An efficient approach to detect lack of synchronization in acyclic workflow graphs

Cédric Favre

IBM Research — Zurich,
ced@zurich.ibm.com

Abstract. Control-flow analysis of business process models requires to check the absence of lack of synchronization. We use workflow graphs, which may contain inclusive OR gateways, to represent the control-flow of business process models. We structurally characterize lack of synchronization in an acyclic workflow graph. Based on this characterization, we show how to detect lack of synchronization in quadratic time.

1 Introduction

A business process model can be simulated, used for automated code generation, or directly executed by a workflow engine. With the increasing popularity of these use cases, ensuring that a process is free of control-flow errors is essential. A process is *sound* [1] if and only if it is free of two control-flow errors: the *deadlock* and the *lack of synchronization*. The control-flow of a process can be modeled by a workflow graph or a workflow net [1]. We use workflow graphs with the inclusive OR (IOR) gateways.

In this paper, we show how to efficiently check the absence of lack of synchronization in acyclic workflow graphs with IOR gateways. In Section 3, we show that a lack of synchronization in acyclic workflow graphs with IOR gateways can be fully characterized in term of *a path with a handle*. In Section 4, we show how to detect a path with a handle by using an extension of the approach of Perl and Shiloach [2]. The time and space complexity of this approach is in $O(|N| \cdot |E|)$, where $|N|$ is the number of nodes and $|E|$ the number of edges of the workflow graph.

Related work: In case a workflow graph does not contain IOR gateways, it can be easily converted into a Petri net and vice versa [1]. However, it is not clear if a workflow graph that contains IOR gateways can be converted into a Petri net. Our notion of handles is similar to the one of Esparza and Silva [3] for nets. If we restrict to workflow graphs without IOR gateways, one of the directions of our characterization follows from a result of Esparza and Silva [3]. The converse direction does not directly follow. Our notion of handles has been described by Aalst [4] who shows that, given a Petri net N, the absence of some type of path

with handle in N is a sufficient condition to the existence of an initial marking i of N such that (N, i) is sound. He points out that path with handles can be computed using a maximum flow approach. Various algorithms exist to compute the maximum flow (see [5] for a list). The complexity of these algorithms ranges between $O(|N| \cdot |E|^2)$ and $O(|N| \cdot |E| \cdot log(|N|))$. The existence of a handle can be checked by applying a maximum flow algorithm to each pair of transition and place of the net. Therefore, the complexity of detecting handles with such approach is at best $O(|N|^3 \cdot |E| \cdot log(|N|))$. Moreover, algorithms to determine a maximum flow are significantly more complex to implement than our approach, especially the ones with the lowest complexity.

2 Preliminaries

We use number of known concepts from set theory, graph theory, and the workflow graph verification which are defined below:

Let U be a set. A *multi-set* over U is a mapping $m : U \to \mathbb{N}$. We write $m[e]$ instead of $m(e)$. For two multi-sets m_1, m_2, and each $x \in U$, we have : $(m_1 + m_2)[x] = m_1[x] + m_2[x]$ and $(m_1 - m_2)[x] = m_1[x] - m_2[x]$. By abuse of notation, we sometimes use a set $X \subseteq U$ in a multi-set context by setting $X[x] = 1$ iff $x \in X$ and $X[x] = 0$ otherwise.

A *directed graph* $G = (N, E)$ consists of a set N of *nodes* and a set E of ordered pairs (s, t) of nodes, written $s \to t$. A *directed multi-graph* $G = (N, E, c)$ consists of a set N of nodes, a set E of *edges* and a mapping $c : E \to (N \cup \{\text{null}\}) \times (N \cup \{\text{null}\})$ that maps each edge to an ordered pair of nodes or null values. If c maps $e \in E$ to an ordered pair $(s, t) \in N$, then s is called the *source* of e, t is called the *target* of e, e is an *outgoing* edge of s, and e is an *incoming* edge of t. If $s = \text{null}$, then we say that e *has no source*. If $t = \text{null}$, then we say that e *has no target*. For a node $n \in N$, the set of incoming edges of n is denoted by $\circ n$. The set of outgoing edges of n is denoted $n\circ$. If n has only one incoming edge e, ${}^\circ n$ denotes e ($\circ n$ would denote $\{e\}$). If n has only one outgoing edge e', n° denotes e'.

A *path* $p = \langle x_0, ..., x_n \rangle$ *from* an element x_0 *to* an element x_n *in* a graph $G = (N, E, c)$ is an alternating sequence of elements x_i in N and in E such that, for any element $x_i \in E$ with $c(x_i) = (s_i, t_i)$, if $i \neq 0$ then $s_i = x_{i-1}$ and if $i \neq n$ then $t_i = x_{i+1}$. If x is an element of a path p we say that p *contains* x. A path is *trivial*, when it is contains only one element. A *cycle* is a path $b = \langle x_0 \dots x_n \rangle$ such that $x_0 = x_n$ and b is not trivial. If there exists a path from an element x_1 to an element x_2 of a graph, we say that x_1 *precedes* x_2, denoted $x_1 < x_2$.

A *workflow graph* $W = (N, E, c, l)$ consists of a multi-graph $G = (N, E, c)$ and a mapping $l : N \to \{AND, XOR, IOR\}$ that associates a *logic* with every node $n \in N$, such that: 1. An edge with null as source is a *source* of the workflow graph and an edge with null as target is a *sink* of the workflow graph. 2. The workflow graph has one source and at least one sink. 3. For each node $n \in N$,

there exists a path from the source to one of the sinks that contains n. W is *cyclic* if there exists a cycle in W.

Figure 1 depicts an acyclic workflow graph. A diamond containing a plus symbol represents a node with AND logic, an empty diamond represents a node with XOR logic, and a diamond with a circle inside represents a node with IOR logic. A node with a single incoming edge and multiple outgoing edges is called a *split*. A node with multiple incoming edges and single outgoing edge is called a *join*. For the sake of simplicity, we use workflow graphs composed of only splits and joins. This syntactic restriction does not reduce the expressiveness of workflow graphs. We usually label the source of the workflow graph s and use workflow graphs with a unique sink labeled t.

Let, in the rest of this section, $W = (N, E, c, l)$ be an acyclic workflow graph.

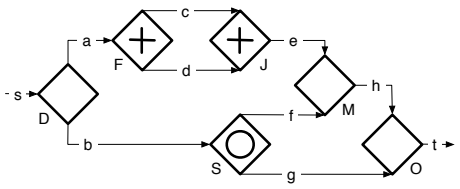

Fig. 1. A workflow graph .

The semantics of workflow graphs is, similarly to Petri nets, defined as a token game. If n has AND logic, executing n removes one token from each of the incoming edges of n and adds one token to each of the outgoing edges of n. If n has XOR logic, executing n removes one token from one of the incoming edges of n and adds one token to one of the outgoing edges of n. If n has IOR logic, n can be executed if and only if at least one of its incoming edges is marked and there is no marked edge that precedes a non marked incoming edge of n. When n executes, it removes one token from each of its marked incoming edge and adds one token to a non-empty subset of its outgoing edges. The outgoing edge or set of outgoing edges to which a token is added when executing a node with XOR or IOR logic is non-deterministic, by which we abstract from data-based or event-based decisions in the process model. In the following, this semantics is defined formally.

A *marking*, $m : E \to \mathbb{N}$, of a workflow graph with edges E is a multi-set over E. When $m[e] = k$, we say that the edge e *is marked with* k *tokens* in m. When $m[e] > 0$, we say that the edge e *is marked*. The *initial marking* m_s of W is such that the source edge s is marked by one token in m_s and no edge, other than s, is marked in m_s.

Let m and m' be two markings of W. A tuple (E_1, n, E_2) is called *transition* if $n \in N$, $E_1 \subseteq {\circ}n$, and $E_2 \subseteq n{\circ}$. A transition (E_1, n, E_2) is *enabled* in a marking m iff for each edge $e \in E_1$ we have $m[e] > 0$ and any of the following proposition:

- $l(n) = AND$, $E_1 = {\circ}n$, and $E_2 = n{\circ}$.
- $l(n) = XOR$, there exists an edge e such that $E_1 = \{e\}$, and there exists an edge e' such that $E_2 = \{e'\}$.

– $l(n) = IOR$, E_1 and E_2 are not empty, $E_1 = \{e \in {}^\circ n \mid m(e) > 0\}$, and, for every edge $e \in {}^\circ n \setminus E_1$, there exists no edge e', marked in m, such that $e' < e$.

A transition T can be *executed* in a marking m iff T is enabled in m. When T is executed in m, a marking m' results such that $m' = m - E_1 + E_2$.

An *execution sequence* of W is an alternate sequence $\sigma = \langle m_0, T_0, m_1, T_1 ... \rangle$ of markings m_i of W and transitions $T_i = (E_i, n_i, E_i')$ such that, for each $i \geq 0$, T_i is enabled in m_i and m_{i+1} results from the execution of T_i in m_i. An *execution* of W is an execution sequence $\sigma = \langle m_0, ..., m_n \rangle$ of W such that $n > 0$, $m_0 = m_s$ and there is no transition enabled in m_n. As the transition between two markings can be easily deduced, we often omit the transitions when representing an execution or an execution sequence, i.e., we write them as sequence of markings.

Let m be a marking of W, m is *reachable from* a marking m' of W iff there exists an execution sequence $\sigma = \langle m_0, ..., m_n \rangle$ of W such that $m_0 = m'$ and $m = m_n$. The marking m is a *reachable marking* of W iff m is reachable from m_s.

A *lack of synchronization* is a reachable marking m of W such that there exists an edge $e \in E$ that carries more than one token in m.

3 Handles and Lack of Synchronization

To characterize the lack of synchronization, we follow the intuition that potentially concurrent paths, i.e., paths starting with an IOR-split or an AND-split, should not be joined by XOR-join. In the following, we formalize this characterization and show that such structure always leads to lack of synchronization in deadlocks free acyclic workflow graphs.

Definition 1 (Path with a AND-XOR or a IOR-XOR handle). *Let $p_1 = \langle n_0, ..., n_i \rangle$ and $p_2 = \langle n_0', ..., n_j' \rangle$ be two paths in a workflow graph $W = (N, E, c, l)$.*

The paths p_1 and p_2 form a path with a handle[1] *iff p_1 is not trivial, $p_1 \cap p_2 = \{n_0, n_i\}$, $n_0 = n_0'$, and $n_i = n_j'$. We say that p_1 and p_2 form a path with an handle* from n_0 to n_i. *The paths p_1 and p_2 form a* path with a AND-XOR handle *iff they form a path with a handle, n_0 is an AND-split, and n_i is an XOR-join. The paths p_1 and p_2 form a* path with a IOR-XOR handle *iff they form a path with a handle, n_0 is an IOR-split, and n_i is an XOR-join. In the rest of this document, we use handle instead of path with a AND-XOR handle or path with a IOR-XOR handle. The node n_0 is the* start node of the handle *and the node n_i is the* end node of the handle.

[1] Strictly speaking, one path is the handle of the other path and *vice versa*.

Theorem 1. *In an acyclic workflow graph that contains no deadlock, there is a lack of synchronization iff there is a handle.*

The outline of the only if direction of the proof of Theorem 1 is that, whenever there is a handle, this handle can be 'executed' in the sense that there exists an execution such that a token reaches the incoming edge of the start node of the handle and then two tokens can be propagated to reach two incoming edges of the end node of the handle to create a lack of synchronization. We believe that, due to its direct relationship with an erroneous execution, the handle is an adequate error message for the process modeler. Note that it is necessary that the workflow graph is deadlocks free in order to show that the handle can be executed and thus a lack of synchronization be observed. However, even if the workflow graph contains a deadlock, a handle is a design error because, once the deadlock is fixed, the handle can be executed and a lack of synchronization can be observed.

4 Handle Detection

Given an acyclic directed graph $G = (N, E)$ and four different nodes $s_1, s_2, t_1, t_2 \in N$, Perl and Shiloach [2] show how to detect two node disjoint paths from s_1 to t_1 and from s_2 to t_2 in $O(|N| \cdot |E|)$. We extend their algorithm in order to detect two edge disjoint paths between two nodes of an acyclic workflow graph.

Our intuitive view of the approach proposed by Perl and Shiloach [2] is to try to move two tokens from a start node to an end node of a handle without allowing the two tokens to visit the same edge, i.e., without allowing the tokens to follow paths that overlap. To achieve this, we build a so called *state graph* which records the state space of the possible combinations of marked edges and a transition relation. Note that the size of the state graph is quadratic with respect to the number of edges of the original graph because we only consider combinations of two edges. Checking the existence of a handle in the workflow graph reduces to check the existence of a special path in the state graph. We ensure that the special path in the state graph does not allow the two tokens to visit the same edge in two ways: 1. We do not represent the states where two tokens are on the same edge. 2. We restrict the transition relation in a way that, if two tokens visit paths that overlap, they visit the overlapping edge synchronously (which is ruled out by 1). This synchronization is achieved by only allowing the token that is the most upstream in the graph to move.

In the remainder of the this section, we describe how to obtain a *line graph*: a directed graph in which the edges are represented as nodes. Based on the line graph and a numbering allowing us to detect which edge is upstream from another, we show how to build the state graph. Finally, we describe how to detect handles using the state graph.

Computing a line graph: Perl and Shiloach [2] describe how to detect two node disjoint paths in a directed graph whereas we want to detect two edge disjoint paths in a workflow graph; a directed multi-graph. To do so, we transform the workflow graph into its *line graph* [6]. A line graph G' of a graph G represents the adjacency between edges of G. The nodes of G' are the edges of G.

For the purpose of checking handles, there are two types of nodes that are of interest: 1. The start nodes of a possible handle $S = \{x \mid x \in N \wedge x$ is an AND-split or an IOR-split$\}$. 2. The end nodes of a possible handle $T = \{x \mid x \in N \wedge$ x is an XOR-join$\}$. We include these nodes in the line graph.

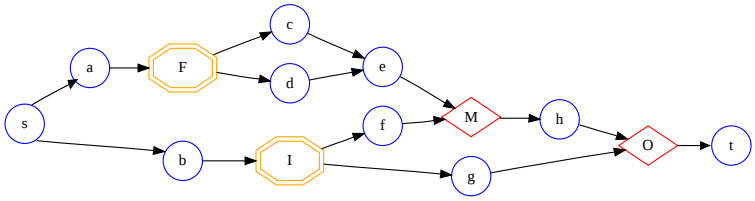

Fig. 2. The line graph of Figure 1.

In the following, for any edge e of a directed graph $t(e)$ references the target node of e and $s(e)$ references the source node of e.

Definition 2 (Line graph). *Let $W = (N, E, c, l)$ be an acyclic workflow graph. The* line graph $L = (N', E')$ *of W is a directed graph such that:*

$$N' = E \cup S \cup T$$
$$E' = \{a \to b \mid a, b \in E \wedge t(a) = s(b)\}$$
$$\cup \{a \to b \mid a, b \in N' \wedge (\ (a \in S \cup T \wedge a = s(b)) \vee (b \in S \cup T \wedge t(a) = b)\ )\}$$

Figure 2 shows the line graph of the workflow graph of Figure 1. The orange octagons with double line are nodes in S. The red diamonds are the nodes in T.

Generating the state graph: We build a *state graph* from the line graph of an acyclic workflow graph. Each node of the state graph is a multi-set containing two nodes of the line graph, i.e., two edges of the original workflow graph. Such multi-set represents a *state* of the line graph where the two nodes carry a token. The edges of the state graph represent the allowed token moves. In order to determine which node of the line graph is upstream of another, i.e., to determine the allowed token moves, we compute a value $v(n)$ given by the reverse post order numbering of the nodes [7]. The token on the node with the lowest reverse post order value is allowed to move. Similarly to Yang *et al.* [8], we perform a straightforward extension of the algorithm of Pearl and Shiloach [2]: We add to the state graph the the states where there are two tokens on a node in S or two tokens on a node in T. This allows us to check for two disjoint paths between

one pair of node in S and T instead of two pairs of nodes as originally described by Perl and Shiloach.

Definition 3 (State graph). *A* state graph *of an acyclic simple directed graph* $L = (N', E')$ *is an acyclic directed graph* $F = (N'', E'')$ *such that:*

$$N'' = \{[x, y] \mid x, y \in N \wedge x \neq y\} \cup \{[x, x] \mid x \in S \cup T\}$$
$$E'' = \{[x, y] \rightarrow [x', y] \mid x \rightarrow x' \in E' \wedge v(x) \leq v(y)\}$$

Figure 3 shows the state graph of the line graph of Figure 2. As shown by Perl and Shiloach [2] the number of edges $|E''|$ in the state graph is in $O(|N| \cdot |E|)$ in terms of the original graph.

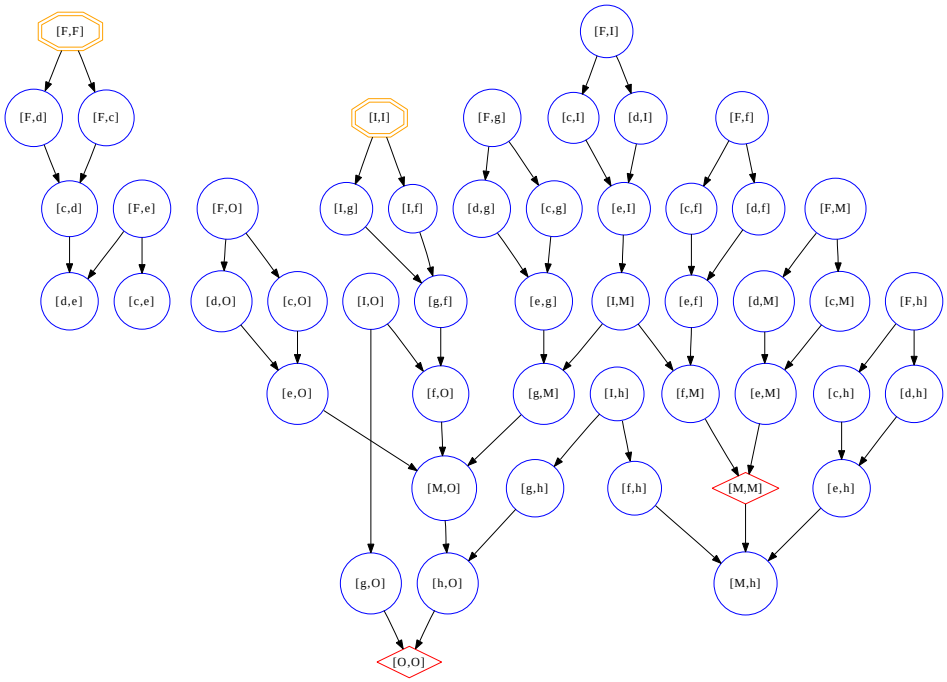

Fig. 3. The state graph of Figure 2 .

Checking for handles: In order to detect the existence of a handle in the original workflow graph, we have to check if there exists a path in the state graph from a state with two tokens on a node in S to a state with two tokens on a node in T. This is achieved by traversing the graph from each unvisited node

in S until reaching a node in $t \in T$ or having visited all the reachable nodes. In the worst case, the graph traversal visits each node of the graph once. The edges that belong to the handle in the workflow graph can be easily retrieved: They are the edges that are on the path from the state with two tokens on a node $s \in S$ to the state with two tokens on a node $t \in T$ of the handle in the state graph. On Figure 3 we observe that there is a path between $[I, I]$ and $[O, O]$ which indicates that there is an handle between I and O in workflow graph of Figure 1.

5 Conclusion

We propose an intuitive structural characterization for lack of synchronization in acyclic workflow graphs that contain inclusive OR logic: the handle. The handle is an adequate error message to the process modeler. We show how to check for handles by building a state graph. The size of the state graph is quadratic with respect to the original graph. All other operations are linear either with respect to the size of the original graph or the state graph. Thus, our approach requires quadratic time and space to check for handles. Note that the approach can be easily adapted to detect potential concurrency between two elements of the process. This has multiple applications. For example, we can detect two tasks accessing concurrently the same data store.

References

1. W. van der Aalst, A. Hirnschall, and H. Verbeek, "An alternative way to analyze workflow graphs," *Lecture Notes in Computer Science*, pp. 535–552, 2002.
2. Y. Perl and Y. Shiloach, "Finding two disjoint paths between two pairs of vertices in a graph," *J. Assoc. Comput. Mach*, vol. 25, no. 1, p. 9, 1978.
3. J. Esparza and M. Silva, "Circuits, handles, bridges and nets," *Advances in Petri nets*, vol. 483, pp. 210–242, 1990.
4. W. van der Aalst, "Workflow verification: Finding control-flow errors using Petri-net-based techniques," *Lecture Notes in Computer Science*, pp. 161–183, 2000.
5. A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum-flow problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, 1988.
6. F. Harary, "Graph Theory. 1969."
7. J. Gross and J. Yellen, *Graph theory and its applications*. CRC press, 2006.
8. B. Yang, S. Zheng, and E. Lu, "Finding two Disjoint Paths in a Network with Normalized α–Min-Sum Objective Function," 2005.