FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

JENAER SCHRIFTEN

ZUR

MATHEMATIK UND INFORMATIK

Eingang: 13.06.2013 Math/Inf/02/2013 Als Manuskript gedruckt

Practical Compiler-based User Support during the Development of Business Processes

Thomas M. Prinz and Wolfram Amme

Friedrich Schiller University Jena, 07743 Jena, Germany
{Thomas.Prinz,Wolfram.Amme}@uni-jena.de

Abstract. An erroneous execution of a business process causes high costs and could damage the prestige of the providing company. Therefore, validation of the correctness of business processes is essential. In general, semantics of business processes are described with Petri net semantics, even though this kind of description is in classical meaning not always exact, because of the missing possibility of modelling data-dependencies in Petri nets.
In this paper, we describe new compiler-based techniques that could be used instead of Petri net semantics for the verification of business processes. Basic idea of our approach is, to start analyses on different points of workflow graphs and to find potential structural errors. These developed techniques improved other known approaches, as it guarantees a precise visualization and explanation of all determined structural errors, which substantially supports the development of business processes.

Keywords: business processes, compiler-based techniques, structural correctness, workflow graphs, soundness

1 Introduction

Business processes are well-established in business management and in the context of service-oriented architectures and cloud computing. Since business processes are described by graphical specification languages like *BPMN 2.0*, there is a need for transformations into more technical representations to allow and outperform analyses and to automatically execute business processes. Because of the similarity to programming languages and its transformations into machine-dependent code, our overall goal is to create a compiler for business processes, which seriously supports the development and produces a common intermediate representation.

Since business processes are frequently used by end users and could have runtimes over months, an erroneous execution of business processes causes high costs and could damage the reputation of the providing company. Therefore, support to develop correct business processes is essential for all business process development tools. Acceptance of such a tool would increase if it fulfills the requirements *completeness*, *fastness*, and *usability*. Thereby, completeness describes the coverage of specification languages by the support, fastness means

the duration to get the support, and finally, usability stands for the quality of support information. More specifically, a high quality tool must cover the standard constructs of the specifiation language, e.g., loops and inclusive gateways, must provide immediate support, must determine at least all statical errors, and should be able to visualize them.

In general, the correctness of business processes can be devided in two parts: the *structural correctness*, which focuses only on the structure of business processes without consideration of data-dependencies, and the *total correctness*, which includes them. Since the total correctness depends on structural correctness and its analyses become easier on structural correct business processes, there must be an assistance to support the development of structural correct business processes.

Business processes can have two kinds of structural errors: *deadlocks* and *lack of synchronization* [1], whereas deadlocks are situations in which the execution within business processes blocks partly, and lack of synchronization are situations in which parts of business processes are executed twice unintentionally. The absence of deadlocks and lack of synchronization in business processes is called *soundness* in the literature [2,3], whereas we prefer to call it structural correctness.

Current soundness checker tools are based on Petri nets, or on workflow graphs, which are similar to control flow graphs using explicit parallelism. Petri net-based techniques are in classical meaning not always exact due to their inability to model data-dependencies in Petri nets. Thus, they are rather inusable within development tools for business processes. Furthermore, most of these Petri net-based techniques [4,5] uses state space exploration to determine structural errors. This allows the determination of exact one runtime error, which even could be unsolvable, since it could be caused by a previous error. Such an information is useless in development tools.

The best known technique is the SESE decomposition [6] which works on workflow graphs and decomposes the graph into subgraphs called *fragments*. For each fragment only a single error may be detected, and this error can be visualized by highlighting the corresponding fragment. In other words, the SESE decomposition cannot find all structural errors in a fragment. Furthermore, there are some complex fragments, which cannot be addressed by this approach.

Overall, there is no development support tool for business processes fulfilling all requirements. In this paper, we describe new compiler-based techniques which work directly on workflow graphs and statically determine deadlocks and lack of synchronization, i.e., independent of previously executed workflow graph parts. Compared to other techniques, it guarantees a precise visualization and explanation of all structural errors, which considerabily assists the development of business processes and fulfilles most of the requirements.

This paper is structured as follows. In Section 2, we refresh the definitions of workflow graphs and structural correctness, followed by an informal description of our approach (Section 3). Section 4 describes the properties of structural errors, whereas Section 5 applying them for determination. The approach will be evaluated in Section 6 and compared to other techniques in Section 7. Eventually, Section 8 concludes the paper.

2 Preliminaries

Formally, a *workflow graph* is a directed graph $WFG = (N, E)$ such that N consists of activities $N_{activities}$, forks N_{forks}, joins N_{joins}, splits N_{splits}, merges N_{merges}, one *start* and *end* node. The end node, each activity, split and fork have exactly one incoming edge; whereas the start node, each activity, merge and join have exactly one outgoing edge. Splits and forks have at least two outgoing edges, and merges and joins have at least two incoming edges. Furthermore, each node lies on a path from the start to the end node.

A workflow graph is called *simple* if for each edge $e = (n_1, n_2) \in E$ either n_1 or n_2 is an activity. If a workflow graph contains also inclusive splits (IOR splits $N_{IORsplits}$) and inclusive joins (IOR joins $N_{IORjoins}$) with structural properties like forks and joins, then the workflow graph is called *complex*.

Figure 1 shows a workflow graph. The start and end nodes are depicted as (thick) circles, whereas an activity is depicted as a rectangle, a fork and a join as thin rectangle, and a split and a merge as (thick) diamond.

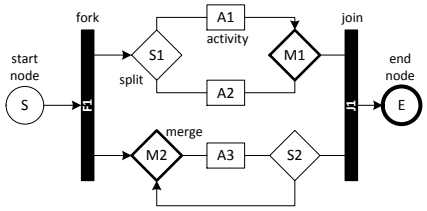

Fig. 1. A workflow graph

The semantics of workflow graphs used in this paper are similar to the semantics of control flow graphs: the execution of a workflow graph starts at the start node and follows the flow described by the directed graph. An activity, a split, a merge, a fork, and the end node can be executed if a control flow reaches an incoming edge of these nodes, whereas a join can only fire if all incoming edges reach a control flow. After executing a split, the split decides *nondeterministically* which outgoing edge will be followed by the control flow in workflow graphs without data-dependencies. It is assumed, that such decisions will be fair, i.e., each outgoing edge could be followed at random. After the execution of a fork, parallel control flows will be built for each outgoing edge, that should be combined with a join.

Without loss of generality, we assume each workflow graph is *simple* for the remainder of this paper, since there is a fast transformation from common to simple workflow graphs, e.g., by placing a new activity on each edge. This allows a description of the incoming and outgoing edges of a node with the direct predecessor and direct successor nodes. We write $\bullet n$ to describe the set of direct predecessor nodes of n, i.e., $\forall n_p \in \bullet n : (n_p, n) \in E$. Furthermore, we write $n\bullet$ to describe the set of direct successor nodes of n, i.e., $\forall n_s \in n\bullet : (n, n_s) \in E$.

Paths will be used to describe control flows within workflow graphs. Formally, a *path* $P = (n_1, n_2, \ldots, n_{m-1}, n_m)$ is a sequence of nodes of N such that $\forall i \in \{1, \ldots, m-1\} : (n_i, n_{i+1}) \in E$. We write $n \in P$ if $n \in \{n_1, \ldots, n_m\}$. A path is called *direct* if $n_2, \ldots, n_{m-1} \notin \{n_1, n_m\}$, and *simple* if all nodes on the path are pairwise different.

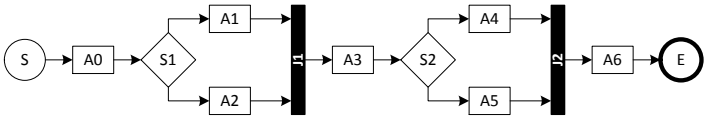

Fig. 2. An unreachable deadlock in join $J2$

The structural correctness will be defined by the absence of deadlocks and lack of synchronization. Thereby, a *deadlock* in a join can be reached if it was not executed as often as each of its direct predecessor nodes and cannot fire in future. Furthermore, a reachable fork causes a *lack of synchronization* if executing it may cause a node to be executed twice in series.

Definition 1 (Structural correctness). *A workflow graph is structurally correct if it has neither deadlocks nor lack of synchronization.*

3 Informal Description

The basic idea of our approach is to start the analysis for structural correctness on different points (nodes) of the workflow graph, called entrypoints. It is comparable to a compiler which tries to find a next safe program point to find further errors after a compile time error was found. For example, Figure 2 shows a workflow graph containing two deadlocks. Starting an analysis in the start node shows only a deadlock at the join $J1$, whereas restarting the analysis at the split $S2$ detects another deadlock in join $J2$.

Each node of a workflow graph can be an entrypoint. In order to avoid wrong analysis results, the entrypoints have to be chosen carefully. For example, the activity $A4$ of Fig. 2 is not a good entrypoint to show a possible deadlock in join $J2$, because it has no path to all direct predecessor nodes of this join. To find suitable entrypoints, they will be chosen with regard to another node, e.g., a join.

Definition 2 (Entrypoint). *A node n_1 is an* entrypoint *of a node n_2 if after an execution of n_1 the execution of n_2* could *follow.*

An entrypoint n_1 of a node n_2 is called safe *if after each execution of n_1 the execution of n_2 follows. Furthermore, an entrypoint n_1 of a node n_2 is called* closest *if on at least one path from n_1 to n_2 lies no other entrypoint of n_2.*

For example, the entrypoints of activity $A1$ are the nodes S, $A0$ and $S1$ in Fig. 2. $A1$ has $S1$ as closest but not safe entrypoint, since not each execution of $S1$ causes $A1$ to be executed. A safe and closest entrypoint of the split $S1$ is the activity $A0$. The joins $J1$ and $J2$ have no entrypoints, since no node within the workflow graph could cause the joins to be executed.

4 Properties of Structural Errors

In this section, we show some properties of structural errors.

Safe entrypoints of joins are excellent entrypoint for the determination of deadlocks, referred to as *activation points*.

Definition 3 (Activation point). *A node n_1 is an (closest) activation point of a node n_2 if n_1 is a (closest) safe entrypoint of n_2.*

With regard to activation points and to joins, the following lemma combines some properties of a join.

Lemma 1 (Properties of a join). *Let WFG be a workflow graph, and $join \in N_{joins}$. Then, the following holds:*

1. *for each activation point pnt of join, $\forall pre \in \bullet join$: pnt is an activation point of pre.*
2. *for each closest activation point pnt of join: $pnt \in N_{forks}$.*

Proof (Lemma 1).

1. This point is self-explanatory.
2. Let *pnt* be a closest activation point of a *join*.
   pnt will be executed
   $\overset{\text{Definition 3}}{\Rightarrow}$ *join* will be executed
   $\overset{\text{Lemma 1.1}}{\Rightarrow}$ all $pre \in \bullet join$ were executed
   $\Rightarrow \exists$ a path from *pnt* to all $pre \in \bullet join$
   Assumption: *pnt* is no split or fork
   $\Rightarrow$ *pnt* has exact one outgoing edge, because it cannot be the end node
   $\Rightarrow$ $suc \in pnt\bullet$ has a path to all $pre \in \bullet join$
   $\Rightarrow$ *suc* is an activation point of *join*
   $\overset{\text{Definition 3}}{\Rightarrow}$ *pnt* cannot be closest $\lightning$
   $\Rightarrow$ *pnt* has to be a split or fork
   Assumption: *pnt* is a split
   $\overset{\text{Definition 3 and semantics of split}}{\Rightarrow}$ $\forall suc \in pnt\bullet$: if *suc* will be executed, then *join* will be executed
   $\Rightarrow$ all $suc \in pnt\bullet$ are activation points of *join*
   $\Rightarrow$ *pnt* cannot be a closest activation point of *join* $\lightning$
   $\Rightarrow$ *pnt* has to be a fork

□

Summarized, all closest activation points of a join should be forks and are activation points of all direct predecessor nodes of that join. Knowing these properties, the following theorem could be used for the determination of deadlocks within workflow graphs without lack of synchronization.

Theorem 1 (Deadlock). *Let WFG be a workflow graph, which is free of lack of synchronization.*

$$A\ join \in N_{joins}\ has\ a\ deadlock$$
$$\Rightarrow$$
on at least one path from the start node to join or from join to itself lies no activation point of join.

Proof (Theorem 1). Proof by contradiction. The assumption has to hold:

$\Rightarrow$ **(1) A** $join \in N_{join}$ **has a deadlock** $\wedge$ **(2) on each path from the start node to** $join$**, or from** $join$ **to itself lies an activation point of** $join$**.**
$join$ has a deadlock $\iff$ $\exists pre \in \bullet join$: pre was executed more than $join$ and $join$ cannot be executed in future
$\Rightarrow \exists pre \in \bullet join$: pre was executed
$\Rightarrow \exists$ a *path* from the start node to pre which was executed
$\overset{(2)}{\Rightarrow}$ an activation point pnt was executed
$\overset{(1),\ \text{Definition 3}}{\Rightarrow}$ $join$ was executed, but there is still a deadlock
$\Rightarrow \exists pre \in \bullet join$: pre was more executed than $join$
$\Rightarrow$ there can be a situation before firing $join$, where pre was fired twice. A lack of synchronization arised$\lightning$ □

In other words, before any control flow ever reaches a join within a workflow graph free of lack of synchronization, an activation point of this join must be executed to prevent a deadlock. The basic idea of the proof is to show that after each execution of an activation point, the join will be executed and a remaining deadlock is only caused by lack of synchronization.

The inverse direction holds if a control flow executes such a path without an activation point additionally.

Theorem 2 (Potential deadlock). *Let WFG be a workflow graph.*

On at least one path from the start node to a $join \in N_{joins}$ *or from join to itself lies no activation point of join, and path can be executed*
$$\Rightarrow$$
join has a deadlock.

Proof (Theorem 2). Proof by contradiction. The assumption has to hold:

$\Rightarrow$ **(1.1) On at least one** *path* **from the start node to a** $join \in N_{joins}$ **or from** $join$ **to itself lies no activation point of** $join$**, and (1.2)** *path* **was executed** $\wedge$ **(2)** $join$ **has no deadlock**
$\overset{(1.2)}{\Rightarrow}$ $\exists pre \in \bullet join$: pre was executed.
$\overset{(2)}{\Rightarrow}$ $join$ will be executed in future.
$\Rightarrow$ all pre were be executed before $join$ was executed.
$\overset{(1.1)}{\Rightarrow}$ $\exists$ a *node* on *path*, in worst case the start node : execution of *node* $\Rightarrow$ execution of all $pre \in \bullet join$.
$\overset{\text{Definition 3}}{\Rightarrow}$ *node* is an activation point of $join\lightning$ □

The entrypoints for the determination of lack of synchronization are forks, since only forks build more than one control flow that can cause an execution of a node twice in series. Indeed, control flows will be described by paths within workflow graphs. In general, these paths can be combined by common nodes, referred to as *intersection points*.

Definition 4 (Intersection point). *Let* $fork \in N_{forks}$ *and* $suc_1, suc_2 \in fork\bullet$, $suc_1 \neq suc_2$.

An intersection point *of* suc_1 *and* suc_2 *is a node* $\cap$*-point with a direct path from* suc_1 *and* suc_2 *to* $\cap$*-point without node* $fork$*. It is called* closest *if it is the first common node of such two direct paths. We write* $\bar{\iota}(suc_1, suc_2)$ *for all closest intersection points of* suc_1 *and* suc_2.

Intersection points can be used to determine lack of synchronization, since they represent combination points of control flows. Furthermore, all control flows from the same fork have to be combined in joins before the fork can be executed again or the end node is reached. This fact will be used in the following theorem.

Theorem 3 (Lack of synchronization). *Let* WFG *be a workflow graph, end its end node and* $fork \in N_{forks}$*. Furthermore, let* $stop_1, stop_2 \in \{fork, end\}$.

$$\begin{gathered}\textit{From the execution of } fork \textit{ follows a lack of synchronization}\\ \Rightarrow\\ \exists suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2\colon\\ \bar{\iota}(suc_1, suc_2) \not\subseteq N_{joins},\ \textit{or}\\ \exists\ \textit{direct}\ path_1 = (suc_1, \ldots, stop_1),\ path_2 = (suc_2, \ldots, stop_2)\colon\\ path_1 \cap path_2 = \emptyset.\end{gathered}$$

The basic idea of the proof is to show that no two control flows built by a fork can ever execute the same node twice in series.

Proof (Theorem 3).

$\Rightarrow$ Proof by contradiction. The assumption has to hold:

$$\begin{gathered}\text{(1) From the execution of } fork \text{ follows a lack of synchronization} \wedge\\ \text{(2) } \forall suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2\colon\\ \text{(2.1) } \bar{\iota}(suc_1, suc_2) \subseteq N_{joins}, \text{ and}\\ \text{(2.2) } \forall \text{ direct } path_1 = (suc_1, \ldots, stop_1),\ path_2 = (suc_2, \ldots, stop_2)\colon\\ path_1 \cap path_2 \neq \emptyset.\end{gathered}$$

Let $suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2$.
$\overset{(2.1),(2.2)}{\Rightarrow} \forall$ such direct paths $path_1, path_2$ of $suc_1, suc_2, \exists \cap$*-point* $\in \bar{\iota}(suc_1, suc_2)\colon$ $\cap$*-point* $\in path_1 \cap path_2$.
$\Rightarrow \forall$ pairs of possible control flows cf_1, cf_2 starting in suc_1, suc_2 holds, that cf_1, cf_2 will be joined in an $\cap$*-point* $\in \bar{\iota}(suc_1, suc_2)$.
$\overset{(2.1)}{\Rightarrow}$ a lack of synchronization can not appear after joining.
$\overset{(1)}{\Rightarrow}$ a lack of synchronization must appear before joining, but this is not possible $\lightning$ $\square$

The inverse direction is not equivalent since the condition induces a lack of synchronization or a deadlock.

Theorem 4 (Potential lack of synchronization). *Let WFG be a workflow graph, end its end node and $fork \in N_{forks}$. Furthermore, let $stop_1, stop_2 \in \{end, fork\}$.*

$$\exists suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2 :$$
$$\bar{\iota}(suc_1, suc_2) \not\subseteq N_{joins}, \text{ or}$$
$$\exists \text{ direct } path_1 = (suc_1, \ldots, stop_1),\ path_2 = (suc_2, \ldots, stop_2) :$$
$$path_1 \cap path_2 = \emptyset$$
$$\Rightarrow$$
from the execution of $fork$ follows a lack of synchronization, or there is a deadlock.

Proof (Theorem 4).

$\Rightarrow$ Constructive proof. Let the following hold:

$$(1.1)\ \exists suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2 :$$
$$(1.2)\ \bar{\iota}(suc_1, suc_2) \not\subseteq N_{joins}, \text{ or}$$
$$(1.3)\ \exists \text{ direct } path_1 = (suc_1, \ldots, stop_1),\ path_2 = (suc_2, \ldots, stop_2) :$$
$$path_1 \cap path_2 = \emptyset$$
$$\Rightarrow$$
$$(2.1) \text{ from the execution of } fork \text{ follows a lack of synchronization, or}$$
$$(2.2) \text{ there is a deadlock}$$

Let assumptions (1.1), (1.2), and (1.3) holds for $fork$. There follow two cases from (1.2) and (1.3).

Case 1: (1.2) holds for $fork$
$\overset{(1.2)}{\Rightarrow} \exists suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2 :\ \bar{\iota}(suc_1, suc_2) \not\subseteq N_{joins}$
$\overset{\text{Definition 4}}{\Rightarrow}$ two direct paths $path_1 = (suc_1, \ldots, \cap\text{-}point)$, $path_2 = (suc_2, \ldots, \cap\text{-}point)$ with $path_1 \cap path_2 = \{\cap\text{-}point\}$, $\cap\text{-}point \notin N_{joins}$
$\Rightarrow$ $fork$, $path_1$ and $path_2$ can be executed, or there is a deadlock
$\overset{(1.2)}{\Rightarrow}$ if $path_1, path_2$ can be executed, then $\cap$-*point* can be executed twice in series
$\Rightarrow$ a lack of synchronization by definition ✓

Case 2: (1.3) holds for $fork$
$\Rightarrow \exists suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2,$
$\exists$ direct $path_1 = (suc_1, \ldots, stop_1),\ path_2 = (suc_2, \ldots, stop_2) :$
$path_1 \cap path_2 = \emptyset$
$\Rightarrow stop_1 \neq stop_2$, without loss of generality, $path_1$ ends in $fork$ and $path_2$ ends in *end*
$\Rightarrow path_1$ describes a backward loop
$\Rightarrow fork$ and $path_1$ can be executed, or there is a deadlock
$\Rightarrow fork$ can be executed again

$\Rightarrow$ suc_2 has been executed twice
$\Rightarrow$ a node on $path_2$ (at least the end node) has been executed twice in series
$\Rightarrow$ a lack of synchronization by definition ✓

□

The conditions used in Theorem 1 and 3 describe a superset of deadlocks and lack of synchronisation, since parts of it never occur at runtime, because forgoing deadlocks will prevent their execution (Theorem 2 and 4). Therefore, we call them *potential*.

Nevertheless, the structural correctness of a business process can be proven if we can show that no potential deadlock and lack of synchronization arise in its corresponding workflow graph. In addition, with the successive elimination of deadlocks and lack of synchronization during the development process, a moment will be reached at which the set of potential errors equals the set of real errors. In this sense, based on conditions used in Theorem 1 and 3, a finite development process could be defined which eventually can be used for the determination of real deadlocks and real lack of synchronization.

5 Determination of Structural Errors

The following section describes how to find structural errors within workflow graphs, more specifically, potential deadlocks and potential lack of synchronization. The basic idea of the overall algorithm is the iteration over two steps until the workflow graph is structurally correct. The first step determines potential lack of synchronization, which are then bug-fixed by the user. Afterwards, the potential deadlocks will be determined, which will also be bug-fixed.

Basically, to determine potential deadlocks, each path from the start node to a join and from this join to itself is checked whether it contains a fork as a closest activation point. If this does not hold true for a certain join, then this join has a potential deadlock. The determination of potential lack of synchronization is straightforward to Theorem 3, i.e., all paths from direct successor nodes of a fork to the end node and to the fork itself will be determined and pairwise examined: if both paths of all pairs have a first common node and this is a join, the fork is said to be free of lack of synchronization.

5.1 Determination of Potential Deadlocks

In the following, let $join \in N_{joins}$. The first step of the algorithm finds all entrypoints for $join$. Thereby, the focus lies on entrypoints which are forks. Afterwards, the closest entrypoints of $join$ will be determined. These closest entrypoints have to be checked to be safe, i.e., they are activation points. Eventually, the algorithm checks the conditions of Theorem 1.

Summarized, the steps of the algorithm are as follows:

1. Determine the entrypoints $\Sigma(join) \subseteq N_{forks}$ of $join$.

2. Determine closest entrypoints ($\Sigma_{closest}(join)$).
3. Determine closest activation points ($\Sigma_{activation}(join)$).
4. Check conditions of Theorem 1.

1. Determine the entrypoints. With regard to Lemma 1, closest activation points of node $join$ can only be forks which have a direct path to each direct predecessor node of $join$. But this basic lemma is not sufficient to prove that a fork is an entrypoint of a join.

For example, a look on Fig. 3 shows, that the fork $F2$ is no entrypoint of $J2$, because no execution of $F2$ follows an execution of $J2$. The execution will be stopped by the join $J1$, because $F2$ is not an entrypoint of $J1$.

Instead of searching the entrypoints for $join$ we determine for each fork the set of nodes for which it is an entrypoint, called the *scope* of a fork.

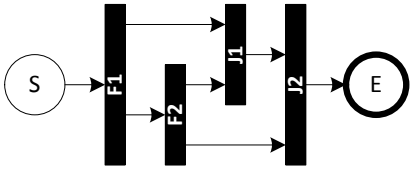

Fig. 3. Two forks and two joins

Definition 5 (Scope of forks). *Let* $fork \in N_{forks}$. *The* scope $\sigma(fork)$ *is a set of nodes with* $\sigma(fork) = \{n : fork \text{ is an entrypoint of } n\}$.

A fork $fork$ is an entrypoint of each of its direct successor nodes, since the workflow graph is simple. Furthermore, if a node n is not a join and has a direct predecessor node for which $fork$ is an entrypoint, then n has $fork$ also as entrypoint, since n can be executed if at least one predecessor node was executed. At last, if the node n is a join and each of its direct predecessor nodes has $fork$ as entrypoint, then $fork$ is also an entrypoint of n, since n can be executed if all of its direct predecessor nodes were executed. Hence, the scope $\sigma(fork)$ of a $fork$ could be determined recursively with the algorithm in Fig. 4.

A fork is an entrypoint of a join if the join is within the scope of the fork. After the determination of each scope of each fork, the set $\Sigma(join)$ can be determined containing all entrypoints being forks of $join$. If $\Sigma(join)$ is empty for $join$, then $join$ cannot have closest activation points, i.e., $join$ has a potential deadlock.

2. Determine closest entrypoints. From the definition of closest entrypoints, there has to be at least one path from an entrypoint of a node n to this node n which contains no other entrypoint of n. The closest entrypoints $\Sigma_{closest}(join) \subseteq \Sigma(join)$ can be determined efficiently with a backward depth-first search. It searches the entrypoints of $join$. If such an entrypoint was reached, then this entrypoint is marked as a closest entrypoint of $join$, and the depth-first search stops the ongoing traverse of this path. If the start node or $join$ was reached, then $join$ has a potential deadlock.

3. Determine closest activation points. Referring to Lemma 1, the closest entrypoint $fork$ is a closest activation point for $join$ if $fork$ is an activation point

```
Input: A fork fork
Output: The scope σ(fork) of fork
 1: σ(fork) is an empty set of nodes
 2: for all suc ∈ fork• do
 3:     determineScope(suc)
 4:
 5: function determineScope(current)
 6:     if current ∉ σ(fork) then
 7:         if current ∉ N_joins then
 8:             σ(fork) ← σ(fork) ∪ {current}
 9:             for all suc ∈ current• do
10:                 determineScope(suc)
11:         else
12:             if •current ⊆ σ(fork) then
13:                 σ(fork) ← σ(fork) ∪ {current}
14:                 for all suc ∈ current• do
15:                     determineScope(suc)
```

Fig. 4. Determine the scope of a fork

for each $pre \in \bullet join$. As mentioned before, $fork$ is an activation point of a $pre \in \bullet join$ if the execution of pre follows after the execution of $fork$. More specifically, there is a direct path from $fork$ to pre which will be guaranteed to be executed. In general, there could be more than one path from $fork$ to pre, e.g., the paths start of different direct successor nodes of $fork$; or decisions (splits) creates a divergence. Therefore, all the direct paths starting in the same direct successor node of $fork$ and ending in pre are merged. This union is called *deliverer*, because it describes how a control could be delivered from a direct successor node of $fork$ to pre.

Definition 6 (Deliverer). *Let $join \in N_{joins}$, $pre \in \bullet join$, $fork \in N_{forks}$ is a closest entrypoint of $join$, and $suc \in fork\bullet$, which has a direct path to pre.*

A deliverer *of $join$ between suc and pre is a set of nodes $\delta(fork, join, suc, pre) = \{n : n$ lies on a direct path from $fork$ to $join$ containing suc and $pre\}$.*

With the help of the definition of deliverers, the safeness of a closest entrypoint, i.e., the closest activation points, could be formulated as follows.

Lemma 2 (Safeness). *Let $join \in N_{joins}$, and $fork$ be a closest entrypoint of $join$.*

$fork$ is a safe and closest entrypoint of $join$ iff $\forall pre \in \bullet join$, $\exists suc \in fork\bullet$: $\delta(fork, join, suc, pre)$ will be guaranteed to be executed.

A deliverer $\delta(fork, join, suc, pre)$ will be guarenteed to be executed if it neither contains a deadlock nor control flows can leave it. Since $fork$ must be a closest activation point of $join$, it has to be an activation point of all joins within

Input: a workflow graph $WFG = (N, E)$
Output: all joins with a potential deadlock

1: $deadlocks \leftarrow \emptyset$
2: **for all** $fork \in N_{forks}$ **do**
3: determine the scope $\sigma(fork)$ and the entrypoints $\Sigma(join)$ of each $join \in N_{joins}$
4: **for all** $join \in N_{joins}$ **do**
5: determine the closest entrypoints $\Sigma_{closest}(join)$ with a backward depth-first search
6: **for all** $entrypoint \in \Sigma_{closest}(join)$ **do**
7: determine all deliverers $\Delta(entrypoint, join)$ for each direct successor node of $entrypoint$ and predecessor node of $join$
8: **for all** $\delta(entrypoint, join, suc, pre) \in \Delta(entrypoint, join)$ **do**
9: determine guaranteed execution of $\delta(entrypoint, join, suc, pre)$
10: **if** $\delta(entrypoint, join, suc, pre)$ is guaranteed to be executed **then**
11: mark pre as $safe$ for $entrypoint$
12: **if** not all $pre \in \bullet join$ are marked as $safe$ for $entrypoint$ **then**
13: eliminate $entrypoint$ from $\Sigma_{closest}(join)$
14: ▷ note, $\Sigma_{last}(join)$ contains noe all safe and last entrypoints of $join$
15: do a backward depth-first search with begin in $join$ and which stops in a traversation of a path on a $fork \in \Sigma_{last}(join)$
16: **if** the start node or $join$ were reached by the depth-first search **then**
17: $deadlocks \leftarrow deadlocks \cup \{join\}$
18: **return** $deadlocks$

Fig. 5. Determine potential deadlocks

this deliverer. Without loss of generality, we assume that $fork$ is an activation point of all these joins.

Furthermore, an execution of $\delta(fork, join, suc, pre)$ is given if the control flow cannot leave this deliverer. The only node where it is possible to leave a deliverer is a split. Thus, if $\delta(fork, join, suc, pre)$ contains a split which has a path outside this deliverer, then an execution is not guaranteed.

Lemma 3 (Guaranteed execution). *Let $\delta(fork, join, suc, pre)$ be a deliverer whose $fork$ is an activation point of all inner joins.*

The execution of $\delta(fork, join, suc, pre)$ is guaranteed *iff* $\forall split \in (\delta(fork, join, suc, pre) \cap N_{splits})\colon\ split\bullet \subseteq \delta(fork, join, suc, pre)$.

Summarized, the safeness of each closest entrypoint of a join can be determined, i.e., the set $\Sigma_{activation}(n_{join})$.

4. Check the conditions of Theorem 1. This could be proved easily by a backward depth search with begin at $join$. It searches the closest activation points of $join$. If such an activation point was found, it stops the further traverse of this path. If it reaches the start node or $join$ itself, $join$ has a potential deadlock.

The overall algorithm is shown in Fig. 5 and has a cubic runtime complexity, although faster implementations are possible.

5.2 Determination of Potential Lack of Synchronization

In the following, let $suc_1, suc_2 \in fork\bullet, suc_1 \neq suc_2$. Furthermore, let $path_1 = (suc_1, \ldots, stop_1)$ and $path_2 = (suc_2, \ldots, stop_2)$ be two direct paths with $stop_1, stop_2 \in \{fork, end\}$, whereas end is the end node. Note, Theorem 3 states that a lack of synchronization will be caused directly by $fork$ if there are two paths with $path1 \cap path2 = \emptyset$, or the closest common node of them is not a join.

Since forks are the entrypoints for the determination of potential lack of synchronzations, the analysis is done for each $fork$. The first step of the algorithm determines for each direct successor node suc of $fork$ the set of all direct paths $paths(suc)$ from suc to $fork$ and from suc to the end node. The next step checks for each pair (suc_1, suc_2) if there is a pair $(path_1, path_2) \in paths(suc_1) \times paths(suc_2)$, where paths $path_1, path_2$ are disjoint or have a closest intersection point not being a join.

Summarized, the steps of the algorithm for each $fork$ and two of its different direct successor nodes suc_1, suc_2 are as follows:

Step 1: Determine the sets $paths(suc_1), paths(suc_2)$.
Step 2: For each $path_1 \in paths(suc_1)$ and for each $path_2 \in paths(suc_2)$ check:
  (a) $path_1 \cap path_2 = \emptyset$, and
  (b) closest intersection point of $path_1$ and $path_2$ is not a join.

1. Find the sets $paths(suc_1), paths(suc_2)$. As mentioned before, for a $suc \in fork\bullet$ holds that $paths(suc) = \{p : p$ is a direct path from suc to $fork$ or from suc to the end node $\}$. Theoretically, there could be any number of such paths, because the workflow graph may contain loops. To address this fact, only the simple paths from a $suc \in fork\bullet$ to $fork$ and to the end node will be determined. Simple paths abstract from possible cycles. Since cycles can start in splits or forks, cycles are not of interest, as either a cycle starting in a fork will be handled when considering lack of synchronization of this fork, or a cycle starting in a split is allowed in general, since the number of control flows does not grow.

Finding all simple paths between two nodes in a directed graph is called an *all simple paths* problem and a performant algorithm can be found in Pahl et al. [7]. A simple algorithm is illustrated in Fig. 6.

2. Checks done for each $(path_1, path_2) \in paths(suc_1) \times paths(suc_2)$. The check $path_1 \cap path_2 = \emptyset$ will be done first, because it guarantees the existence or absence of a closest intersection point. If $path_1 \cap path_2 = \emptyset$, $fork$ has a lack of synchronization or there was a deadlock.

For the second check, it holds that $path_1 \cap path_2 \neq \emptyset$. Furthermore, each node of $path_1 \cap path_2$ is an intersection point of suc_1, suc_2. An intersection point $\cap$-*point* of suc_1, suc_2 in $path_1 \cap path_2$ is closest by definition iff it has a $pre \in \bullet \cap$-*point* with $pre \in path_1$ and $pre \notin path_2$, and vice versa.

Summarized, the closest intersection point of suc_1, suc_2 within $path_1$ and $path_2$ can be determined by iterating over each intersection point within $path_1 \cap path_2$ and applying the definition. If the found closest intersection point is not a join, then $fork$ has a lack of synchronization or there was a deadlock.

```
Input: a fork ∈ N_forks, a suc ∈ fork• and the end node
Output: the set paths(suc) of all paths from suc to node end and from suc to fork
 1: paths(suc) ← ∅
 2: stack ← ∅
 3: push suc on stack
 4: findPath(suc, stack)
 5:
 6: function findPath(current, stack)
 7:     for all suc ∈ current• do
 8:         if suc ∉ stack then
 9:             push suc on stack
10:             if suc = end or suc = fork then
11:                 create path from stack and put it into paths(suc)
12:             else
13:                 findPath(suc, stack)
14:             pop stack
```

Fig. 6. Determine all simple paths

The overall algorithm will be shown in Fig. 7. The implementation of the algorithm was presented at this point for a better understanding. Although the runtime complexity of the algorithm looks inacceptable, it is quite possible to build an algorithm which runs in quadratic time, like used in our implementation. In this implementation, all paths of direct successor nodes will be abstracted by a subgraph. To handle divergences, also the subgraphs of direct successor nodes of splits are considered. Eventually, by handling all direct successor nodes of the fork together, the overall runtime is quadratic (see appendix 9.2).

5.3 Complex Workflow Graphs

By definition, a complex workflow graph contains IOR splits and IOR joins. Before handling these special nodes, their semantics will be explained. An IOR split builds a new control flow for at least one of its outgoing edges if a control flow reaches it. Thus, it is an in-between of a split (XOR semantic) and a fork (AND semantic). An IOR join will be activated to be executed if a control flow reaches it. However, it will not execute until a control flow can ever reach it. Thus, an IOR join can only have a deadlock if it describes the end node of a cycle. Since the semantics of cycles ending in IOR joins are not yet defined [8], it is assumed, that workflow graphs have no cycles ending in IOR joins in the following. The existance of such cycles will be expressed as warnings. The algorithms and definitions above must be reused and extended to determine the potential deadlocks and potential lack of synchronization of complex workflow graphs.

Since by definition an IOR join can never cause a potential deadlock, these nodes are not be handled separately. However, an IOR split can influence the closest activation points of common joins. As mentioned before, a deliv-

Input: a workflow graph $WFG = (N, E)$
Output: all forks which could cause a potential lack of synchronization

```
 1: for all fork ∈ N_forks do
 2:     for all suc ∈ fork• do
 3:         determine paths(suc)
 4:     for all (suc_1, suc_2) ∈ (fork• × fork•), suc_1 ≠ suc_2 do
 5:         for all (path_1, path_2) ∈ paths(suc_1) × paths(suc_2) do
 6:             if path_1 ∩ path_2 = ∅ then
 7:                 lackOfSync ← lackOfSync ∪ {fork}
 8:             else
 9:                 Find closest intersection point ∩-point within path_1 and path_2
10:                 if ∩-point ∉ N_joins then
11:                     lackOfSync ← lackOfSync ∪ {fork}
12: return lackOfSync
```

Fig. 7. Determine potential lack of synchronization

erer $\delta(fork, join, suc, pre)$ must be guaranteed to be executed. Since an IOR split has the same property as a split, i.e., that not all of its direct successor nodes is guaranteed to be executed, Lemma 3 has to be extended for IOR splits so, that the execution of $\delta(fork, join, suc, pre)$ is guaranteed if for all $split \in \delta(fork, join, suc, pre) \cap (N_{splits} \cup N_{IORsplits})$ holds that $split\bullet \subseteq \delta(fork, join, suc, pre)$.

Furthermore, an IOR split can produce more than one control flow. Thus, it can cause potential lack of synchronization. Since Theorem 3 descibes potential lack of synchronization for non-complex workflow graphs, this theorem and the definition of intersection points must be extended. Each fork within the definition of intersection points and within the theorem has to be considered a synonym for a common fork or an IOR split. Furthermore, a closest intersection point of two direct successor nodes of a fork and of an IOR split can also be an IOR join.

6 Evaluation

We have implemented the algorithms to detect structural errors in a (complex) workflow graph in Java. To check the practical application of the approach, we have evaluated it twice, (1) in the Activiti BPMN 2.0 designer, a modeler for business processes, and (2) as a soundness verification tool.

Activiti BPMN 2.0 designer. To verify the usebility of the structural correctness approach, we have implemented the algorithms for the Activiti BPMN 2.0 designer (`http://activiti.org`). Therefore, algorithms were changed to allow full detailed failure information used to visualize the error conditions to the user.

The designer itself includes a disabled verification extension point, which was enabled to create some stand-alone extensions. These extensions transform the graphical model of Activiti into a simple (complex) workflow graph. Therefore, most nodes of the workflow graph has correspond Activiti nodes to represent the

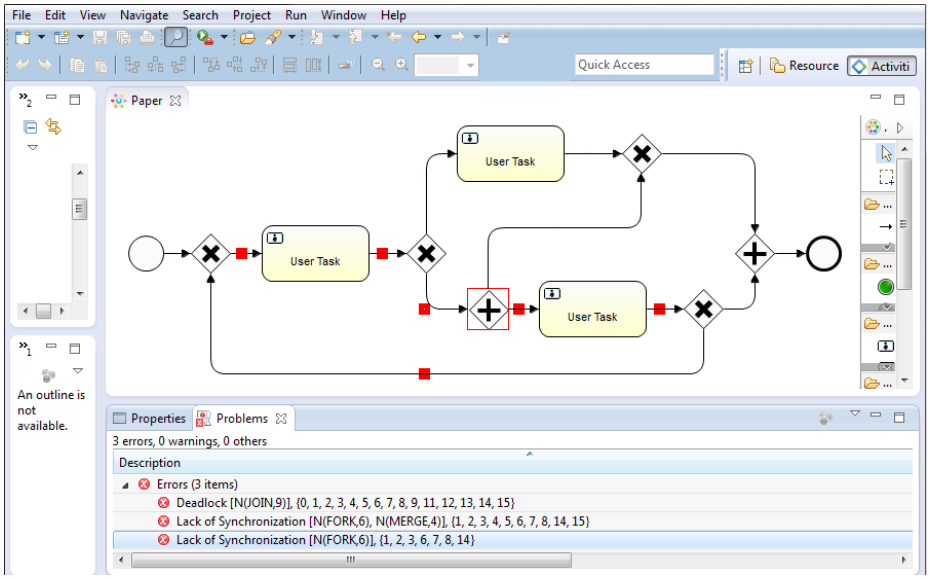

Fig. 8. Visualizing control-flow errors in Activiti

analysis results back to the graphical model. After the transformation process, the structural correctness analysis starts and directly visualize possible failures back into the graphical model of Activiti.

Figure 8 depicts an illustration of the tool highlighting a detected lack of synchronization within the graphical model, and showing a list of all errors. Impressively, the structural correctness analysis is upon every change to the graphical model without a visible delay. Furthermore, the detailed failure information allows an excellent failure analysis, failure understanding and bug-fixing of the end user.

Soundness verification tool. The comparison of the processing time to other soundness verification approaches was the primary goal of the evaluation of the algorithms as soundness verification tool.

The benchmark was taken from `http://www.service-technology.org/soundness` and contains real-world business processes of IBM [3]. It is splitted in 5 libraries, i.e., A, B1, B2, B3 and C, where B1, B2 and B3 describes changing processes in three generations. This benchmark was also used by Fahland et al. [3]. A PNML [9] file was used as input describing a Petri net. As shown by Fahland et al. [3], all Petri nets in the benchmark are so-called free-choice Petri nets, so the transformation into workflow graphs can be performed easily. By using Petri nets, we can directly compare the results with other tools like LoLA (`http://www.informatik.uni-rostock.de/tpp/lola/`).

Table 1. Results of the benchmark evaluation

Library:	A	B1	B2	B3	C
Analysis time [ms]	16.4	15.4	20.7	28.4	1.7
Analysis time LoLA [ms]	2373.0	2395.9	3126.1	3651.3	303.8
Per process avg./max. [ms]	0.06/0.28	0.06/0.36	0.06/0.47	0.07/0.69	0.06/0.31
Per process LoLA avg. [ms]	8.5	8.4	8.7	8.7	9.5

For benchmark evaluation, we have changed our algorithms to stop structural analysis upon first error. This follows the approach of Fahland et al. [3]. Furthermore, the algorithm was tuned to answer the yes-no question if the workflow graph is structurally correct or incorrect.

Our runtime environment was a 64 bit Intel® Core™2 CPU E6300 processor and 2 GB main memory. The system ran Linux 3.1.0. Because of the hot spot compiler of the JRE, we created a startup as long as the optimization system needed to optimize the most hot methods. We ran each of the 5 libraries 10 times, removed the two best and worst results and calculated the average run time. The highest deviation of the best and worst result, respectively, compared with the average run time was 6 %.

We have chosen LoLA to compare our solution with existing tools. The SESE decomposition approach is hard to compare, because a standalone implementation was not available and it depends on other soundness verification approachs. Table 1 shows the results of the benchmark evaluation.

All the structural correct (sound) and uncorrect (unsound) processes of the benchmarks were detected successfully.

Compared to LoLA, our algorithm is 150 times faster. This is not the main result, because it is difficult to compare LoLA with our approach. LoLA was not build to verify business processes and has a different focus. Since Fahland et al. [3] have shown that SESE decomposition and the Woflan tool have comparable runtimes, our approach is faster than the state-of-the-art tools compared by Fahland et al. [3].

7 Related Work

In literature exist many approaches to prove the structural correctness, i.e., the soundness, of a workflow graph or a corresponding workflow net. To compare the different approaches among each other, they will be catogorized with regard to the three requirements: (1) completeness, (2) fastness, and (3) usability. If a requirement is fulfilled, then the technique gets a ++. A + is given if the requirement is semi fulfilled. The technique gets a minus − if the requirement is unfilled. Table 2 shows the results of the technique comparison.

Summarized, these techniques are separated into techniques which transform the workflow graph into a free choice Petri net, and techniques which works on the workflow graph directly.

Table 2. Comparison of soundness checking techniques

Technique	Completeness	Fastness	Usability	Sum
Free choice Petri nets				
Rank theorem [2]	++	++	−	+++
Model Checking [4,5]	++	−	+	+++
Workflow Graphs				
Integer programming [10]	+	−	++	+++
Acyclic graphs [11,12]	−	++	++	++++
SESE decomposition [6]	+	++	++	+++++

The fastest free choice Petri net soundness verification approach uses the rank theorem [2], i.e., a mathematical theorem of linear algebra. It has at least a cubical time complexity in the size of the workflow graph, but does not provide diagnostic information. The other approach to determine the soundness on free choice Petri nets is model checking with tools like Woflan [4] or LoLA [5]. Thus, a search on the state space of the free choice Petri net will be performed. This technique can lead to an exponential processing time in the size of the free choice Petri net. However, it supplies a failure trace (or execution sequence) that leads to the first error found. It is not possible to detect all failures with this technique.

Techniques working directly on the workflow graph are (1) building an integer programming [10], (2) working on acyclic or restricted workflow graphs, e.g., [11,12], or (3) performing a SESE decomposition [6]. The integer programming technique of Eshuis et al. [10] has an exponential worst case time complexity. In contrast, it returns very detailed and localized failure information, but the processing time makes it inapplicable to development tools for end users.

The approach of working on acyclic or restricted workflow graphs restricts the completeness of the soundness checking tool, rendering it inapplicable, although it could have a very fast processing time and could provide very detailed failure information.

The best known technique for soundness checking is performing a SESE decomposition [6]. It decompose the workflow graph in subgraphs which have a single entry and a single exit. This decomposition could be done in linear time by constructing a Refined Process Structure Tree [13]. Each of the subgraphs will be checked first by the application of heuristics [6]. Uncovered subgraphs then will be checked by other techniques, like space state exploration. Because fragments are usually smaller than the entire workflow graph, the state space exploration performs fast [3]. However, an exponentially processing time in the size of the workflow graph is still possible. Summarized, the SESE decomposition in addition to the heuristics works fast and gives detailed and localized failure information, but the heuristics do not cover all cases.

Our new approach to verify structural correctness is comparable to the SESE decomposition approach of Vanhatalo et al. [6]. Both techniques find failures in isolation. However, the SESE decomposition found only one failure per fragment, while our approach found all potential errors. Furthermore, the SESE

decomposition does not always find the structural reason of failures. Therefore, an end user cannot repair these structures. In conclusion, our approach is complete and can handle complex workflow graphs.

8 Conclusion

In this paper, new compiler-based techniques to determine the structural correctness, i.e., the soundness, of a workflow graph were introduced. They directly work on workflow graphs, in order that they guarantee a precise visualization and explanation of all determined structural errors, which substantially supports building business processes. Furthermore, the delevoped techniques demonstrate that well-known compiler techniques can be used for business processes. It is possible to perform a structural correctness analysis in each development step, which directly visualizes errors within the editor and shows only failures which must be fixed.

Main issues for future work are data-dependencies by transforming business processes into CSSA-based workflow graphs [14,15]. Then, an analysis of data-dependencies could be used to solve other actual problems, e.g., the operability of workflow graphs, and the data-dependent termination analysis.

References

1. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. **25**(2) (April 2000) 117–134
2. Aalst, W.M.P.v.d., Hirnschall, A., Verbeek, H.M.W.E.: An alternative way to analyze workflow graphs. In: Proceedings of the 14th International Conference on Advanced Information Systems Engineering. CAiSE '02, London, UK, UK, Springer-Verlag (2002) 535–552
3. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: Instantaneous soundness checking of industrial business process models. Data Knowl. Eng. **70**(5) (May 2011) 448–466
4. Verbeek, E., van der Aalst, W.M.P.: Woflan 2.0: a petri-net-based workflow diagnosis tool. In: Proceedings of the 21st international conference on Application and theory of petri nets. ICATPN'00, Berlin, Heidelberg, Springer-Verlag (2000) 475–484
5. Wolf, K.: Generating petri net state spaces. In: Proceedings of the 28th international conference on Applications and theory of Petri nets and other models of concurrency. ICATPN'07, Berlin, Heidelberg, Springer-Verlag (2007) 29–42
6. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through sese decomposition. In: ICSOC. (2007) 43–55
7. Pahl, P.J., Damrath, R.: Mathematical Foundations of Computational Engineering: A Handbook. 1st edn. Springer Berlin / Heidelberg, Berlin, Heidelberg (jul 2001)
8. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. In: BPM. (2010) 294–309
9. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The petri net markup language: Concepts, technology, and tools. In: ICATPN. (2003) 483–505

10. Eshuis, R., Kumar, A.: An integer programming based approach for verification and diagnosis of workflows. Data Knowl. Eng. **69**(8) (August 2010) 816–835
11. Perumal, S., Mahanti, A.: A graph-search based algorithm for verifying workflow graphs. 2012 23rd International Workshop on Database and Expert Systems Applications **0** (2005) 992–996
12. Favre, C., Völzer, H.: Symbolic execution of acyclic workflow graphs. In: Proceedings of the 8th international conference on Business process management. BPM'10, Berlin, Heidelberg, Springer-Verlag (2010) 260–275
13. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Proceedings of the 6th International Conference on Business Process Management. BPM '08, Berlin, Heidelberg, Springer-Verlag (2008) 100–115
14. Amme, W., Martens, A., Moser, S.: Advanced verification of distributed ws-bpel business processes incorporating cssa-based data flow analysis. International Journal of Business Process Integration and Management **4**(1) (2009) 47–59
15. Heinze, T.S., Amme, W., Moser, S.: A restructuring method for ws-bpel business processes based on extended workflow graphs. In: BPM. (2009) 211–228

9 Appendix

9.1 Data-flow analysis

A fast implementation of the algorithm in Fig. 7 is based on data-flow analyses. In general, a data-flow analysis can be described by the following four data-flow equations.

$$gen_{name}(node) : \text{the information generated by } node \tag{1}$$
$$kill_{name}(node) : \text{the information destroyed by } node \tag{2}$$
$$in_{name}(node) = \bigsqcup_{n \in prev(node)} out_{name}(node) \tag{3}$$
$$: \text{the incoming information of } node \tag{4}$$
$$out_{name}(node) = gen_{name}(node) \cup (in_{name}(node) \setminus kill_{name}(node)) \tag{5}$$
$$: \text{the outgoing information of } node \tag{6}$$

The set $prev$ is defined by the direction of the data-flow analysis, i.e., it is a forward or backward analysis. It holds $prev(node) = \bullet node$ in a forward and $prev(node) = node\bullet$ in a backward analysis. The $\bigsqcup$ operator defines the effect of confluences and is called data-flow confluence operator.

The following algorithm solves a data-flow analysis described by the previous data-flow equations for a given workflow graph WFG, a given data-flow confluence operator $\bigsqcup$, and a direction $prev$.

```
1: function dataFlowAnalysis(in, out, gen, kill, WFG, ⨆, prev)
2:     stable
3:     repeat
4:         stable ← true
5:         for all cur ∈ N do
6:             new ← ⨆_{n∈prev(cur)} out(suc)
7:             if new ≠ in(cur) then
8:                 stable ← false
9:                 in(cur) ← new
10:                out(cur) ← gen(cur) ∪ (in(cur) \ kill(cur))
11:    until stable = true
```

The time complexity of performing such a data-flow analysis is linearly or quadraticly in the typical case depending on the ordering and implementation.

9.2 Potential lack of synchronization algorithm

Input: a workflow graph $WFG = (N, E)$
Output: all forks with a potential lack of synchronization

```
1: ▷ Declare a set of lack of synchronization.
2: lackOfSync ← ∅
```

3: ▷ The algorithm is splitted in three parts,
(1) determine necessary data,
(2) check disjoint paths, and
(3) check and find closest intersection points.

4: ▷ (1) Determine necessary data.
Initialize the sets for a data-flow analysis. The data-flow analysis determines for each direct successor node of a fork or a split the set of successor nodes, which are reachable without passing this fork or this split.

5: $gen_{succ^*}(node) \leftarrow \begin{cases} \{node\} & ,\exists div \in N_{forks} \cup N_{splits} \colon node \in div\bullet \\ \emptyset & ,\text{otherwise} \end{cases}$

6: $kill_{succ^*}(node) \leftarrow \begin{cases} node\bullet & ,node \in N_{forks} \cup N_{splits} \\ \emptyset & ,\text{otherwise} \end{cases}$

7: **for all** $node \in N$ **do**

8: $\quad in_{succ^*}(node), out_{succ^*}(node) \leftarrow \emptyset$

9: ▷ Perform a forward data-flow analysis.

10: dataFlowAnalysis(in_{succ^*}, out_{succ^*}, gen_{succ^*}, $kill_{succ^*}$, WFG, $\bigcup$, $\bullet x$)

11:

12: ▷ Based on the results of the previous data-flow analysis determine all nodes between a fork and itself and between a fork and the end node, respectively. Let $(suc, stop)$ be the information.

13: **for all** $node \in N$ **do**

14: $\quad gen_{to}(node), kill_{to}(node), in_{to}(node), out_{to}(node) \leftarrow \emptyset$

15: $\quad$ **for all** $info \in (\bigcup_{fork \in N_{forks}} fork\bullet) \setminus in_{succ^*}(node)$ **do**

16: $\quad\quad kill_{to}(node) \leftarrow kill_{to}(node) \cup \{(info, end), (info, fork)\}$

17: $\quad$ **if** $node \in N_{forks}$ **then**

18: $\quad\quad$ **for all** $info \in (in_{succ^*}(node) \cap node\bullet)$ **do**

19: $\quad\quad\quad gen_{to}(node) \leftarrow gen_{to}(node) \cup \{(info, node)\}$

20: $\quad$ **if** $node = end$ **then**

21: $\quad\quad$ **for all** $info \in ((\bigcup_{fork \in N_{forks}} fork\bullet) \cap in_{succ^*}(node))$ **do**

22: $\quad\quad\quad gen_{to}(node) \leftarrow gen_{to}(node) \cup \{(info, node)\}$

23: ▷ Perform a backward data-flow analysis.

24: dataFlowAnalysis(in_{to}, out_{to}, gen_{to}, $kill_{to}$, WFG, $\bigcup$, $x\bullet$)

25:

26: ▷ (2) Check disjoint paths.
It will be checked, that on each path from a fork to itself and from a fork to the end node lies a node (an intersection point) that covers all direct successor nodes of this fork.

27: **for all** $node \in N$ **do**

28: $\quad succ^*(node), succ^*_{end}(node), succ^*_{fork}(node) \leftarrow \emptyset$

29: **for all** $node \in N$ **do**

30: $\quad$ **for all** $info \in out_{to}(node)$ **do**

31: $succ^*(info.suc) \leftarrow succ^*(info.suc) \cup \{node\}$
32: **if** $info.stop = end$ **then**
33: $succ^*_{end}(info.suc) \leftarrow succ^*_{end}(info.suc) \cup \{node\}$
34: **else**
35: $succ^*_{fork}(info.suc) \leftarrow succ^*_{fork}(info.suc) \cup \{node\}$
36: **for all** $fork \in N_{forks}$ **do**
37: $allIntersect(fork) \leftarrow \bigcap_{suc \in fork\bullet} succ^*_{end}(suc)$
38: **if** $\exists suc \in fork\bullet$: $succ^*_{fork}(suc) \neq \emptyset$ **then**
39: $allIntersect(fork) \leftarrow allIntersect(fork) \cap \bigcap_{suc \in fork\bullet} succ^*_{fork}(suc)$
40: ▷ In general, the test will be done with a data-flow analysis.
41: **for all** $node \in N$ **do**
42: $gen_{reach}, kill_{reach}, in_{reach}(node), out_{reach}(node) \leftarrow \emptyset$
43: **for all** $fork \in N_{forks}$ **do**
44: $gen_{reach}(fork) \leftarrow \{fork\}$
45: **if** $node \in allIntersect(fork)$ **then**
46: $kill_{reach}(node) \leftarrow kill_{reach}(node) \cup \{fork\}$
47: ▷ Perform the forward data-flow analysis.
48: dataFlowAnalysis(in_{reach}, out_{reach}, gen_{reach}, $kill_{reach}$, WFG, $\bigcup$, $\bullet x$)
49:
50: ▷ If any fork information reaches the end node or itself, then there are at least two paths which are disjoint.
51: **for all** $fork \in in_{reach}(end)$ **do**
52: $lackOfSync \leftarrow lackOfSync \cup \{fork\}$
53: **for all** $fork \in N_{forks}$ **do**
54: **if** $fork \in in_{reach}(fork)$ **then**
55: $lackOfSync \leftarrow lackOfSync \cup \{fork\}$
56:
57: ▷ (3) Check and find closest intersection points. Instead of determine all closest intersection points directly, the *direct* closest intersection points will be determined, i.e., the closest intersection points which are reachable from a direct successor node without passing another intersection point.
58: **for all** $div \in N_{forks} \cup N_{splits}$ **do**
59: $dirClosest(div) \leftarrow \emptyset$
60: **for all** $node \in N$ **do**
61: **for all** $div \in N_{forks} \cup N_{splits}$ **do**
62: ▷ Which direct successor nodes of *div* reaches this node?
63: $isection \leftarrow info(node) \cap div\bullet$
64: **if** $|isection| > 1$ **then**
65: ▷ There is an intersection since more than two direct successor nodes reaches it.
66: ▷ Check if it is a direct and closest intersection point.
67: $closest \leftarrow false$

68: **for all** $pre \in \bullet node$ **do**
69: $cap \leftarrow isection \cap out_{succ^*}(pre)$
70: **if** $cap \neq isection$ and $cap \neq \emptyset$ **then**
71: $closest \leftarrow true$
72: **if** $closest = true$ **then**
73: $dirClosest(div) \leftarrow dirClosest(div) \cup \{node\}$
74:
75: ▷ Determine all closest intersection points.
76: **for all** $fork \in N_{forks}$ **do**
77: $closest(fork) \leftarrow dirClosest(fork)$
78: **for all** $div \in (\bigcup_{suc \in fork\bullet} succ^*(suc)) \cap (N_{forks} \cup N_{splits})$ **do**
79: **for all** $direct \in dirClosest(div)$ **do**
80: **if** $|in_{succ^*}(div) \cap fork\bullet| < |in_{succ^*}(direct) \cap fork\bullet|$ **then**
81: $closest(fork) \leftarrow closest(fork) \cup \{direct\}$
82: ▷ Check, that each closest intersection point is a join.
83: **if** $closest(fork) \not\subseteq N_{joins}$ **then**
84: $lackOfSync \leftarrow lackOfSync \cup \{fork\}$
85:
86: **return** $lackOfSync$