merge patterns, where the split/choice informs the corresponding merge about the number of incoming arcs) to global analysis (e.g. delay synchronizing merge while other transitions are possible). Further discussions regarding the synchronizing merge pattern can be found in [22,23]. Arbitrary cycles are only partly supported in MWB as well as ABC. These tools fail at deciding processes with loops generating an infinite number of $\downarrow_{\overline{o}}$. This is indeed a tool related issue, as the reasoning could be stopped immediately after more than one $\overline{o}$ has been observed, instead of creating the full space state. A related issue concerns multiple instances with a dynamic number of instances, either runtime or without a priori knowledge. MWB as well as ABC fail for unknown reasons at detecting the contained cycles, while they work at simple loops. However, both issues are tool related and do not disturb the theory. For all patterns containing cancellation, i.e. cancel activity, cancel case, event-based rerouting, we can not actually stop the unobservable action τ, only immediately reroute the control flow and cancel all related outgoing flows.

Regarding the performance of deciding weak open bisimulation, we are currently investigating existing tools. First practical results for business processes containing different patterns have been collected in table 1.[3] Some processes have been converted to BPMN and can be found in the cited references. Figure 2 and the modified version have been discussed in this paper. Figure A6 from [6] contains arbitrary cycles. Figure 2 from [16] contains event-based rerouting and deferred choice patterns. The bookstore process from [24] contains multiple deferred choices and arbitrary cycles.

7 Conclusion

In this paper, we introduced and discussed a new correctness criterion for business processes, called lazy soundness. Lazy soundness proves a business process

[3] Rough estimations measured on an Apple PowerBook G4 1.5GHz with 1.25GB RAM.

to be deadlock and livelock free, but does not cover dead activities, or requires all activities to be finished when a final activity is reached. It can be classified below weak soundness and soundness, i.e. all sound and all weak sound business processes are lazy sound, and beside relaxed soundness, i.e. a relaxed sound business process can be lazy sound. A stronger kind of lazy soundness is weak soundness, forcing all activities to finish before the final activity is reached.

Lazy soundness is an important correctness criterion for business processes, as it supports reasoning on deadlock and livelock freedom without being to restrictive regarding so called *clean-up* or *lazy-activities* that can be left behind. Our reasoning framework presented supports the original semantics of the workflow patterns discriminator, n-out-of-m-join, and multiple instances without synchronization. It achieves this by utilizing the π-calculus as formal foundation. All existing workflow patterns [8] as well as new routing patterns [16] can be represented in this calculus. It has strong theoretical reasoning capabilities based on different kinds of bisimulation [14,15], that can be used to prove lazy soundness. We already achieved first feasibility results using a tool chain for converting BPMN business process diagrams to π-calculus processes that can be analyzed using existing π-calculus tools. Obviously, the underlying concepts of lazy soundness as discussed in section 2 can also be adapted to other formalizations like workflow nets.

Further work will focus on complete support for soundness and relaxed soundness, as well as reasoning on interacting business processes. Therefore, a special capability of the π-calculus, namely *channel-passing*, will be of special interest as it allows support for dynamic routing patterns [7]. Alongside, we will improve tool development focusing on weak open bisimulation for π-calculus processes representing workflow patterns.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H., Weske, M.: Business Process Management: A Survey. In van der Aalst, W.M.P., ter Hofstede, A.H., Weske, M., eds.: Proceedings of the 1st International Conference on Business Process Management, volume 2678 of LNCS, Berlin, Springer-Verlag (2003) 1–12
2. Burbeck, S.: The Tao of e-business services. Available at: `http://www-128.ibm.com/developerworks/library/ws-tao/` (2000)
3. van der Aalst, W.M.P.: Verification of Workflow Nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets, volume 1248 of LNCS, Berlin, Springer-Verlag (1997) 407–426
4. Dehnert, J., Rittgen, P.: Relaxed Soundness of Business Processes. In Dittrich, K., Geppert, A., Norrie, M.C., eds.: CAiSE 2001, volume 2068 of LNCS, Berlin, Springer-Verlag (2001) 157–170
5. Martens, A.: On Compatibility of Web Services. Petri Net Newsletter **65** (2003) 12–20
6. van der Aalst, W., van Hee, K.: Workflow Management. MIT Press (2002)
7. Barros, A., Dumas, M., ter Hofstede, A.: Service Interaction Patterns. In van der Aalst, W., Benatallah, B., Casati, F., eds.: Proceedings of the 3rd International Conference on Business Process Management, volume 3649 of LNCS, Berlin, Springer-Verlag (2005) 302–318

8. Puhlmann, F., Weske, M.: Using the Pi-Calculus for Formalizing Workflow Patterns. In van der Aalst, W., Benatallah, B., Casati, F., eds.: Proceedings of the 3rd International Conference on Business Process Management, volume 3649 of LNCS, Berlin, Springer-Verlag (2005) 153–168
9. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Part I/II. Information and Computation **100** (1992) 1–77
10. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.: Workflow Patterns. Technical Report BETA Working Paper Series, WP 47, Eindhoven University of Technology (2000)
11. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language (Revised version. Technical Report FIT-TR-2003-04, Queensland University of Technology, Brisbane (2003)
12. Verbeek, H., van der Aalst, W., ter Hofstede, A.: Verifying Workflows with Cancellation Regions and OR-joins: An Approach based on Invariants, BETA Working Paper Series, WP 156. Technical report, Eindhoven University of Technology, Eindhoven, The Netherlands (2006)
13. Puhlmann, F.: Why do we actually need the Pi-Calculus for Business Process Management? In: Proceedings of the 9th International Conference on Business Information Systems. (2006) (to appear)
14. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Paperback edn. Cambridge University Press, Cambridge (2003)
15. Sangiorgi, D.: A Theory of Bisimulation for the Pi-Calculus. In: CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory, Berlin, Springer-Verlag (1993) 127–142
16. Overdick, H., Puhlmann, F., Weske, M.: Towards a Formal Model for Agile Service Discovery and Integration. In Verma, K., Sheth, A., Zaremba, M., Bussler, C., eds.: Proceedings of the International Workshop on Dynamic Web Processes (DWP 2005). IBM technical report RC23822, Amsterdam (2005)
17. BPMI.org: Business Process Modeling Notation. 1.0 edn. (2004)
18. Knuth, D.E.: The Art of Computer Programming. 3rd edn. Volume 1. Addison–Wesley (1997)
19. Björn Victor, Faron Moller, M.D., Eriksson, L.H.: The Mobility Workbench. Available at: `http://www.it.uu.se/research/group/mobility/mwb` (2005)
20. Briais, S.: ABC Bisimulation Checker. Available at: `http://lamp.epfl.ch/~sbriais/abc/abc.html` (2003)
21. Frendrup, U., Jensen, J.N., Hüttel, H.: OBC Workbench. Available at: `http://www.cs.auc.dk/research/FS/ny/PR-pi/` (2001)
22. Wynn, M., Edmond, D., van der Aalst, W., ter Hofstede, A.: Achieving a General, Formal and Decidable Approach to the OR-join in Workflow using Reset nets (2005)
23. Kindler, E.: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In Desel, J., Pernici, B., Weske, M., eds.: Proceedings of the 2nd International Conference on Business Process Management, volume 3080 of LNCS, Berlin, Springer-Verlag (2004) 82–97
24. van der Aalst, W.M.P., Weske, M.: The P2P Approach to Interorganizational Workflow. In Dittrich, K., Geppert, A., Norrie, M., eds.: Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01), volume 2068 of LNCS, Berlin, Springer-Verlag (2001) 140–156