Fast soundness verification of workflow graphs

Thomas M. Prinz
Friedrich Schiller University Jena, Germany

Abstract. This paper shows a new approach to check the soundness of workflow graphs. The algorithm is complete and allows a very good localization of the structural conflicts, i.e. local deadlocks and lack of synchronizations. The evaluation shows a linear processing time in the average and an up to quadratic in the worst case.

Keywords: Soundness, Workflow Graphs, Deadlock, Lack of Synchronization, Localization

1 Introduction

Creating a business process is accompanied by control-flow errors, which is shown by various studies [1, 2]. These errors evoke wrong or unexpected results and restrict the correct simulation and execution of business processes [1].

We deal with business processes as workflow graphs [3–5]. Formally, a workflow graph is a directed graph WG = (N, E) at which N consists of activities, fork, join, split, merge nodes and one start node as well as one end node. (1) An activity has exactly one incoming and exactly one outgoing edge, (2) a fork or split node has exactly one incoming and at least two outgoing edges, (3) a join or merge node has at least two incoming edges and exactly one outgoing edge, and (4) each node n ∈ N lies on a path from the start to the end node. Figure 1 illustrates an example of a workflow graph. We use the notion •n to describe all predecessor nodes and n• to describe all successor nodes of a node n.

We call a workflow graph simple if and only if the predecessors and successors of each fork, join, split, merge, start or end node are activities, e.g., the workflow graph of Figure 1 is simple. Workflow graphs observed by this paper are simple, contain one start and one end node as well as cycles. All splits and merges have an XOR semantic and all forks and joins have an AND semantic. The restriction of simple workflow graphs is not a limitation, because a transformation of a common workflow graph to a simple one is possible in O(E) (see appendix). The execution of such a workflow graph starts in an initial state that means the outgoing edge of the start node owns one token.

As structural correctness criterion of workflow graphs, the classical notion of soundness [4] is used in this paper. This notion was introduced on workflow nets. [1] has shown that a workflow graph is a so called free-choice workflow net and the soundness of such a graph corresponds with the absence of local deadlocks (for short LD) and lack of synchronizations (for short LoS) [3, 1]. To introduce
both error types we take a look at the simple example of Figure 1 taken from [1] that includes a LoS and a LD.

If a token travels the true edge leaving the split $D_1$ in our example, then the token will reach the join $J$ via the upper incoming edge. However, there is no other token that will ever arrive at the lower incoming edge of $J$. Such a reachable state is called local deadlock. Formally, a local deadlock is a reachable state $s$ of the workflow graph that has a token on an incoming edge $e$ of a join node such that each reachable state from $s$ also contains a token on $e$ [1].

When a token reaches the false edge leaving the split $D_1$, the token enables the firing of fork $F$. So one token reaches $M_2$ and it is possible that the other token will firing the cycle $D_2, M_1, D_1, F$. The result is a possible state with multiple tokens on the incoming edge of $M_2$. Such a reachable state is called lack of synchronization. Formally, a lack of synchronization is a reachable state $s$ containing an edge that has more than one token [1].

The analysis of the absence of LoSs and LDs is well established by various studies (see [3–6] for example). However, the processing time of the current approaches is at least cubic or the workflow graphs are critically restricted (acyclic, incomplete or well-formed). The diagnostic information also ranges from no information (fast algorithms) over displaying the failure trace (Petri net based techniques) to detailed information (for restricted workflow graphs). A good overview over such techniques gives the introduction of [6]. Therefore, it exists a divergence of fast or informative algorithms.

The remainder of this paper is structured as follows: Sect. 2 introduces the basic ideas and marginal cases of our new approach. These ideas will be complemented in Sect. 3 following an evaluation of our approach in Sect. 4. Finally, Sect. 5 concludes the paper.

### 2 Basic idea and marginal cases

Our basic idea is to detect LD and LoS structurally by finding entry points which control the firing of a join node. Then we assume a state allowing the firing of such an entry point. Further, we find each LD and LoS isolated. That means, such an error can not occur if a LD or LoS occurs before. These isolated LDs and LoSs are called potential. The task is to determine entry points to detect potential LDs and LoSs.
Therefore, we introduced so-called *bottle necks*, which are fork or split nodes, e.g., the nodes $D_1$, $F$ and $D_2$ of Figure 1. A *bottle* of a bottle neck $n_b$ is a subgraph $G_{n_b} = (N_{n_b}, E_{n_b})$ of the workflow graph such that each path from the start node to a node $n \in N_{n_b}$ contains $n_b$ or a merge node $n_m \in N_{n_b}$. From this it follows that, $N_{n_b}$ contains only a join node $n_j$ if $\bullet n_j \subseteq N_{n_b}$ is valid. If the bottle $n_b$ contains a merge node $n_m$ that has a predecessor node $n_p$ being not in $N_{n_b}$, then no state should be reachable from the initial state which has a token on the incoming edge of $n_b$ and on the edge $(n_p; n_m)$ (see Figure 2). A LoS is reachable having two tokens on the outgoing edge of $n_m$.

For example the bottle of $D_1$ of Figure 1 is the sub graph containing all nodes except $S$ and $A_1$ and the bottle of $F_2$ of Figure 3 contains the nodes $A_3$ and $A_4$.

![Diagram](image1.png)  
**Fig. 2:** A bottle that contains a merge node  

![Diagram](image2.png)  
**Fig. 3:** A sub graph of a workflow graph

After all, an *entry point* of a join node $n_j$ is a bottle neck with a bottle containing $n_j$, e.g., the entry points of $J$ of Figure 1 are $D_1$, $F$ and $D_2$ and an entry point of $J$ of Figure 3 is $F_1$.

If a join node $n_j$ has no entry point, then there exists no bottle neck, which has a bottle containing all predecessors of $n_j$. That means at least one predecessor of $n_j$ is only reachable from the start node when $n_j$ fires. A LD is reachable.

**Lemma 1.** *If a join node has no entry point, a potential local deadlock occurs.*

An entry point of a join node is named *immediate* if and only if there exists at least one path from the entry point of the join node containing no other entry point of the join. If we determine all immediate entry points of a join node, then the join node is only reachable over these nodes. So all entry points of $J$ in our example of Figure 1 are immediate, e.g., there is a path $(A_5, M_2, A_4)$ from $F$ to $J$ with no entry point.

Now, we consider an immediate entry point of a join node $n_j$ being a split node $n_s$. There is a path from $n_s$ to $n_j$ containing no other immediate entry points. So if no LD occurs on this path, it is possible that after firing $n_s$ a token reaches exactly one incoming edge of $n_j$. $n_j$ cannot fire. If we assume that, at this point the entry point or another entry point fires, a token can reach the same incoming edge of $n_j$ over again. So either a LD occurs or a LoS is possible.
Lemma 2. If a join node has at least one split node as immediate entry point, a potential local deadlock is reachable.

We call an entry point of a join node $n_j$ complete if and only if no path from the entry point to $n_j$ contains an other entry point of $n_j$. We assume that, $n_j$ has a non-split immediate entry point $n_e$ which is not complete. We know, there is a path $P_1$ from $n_e$ to $n_j$ which contains no other immediate entry point. Let us assume that, $P_1$ contains the edge $e = (x, n_j)$. We know, there is also a path $P_2$ from $n_e$ to $n_j$ containing another immediate entry point $n_i$. Because $n_i$ is also an entry point of $n_j$, there is a path $P_3$ from $n_i$ to $n_j$ containing also the edge $e$. If $n_e$ fires and there is no LD on the observed paths, then a token reaches $e$ and the incoming edge of $n_i$. $n_i$ can also fire and a token reaches also $e$. A LoS occurs. This situation is only avoidable if a LD occurs.

Lemma 3. If a join node has at least one immediate entry point being not complete and not a split node, a potential lack of synchronization is reachable.

Definition 1 (Well-formed). The immediate entry points of a join node are called well-formed if and only if none of the lemmata 1, 2 and 3 is valid.

3 Processes

Let us consider all paths from all immediate entry points to exactly one predecessor of a join node $n_j$ (and so to an incoming edge) which contains no entry point of $n_j$. We merge all these paths to a sub graph $P = (N_P, E_P)$ called process of $n_j$, because (related to a business process) such a process can be executed from one entity (like a human or machine). $n_j$ is named terminator $T(P)$ of the process, because this node ends the execution of the process. Supplementary, we define a main process for generalization. A main process is the process containing all nodes of a workflow graph except the start and end node. For example, the processes in figure 1 are $\{A_3, M_2, A_4, A_5\}$ and $\{A_7, D_2, A_9\}$ and the main process is $N \setminus \{S, E\}$. At this point, we need the simpleness of a workflow graph, because every process should have at least one node.

Hence, we observe a join node $n_j$ having only well-formed immediate entry points. If we assume that on the join node $n_j$ with its complete entry points $N_{EP}$ occurs a LD, then we conclude that at least one process $P$ of $n_j$ contains a split node $n_s$ with $n_s \cdot \notin N_P$. On a LD is at least one token on an incoming edge and at least no token on another. We assume $P$ is the process which contains the predecessor node $n_p$ of $n_j$ and $(n_p, n_j)$ bears no token. One entry point $n_e \in N_{EP}$ has to been fired. There exists a path from $n_e$ to $n_p$. So if we assume there is no LD on this path, a token can travel from the outgoing edge of $n_e$ to the outgoing edge of $n_p$. How could a token leave this path? The only way is a split node which lies on the path and has a successor node with no path from this node to $n_p$. That means this successor node is not in the process. So our assertion is correct.
Lemma 4. If a process $P = (N_P, E_P)$ contains a split node $n_s$ with $n_s \cdot \not\subseteq N_P$ a potential local deadlock is reachable.

For example the process $\{A_7, D_2, A_9\}$ contains the split node $D_2$ which satisfies the conditions to cause a potential local deadlock.

Processes have nice properties on closer inspection. If a process $P$ contains a join node $n_j$, each process $P'$ which ends in $n_j$ is a sub graph of $P$ called sub process (written $P' \subset P$) and $P$ is called super process respectively. This results on the definitions of processes and entry points. Furthermore the process $P$ is called active on a node $n$ if there is no sub process of $P$ containing $n$. We called it active, because an entity processes $P$ actively regarding a business process engine, while passive processes have to wait.

Each LoS starts in a fork node which produces two tokens joining the same edge later. So the detection of LoSs is the detection of processes which are active on the successor nodes twice or more. Because of the definition of an active process, there is no join node on the path to its terminator splitting the process. It has to be a merge node. This results in a LoS (see Figure 4 and 7). But an active process could be hidden by another active process (see Figure 5 and 6). If a sub process $P'$ of a process $P$ starts in a fork node $n_f$, we can assume a (maybe infinitesimal) short time between the firing of $n_f$ and the activation of $P'$, in that $P$ is also active. We name these hidden active processes simply hidden active. Altogether we can express the following lemma.

![Fig. 4: Case 1](image1.png) ![Fig. 5: Case 2](image2.png) ![Fig. 6: Case 3](image3.png) ![Fig. 7: Case 4](image4.png)

Lemma 5. Let $M$ be a multi set containing all active and hidden active processes of all successors of a fork node. If $M$ contains a process twice or more, then a potential lack of synchronization is reachable.

For example the fork node $F$ in our example contains the active main process twice in its successor nodes (once active and once hidden active).

Now we have determined all potential LDs and LoSs by handling marginal cases and processes and can exactly localize them by the entry points, the terminators, the processes and split and fork nodes to visualize them. Because a potential LD is not only reachable from the initial state if another potential LD or LoS occurs, and a potential LoS is not only reachable if a potential LD occurs, we can determine the soundness of a workflow graph. Altogether we can formalize the following lemma.

Theorem 1. A workflow graph is sound if and only if it contains neither a potential local deadlock nor a potential local deadlock.
The basic algorithms to determine the entry points, immediate entry points and processes are in the appendix A. It is reasonable that $|E| \leq 2 \times |N|$ is valid for simple workflow graphs. Altogether, the processing time is in worst case quadratic to $N$.

4 Evaluation

We have implemented the algorithm in Java and stopped the soundness verification if the algorithm found the first potential error. It followed the approach of [1]. The benchmark was taken from http://www.service-technology.org/soundness, is splitted in 5 libraries (A, B1, B2, B3 and C) and was also used by [1]. The input was a PNML file which was parsed and transformed from a Petri net to a workflow graph. So we could compare it with other tools like LoLA (http://www.informatik.uni-rostock.de/tpp/lola/).

Our runtime environment was a system with a 64 bit Intel® Core™2 CPU E6300 processor and 2 GB main memory running a Linux 3.1.0-1.2-desktop x86_64 kernel and an Oracle OpenJDK JRE 1.6.0_22. Because of the hot spot compiler of the JRE, we created a startup as long as the optimizing 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. Furthermore the number of nodes, edges and explored nodes were determined. We chose LoLA for comparison and ran them on the same machine like our algorithm and subtracted the read and build time of the petri net. Altogether we determined only the validation time and no transformations. Table 1 shows the results.

<table>
<thead>
<tr>
<th>Library:</th>
<th>A</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes/Sound</td>
<td>282/152</td>
<td>288/107</td>
<td>363/161</td>
<td>421/207</td>
<td>32/15</td>
</tr>
<tr>
<td>Avg./Max. $</td>
<td>N</td>
<td>$</td>
<td>84/292</td>
<td>83/437</td>
<td>93/501</td>
</tr>
<tr>
<td>Avg./Max. $</td>
<td>E</td>
<td>\setminus</td>
<td>N</td>
<td>$</td>
<td>1.3/1.9</td>
</tr>
<tr>
<td>Avg./Max. $</td>
<td>\text{vis.} N</td>
<td>\setminus</td>
<td>N</td>
<td>$</td>
<td>3.6/7.1</td>
</tr>
<tr>
<td>Analysis time [ms]</td>
<td>16.4</td>
<td>15.4</td>
<td>20.7</td>
<td>28.4</td>
<td>1.7</td>
</tr>
<tr>
<td>Analysis time LoLA [ms]</td>
<td>2373.0</td>
<td>2395.9</td>
<td>3126.1</td>
<td>3651.3</td>
<td>303.8</td>
</tr>
<tr>
<td>Per process avg./max. [ms]</td>
<td>0.06/0.28</td>
<td>0.06/0.36</td>
<td>0.07/0.69</td>
<td>0.06/0.31</td>
<td></td>
</tr>
<tr>
<td>Per process LoLA avg. [ms]</td>
<td>8.5</td>
<td>8.4</td>
<td>8.7</td>
<td>8.7</td>
<td>9.5</td>
</tr>
</tbody>
</table>

Table 1: Results of the benchmark evaluation

Compared to the results of LoLA we determined the same sound and unsound processes. So it accompanies with the correctness of our approach.

The results showed the number of edges grow linear with the number of nodes and that the number of split, fork and join nodes is less ($< 20\%$). It accompanies with the nearly linear number of max. approx. 11-times inspections of nodes. So the runtime results showed that our approach is nearly linear in the worst case. This is also represented by the analysis times of the libraries. They were approx. 150-times faster as the times of LoLA in average although the comparison was
difficult regarding the different implementation technologies. Altogether a single process took always less than one millisecond to be validated.

To verify these results we added our complete algorithm to the Activiti BPMN 2.0 designer (http://activiti.org). When drawing a workflow graph-like business process, the plugin verifies the process in-time and visualizes all errors. It underscores the efficiency of our algorithm and the very good error localisation.

5 Conclusion and Outlook

In this paper we presented and evaluated a new approach to detect structural conflicts in workflow graphs, i.e. the soundness. The approach and the resulting algorithm found all structural conflicts and was able to localize them. We showed a soundness verification is possible during the drawing of a business process. This is possible, because the algorithm has an up to quadratic average run time.

The main issue for future work are to present a complete proof of our approach, to use the introduced processes to verify the operability of workflow graphs and to transform data-extended workflow graphs into a CSSA-based form.

Acknowledgments

This paper was written in the context of the SimProgno research project (support code: 01IS10042B) funded by the German Federal Ministry of Education and Research. Special thanks to Wolfram Amme for discussing the ideas of this paper.

References

A Appendix

Algorithm 1 Transforming a workflow graph into a simple one with processing time: $O(E)$.

**Input:** A workflow graph $WG = (N, E)$

**Output:** A simple workflow graph $WG' = (N', E')$

1: for all $e = (n_s, n_t) \in E$ do
2: $N' \leftarrow N' \cup \{n_s, n_t\}$
3: if $n_s$ and $n_t$ are not activities then
4: $N' \leftarrow N' \cup \{n_a\}$, $n_a$ is a new activity
5: $E' \leftarrow E' \cup \{(n_s, n_a), (n_a, n_t)\}$
6: else
7: $E' \leftarrow E' \cup \{e\}$

Algorithm 2 Determining a bottle for one bottle neck with processing time: $O(N + E)$.

**Input:** A fork or split node $n$ of a workflow graph $WG = (N, E)$

**Output:** A set $N_B$ of nodes of the bottle of $n$

1: for all $n_s \in n \bullet$ do
2: determineBottle($n_s$)

3: function determineBottle($n_c$)
4: if $n_c \notin N_B$ then
5: if $n_c \in N_e$ then
6: $N_B \leftarrow N_B \cup \{n_c\}$ and mark $n$ as entry point of $n_c$
7: for all $n_s \in n_c \bullet$ do
8: determineBottle($n_s$, $N_B$)
9: else
10: $N_B \leftarrow N_B \cup \{n_c\}$
11: for all $n_s \in n_c \bullet$ do
12: determineBottle($n_s$, $N_B$)

Algorithm 3 Determining a subset of immediate entry points and one process of a join node respectively with processing time: $O(N + E)$.

**Input:** Predecessor node $n_p$ of a join node $n_j$ and the entry points $N_e$ of $n_j$

**Output:** Process $P = (N_P, E_P)$ and a subset of immediate entry points $N_i$ of $n_j$

1: determineProcess($n_p$)
2: function determineProcess($n_c$)
3: if $n_c \notin N_P$ then
4: if $n_c \in N_e$ then
5: $N_i \leftarrow N_i \cup \{n_c\}$
6: else
7: $N_P \leftarrow N_P \cup \{n_c\}$
8: for all $n_p \in n_c \bullet$ do
9: determineProcess($n_p$)