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 business processes causes high costs and could damage the prestige of the providing company. Therefore, validation of the correctness of business processes is essential. In general, business processes are described with Petri nets semantics, even though this kind of description allows only algorithms with a worse processing time and bad failure information to this moment.

In this paper, we describe new compiler-based techniques that could be used instead of Petri net algorithms 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.

1 Introduction

Business processes, e.g., service orchestrations, 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 like Sadiq and Orloswka [1], since soundness describes the overall correctness.

Current soundness checker tools are based on Petri nets, or on workflow graphs, which are similar to control flow graphs using explicit parallelism. Most Petri net-based techniques [4,5] use state space exploration to determine structural errors. This allows the determination of exactly one runtime error, which even could be unsolvable, since it could be caused by a previous error. Take the business process in BPMN notation of Fig. 1 as example. It is possible, that after the execution of the parallel diverging gateway $F1$ the parallel converging gateway $J1$ will be executed, however, the task $T1$ has still a control flow, since there is a classical lack of synchronization situation. If this control flow arrives at $J1$, then there is a deadlock situation. The state space exploration could find the deadlock situation firstly, however, bug-fixing this deadlock seems not to be the

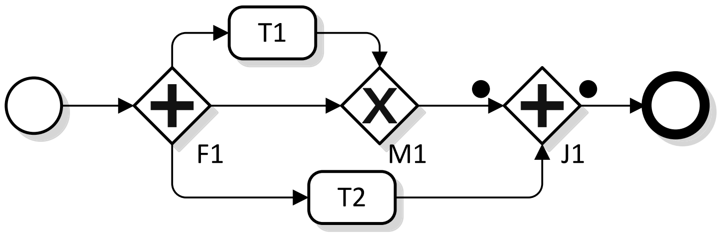


Fig. 1. A lack of synchronization causes a deadlock

best solution to get a correct business process. Therefore, such an information is useless in development tools. Furthermore, state space exploration can lead to an exponential processing time in the size of the Petri net in general. Summarized, they are rather unusable within development tools for business processes.

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 being fast, complete and informative. 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 has exactly one incoming edge; whereas the start node, each activity, merge, and join has 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 $e = (n_1, n_2) \in E$ the source n_1 or the target n_2 is an activity.

Figure 2 shows an example workflow graph. The start and end node are depicted as (thick) circles, and an activity is depicted as rectangle. Forks and joins are illustrated as thin rectangles, whereas splits and merges are depicted as (thick) diamonds.

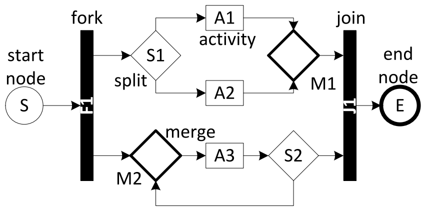


Fig. 2. A workflow graph

The semantics of workflow graphs used in this paper is similar to the semantics of control flow graphs. The execution of a workflow graph begins 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 when a control flow reaches an incoming edge of these nodes, whereas a join can only fire if all incoming edges are reached by a control flow. After executing a split, it decides *nondeterministically*, which outgoing edge will be followed by the control flow in workflow graphs without data aspects. After the execution of a fork, parallel control flows will be built for each outgoing edge.

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* $P = (n_1, n_2, \dots, n_{m-1}, n_m)$ is a sequence of nodes of N such that $\forall i \in \{1, \dots, m-1\} : (n_i, n_{i+1}) \in E$. A path is called *direct* if $n_2, \dots, n_{m-1} \neq n_1, n_m$; and *simple* if all nodes on the path are pairwise different.

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* when its execution may cause a node to be executed twice in series. A workflow graph is structurally correct if it has neither deadlocks nor lack of synchronization.

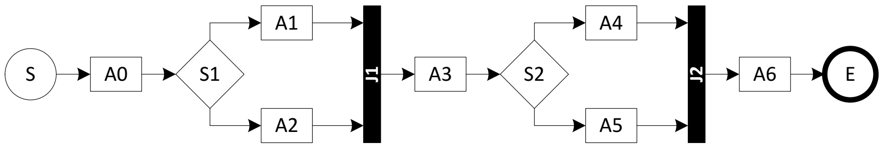


Fig. 3. An unreachable deadlock in join $J2$

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 3 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. 3 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 1 (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. 3. $A1$ 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. The proofs are out of the scope of this paper, however, the interested reader may find them in the technical report [7].

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

Definition 2 (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. Then, the following holds:*

1. *each activation point of a join is an activation point of the joins direct predecessor nodes.*
2. *each closest activation point of a join is 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.*

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

In other words, before any control flow ever arrives at 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 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.

Definition 3 (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 $\overline{\iota}(suc_1, suc_2)$ 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 2 (Lack of Synchronization). *Let WFG be a workflow graph, end its end node and $fork \in N_{forks}$. Furthermore, let $stop_1, stop_2 \in \{fork, end\}$.*

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

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.

The conditions used in Theorem 1 and 2 describe a superset of deadlocks and lack of synchronisation, since parts of it never occur at runtime, because forgoing deadlocks will prevent their execution. 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 2, 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 basic idea of the overall algorithm for detecting structural errors 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 2, 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 causing 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.

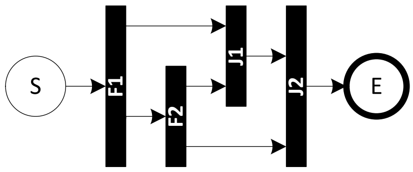


Fig. 4. Two forks and two joins

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 proof that a fork is an entrypoint of a join.

For example, a look on Fig. 4 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.

Definition 4 (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 when 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. 5.

```
Input: A fork $fork$ and $\sigma(fork) \leftarrow \emptyset$
Output: The scope $\sigma(fork)$ of $fork$
for all $suc \in fork\bullet$ do
    determineScope($suc$)

function DETERMINESCOPE($current$)
    if $current \notin \sigma(fork)$ then
        if $current \notin N_{joins}$ then
            commonCase($current$)
        else
            joinCase($current$)
function COMMONCASE($current$)
    $\sigma(fork) \leftarrow \sigma(fork) \cup \{current\}$
    for all $suc \in current\bullet$ do
        determineScope($suc$)

function JOINCASE($current$)
    if $\bullet current \subseteq \sigma(fork)$ then
        $\sigma(fork) \leftarrow \sigma(fork) \cup \{current\}$
        for all $suc \in current\bullet$ do
            determineScope($suc$)
```

Fig. 5. Determine the scope of a fork

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 entry-point $fork$ is a closest activation point for $join$ if $fork$ is an activation point for each $pre \in \bullet join$. As mentioned before, $fork$ is an activation point of a $pre \in \bullet join$ when 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, we merge all the direct paths starting in the same direct successor node of $fork$ and ending in pre. This union of paths is called *deliverer*, because it describes how a control could be delivered from a direct successor node of $fork$ to pre.

Definition 5 (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 $\delta(fork, join, suc, pre) = \{n\colon n$ 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 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

```
Input: a workflow graph $WFG = (N, E)$
Output: all joins with a potential deadlock
1: determine scope $\sigma$ for all forks and entrypoints $\Sigma$ for all joins
2: for all $join \in N_{joins}$ do
3:     determine the last entrypoints $\Sigma_{last}(join)$ with a backward depth-first search
4:     for all $entrypoint \in \Sigma_{last}(join)$ do
5:         determine all deliverers $\Delta(entrypoint, join)$ for each direct successor node
           of $entrypoint$ and predecessor node of $join$
6:         for all $\delta(entrypoint, join, suc, pre) \in \Delta(entrypoint, join)$ do
7:             determine guaranteed execution of $\delta(entrypoint, join, suc, pre)$
8:             if $\delta(entrypoint, join, suc, pre)$ is guaranteed to be executed then
9:                 mark $pre$ as $safe$ for $entrypoint$
10:        if not all $pre \in \bullet join$ are marked as $safe$ for $entrypoint$ then
11:            eliminate $entrypoint$ from $\Sigma_{last}(join)$
12:    do a backward depth-first search with begin in $join$ and which stops in a
       traversation of a path on a $fork \in \Sigma_{last}(join)$
13:    if the start node or $join$ were reached by the depth-first search then
14:        mark $join$ as deadlock
```

Fig. 6. Determine potential deadlocks

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 $\forall split \in (\delta(fork, join, suc, pre) \cap N_{splits})$: $split\bullet \subseteq \delta(fork, join, suc, pre)$.*

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-first 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. 6 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 2 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 synchronzation, 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.

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.

Finding all simple paths between two nodes in a directed graph is called an *all simple paths* problem and a fast algorithm can be found in Pahl et al. [8].

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 guaranteeing the absence of closest intersection points. If $path_1 \cap path_2 = \emptyset$, $fork$ has a potential lack of synchronization.

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, when 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 potential lack of synchronization.

The overall algorithm will be shown in Fig. 7. This implementation of the algorithm was presented at this point for a better understanding. Although the runtime complexity of the algorithm looks inacceptable, it is possible to build an algorithm which runs in quadratic time, like used in our implementation [7,9].

6 Evaluation

We have implemented the algorithms in Java to detect structural errors in workflow graphs. 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. Tools and benchmarks are available on `www.bpmn-compiler.org` and `https://sourceforge.net/projects/bpmojo`

Activiti BPMN 2.0 designer. To verify the usability of the structural correctness approach, we have implemented the algorithms for the Activiti BPMN 2.0 designer (`http://activiti.org`).

```
Input: a workflow graph $WFG = (N, E)$
Output: all forks which could cause a potential lack of synchronization
 1: for all $fork \in N_{forks}$ do
 2:     for all $suc \in fork\bullet$ do
 3:         determine $paths(suc)$
 4:     for all $(suc_1, suc_2) \in (fork \bullet \times fork\bullet), suc_1 \neq suc_2$ do
 5:         for all $(path_1, path_2) \in paths(suc_1) \times paths(suc_2)$ do
 6:             if $path_1 \cap path_2 = \emptyset$ then
 7:                 mark $fork$ as lack of synchronization
 8:             else
 9:                 Find closest intersection point $\cap$-$point$ within $path_1$ and $path_2$
10:                 if $\cap$-$point \notin N_{joins}$ then
11:                     mark $fork$ as lack of synchronization
```

Fig. 7. Determine potential lack of synchronization

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. Practically, the structural correctness analysis is upon every change to the graphical model without a visible delay.

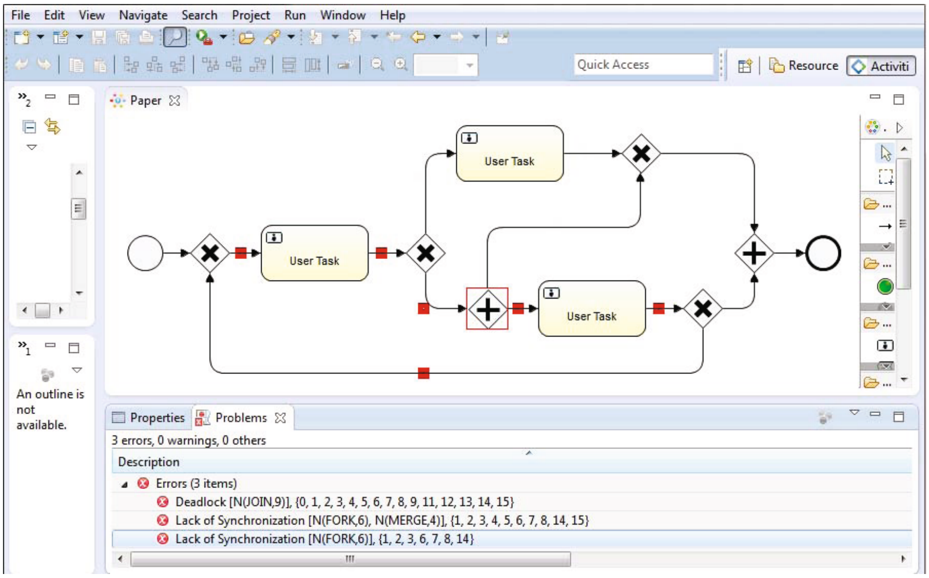
Soundness verification tool. The comparison of the processing time to other soundness verification approaches was the primary goal of the evaluation of the algorithms as soundness verification tool. The benchmark contains real-world business processes of IBM [3]. It is splitted in 5 libraries, i.e., A, B1, B2, B3 and C. This benchmark was also used by Fahland et al. [3]. A PNML [10] file was used as input describing a Petri net and then transformed into a workflow graph. By using Petri nets, we can directly compare the results with other tools like LoLA [5].

For benchmark evaluation, we have changed our algorithms to stop structural analysis upon first error. 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^{TM}2 CPU E6300 processor and 2 GB main memory Linux 3.1.0 system. We ran each of the 5 libraries 10 times, removed the two best and worst results and calculated the average time.

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.

Compared to LoLA, our algorithm is 150 times faster. This is not the major result, since LoLA was not build to verify business processes. Fahland et al. [3] have shown that SESE decomposition and the Woflan tool have comparable runtimes like LoLA. Summarized, our approach is faster than the state-of-the-art tools compared by Fahland et al. [3].


File Edit View Navigate Search Project Run Window Help
Quick Access
Resource
Activiti
Paper
User Task
User Task
User Task
An outline is not available.
Properties
Problems

```
3 errors, 0 warnings, 0 others
Description
Errors (3 items)
    Deadlock [N(JOIN,9)], {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15}
    Lack of Synchronization [N(FORK,6), N(MERGE,4)], {1, 2, 3, 4, 5, 6, 7, 8, 14, 15}
    Lack of Synchronization [N(FORK,6)], {1, 2, 3, 6, 7, 8, 14}
```

Fig. 8. Visualizing control-flow errors in Activiti

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

7 Related Work

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 of 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.

Primarily, techniques working directly on the workflow graph restrict them to acyclic or restricted, e.g., Perumal and Mahanti [11]. Although they could have a very fast processing time and could provide very detailed failure information, they restrict the completeness of the soundness checking tool directly, rendering it inapplicable. An exception and 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 complexity by constructing a Refined Process Structure Tree [12]. Each of the subgraphs will be checked first by the application of heuristics. Uncovered subgraphs then will be checked by other techniques, like space state exploration. Because subgraphs 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 user cannot repair these structures. In conclusion, our approach is complete, i.e., it finds all structural failures.

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 to 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.

Major issues for future work are including data aspects in our techniques by transforming business processes into CSSA-based workflow graphs [13,14].

References

1. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. 25(2), 117–134 (2000)
2. van der Aalst, W.M.P., Hirnschall, A., Verbeek, H.M.W(E.): An alternative way to analyze workflow graphs. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 535–552. Springer, Heidelberg (2002)
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), 448–466 (2011)
4. Verbeek, H.M.W(E.), van der Aalst, W.M.P.: Woflan 2.0 A petri-net-based workflow diagnosis tool. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 475–484. Springer, Heidelberg (2000)

5. Wolf, K.: Generating petri net state spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)
6. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through SESE decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
7. Prinz, T.M., Amme, W.: Practical compiler-based user support during the development of business processes. Technical Report Math/Inf/02/13. (June 2013), `http://www.bpmn-compiler.org`
8. Pahl, P.J., Damrath, R.: Mathematical Foundations of Computational Engineering: A Handbook, 1st edn. Springer, Heidelberg (2001)
9. Prinz, T.M., Spieß, N., Amme, W.: A first step towards a compiler for business processes. In: Cohen, A. (ed.) CC 2014 (ETAPS). LNCS, vol. 8409, pp. 238–243. Springer, Heidelberg (to be published, 2014)
10. 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: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
11. Perumal, S., Mahanti, A.: A graph-search based algorithm for verifying workflow graphs. In: 2012 23rd International Workshop on Database and Expert Systems Applications, pp. 992–996 (2005)
12. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 100–115. Springer, Heidelberg (2008)
13. 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), 47–59 (2009)
14. Heinze, T.S., Amme, W., Moser, S.: A restructuring method for WS-BPEL business processes based on extended workflow graphs. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 211–228. Springer, Heidelberg (2009)