RZ 3780 (# Z1012-002) 12/09/2010
Computer Science 24 pages

Research Report

Symbolic Execution of Acyclic Workflow Graphs

C. Favre, H. Völzer

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

Symbolic Execution of Acyclic Workflow Graphs

Cédric Favre and Hagen Völzer

IBM Research — Zurich
{ced,hvo}@zurich.ibm.com

Abstract. We propose a new technique to analyze the control-flow, i.e., the workflow graph of a business process model, which we call *symbolic execution*. We consider acyclic workflow graphs that may contain inclusive OR gateways and define a symbolic execution for them that runs in quadratic time. The result allows us to decide in quadratic time, for any pair of control-flow edges or tasks of the workflow graph, whether they are sometimes, never, or always reached concurrently. This has different applications in finding control- and data-flow errors. In particular, we show how to decide soundness of an acyclic workflow graph with inclusive OR gateways in quadratic time. Moreover, we show that symbolic execution provides diagnostic information that allows the user to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process.

1 Introduction

With the increased use of business process models in simulation, code generation and direct execution, it becomes more and more important that the processes are free of control- and data-flow errors. Various studies (see [1] for a survey) have shown that such errors frequently occur in process models.

Some of these errors can be characterized in terms of relationships between control-flow edges or tasks of the process. For example, a process is free of *deadlock* if any two incoming edges of an AND-join are always marked concurrently. We can say that such a pair of edges is *always concurrent*. A process is free of *lack of synchronization* if any two incoming edges of an XOR-join are *mutually exclusive*, i.e., they never get marked concurrently. A data-flow hazard may arise if two conflicting operations on the same data object are executed concurrently. That can happen only if the tasks containing the data operations are *sometimes concurrent*, i.e., not mutually exclusive. Similar relationships have also been proposed for a behavioral comparison of processes [2].

Such control-flow relations can be computed by enumerating all reachable control-flow states of the process by explicitly executing its *workflow graph*, i.e., its control-flow representation. However, there can be exponentially many such states, resulting in a worst-case exponential time algorithm. We propose in this paper a form of symbolic execution of a workflow graph. We consider acyclic workflow graphs that may contain inclusive OR (IOR) gateways and define a symbolic execution of such graphs that runs in quadratic time. It captures enough information to allow us to decide, using a complementing graph analysis technique, the above mentioned relationships for any pair of control-flow edges in quadratic time. In particular, we obtain a control-flow

analysis that decides *soundness*, i.e., absence of deadlock and lack of synchronization in quadratic time for any acyclic graph that may contain IOR gateways.

The symbolic execution keeps track of which decision outcomes within the process flow lead to which edge being marked. Therefore, it can provide information, in case of a detected error, about which decisions potentially lead to the error. We show how this leads to more compact diagnostic information than obtained with prior techniques. In particular, we show how this allows the user to efficiently deal with spurious errors that arise due to over-approximation of the data-based decisions in the process.

Some existing techniques can decide soundness of a workflow graph without IOR gateways, or equivalently a Free Choice Petri net, in polynomial time: A technique based on the *rank theorem* [3] in cubic time and techniques based on a complete reduction calculus [4] in more than cubic time. However, diagnostic information is not provided by the former technique and was not yet worked out for the latter.

Techniques based on *state space exploration* return an *error trace*, i.e., an execution that exhibits the control-flow error, but they have exponential worst-case time complexity. It has been shown [5] for industrial processes without IOR gateways that the latter problem can be effectively addressed in practice using various reduction techniques. Various additional structural reduction techniques exist in the literature, e.g., [6, 7].

Wynn *et al.* [8] provide a study of verifying processes with IOR gateways. They apply state space exploration and use a different IOR-join semantics.

We are not aware of approaches that provide diagnostic information to deal with the over-approximation due to data abstraction in workflow graphs. Existing approaches to check other notions of soundness such as *relaxed soundness* [9] or *weak soundness* [10] have exponential complexity.

The paper is structured as follows: After setting the preliminary notions, we introduce symbolic execution in Sect. 3 and show how the relationship ‘always-concurrent’ and the absence of deadlock can be decided. Then, we discuss the ‘sometimes-concurrent’ and ‘mutually-exclusive’ relationships and lack of synchronization in Sect. 4. In Sect. 5, we show how the diagnostic information provided by symbolic execution can be used to deal with the over-approximation that results from abstracting from data-based decisions.

This research reports extends a conference paper [11] with proofs that the conference paper omits to satisfy space constraints.

2 Preliminaries

In this section, we define preliminary notions which include workflow graphs and their soundness property.

2.1 Basic notions

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 $(m_1 - m_2)[x] = m_1[x] - m_2[x]$. The *scalar product* is defined by $m_1 \otimes m_2 =$

$\sum_{x \in U}(m_1[x] \times m_2[x])$. By abuse of notation, we sometimes use a set $X \subseteq U$ in a multi-set context by setting $X[x] = 1$ if $x \in X$ 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 $s \rightarrow t$. A *directed multi-graph* $G = (N, E, c)$ consists of a set N of nodes, a set E of *edges* and a mapping $c : E \rightarrow (N \cup \{\text{null}\}) \times (N \cup \{\text{null}\})$ 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 is a *source* of the graph. If $t = \text{null}$, then we say that e is a *sink* of the graph. 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*, if 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.

2.2 Workflow graphs

A *workflow graph* $W = (N, E, c, l)$ consists of a multi-graph $G = (N, E, c)$ and a mapping $l : N \rightarrow \{\text{AND}, \text{XOR}, \text{IOR}\}$ that associates a *logic* with every node $n \in N$, such that: 1. The workflow graph has exactly one source and at least one sink. 2. 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.

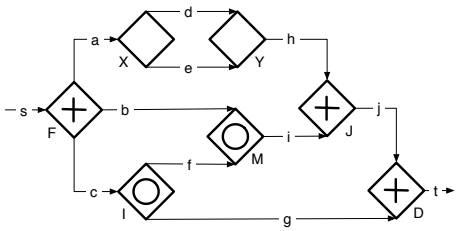

Fig. 1. A workflow graph.

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 presentation simplicity, we use workflow graphs composed of only splits and joins.

Let, in the rest of this section, $W = (N, E, c, l)$ be an acyclic graph. Let $x_1, x_2 \in N \cup E$ be two elements of W such that there is a path from x_1 to x_2. We then say that x_1 *precedes* x_2, denoted $x_1 < x_2$, and x_2 *follows* x_1. Two elements $x_1, x_2 \in N \cup E$ of W are *unrelated*, denoted $x_1 \parallel x_2$, if $x_1 \neq x_2$ and neither $x_1 < x_2$ nor $x_2 < x_1$. A *prefix* of W is a workflow graph $W' = (N', E', c', l')$ such that $N' \subseteq N$, for each pair of nodes $n_1, n_2 \in N$, if $n_2 \in N'$ and $n_1 < n_2$ then $n_1 \in N'$, an edge e belongs to E' if there exists a node $n \in N'$ such that e is adjacent to n, for each node $n \in N'$, we have $l'(n) = l(n)$, and for each edge $e \in E'$, we have $c'(e) = (s', t')$, $c(e) = (s, t)$, $s' = s$, $t' = t$ if $t \in N'$, and $t' = \text{null}$ otherwise.

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 edges and adds one token to a non-empty subset of its outgoing edges. This IOR semantics, which is explained in detail elwhere [12], complies with the BPMN standard and BPEL's dead path elimination.

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. 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* in m. The *initial marking* m_s of W is such that the source edge is marked with one token in m_s and no other edge is marked in m_s.

Let m and m' be two markings of W. A tuple (E_1, n, E_2) is called a *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 if for each edge $e \in E_1$ we have $m[e] > 0$ and any of the following propositions:

- $l(n) = \text{AND}$, $E_1 = {\circ}n$, and $E_2 = n{\circ}$.
- $l(n) = \text{XOR}$, there exists an edge e such that $E_1 = \{e\}$, and there exists an edge e' such that $E_2 = \{e'\}$.
- $l(n) = \text{IOR}$, E_1, E_2 are non-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 if 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 if 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 if m is reachable from m_s.

2.3 Soundness

A deadlock occurs when a token stays 'blocked' on one edge of the workflow graph: A *deadlock* of W is a reachable marking m of W such that there exists a non-sink edge $e \in E$ that is marked in m and e is marked in all the markings reachable from m. We

say that *W contains* a deadlock if and only if there exists a reachable marking m of W such that m is a deadlock. The workflow graph in Fig. 1 permits the execution $\sigma = \langle [s], [a,b,c], [b,c,d], [b,c,h], [b,f,h], [h,i], [j] \rangle$. The marking $[j]$ is a deadlock.

A *lack of synchronization* of W is a reachable marking m of W such that there exists an edge $e \in E$ that is marked by more than one token in m. We say that a workflow graph W *contains* a lack of synchronization if and only if there exists a reachable marking m of W such that m is a lack of synchronization.

A workflow graph is *sound* if it contains neither a deadlock nor a lack of synchronization. Note that this notion of soundness is equivalent to the notion presented by van der Aalst [13] for workflow nets.

3 Symbolic Execution and Always-Concurrent Edges

In this section, we introduce *symbolic execution* and show how we use it to detect deadlocks and determine whether two edges are always-concurrent. We start by giving a characterization of deadlock, then introduce the symbols and the propagation rules of the symbols, we show how to compute a normal form of a symbol and discuss the complexity of the proposed technique.

Let, in this section, $W = (N, E, c, l)$ be an acyclic workflow graph prefix that is free of lack of synchronization. We describe in Sect. 4.3 how we determine such prefix.

3.1 Equivalence of edges and a characterization of deadlock

A deadlock arises at an AND-join when one of its incoming edges e is marked during an execution σ but another edge e' does not get marked during σ because, as e' never gets marked during σ, the AND-join cannot execute and the token marking e is 'blocked'. Thus, in order to have no deadlock, the incoming edges of an AND-join need to get marked 'together' in each execution. We can precisely capture this through *edge equivalence* or the notion *always-concurrent*. In an acyclic workflow graph, only an AND-join can can '*cause*' a deadlock. An IOR-join can 'block' a token if and only if there exists a preceding node that blocks another token. Thus, whenever there is a deadlock in an acyclic workflow graph, there exists an AND-join with non-equivalent incoming edges. Nodes that are nor AND-join or IOR-join cannot block a token. To introduce edge equivalence, we define the *Parikh vector* of an execution, which records, for each edge, the number of tokens that are produced on that edge during the execution.

Definition 1 (Parikh vector). *The* Parikh vector *of an execution $\sigma = \langle m_0, T_0, ... \rangle$, written $\vec{\sigma}$, is the multi-set of edges such that $\vec{\sigma}[s] = 1$ for the source s of W and otherwise $\vec{\sigma}[e] = k$ such that $k = |\{i \mid T_i = (E, n, E') \wedge e \in E'\}|$.*

For example, given the execution $\sigma = \langle [s], (\{s\}, F, \{a,b,c\}), [a,b,c], (\{a\}, X, \{d\}), [b,c,d], (\{d\}, Y, \{h\}), [b,c,h], (\{c\}, I, \{f\}), [b,f,h]\ (\{b,f\}, M, \{i\}), [h,i], (\{h,i\}, J, \{j\}), [j] \rangle$ of the workflow graph of Fig. 1, we have $\vec{\sigma}[s] = \vec{\sigma}[a] = \vec{\sigma}[b] = \vec{\sigma}[c] = \vec{\sigma}[d] = \vec{\sigma}[f] = \vec{\sigma}[h] = \vec{\sigma}[i] = \vec{\sigma}[j] = 1$ and $\vec{\sigma}[e] = \vec{\sigma}[g] = 0$.

Definition 2 (Edge equivalence, always-concurrent).

- *Two edges are* parallel *in an execution σ if there is a marking in σ in which both edges are marked. Two executions σ, σ' are* interleaving equivalent *if $\vec{\sigma} = \vec{\sigma}'$. Two edges are* concurrent *in σ if there is an execution σ' such that σ and σ' are interleaving equivalent and the edges are parallel in σ'. Two edges are* always-concurrent *if they are concurrent in every execution of W.*
- *Two edges e_1 and e_2 of W are* equivalent, *written $e_1 \equiv e_2$, if for any execution σ of W, we have $\vec{\sigma}[e_1] = \vec{\sigma}[e_2]$.*

Two executions that are interleaving equivalent execute the same transitions; possibly in a different order. Note that these definitions are founded only for acyclic workflow graphs.

Proposition 1. *Two edges e_1, e_2 are always-concurrent iff $e_1 \equiv e_2$ and $e_1 \parallel e_2$.*

In the workflow graph depicted by Fig. 1, we have $a \equiv b \equiv h$ and $a \not\equiv d \not\equiv g$. Note that we have discussed earlier an execution of the workflow graph of Fig. 1 where $\vec{\sigma}[a] = \vec{\sigma}[d]$. However, there exist another execution such that $\vec{\sigma}[a] \neq \vec{\sigma}[d]$ and therefore $a \not\equiv d$. Moreover, a is always-concurrent to b but not to h.

Proposition 2. *W contains a* deadlock *iff there exist two incoming edges of an AND-join of W that are not equivalent, or equivalently, that are not always-concurrent.*

In the workflow graph depicted by Fig. 1, the edges j and g are not always-concurrent. Therefore, we get a deadlock at the AND-join D.

In the following, we show how we can compute edge equivalence and therefore also whether two edges are always-concurrent.

3.2 Symbolic execution

The first step to compute edge equivalence is the symbolic execution of the workflow graph. During symbolic execution, each edge is labeled with a symbol, which is a set of *outcomes* of the workflow graph. An *outcome* is the source edge, an outgoing edge of an XOR-split, or an outgoing edge of an IOR-split in the graph. Figure 2 shows the labeling of the workflow graph of Fig. 1 that results from its symbolic execution.

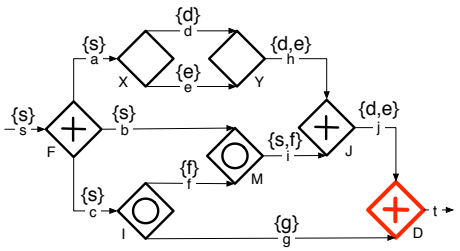

Fig. 2. The assignment resulting from the symbolic execution of the workflow graph of Fig. 1.

The symbolic execution starts with labeling the source s with $\{s\}$. All other edges are yet unlabeled. If all incoming edges of a node are labeled, we may label the outgoing edges of the node by applying one of the propagation rules depicted by Fig. 3, depending on the logic of the node.

The intuition behind symbolic execution is to label an edge e with a set S of outcomes such that e is marked during an execution σ if and only if some of the outcomes in S get marked during σ. In general, the label of the outgoing edges depends on the labels of the incoming edges. However, if the node is an XOR-split or an IOR-split, then the symbol that is assigned to one of the outgoing edges only contains that outgoing edge. The symbol associated to the incoming edge of the node is then ignored. In case of an AND-join, the propagation rule additionally requires the symbol labeling its incoming edges to be equivalent (which we will describe in Sect. 3.3) in order to be applied. The AND-join rule then chooses one of the labels of the incoming edges non-deterministically as the label for the outgoing edge. The symbol labeling an outgoing edge of a node that is an XOR-join or an IOR-join, is the union of the symbols labeling the incoming edges of the node. The symbolic execution terminates when there is no progation rule that can be applied. In the following, we define these propagation rules formally.

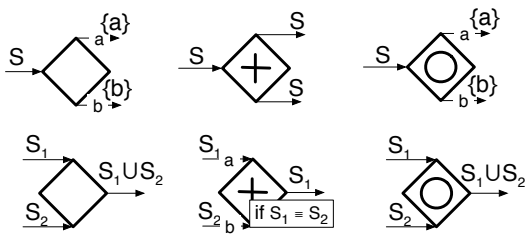

Fig. 3. The propagation rules.

Definition 3 (Symbolic execution). *An* outcome *of W is the source, an outgoing edge of some XOR-split, or an outgoing edge of some IOR-split of W. A* symbol *of W is a set of outcomes of W. An* assignment *is a mapping φ that assigns a symbol to each edge of some prefix of W. If e is an edge of that prefix, we say that e is* labeled under *φ.*

For every node n of a workflow graph, we describe the propagation *by the node n from an assignment φ to an assignment φ', written $\varphi \xrightarrow{n} \varphi'$. The propagation $\varphi \xrightarrow{n} \varphi'$ is* activated *when all the incoming edges of n are labeled under φ and no outgoing edge is labeled under φ. Additionally, if n is an AND-join, the symbol associated to each incoming edges of n must be equivalent (according to Def. 4) for the propagation to be activated. If n is activated in φ, we have $\varphi \xrightarrow{n} \varphi'$ where φ' is obtained as follows, for any edge e of W:*

- *If $l(n)$ = AND and there exists an edge $e' \in \circ n$, then $\varphi'(e) = \varphi(e')$ for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*
- *If n is an XOR-split or an IOR-split, then $\varphi'(e) = \{e\}$ for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*
- *If n is an XOR-join or an IOR-join, for $\varphi'(e) = \bigcup_{e' \in \circ n} \varphi(e')$for $e \in n\circ$ and $\varphi'(e) = \varphi(e)$ otherwise.*

As said above, the propagation rules establish that an edge e is marked during an execution σ if and only if some of the outcomes in $\varphi(e)$ are marked during σ:

Lemma 1. *For any execution σ of W and any edge $e \in E$, $\vec{\sigma} \otimes \varphi(e) > 0 \Leftrightarrow \vec{\sigma}[e] > 0$.*

Proof. We perform an induction on the number of edges labeled under φ. We increase the number of edges labeled under φ using propagation rules of Def. 3.

Base case: *Only the source edge s of W is labeled under φ with the symbol $\{s\}$. By the definition of the Parikh vector (Def. 1), $\vec{\sigma}[s] = 1$. Trivially, we have $\vec{\sigma} \otimes \varphi(s) = 0 \Leftrightarrow \vec{\sigma}[s] = 0$.*

Induction step: *Assuming an assignment φ of W according to Def. 3 and that $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow \vec{\sigma}[e] = 0$ holds for each execution σ of W and edge e labeled under φ. We want to show that, when labeling the outgoing edge(s) of any node $n \in N$ according to the assignment propagation rules, i.e., applying one transition rule to increase φ, $\vec{\sigma} \otimes \varphi(e) = 0 \Leftrightarrow \vec{\sigma}[e] = 0$ for each execution σ of W and labeled edge e. As the assigned symbols do not change, we only have to show the latter proposition for the freshly labeled edges.*

For each execution σ, each incoming edge $e_I \in \circ n$, and each outgoing edge $e_O \in n\circ$ of n, we consider the four following cases:

1. *When n is an XOR-split or IOR-split: By the assignment propagation rules (Def. 3), we have $\varphi(e_O) = [e_O]$. Thus, it follows directly from the definition of the Parikh vector (Def. 1) that $\varphi(e_O) \otimes \vec{\sigma} = \vec{\sigma}[e_O]$.*
2. *When n is an XOR-join:*
   $\varphi(e_O) \otimes \vec{\sigma} = 0$
   $\Leftrightarrow \sum_{e_{Ik} \in \circ n} \varphi(e_{Ik}) \otimes \vec{\sigma} = 0$, *by the assignment propagation rules (Def. 3).*
   $\Leftrightarrow \sum_{e_{Ik} \in \circ n} \vec{\sigma}[e_{Ik}] = 0$, *by the induction hypothesis.*
   $\Leftrightarrow \vec{\sigma}[e_O]$, *by the workflow graph semantics.*
3. *When n is an IOR-join:*
   $\Leftrightarrow \varphi(e_O) \otimes \vec{\sigma} = 0$,
   $\Leftrightarrow \bigcup_{e_I \in \circ n} (\varphi(e_I)) \otimes \vec{\sigma} = 0$, *by the deadlock assignment propagation rule for the IOR-join (Def. 3).*
   $\Leftrightarrow$ *for each edge $e_I \in \circ n$, $\varphi(e_I) \otimes \vec{\sigma} = 0$*
   $\Leftrightarrow \sum_{e_I \in \circ n} (\varphi(e_I) \otimes \vec{\sigma}) = 0$.
   $\Leftrightarrow \sum_{e_I \in \circ n} (\vec{\sigma}[e_I]) = 0$, *by the induction hypothesis.*
   $\Leftrightarrow \vec{\sigma}[e_O] = 0$, *by the workflow graph execution semantics and the assignment propagation rules, which requires that there is no deadlock in the labeled prefix of W because, for each AND-join j in the prefix, all the incoming edges of j are equivalent.*
4. *When $l(n) = AND$:*
   $\varphi(e_O) \otimes \vec{\sigma} = 0$
   $\Leftrightarrow \varphi(e_I) \otimes \vec{\sigma} = 0$ *because, by the assignment propagation rules (Def. 3), there exist an edge e_I such that $\varphi(e_O) = \varphi(e_I)$.*
   $\Leftrightarrow \vec{\sigma}[e_I] = 0$, *by the induction hypothesis.*
   $\Leftrightarrow \vec{\sigma}[e_O] = 0$, *because the workflow graph semantics implies that $\Leftrightarrow \vec{\sigma}[e_0] = \min_{e_I \in \circ n}(\vec{\sigma}[e_I])$. Moreover, the assignment propagation rules requires all edges $e \in \circ n$ to be equivalent so that the propagation is activated. Thus, we have $\vec{\sigma}[e_I] = \vec{\sigma}[e_O] = 0$*

□

3.3 A normal form for symbols

To detect a deadlock or to label the outgoing edge of an AND-join, we need to check edge equivalence. If two incoming edges of an AND-join are not equivalent, we have found a deadlock.

We will exploit that the equivalence of edges corresponds to an equivalence of the symbols they are labeled with. This symbol equivalence can be defined as follows:

Definition 4 (Symbol equivalence). *Two symbols S_1, S_2 are* equivalent *w.r.t. W, written $S_1 \equiv S_2$ if, for any execution σ of W, $S_1 \otimes \vec{\sigma} = 0 \Leftrightarrow S_2 \otimes \vec{\sigma} = 0$.*

As W is free of lack of synchronization, for any edge e and for any execution σ, we have $\vec{\sigma}[e] = 1$ or $\vec{\sigma}[e] = 0$. Thus, given two edges e_1, e_2 labeled under φ, the edges e_1 and e_2 are equivalent if and only if the symbols $\varphi(e_1)$ and $\varphi(e_2)$ are equivalent.

We will decide the equivalence of two symbols by computing a normal form for each of them. The normal form of a symbol S is the 'largest' set of outcomes that is equivalent to S. Two symbols are equivalent if and only if they have the same normal form. To show this, we define:

Definition 5 (Maximal equivalent extension, Closure). *Let φ be an assignment of W and e be an edge such that e is labeled under φ. Let O be the set of outcomes of W that are labeled under φ.*

- *A* maximal equivalent extension *of $\varphi(e)$ w.r.t. φ is a set $\varphi^*(e) \subseteq O$ such that $\varphi^*(e) \equiv \varphi(e)$ and there exist no other set $S \subseteq O$ such that $\varphi^*(e) \subsetneq S$ and $S \equiv \varphi(e)$.*
- *The* closure *of $\varphi(e)$ w.r.t. φ is the smallest set $\overline{\varphi}(e)$ such that $\varphi(e) \subseteq \overline{\varphi}(e)$ and for each XOR- or IOR-split n such that e' is labeled under φ for each $e' \in n\circ$, we have $\varphi(\circ n) \subseteq \overline{\varphi}(e)$ iff $n\circ \subseteq \overline{\varphi}(e)$.*

The existence of a maximal equivalent extension is clear. We can also show that it is unique.

Lemma 2. *Let φ be an assignment of W and e an edge that is labeled under φ. Then $\varphi^*(e)$ is unique.*

Proof. There exists some maximal equivalent extension of $\varphi(e)$ because e is labeled under φ and $\varphi(e)$ is trivially equivalent to $\varphi(e)$. Thus some outcomes can be added to $\varphi(e)$ until obtaining a maximal equivalent extension. As the number of outcomes is finite, the size of the maximal equivalent extension is finite.

We show that $\varphi^*(e)$, the maximal equivalent extension of $\varphi(e)$, is unique by contradiction: Suppose that there exist two sets of outcomes S_1 and S_2 such that $S_1 \neq S_2$ and S_1 and S_2 are both maximal equivalent extensions of $\varphi(e)$.

Because $S_1 \neq S_2$, we can assume without loss of generality that there exists an edge $e' \in S_2 \setminus S_1$. We show that $S_1 \cup S_2 \equiv \varphi(e)$ which contradicts the maximality of S_1.

As S_1 and S_2 are maximal equivalent extensions of $\varphi(e)$ (Def. 5), $S_1 \equiv \varphi(e)$ and $S_2 \equiv \varphi(e)$. It follows from the definition of equivalence (Def. 4) that, for any execution σ of W, $S_1 \otimes \vec{\sigma} = 0 \Leftrightarrow \varphi(e) \otimes \vec{\sigma} = 0$ and $S_2 \otimes \vec{\sigma} = 0 \Leftrightarrow \varphi(e) \otimes \vec{\sigma} = 0$. Thus, by definition of the scalar product, for any execution σ of W, $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow (\sum_{e_1 \in S_1} \{e_1\} \otimes \vec{\sigma} =$

$0) \wedge (\sum_{e_2 \in S_2} \{e_2\} \otimes \vec{\sigma} = 0)$. *Moreover, by definition of the scalar product* $(S_1 \cup S_2) \otimes \vec{\sigma} = 0 \Leftrightarrow \sum_{e^* \in S_1 \cup S_2} \{e^*\} \otimes \vec{\sigma} = 0$. *Thus, for any execution* σ *of* W, $\varphi(e) \otimes \vec{\sigma} = 0 \Leftrightarrow (S_1 \cup S_2) \otimes \vec{\sigma} = 0$ *which is the definition of* $\varphi(e) \equiv S_1 \cup S_2$ *(Def. 4).* □

It is clear that the closure exists and is unique. We show that the closure of a symbol $\overline{\varphi}(e)$ is equivalent to $\varphi(e)$:

Lemma 3. *Let* $W = (N, E, c, l)$ *be an acyclic workflow graph,* φ *be an assignment of* W*, and* $e \in E$ *be an edge labeled under* φ*. We have* $\overline{\varphi}(e) \equiv \varphi(e)$

Proof. By the workflow graph execution semantics we have, for any XOR- or IOR-split d, $\vec{\sigma}[^\circ d] = 0 \Leftrightarrow \sum_{e \in d^\circ} (\vec{\sigma}[e]) = 0$. *Thus, by Lemma 1,* $\varphi(^\circ d) \otimes \vec{\sigma} = 0 \Leftrightarrow \bigcup_{e \in d^\circ} \varphi(e) \otimes \vec{\sigma} = 0$. *Which, by the assignment propagation rules (Def. 3), is equivalent to* $\varphi(^\circ d) \otimes \vec{\sigma} = 0 \Leftrightarrow (d^\circ) \otimes \vec{\sigma} = 0$. *To compute the closure, we add* d° *if the symbol contains* $\varphi(^\circ d)$ *or we add* $\varphi(^\circ d)$ *if the symbol contains* d°*, it is clear that we stay in the same equivalence class at each operation needed to obtain the closure.* □

We can now prove that the closure is equal to the maximal equivalent extension:

Theorem 1. *Let* φ *be an assignment of* W*. For every edge* e *that is labeled under* φ*, we have* $\varphi^*(e) = \overline{\varphi}(e)$.

Proof. We prove that $\varphi^*(e) = \overline{\varphi}(e)$. *By Lemma 3,* $\overline{\varphi}(e) \equiv \varphi(e)$. *Thus any edge in* $\overline{\varphi}(e)$ *is also in* $\varphi^*(e)$ *by definition of the maximal equivalent extension of* $\varphi(e)$ *(Def. 5). It is left to show that each edge in* $\varphi^*(e)$ *is also in* $\overline{\varphi}(e)$.

We prove by contradiction that there exists no edge e' *such that* $e' \in \varphi^*(e)$ *and* $e' \notin \overline{\varphi}(e)$, *i.e.,* $\varphi^*(e) \setminus \overline{\varphi}(e) \neq \emptyset$.

Suppose that there exists an edge e' *such that* $e' \in \varphi^*(e) \setminus \overline{\varphi}(e)$.

We show how to build an execution σ *of* W *such that* $\vec{\sigma} \otimes \varphi^*(e) > 0$ *and* $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$. *The existence of such an execution shows that* $\varphi^*(e) \not\equiv \overline{\varphi}(e)$ *(Def. 4), which contradicts* $\varphi^*(e) \equiv \varphi(e)$ *because* $\overline{\varphi}(e) \equiv \varphi(e)$ *by Lemma 3.*

The construction of σ *is performed in two steps: First, we define a path* p *from the source to* e'*. Second, we construct the execution* σ *itself. The construction strategy is to avoid taking any outcome in* $\overline{\varphi}(e)$ *and thus* $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$. *The execution follows* p *to ensure that a token reaches the edge* e' *and thus* $\vec{\sigma} \otimes \varphi^*(e) > 0$.

1. *The path* p *is defined inductively from its last edge* e*. The induction stops when reaching* s*. For each edge* e_p *of* p*, we ensure that* $\varphi(e_p) \setminus \overline{\varphi}(e) \neq \emptyset$.
   As $e' \in \varphi^*(e)$, e' *is an outcome, i.e.,* e' *is the outgoing edge of an IOR-split or an XOR-split. By the assignment propagation rules (Def. 3),* $\varphi(e') = \{e'\}$. *Therefore* $\varphi(e') \setminus \overline{\varphi}(e) \neq \emptyset$ *by assumption.*
   The previous elements on the path is defined based on the current element as follows:
   - *if the current element is an edge* e_c*, the previous element is the source node of* e_c.

– *if the current element is a node n, the previous element is any of the incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$.*
We show that an incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$ always exists. By induction hypothesis of p there exists an outgoing edge e_o of n such that $\varphi(e_o) \setminus \overline{\varphi}(e) \neq \emptyset$. Let's consider three cases based on the type of n:
  (a) *If $l(n) = AND$: By the assignment propagation rules (Def. 3), there exists an incoming edge e_i of n such that $\varphi(e_i) = \varphi(e_o)$.*
  (b) *XOR and IOR joins: By the assignment propagation rules (Def. 3), $\varphi(e_o) = \bigcup_{e_i \in \circ n} \varphi(e_i)$. Thus there exists an incoming edge e_i of n such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$.*
  (c) *XOR and IOR splits: The incoming edge e_i is such that $\varphi(e_i) \setminus \overline{\varphi}(e) \neq \emptyset$. If it was not the case, then $\varphi(e_i) \subseteq \overline{\varphi}(e)$ and, by the definition of the closure (Def. 5), $\varphi(e_o) \subseteq \overline{\varphi}(e)$ because if $\varphi(e_i) \subseteq \overline{\varphi}(e)$, then $n\circ \in \overline{\varphi}(e)$, which contradicts $\varphi(e_o) \setminus \overline{\varphi}(e) \neq \emptyset$*

By the definition of workflow graph, for every edge e_g of W, there exists a path from s to e_g. As W is acyclic, we can always build the finite path p starting from s to e′ such that, for every edge e_p of p, $\varphi(e_p) \setminus \overline{\varphi}(e) \neq \emptyset$.

2. *We build inductively an execution σ such that $\vec{\sigma} \otimes \overline{\varphi}(e) = 0$ and $\vec{\sigma} \otimes \varphi^*(e) > 0$. The insight in building σ is to not mark any edge in $\overline{\varphi}(e)$ and to mark e′. For each marking m of σ, we maintain that every edge e_σ marked in m $\varphi(e_\sigma) \setminus \overline{\varphi}(e) \neq \emptyset$.*
The execution starts with the initial marking $[s]$. As $\varphi(s) = \{s\}$ and s is on p, $s \notin \overline{\varphi}(e)$ and thus $\varphi(s) \setminus \overline{\varphi}(e) \neq \emptyset$.
We define how any activated node n is executed during σ. The marking m changes to the marking m′ as follows:
   – *If n has AND logic, is an XOR-join, or an IOR-join: There is a unique execution step possible and the marking changes from m to m′ according to the workflow graph semantics. Executing such type of node cannot mark an edge in $\overline{\varphi}(e)$ because, by the assignment propagation rules (Def. 3), edges in $\overline{\varphi}(e)$ are exclusively outcomes.*
   – *If n is an XOR-split or an IOR-split: We distinguish two cases:*
     (a) *If there is one of the outgoing edges e_o of n that is on p, then $m' = m - [^\circ n] + [e_0]$, i.e., executing n propagates the token from $^\circ n$ to e_0. By construction of p, $e_o \notin \overline{\varphi}(e)$.*
     (b) *If there is no outgoing edge of n that is on p, we pick any outcome e_o such that $e_o \notin \overline{\varphi}(e)$. Again $m' = m - [^\circ n] + [e_0]$. By induction hypothesis $\varphi(^\circ n) \not\subseteq \overline{\varphi}(e)$. Therefore, there is always an outgoing edge e_o of n such that $e_o \notin \overline{\varphi}(e)$ because, by definition of the closure (Def. 5), if for any edge $n\circ \subseteq \overline{\varphi}(e)$, then $\varphi(^\circ n) \subseteq \overline{\varphi}(e)$, which contradicts $\varphi(^\circ n) \setminus \overline{\varphi}(e) \neq \emptyset$.*

□

That is, we obtain a unique normal form that is equivalent with a given label of an edge. We show in Sect. 3.4 that the closure can be computed in linear time. Thus, from the characterization as a closure, we can compute the normal form in linear time. Moreover, the normal form has the desired property:

Theorem 2. *$\overline{\varphi}(e) = \overline{\varphi}(e')$ whenever e and e' are equivalent.*

We are now able to compute the closure for the edges g, h, i, and j of the example from Fig. 2. We have $\overline{\varphi}(g) = \{g\}$, $\overline{\varphi}(h) = \overline{\varphi}(i) = \overline{\varphi}(j) = \{s, d, e, f, g\}$. As $\overline{\varphi}(h) = \overline{\varphi}(i)$, h and i are equivalent. Thus, there is no deadlock at the AND-join J. On the contrary, $\overline{\varphi}(g)$ differs from $\overline{\varphi}(j)$ which implies, by Thm. 2, that g and j are not equivalent. Therefore, we detect a deadlock located at the AND-join D.

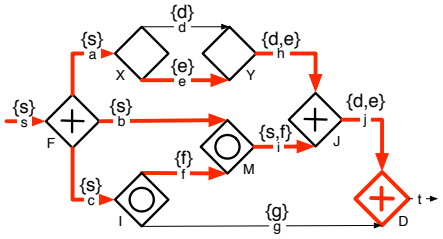

Fig. 4. Display of a deadlock.

When we detect a deadlock because two incoming edges of an AND-join are not equivalent, we say that the AND-join is the *location* of the deadlock. To display the deadlock, we can, based on the assignment, generate in linear time an execution, called *error trace*, that exhibits the deadlock. Figure 4 depicts how we would display a deadlock: we highlight the location of the deadlock and the error trace, i.e., the edges marked during the execution leading to the deadlock. We discuss in Sect. 5 a form of diagnostic information and user interaction that goes beyond this error trace.

3.4 Complexity of the computation

In this section, we first describe an approach to compute the closure in linear time and then discuss overall the complexity of symbolic execution.

Let, in this section, φ be an assignment of a workflow graph W and D be the set of IOR-splits and XOR-splits of W such that, for every node $d \in D$, every edge in $d\circ$ is labeled under φ. We define a *closure operation of* a node $d \in D$ *on* a symbol S that changes S *to* a symbol S' such that $S' = S \cup \varphi(\circ d) \cup d\circ$. A closure operation of a node n changing S to S' is *enabled* when $\varphi(\circ d) \in S$ or $d\circ \in S$ and $S \neq S'$. The computation of the closure comprises two phases:

1. We go through the nodes from the maximal to the minimal element in D w.r.t. the precedence relation $<$, i.e., from the right most nodes in the graph to the left most nodes of the graph. For each node n, we execute the closure operation of n if it is enabled.
2. We go through the nodes from the minimal to the maximal element in D w.r.t. the precedence relation $<$. For each node n, we execute the closure operation of n if it is enabled.

It is clear that this computation requires linear time. Note that D must be sorted. This sorting is obtained once and for all before performing symbolic execution and requires linear time with respect to the size of the workflow graph. Moreover, we show that the this sequence of phases is sufficient to ensure completeness of the closure computation:

Lemma 4. *After performing phase 1 and phase 2 on a symbol S, there exists no node $n \in D$ such that a closure operation of d is enabled on S.*

Proof. We have to ensure that after the two phases there is no more closure operation that is enabled:

We say that a closure operation of a node $d \in D$ on a symbol S is right enabled *iff $d\circ \subseteq S$ and $S \neq S'$. It is* left enabled *iff $\varphi(^\circ d) \subseteq S$ and $S \neq S'$.*

A closure operation of a node d is enabled only once because, by definition of left enabled and right enabled, an operation is enabled only if $S \neq S'$, a closure operation grows S by adding to it $\varphi(^\circ d) \cup d\circ$, and S is monotonously growing during the computation of the closure.

We show four propositions that help us in the proof:

1. *A right enabled closure operation of $d \in D$ on S to S' cannot right enable a closure operation of a node $d' \in D$ such that $d < d'$ because a right enabled closure operation of d only adds to the symbol outcomes that precede d.*
2. *Similarly, a left enabled closure operation of $d \in D$ on S to S' only adds outcomes that follow d. Thus, it cannot left enable a closure operation of a node that precedes d.*
3. *We show now that a left enabled closure operation cannot right enable a closure operation: Assume that we perform a left enabled closure operation of a node $d \in D$ on S to S'. We have $S' = S \cup d\circ$. For any $d' \in D$ such that $d \neq d'$, we have $d \circ \cap d'\circ = \emptyset$. Thus it cannot right enable a closure operation of d'. Moreover, it cannot right enable a closure operation of d because a closure operation of a node cannot be enabled twice.*
4. *By 1 and 3 we have that a closure operation of a node d can only right enable a node that precedes d.*

We first show by contradiction that at the end of phase 1 *all the right enabled closure operations are performed, i.e,* no closure operation is right enabled and it is not possible to right enable a closure operation*:*

Assume that there exists a right enabled closure operation at the end of phase 1*. Consider the node d that is the minimal node with respect to $<$ (,i.e., the left most node) such that a closure operation of d is right enabled at the end of phase* 1*. As a closure operation of d is enabled only once, the closure operation was not enabled when phase* 1 *visited d. Thus, a closure operation on a node preceding d enabled right the closure operation of d, which is in contradiction with 4.*

We have shown that there is no closure operation that is right enabled at the end of phase 1*. Moreover, by 3 we have that there will not be right enabled closure operation anymore.*

We show now that at the end of phase 2 there is no more left enabled closure operation*. We again show this by contradiction:*

Assume that at the end of phase 2 *there exists a left enabled closure operation. Consider the node d that is the maximal node with respect to $<$ (,i.e., the right most node) such that a closure operation of d is left enabled at the end of phase* 2*. As a closure operation of d is enabled only once, the closure operation was not enabled when phase* 2 *visited d. Thus, a closure operation on a node following d enabled left the closure operation of d, which is in contradiction with 2 and that there will not be right enabled closure operation anymore.* □

Symbolic execution needs just one traversal of the workflow graph. The closure is the most expensive operation. We have shown how to compute the closure of a symbol in linear time with respect to the size of D. Therefore, each transition takes at most linear time and the overall worst-case time complexity is quadratic.

4 Lack of Synchronization and Sometimes-Concurrent Edges

The workflow graph depicted in Fig. 5 permits the execution $\sigma = \langle [s], [a,b], [b,d], [d,e], [e,g], [g,g], ... \rangle$. The edge g is marked by two tokens in the marking $[g,g]$. Thus, the workflow graph depicted by Fig. 5 contains a lack of synchronization. In this section, we describe an algorithm that detects lack of synchronization and sometimes-concurrent edges. The technique has quadratic time complexity.

We first give a characterization of lack of synchronization in terms of *handles* of the graph and then show how handles can be computed in quadratic time. We describe how to combine the symbolic execution and handle detection to detect control-flow errors. Finally, we show how to compute whether two edges are sometimes-concurrent, which has separate applications such as data-flow analysis.

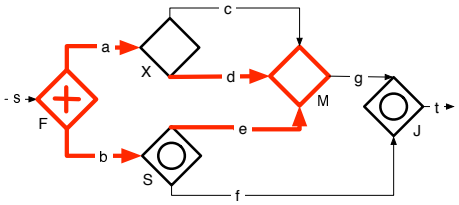

Fig. 5. A workflow graph that contains a lack of synchronization.

4.1 Handles and lack of synchronization

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

Definition 6 (Path with an AND-XOR or an 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] *if 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 a handle* from n_0 to n_i. *We speak of a* path with an IOR-XOR handle *if n_0 is an IOR-split and n_i is an XOR-join. We speak of a* path with an AND-XOR handle *if n_0 is an AND-split, and n_i is an XOR-join. In the rest of this document, we use handle instead of path with an AND-XOR handle or path with an 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.*

Definition 7 (Minimal handle). *A handle h from n_0 to n_i is* minimal *iff there exists no other handle h' from n'_0 to n'_j such that $n'_j < n_i$, or $n_i = n'_j$ and $n_0 < n'_0$.*

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

Theorem 3. *In an acyclic workflow graph that contains no deadlock, there is a lack of synchronization iff there is a handle.*

Proof. Let an acyclic workflow graph $W = (N, E, c, l)$ that does not contain any deadlock.

$\Rightarrow$

We show that if there is a lack of synchronization in W, then there is a handle in W.

Assume that there is a lack of synchronization in W, i.e., there is a reachable marking m^* such that an edge is marked with more than one token in m^*.

Therefore, there exists a reachable marking m such that there is an edge $e \in E$ that a is marked with more than one token in m and there exists no other marking m' such that an edge $e' \in E$ precedes e and is marked with more than one token in m'.

By the definition of reachable marking, there exists an execution σ of W such that $m \in \sigma$.

By the semantics of workflow graphs, the source node n of e is an XOR-join. n cannot be another type of node as if it was of another type n would have been executed twice and thus a reachable marking m' would exist with an edge e' preceding e such that e' is marked with more than one token in m'.

By the semantics of workflow graphs and as none of the edges preceding e can be marked with more than one token, there has to be two edges $e_1, e_2 \in {}^\circ n$ that are marked with one token in σ.

We use σ to define two paths p_1 and p_2 in W such that every edge in p_1 or p_2 is marked during σ. The path p_1 and p_2 start at the source edge of W. The path p_1 ends with $\langle e_1, n\rangle$, p_2 with $\langle e_2, n\rangle$. We define p_1 recursively from e_1, p_2 recursively from e_2.

Each node $n_k \in p_1$ is defined by the edge e_k that follows n in p_1 such that n_i is the source of e_i. Each edge $e_{k-1} \in p_1$ is defined by the node n_k that follows e_{k-1} in p_1 as follows:

$$e_{k-1} = \begin{cases} {}^\circ n_k & \text{if } n_k \text{ is an XOR-split, IOR-split, or an AND-split, (1)} \\ e_n & \text{if } n_k \text{ is an XOR-join or an IOR-join, } e_n \in {}^\circ n_k, \text{ and there exists a} \\ & \text{marking } m_n \text{ in } \sigma \text{ such that } e_n \text{ is marked in } m_n, \text{ (2)} \\ e_n & \text{if } n_k \text{ is an AND-join and } e_n \in {}^\circ n_k. \text{ (3)} \end{cases}$$

In case (1), *p_1 contains the only incoming edge of n_k. It follows from the workflow graph semantics that if an outgoing edge of n_k is marked during σ, then the incoming edge of n_k is marked during σ.*

In case (2), *p_1 contains e_n that was marked during σ. By the workflow graph semantics, e_n exists because there is a token on the outgoing edge of n_k.*

In case (3), *p_1 contains any of the incoming edges of n_k. By the workflow graph semantics, every incoming edge of n_k is marked during σ because the outgoing edge of the n_k is marked during σ.*

Each node and edge of p_2 is defined similarly to p_1 using e_2 instead of e_1 as basis for the recursion.

There exists a node f such that f belongs to p_1 and p_2, and there exists no node, other than n, that follows f and that belongs to p_1 and p_2.

It follows from the workflow graph semantics that f is an AND-split or an IOR-split because two of its outgoing edges get marked during σ and, by assumption, there is no edge preceding e that is marked with more than one token in any execution of W.

Thus, there exist two paths between the AND-split f and the XOR-join n that are disjoint aside from f and n. By Def. 6 they form a handle.

⇐

We show that if there is a handle in W, then there is a lack of synchronization in W.

Assume that there is a handle in W, i.e., there exist two paths $p = \langle n_0, e_0, ..., e_{n-1}, n_n \rangle$ and $p' = \langle n'_0, e'_0, ..., e'_{n-1}, n'_k \rangle$ such that n_0 is an AND-split or an IOR-split, n_n is an XOR-join, $n_0 = n'_0$, $n_n = n'_k$ and the two paths are disjoint aside from n_0 and n_n (Def. 6).

By the definition of workflow graphs, there is a path from the source edge to n_0. Thus, as there is no deadlock, a marking m_0 where the incoming edge of n_0 is marked is reachable in W.

By the semantics of workflow graph, n_0 can execute and its execution can result in a marking m_1 where e_0 and e'_0 are marked.

There exists a set of marking M reachable from m_1 such that for each marking $m \in M$ there exist an edge of p and an edge of p' that are marked in m.

We show that there exists a reachable marking in M such that e_{n-1} and e'_{n-1} are both marked.

We show for any m_i in M such that $m_i[e_{n-1}] = 0$ or $m_i[e'_{n-1}] = 0$, there exists a transition that changes m_i into a marking m_{i+1} that belongs to M.

As there is no deadlock, there is always at least one node n that is activated in m_i, such that $n \neq n_n$. Note that if n_n is activated in m_i, there exists another node n activated in m_i. This is because, by definition of m_i, there is an edge e_j marked prior to e_{n-1} or e'_{n-1} and, as there is no cycle, there is no path from n_n to the target t of e_j. Thus, the execution of t cannot require the prior execution of n_n and, as there is no deadlock, either t is activated in m_i or a node prior to t is activated in m_i.

By assumption on m_i, there exists an edge e_i on p and an edge e'_i on p' such that both are marked in m_i and either $e_i \neq e_{n-1}$ or $e'_i \neq e'_{n-1}$.

We distinguish three cases:

1. *$e_i \notin \circ n$ and $e'_i \notin \circ n$*
   Executing n in m_i results in a marking m_{i+1}. By the workflow graph semantics, e_i and e_i are both marked in m_{i+1} because they are not incoming edges of n.
2. *$e_i \in \circ n$ $e'_i \notin \circ n$*
   By the definition of path, as e'_i is in p, the target n is in p and at least one of its outgoing edge e_{i+1} is in p. By the workflow graph semantics, there exists a transition T such that $m_i \xrightarrow{T} m_{i+1}$ and e'_{i+1} is marked in m_{i+1}.
   By the workflow graph semantics, e'_i is marked in m_{i+1} because e'_i is not an incoming edge of n.
   Thus, m_{i+1} is in M.

3. $e_i \notin {\circ}n\ e'_i \in {\circ}n$
   A similar reasoning as for the case $e_i \in {\circ}n\ e'_i \notin {\circ}n$ *can be applied to show that there exists a transition* T *and a marking* $m_{i+1} \in M$ *such that* $m_i \xrightarrow{T} m_{i+1}$.

Because W is acyclic, M is a finite set and we cannot reach the same marking twice. Therefore, a marking m_t *such that* $m_t[e_{n-1}] = 1$ *or* $m_t[e'_{n-1}] = 1$ *is reachable.*

By the definition of workflow graph semantics: In marking m_t*, executing* n_n *once results in a marking* m'_t *where either* e_{n-1} *or* e_k *are marked. Thus* n_n *can also be executed in* m'_t*. Executing* n_n *twice results in the outgoing edge of* n_n *being marked with two tokens. Thus exists a reachable state of W where an edge is marked with more than one token, i.e., there is a lack of synchronization in W.* □

The outline of the 'only if' direction of the proof of Thm. 3 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. In Fig. 5, the handle corresponding to the lack of synchronization is highlighted. We say that the end node of the handle is the *location* of the lack of synchronization. Note that it is necessary that the workflow graph is deadlock-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.

Our notion of handles is similar to the one of Esparza and Silva [14] for Petri nets. If we restrict ourselves to workflow graphs without IOR gateways, one of the directions of our characterization follows from a result of Esparza and Silva [14]. The converse direction does not directly follow. Our notion of handles has been described by van der Aalst [13] who shows that, given a Petri net N, the absence of some type of 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 [15] 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 an approach is at best $O(|N|^3 \cdot |E| \cdot log(|N|)$.

4.2 Computing handles

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 [16] 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. We sketch our extension here while the details can be found in a separate report [17].

Perl and Shiloach [16] 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 which is a directed multi-graph. To do so, we transform the workflow graph into its *line graph*. A line graph G' of a graph G represents the adjacency between edges of G. Each edge of G becomes a node of G'. Additionally, we carry over those nodes from G to G' that can be start or end nodes of a handle, i.e., $S = \{x \mid x \in N \wedge x$ is an AND-split or an IOR-split$\}$ and $T = \{x \mid x \in N \wedge$ x is an XOR-join$\}$. The edges of G' are such that the adjacency in G is reflected in G'. For the workflow graph in Fig. 5, we obtain the line graph shown in Fig. 6. The line graph has two node-disjoint paths from an AND- or IOR-split to an XOR-join if and only if the workflow graph has a handle from that split to that join.

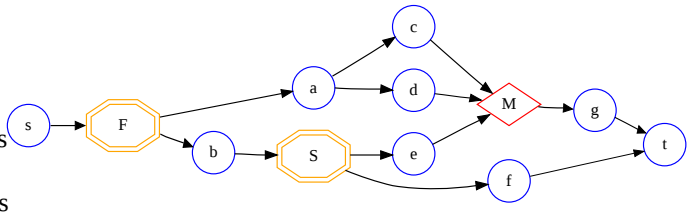

Fig. 6. The line graph for the workflow graph in Fig. 5.

To decide whether there are such two node-disjoint paths in the line graph, we can now apply the approach by Perl and Shiloach [16], which is the construction of a graph that we call the *state graph*. To this end, we extend the partial ordering $<$ of the nodes in the line graph to a total ordering $\prec$. A node of the state graph is a pair (n, m) of nodes of the line graph such that either $n = m \in S \cup T$ or $n \neq m$ and $n \prec m$. There is an edge in the state graph from (n, m) to (n', m) (or to (m, n')) if there is an edge from n to n' in the line graph.

d

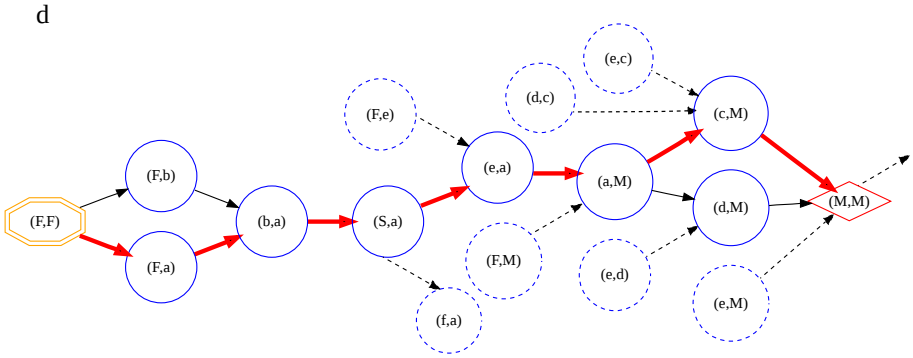

Fig. 7. A portion of the state graph for the line graph in Fig. 6

Figure 7 depicts a portion of the state graph for the line line graph in Fig. 6. We have two node-disjoint paths from an AND- or IOR-split s to an XOR-join j in the line graph if and only if there is a path from (s, s) to (j, j) in the state graph. In Fig. 7, one such path is highlighted which indicates two disjoint paths from the AND-split F to the XOR-join M. The number of edges in the state graph is in $O(|N| \cdot |E|)$ and the number of nodes is in $O(|N|^2)$ in terms of the line graph [16]. The entire algorithm can be implemented to run in quadratic time in the size of the workflow graph, cf. [17].

4.3 Combining symbolic execution with handle detection

Symbolic execution detects deadlocks in a prefix of the workflow graph that is free of lack of synchronization. Therefore, we first check the workflow graph for handles. We use the end nodes of the handles to delimit a maximum prefix of the workflow graph that is free of handles. We perform a symbolic execution of this prefix. If a deadlock is detected, we report the deadlock. If symbolic execution labels the incoming edges of the end node of a handle, we report the corresponding lack of synchronization. If no deadlock is detected and there is no handle detected, the workflow graph is sound.

4.4 Sometimes-concurrent

A data-flow hazard may arise if two conflicting operations on the same data object are executed concurrently. This can happen only if the tasks containing the data operations are sometimes-concurrent. A task of a process is represented as an edge in the corresponding workflow graph. Thus for the purpose of data-flow analysis, we are interested in detecting sometimes-concurrent edges for data-flow analysis.

Definition 8. *Two edges are* sometimes-concurrent *if there exists an execution in which they are parallel. They are* mutually-exclusive *or* never-concurrent *if they are not sometimes-concurrent.*

The notion of sometimes-concurrent edges is tightly related to lack of synchronization: It follows from the proof of Thm. 3 that two incoming edges e, e' of an XOR-join are sometimes-concurrent if and only if there is handle to this XOR-join such that one path goes through e and the other goes through e'. To decide whether two arbitrary edges of a sound graph are sometimes-concurrent, we show the following:

Lemma 5. *In a sound prefix of the workflow graph W, if two edges e_1, e_2 are sometimes-concurrent, then $e_1 \parallel e_2$.*

Proof. We this lemma by contradiction: Without loss of generality, assume that $e_1 < e_2$. As e_1 and e_2 are sometimes-concurrent, there exists a reachable marking m such that $m[e_1] = m[e_2] = 1$. As there is no deadlock, we can move the token on e_1 on the path to e_2 until reaching a marking m' such that $m'[e_2] = 2$. The marking m' is a lack of synchronization which is ruled out by the soundness assumption. □

by contradiction: Without loss of generality, assume that $e_1 < e_2$. As e_1 and e_2 are sometimes-concurrent, there exists a reachable marking m such that $m[e_1] = m[e_2] = 1$. As there is no deadlock, we can move the token on e_1 on the path to e_2 until reaching a marking m' such that $m'[e_2] = 2$. The marking m' is a lack of synchronization which is ruled out by the soundness assumption.

We can now determine whether two edges are sometimes-concurrent: Let W^* be the graph obtained by removing all the elements of the workflow graph that follow either e_1 or e_2 and add an XOR-join x to be the target of e_1 and e_2. The edges e_1 and e_2 are sometimes-concurrent if and only if x is the end node of a handle in W^*. As we can check for handles in quadratic time with respect to the size of the workflow graph, we obtain:

Theorem 4. *It can be decided in quadratic time in the size of the workflow graph whether a given pair of edges is sometimes-concurrent.*

Kovalyov and Esparza [18], propose a technique to detect sometimes-concurrent edges for sound workflow graphs that do not contain IOR logic in cubic time.

5 Dealing with Over-Approximation

In this section, we show how the labeling that is computed in the symbolic execution can be leveraged to deal with errors that are detected in the workflow graph but may not arise in a real execution of the process due to the correlation of data-based decisions.

5.1 User interaction to deal with over-approximation

When we capture the control-flow of a process in a workflow graph, we abstract from the data-based conditions that are evaluated when executing an XOR-split or an IOR-split of the process. Such a data-based decision can be, for example, isGoldCustomer(*client*). The data-abstraction may result in errors that occur in the workflow graph but not in an *actual* execution of the process. We use in the following the term *actual execution* to refer to an execution of the real process as opposed to its workflow graph, which is an abstraction of the process.

For example, the graph in Fig. 8 contains a deadlock located at J. However, if the data-based decisions in all actual executions are such that outcome d is taken whenever e is taken, this deadlock would never occur in an actual execution. For example, the data-based condition on d could be exactly the same as on e. The user should therefore have the opportunity to inspect the deadlock and decide whether outcomes d and e are related as mentioned above and then dismiss the deadlock. Analysis of the graph should then continue.

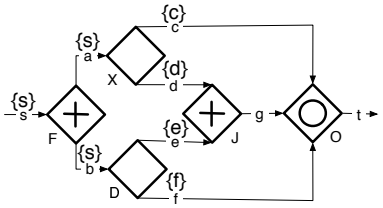

Fig. 8. A deadlock.

To inspect a deadlock, we provide the AND-join, two incoming edges e, e' of the join, and their non-equivalent labels $\varphi(e), \varphi(e')$ to the user. Then, she has to decide whether for each outcome $o \in \varphi(e)$ and each actual execution where o is taken, there is an outcome $o' \in \varphi(e')$ that is also taken in that execution and vice versa. If the user affirms the latter, she can dismiss the deadlock. This basically postulates the equivalence of the two symbols in actual executions. Henceforth, we continue the symbolic execution by treating, internally to the analysis, the AND-join as an IOR-join.

To inspect a lack of synchronization, we provide the XOR-join that terminates the detected handle and the two incoming edges e, e' of the XOR-join that are part of the handle to the user.

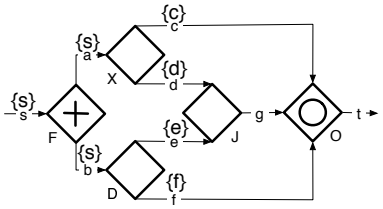

Fig. 9. A lack of synchronization.

Furthermore, we provide the labels $\varphi(e), \varphi(e')$. Then, the user has to determine that for each pair of outcomes $o \in \varphi(e)$ and $o' \in \varphi(e')$, we have that o is taken in an actual execution implies that o' is not taken in that execution. If the user affirms the latter, she can dismiss the lack of synchronization. This basically postulates that o and o' are mutually-exclusive in actual executions. If this is done for all incoming edges of the XOR-join, we can henceforth continue the symbolic execution by treating, internally to the analysis, the XOR-join as an IOR-join. Figure 9 shows an example with a lack of synchronization located at J. The user may dismiss it because for example, the conditions on c and e are the same, i.e., d and e are mutually-exclusive.

Figure 10 shows another example where the deadlock can be dismissed if b and c are deemed to be equivalent. Once the user dismissed the deadlock, we continue the symbolic execution and label the edge d with the symbol $\{b, c\}$ according to the IOR-join propagation rule. To dismiss the lack of synchronization at M, the user then has to check the pair a, b and the pair a, c for mutual exclusion.

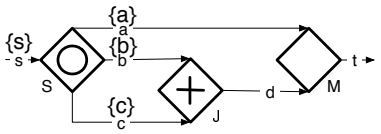

Fig. 10. A deadlock and a lack of synchronization.

The deadlock displayed on Fig. 4, can be dismissed if g is equivalent to s, i.e., g is deemed to be marked in every execution of the process.

Note that, if we provided an execution, i.e., an error trace, rather than the symbolic information to dismiss an error, we would present exponentially many executions that contain the same error in the worst case. The analysis of the outcome sets precisely gives the conditions under which one deadlock or one lack of synchronization occurs. It does not contain information that is irrelevant for producing the error.

5.2 Relaxed soundness

In some cases, the user should not be allowed to dismiss an error. Figure 11 shows a deadlock that cannot be avoided unless d and e are never taken which clearly indicates a modeling error. This is related to the notion of *relaxed soundness* [9]. A workflow graph is *relaxed sound* if for every edge e, there is a *sound execution* that marks e, where an execution is *sound* if no interleaving equivalent execution neither reaches a deadlock nor a lack of synchronization.

The graph in Fig. 11 is not relaxed sound. We do not know any polynomial-time algorithm to decide relaxed soundness for acyclic workflow graphs. However, we provide here necessary conditions for relaxed soundness that can be checked in polynomial time.

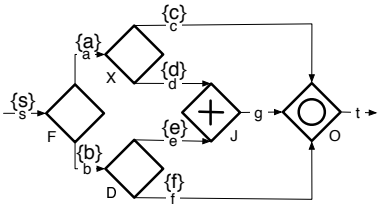

Fig. 11. A deadlock located at J that should not be dismissed.

One necessary condition for relaxed soundness is that for every AND-join J and every pair of incoming edges e, e' of J, e and e' are sometimes-concurrent. Likewise, for every XOR-join J and every pair of incoming edges e, e' of J, e and e' must not be always-concurrent. Moreover, we have the following stronger necessary conditions:

Theorem 5. *Let W be an acyclic workflow graph.*

1. *If for an AND-join J, and a pair of incoming edges e, e' of J and one outcome $o \in \varphi(e)$, we have that all outcomes $o' \in \varphi(e')$ are mutually-exclusive with o, then W is not relaxed sound.*
2. *If for an XOR-join J, and a pair of incoming edges e, e' of J, we have $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$, then W is not relaxed sound.*

Proof.

1. *Assume that there exist an AND-join J, and a pair of incoming edges e, e' of J and one outcome $o \in \varphi(e)$, we have that all outcomes $o' \in \varphi(e')$ are mutually-exclusive with o. When o carries a token during an execution σ, then e carries a token by Lemma 1. As the outcomes in $\varphi(e')$ are mutually exclusive with o, e' is not marked during σ. By Proposition 2, there is a deadlock located at J during σ. Thus, there exists no sound execution that marks o, i.e., W is not relaxed sound.* □
2. *Assume that for an XOR-join J, and a pair of incoming edges e, e' of J, we have $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$. Then, there exists an outcome $o \in \overline{\varphi}(e) \cap \overline{\varphi}(e')$. By Lemma 1, when o is marked during an execution σ, e and e' get marked during σ. And therefore there exists an exectution σ' that is interleaving equivalent to σ which leads to a lack of synchronization. Thus, there exists no sound execution that marks o, i.e., W is not relaxed sound.* □

Based on the previous results in this paper, we can compute these necessary conditions for relaxed soundness in polynomial time. If one of them is true, the corresponding error should not be dismissible. For example, the deadlock in the workflow graph depicted by Fig. 11 cannot be dismissed because d and e are mutually-exclusive. The lack of synchronization located at J in the workflow graph depicted by Fig. 12 cannot be dismissed because $\overline{\varphi}(d) = \{d\}$ and $\overline{\varphi}(b) = \{s, c, d\}$ and thus $\overline{\varphi}(e) \cap \overline{\varphi}(e') \neq \emptyset$.

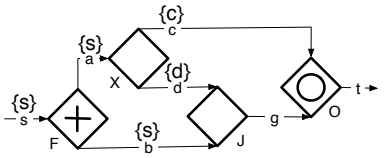

Fig. 12. A lack of synchronization that should not be dismissed.

Note that, deciding soundness and relaxed soundness complement each other. If we only decided relaxed soundness, we would not detect the deadlock that may be present in an actual execution of Fig. 8 for example.

6 Conclusion

We have shown how basic relationships between control-flow edges of a process can be decided in polynomial time for acyclic workflow graphs with inclusive OR gateways. This has various applications, for example, to detect control-flow errors, to perform data-flow analysis, or to compare processes at a behavioral level. Moreover, we have proposed a control-flow analysis that decides soundness in quadratic time and gives concise error information that precisely characterizes a single error. We outlined how

the diagnostic information can be used to efficiently dismiss spurious errors that may not occur in actual executions of the process due to correlated data-based decisions.

Note that, to increase the applicability of this approach, we can combine it with workflow graph parsing using the Refined Process Structure Tree [19], which allows us to decompose the workflow graph into fragments and to analyze each fragment in isolation (see [5] for details). Thus, our approach can be used to analyze every acyclic fragment of a cyclic workflow graph. However, it has to be worked out how the user interaction proposed in Sect. 5 can be extended to that class. Some cyclic fragments can be analyzed using suitable heuristics [20] which can be applied in linear time. Moreover, we would like to extend symbolic execution to cyclic workflow graphs in future work.

References

1. Mendling, J.: Empirical Studies in Process Model Verification. T. Petri Nets and Other Models of Concurrency (ToPNoC) **2** (2009) 208–224
2. Weidlich, M., Polyvyanyy, A., Mendling, J., Weske, M.: Efficient Computation of Causal Behavioural Profiles using Structural Decomposition. Technical Report BPT 10, HPI (2010)
3. Desel, J., Esparza, J.: Free choice Petri nets. Volume 40 of Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge (1995)
4. Esparza, J.: Reduction and synthesis of live and bounded free choice Petri nets. Information and Computation **114**(1) (October 1994) 50–87
5. Fahland, D., Favre, C., Jobstmann, B., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous soundness checking of industrial business process models. In: BPM. Volume 5701 of LNCS., Springer (2009) 278–293
6. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. Inf. Syst **25**(2) (2000) 117–134
7. Mendling, J.: Detection and Prediction of Errors in EPC Business Process Models. PhD thesis, Vienna University of Economics and Business Administration, Vienna, Austria (2007)
8. Wynn, M., Verbeek, H., Aalst, W., Hofstede, A., Edmond, D.: Business process verification-finally a reality! Business Process Management Journal **15**(1) (2009) 74–92
9. Dehnert, J., Rittgen, P.: Relaxed soundness of business processes. In: CAiSE. Volume 2068 of LNCS., Springer (2001) 157–170
10. Martens, A.: On compatibility of web services. Petri Net Newsletter **65** (2003) 12–20
11. Favre, C., Völzer, H.: Symbolic execution of acyclic workflow graphs. In Hull, R., Mendling, J., Tai, S., eds.: BPM. Volume 6336 of Lecture Notes in Computer Science., Springer (2010) 260–275
12. Völzer, H.: A new semantics for the inclusive converging gateway in safe processes. This volume
13. van der Aalst, W.: Workflow verification: Finding control-flow errors using Petri-net-based techniques. Lecture Notes in Computer Science (2000) 161–183
14. Esparza, J., Silva, M.: Circuits, handles, bridges and nets. Advances in Petri nets **483** (1990) 210–242
15. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM **35**(4) (1988) 921–940
16. Shiloach, Y., Perl, Y.: Finding two disjoint paths between two pairs of vertices in a graph. Journal of the ACM (JACM) **25**(1) (1978) 1–9
17. Favre, C.: An efficient approach to detect lack of synchronization in acyclic workflow graphs. In: ZEUS. Volume 563 of CEUR Workshop Proceedings. (2010) 57–64

18. Kovalyov, A., Esparza, J.: A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs, IEE
19. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. Data Knowl. Eng. **68**(9) (2009) 793–818
20. Vanhatalo, J.: Process Structure Trees: Decomposing a Business Process Model into a Hierarchy of Single-Entry-Single-Exit Fragments. PhD thesis, Universität Stuttgart (2009)