Fast soundness verification of workflow graphs

Thomas M. Prinz

Friedrich Schiller University Jena, Germany
Thomas.Prinz@uni-jena.de

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 restricts the correct simulation and execution of business processes [1].

We deal with business processes as workflow graphs [3,4,5]. 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 $n \in N$ lies on a path from the start to the end node. Figure 1 illustrates an example of a workflow graph. We use the notion $\bullet n$ to describe all predecessor nodes and $n\bullet$ 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* and could have 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*

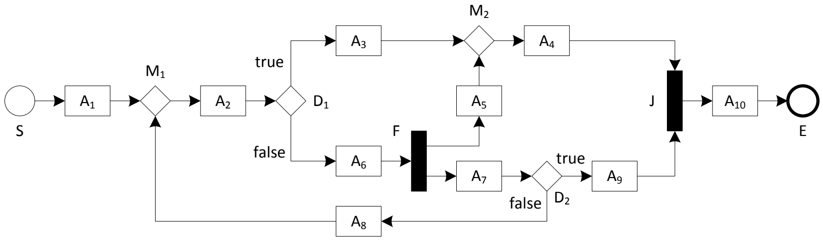

Fig. 1. A workflow graph containing a local deadlock and lack of synchronization

deadlocks and *lack of synchronizations* [3,1]. To introduce both error types, we take a look at the simple example of Figure 1 taken from [1] that includes a lack of synchronization and a local deadlock.

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 lack of synchronizations and local deadlocks is well established by various studies (see [3,4,5,6] 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 structered 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 local deadlocks and lack of synchronizations 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 local deadlock and lack of synchronization *isolated.* That means, such an error can not occur if a local deadlock or lack of synchronization occurs before. These isolated local deadlocks and lack of synchronizations are called *potential.* The task is to determine entry points to detect *potential LD*s and lack of synchronizations.

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 beeing 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 lack of synchronization 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.

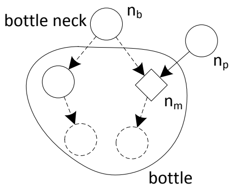

Fig. 2. A bottle that contains a merge node

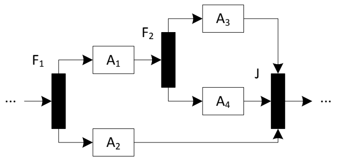

Fig. 3. A sub graph of a workflow graph

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 local deadlock 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 point of n_j. So if no local deadlock 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 local deadlock occurs or a lack of synchronization 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 local deadlock 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 lack of synchronization occurs. This situation is only avoidable if a local deadlock 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 local deadlock, then we conclude that at least one process P of n_j contains a split node n_s with $n_s\bullet \not\subseteq N_P$. On a local deadlock 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 local deadlock 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 $n_s \bullet \nsubseteq N_P$ 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 to a business process engine, while *passive* processes have to wait.

Each lack of synchronization starts in a fork node which produces two tokens joining the same edge later. So the detection of lack of synchronizations 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 lack of synchronization (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.

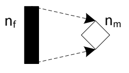

Fig. 4. Case 1

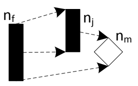

Fig. 5. Case 2

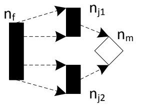

Fig. 6. Case 3

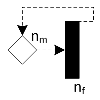

Fig. 7. Case 4

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 local deadlocks and lack of synchronizations by handling marginal cases and processes. Thus, it is possible to exactly localize them by the entry points, the terminators, the processes and split and fork nodes. Because of the fact, that a potential local deadlock is not only reachable from the initial state when another potential local deadlock or lack of synchronization occurs, and a potential lack of synchronization is not only reachable when a potential local deadlock 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 lack of synchronization.*

The basic algorithms to determine the entry points, immediate entry points and processes are in the appendix. It is reasonable that $|E| \leq 2 * |N|$ 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 when 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$^{\text{TM}}$2 CPU E6300 processor and 2 GB main memory. The system rans 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 it 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.

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 grows 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,

Library:	A	B1	B2	B3	C
Processes/Sound	282/152	288/107	363/161	421/207	32/15
Avg./Max. $\|N\|$	84/292	82/402	83/437	93/501	136/567
Avg./Max. $\|E\| \setminus \|N\|$	1.3/1.9	1.3/1.9	1.3/1.9	1.2/1.7	1.1/1.3
Avg./Max. $\|vis.N\| \setminus \|N\|$	3.6/7.1	3.4/7.9	3.5/10.8	3.4/7.7	2.4/7.1
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

Table 1. Results of the benchmark evaluation

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 efficiancy 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 were 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 worst case processing time.

The main issues for future work are to present a complete proof of our approach, solving the soundness for OR-splits and OR-merges, 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

1. 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
2. Mendling, J.: Empirical studies in process model verification. In Jensen, K., Aalst, W.M., eds.: Transactions on Petri Nets and Other Models of Concurrency II, Berlin, Heidelberg, Springer-Verlag (2009) 208–224

3. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst. **25**(2) (April 2000) 117–134
4. 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
5. Eshuis, R., Kumar, A.: An integer programming based approach for verification and diagnosis of workflows. Data Knowl. Eng. **69**(8) (August 2010) 816–835
6. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and more focused control-flow analysis for business process models through sese decomposition. In: Proceedings of the 5th international conference on Service-Oriented Computing. ICSOC '07, Berlin, Heidelberg, Springer-Verlag (2007) 43–55

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 $WG' = (N', E')$

```
for all $e = (n_s, n_t) \in E$ do
    $N' \leftarrow N' \cup \{n_s, n_e\}$
    if $n_s$ and $n_t$ are not activities then
        $N' \leftarrow N' \cup \{n_a\}, n_a$ is a new activity
        $E' \leftarrow E' \cup \{(n_s, n_a), (n_a, n_t)\}$
    else
        $E' \leftarrow E' \cup \{e\}$
```

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

```
for all $n_s \in n\bullet$ do
    determineBottle($n_s$)
function DETERMINEBOTTLE($n_c$)
    if $n_c \notin N_b$ then
        if $n_c$ is a join node then
            if $\bullet n_c \subseteq N_B$ then
                $N_B \leftarrow N_B \cup \{n_c\}$ and mark $n$ as entry point of $n_c$
                for all $n_s \in n_c\bullet$ do
                    determineBottle($n_s$, $N_B$)
        else
            $N_B \leftarrow N_B \cup \{n_c\}$
            for all $n_s \in n_c\bullet$ do
                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

```
determineProcess($n_p$)
function DETERMINEPROCESS($n_c$)
    if $n_c \notin N_P$ then
        if $n_c \in N_e$ then
            $N_i \leftarrow N_i \cup \{n_c\}$
        else
            $N_P \leftarrow N_P \cup \{n_c\}$
            for all $n_p \in \bullet n_c$ do
                determineProcess($n_p$)
```