Complex Systems Informatics and Modeling Quarterly (CSIMQ)
eISSN: 2255-9922
Published online by RTU Press, https://csimq-journals.rtu.lv
Article 154, Issue 27, June/July 2021, Pages 1–44
https://doi.org/10.7250/csimq.2021-27.01

# Control-Flow-Based Methods to Support the Development of Sound Workflows

Thomas M. Prinz[1]* and Wolfram Amme[2]

[1]Course Evaluation Center, Friedrich Schiller University Jena, Am Steiger 3, Haus 1, 07743 Jena, Germany
[2]Research group Program Analysis and Optimization, Friedrich Schiller University Jena, Ernst-Abbe-Platz 2, 07743 Jena, Germany

Thomas.Prinz@uni-jena.de, Wolfram.Amme@uni-jena.de

**Abstract.** Workflows describe sequences of tasks to achieve goals. These sequences can contain decisions, loops, and parallelisations and are, therefore, similar to computer programs. Experts in the domain of workflow application usually design these workflows. However, these experts are rarely IT experts. For this reason, after automation by a computer, workflows can exhibit undesired behaviors. Such behaviors can be expensive and dangerous and should be avoided. The notion of soundness describes the absence of the undesired behaviors of deadlocks and abundances. The state of the art in workflow verification can detect such behaviors, but gives no indication of causes, does not provide detailed diagnostic information, or is slow. This article introduces two new compiler-based techniques to find causes of deadlocks and abundances. These techniques provide detailed diagnostic information and have a cubic asymptotic complexity of runtime. Their efficiency and quality is evaluated using a benchmark of over thousand real-world workflows together with two leading state-of-the-art approaches.

**Keywords:** Workflow, Verification, Soundness, Causes.

## 1 Introduction

Processes are omnipresent, whether in public authorities, companies, hospitals, or in everyday life. In many cases, it is profitable to recognize, observe, and write down such processes. This contributes to speeding up public authorities, securing corporate goals, and saving the lives of patients in hospitals. When we formally describe processes, they are called *workflows* and they mainly define a sequence of different tasks to achieve a goal [1], [2].

This article uses the following example of a (simplified) process of treating a patient in a hospital: A patient enters a hospital with complaints. First, the patient is examined by a doctor. On the basis of this examination, the doctor decides whether the treatment is simple or not. If it is simple, the doctor will take care of the patient immediately.

* Corresponding author

Reference: T. M. Prinz and W. Amme, "Control-Flow-Based Methods to Support the Development of Sound Workflows," Complex Systems Informatics and Modeling Quarterly, CSIMQ, no. 27, pp. 1–44, 2021. Available: https://doi.org/10.7250/csimq.2021-27.01

Additional information. Author ORCID iD: T. M. Prinz – https://orcid.org/0000-0001-9602-3482. PII S225599222100154X. Received: 23 April 2021. Accepted: 21 July 2021. Available online: 30 July 2021.

Otherwise, the doctor will inform the senior physician and start treatment at the same time to alleviate the symptoms. In the meantime, the senior physician reviews the treatment and the complaints. If other treatments are required, the senior physician informs the doctor, who then re-examines the patient. Otherwise, in the simple case, the senior physician will confirm this to his colleague. During the last examination, the patient is examined again and the doctor writes the patient's discharge papers. The patient leaves the hospital.

A *domain* expert can formalize this process into a workflow in the modeling language Business Process Model and Notation (BPMN) 2.0 [3]. Figure 1 illustrates the workflow. In most modeling languages, the workflow is a graph of nodes and edges, with nodes having different semantics. There are start and end events (circles with thin and thick lines) and tasks (rectangles) as well as decision nodes (diamonds with crosses) and parallelism nodes (diamonds with plus signs). In the figure, roles are also shown as lanes with the labels "patient", "doctor", and "senior physician".

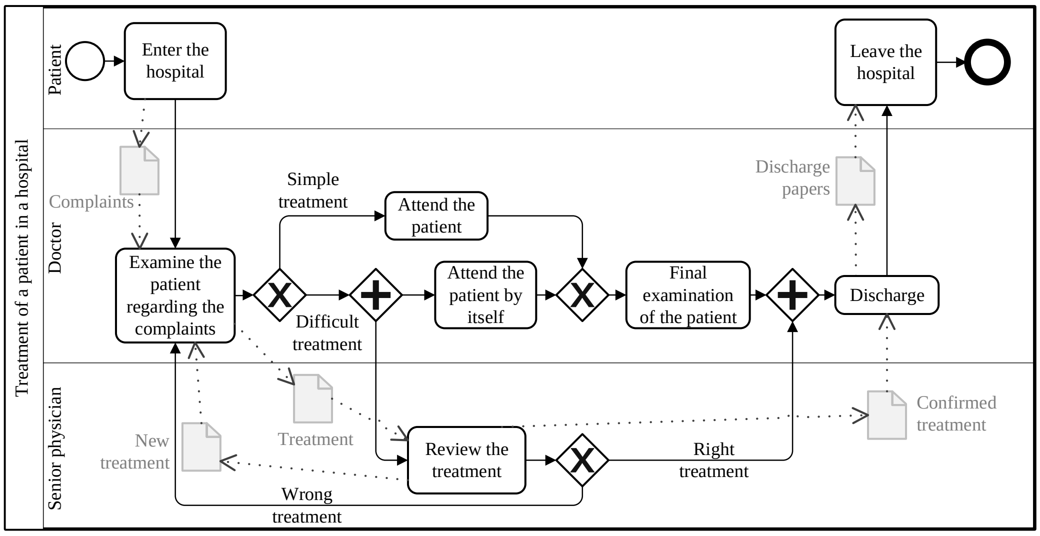


**Figure 1.** A workflow in BPMN

Usually, *domain* experts are not *IT* experts. For this reason, undesired behavior can occur in extracted workflows when automated by a computer. Finding these undesired behaviors in complex workflows is difficult. But in the worst case they can lead to false or wrong government documents, the non-attainment of business goals or loss of human lives. Undesired behavior is expensive and dangerous and should be avoided [4]. This article considers such undesired behaviors and recognizes them *before* they cause harm.

The state of the art considers different behavioral failures in workflows that lead to different notions of correctness. The notion used in this article — the *soundness* — describes the absence of (local) *deadlocks* and *abundances* [5], [6]. In an abundance, the workflow is in a situation where the same task can be performed undesirably more than once in a row. A deadlock blocks partly the execution of a workflow so that the workflow cannot be successfully terminated.

The workflow of Figure 1 contains undesired behavior in the form of abundances and deadlocks and is unsound. A deadlock occurs, for instance, when the treatment of the patient is simple. Figure 2 illustrates this fact on a simplified version of the workflow — the execution (represented by a bold grey path) is blocked in front of the right diamond with a plus sign. This happens since this node tries to synchronize the parallelism, which, however, does not exist.

To avoid undesired behavior in workflows, various techniques can be applied to determine deadlocks and abundances. In general, they take into account situations that can be reached in workflows. But sometimes any number of such situations are possible, so some of these techniques do not finish [4]. The more crucial problem, however, is that most techniques consider only the *errors* instead of the *faults* [7], [8]. In other words, the techniques find only deadlocks or abundances instead of the reason "*why*" they happen. But we need that *why* to locate the modeling failures and repair the workflow. The derivation of the fault from an error is difficult due to the *fault distance* known from software quality. Also, algorithms cannot find all errors since they are hidden by previous errors. For instance, a deadlock can *block* the reaching of another error [9]. In other situations, unnoticed errors can repair and *mask* another error [7], which is therefore hidden. Sometimes the fault cannot be determined, because the undesired behavior does not seem to fit the fault, e. g., a deadlock was detected whose origin is an abundance fault, *fault illusion* [10]. The interested reader can learn more about fault blocking, masking, distance, and illusion in previous work [11].

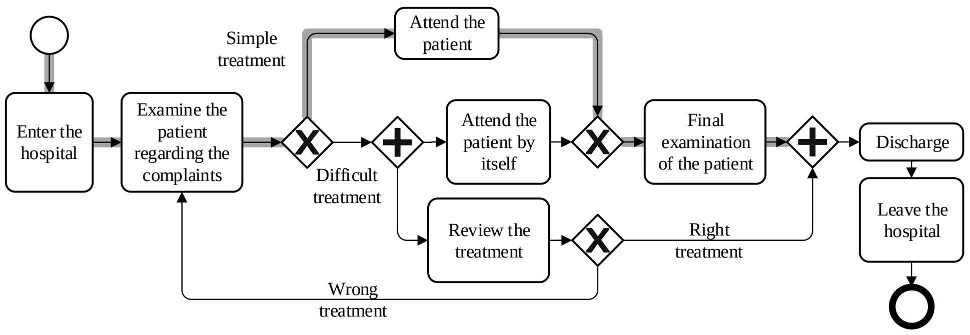


**Figure 2.** A deadlock (grey path) in the example workflow

If the soundness checkers only detect errors such as deadlocks and abundances instead of their faults: *How* should a developer know, *what* is the reason for undesired behavior and *where* is its cause? This article answers these questions for the first time to the best of the authors' knowledge. It shows that workflows are sound if they have neither *causes* of abundances nor deadlocks. In addition, the article presents two algorithms that find causes of abundances and deadlocks in cyclic workflows. These algorithms have the following properties:

1. Their runtime is $O(E^3)$ with $E$ being the number of edges of the workflow.
2. They provide accurate and detailed diagnostic information.
3. They detect causes of deadlocks and abundances behind other causes of deadlocks and abundances.

This article has the following structure: After the introduction, Section 2 repeats the basics of workflows. Section 3 gives an overview of the state of the art in soundness checking. Subsequently, Section 4 explains partial analyses of workflows. Section 5 examines causes of deadlocks, while Section 6 focuses on causes of abundances. Section 7 introduces our analysis tool *Mojo*. This tool is then used in an evaluation (Section 8). The article ends with a conclusion and a look at the future work in Section 9.

## 2 Preliminaries

This section introduces the notions used in this article.

## 2.1 Multisets

A *multiset* describes a set-like construct where each element can appear more than once. Each *multiset* $M$ over a set $S$ is a total function from $S$ to the natural numbers $\mathbb{N}$ ($M\colon S \mapsto \mathbb{N}$). Therefore, $M$ applies a natural number to each element of $S$ that describes the absolute frequency of each element in $M$. We write $\langle m_0, \dots, m_k\rangle, k \geq 0$, $\{m_0, \dots, m_k\} \subseteq S$ [12]. For instance, $M' = \langle a, a, b, b, b, d\rangle$ is a multiset over $\{a, b, c, d\}$ with $M'(a) = 2$, $M'(b) = 3$, $M'(c) = 0$, and $M'(d) = 1$. The *support* $\mathit{Support}(M)$ of $M$ is defined as $\{m \in S\colon M(m) \geq 1\}$. It describes the set of all elements of $S$ that appears at least once in $M$. Each $m \in \mathit{Support}(M)$ is also called *element* of $M$, $m \in M$. As for the last example, $\mathit{Support}(M') = \{a, b, d\}$ is the support of $M'$ and $a, b, d \in M'$, $c \notin M'$.

Multisets can be used like normal sets. The operations $\subseteq$, $\cap$, $\cup$, and $\setminus$ are naturally defined as follows (inspired by Rosen [13] and Reisig [12]):

$$\begin{aligned}
M \subseteq O &\iff \forall m \in \mathit{Support}(M)\colon M(m) \leq O(m) && O \text{ is a multiset}\\
M \cap O = C &\iff \forall m \in \mathit{Support}(M)\colon C(m) = \min(M(m), O(m)) && O, C \text{ are multisets}\\
M \cup O = V &\iff \forall m \in (\mathit{Support}(M) \cup \mathit{Support}(O))\colon V(m) = M(m) + O(m) && O, V \text{ are multisets}\\
M \setminus O = K &\iff \forall m \in \mathit{Support}(M)\colon K(m) = \max\left(M(m) - O(m), 0\right) && O, K \text{ are multisets}
\end{aligned}$$

Operations between multisets and ordinary sets are easily possible by converting the sets to multisets before applying an operation. This is done implicitly throughout this article. For the two multisets $M_1 = \langle a, a, b, b, b, c\rangle$ and $M_2 = \langle a, a, a, b, b, b, b, b, c, d, d\rangle$, the introduced operations result in:

$$\begin{aligned}
&M_1 \subseteq M_2 \quad M_2 \nsubseteq M_1 && M_1 \cap M_2 &&= M_1 = \langle a, a, b, b, b, c\rangle\\
&M_1 \cup M_2 \quad = \langle a, a, a, a, a, b, b, b, b, b, b, b, b, c, c, d, d\rangle && M_2 \setminus M_1 &&= \langle a, b, b, d, d\rangle
\end{aligned}$$

## 2.2 Graphs and Paths

A *directed graph* (digraph) $G$ consists of a set $N = N(G)$ called the *nodes* of $G$, and a set $E = E(G)$ of ordered pairs, $E \subseteq N \times N$, called the *edges* of $G$ [13], [14], [15]. We write $G = (N, E)$. For an edge $e = (s, t) \in E$, $s$ is the *source* of $e$, $s = \mathit{src}(e)$, and $t$ is the *target* of $e$, $t = \mathit{tgt}(e)$. The *incoming edges* of a node $n$ define the set of edges with $n$ as target: $\triangleright n = \{(p, n) = e \in E\}$. Similarly, $n\triangleleft = \{(n, s) = e \in E\}$ describes all *outgoing edges* of $n$.

A *path* $P$ of $G$ is a sequence $(e_0, \dots, e_m)$, $m \geq 0$, of edges of $E$, where each target of an edge corresponds to the source of the next edge in sequence: $\forall 0 \leq i < m\colon \mathit{tgt}(e_i) = \mathit{src}(e_{i+1})$ [16]. An edge $e$ is an element of $P$ if it is part of $P$'s sequence, $e \in P$. The *length* of $P$ is the length of the sequence, $|P| = m$. The notion $P_{a \to b}$ is used to describe a path from $a$ to $b$. In this context, $\mathcal{P}_{a \to b}$ describes the set of *all* paths from $a$ to $b$. Each *path*, in which an edge appears twice, has a *loop*. A graph is called *cyclic* if there is at least one path with a loop. Otherwise, the graph is called *acyclic*.

*Control-flow graphs* are a special subset of all graphs (e. g., Zima et al.[17]). A *control-flow graph* $CFG = (N, E)$ is a digraph $(N, E)$ with the following properties:

1. There is exactly one *start node* $n_{\mathit{Start}}$ without any incoming edge, $\triangleright n_{\mathit{Start}} = \emptyset$, but with exactly one outgoing edge, $n_{\mathit{Start}}\triangleleft = \{e_{\mathit{Start}}\}$, the *start edge*.
2. There is exactly one *end node* $n_{\mathit{End}}$ with exactly one incoming edge, $\triangleright n_{\mathit{End}} = \{e_{\mathit{End}}\}$, the *end edge*, but without any outgoing edge, $n_{\mathit{End}}\triangleleft = \emptyset$.
3. Each edge $e \in E$ lies on a path from $e_{\mathit{Start}}$ to $e_{\mathit{End}}$: $\quad \forall e \in E\colon \quad \exists P \in \mathcal{P}_{e_{\mathit{Start}} \to e_{\mathit{End}}}\colon e \in P$

An example control-flow graph is shown in Figure 3. It consists of the nodes $n_1, n_2, \dots, n_{15}$ and the edges $(n_1, n_2), (n_2, n_3), \dots$ between these nodes. The start node is $n_1$ and the end node is $n_9$. The start edge is $(n_1, n_2)$ and the end edge is $(n_8, n_9)$. Obviously, each edge lies on a path of $\mathcal{P}_{(n_1,n_2) \to (n_8,n_9)}$.

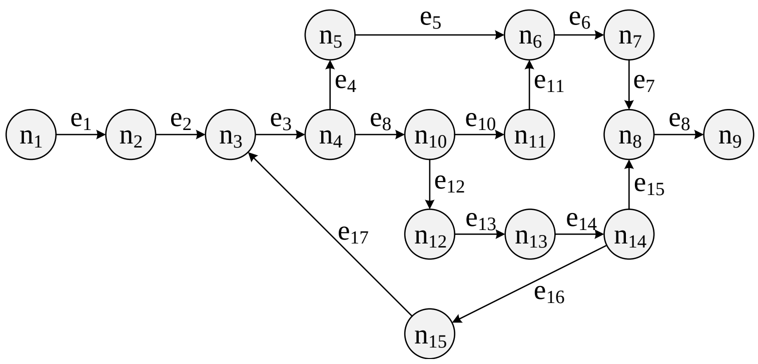


**Figure 3.** A control-flow graph

## 2.3 Workflow Graphs

*Workflow graphs* are simplifications of complex *workflows* for control-flow analysis. They were introduced by Sadiq and Orlowska [5] and later formalized by other researchers. The definition in this article is based on control-flow graphs:

**Definition 1 (Workflow graph).** A *workflow graph WFG* consists of a control-flow graph $(N, E)$ and a total function

$$l\colon N \mapsto \{\mathit{Start}, \mathit{End}, \mathit{Task}, \mathit{Split}, \mathit{Merge}, \mathit{Fork}, \mathit{Join}\} \tag{1}$$

that assigns a *label* to each node. The label defines the kind of node and its semantics. A workflow graph is written as $\mathit{WFG} = (N, E, l)$. The set of nodes consists of distinct subsets:

$$N = \{n_{\mathit{Start}}, n_{\mathit{End}}\} \cup N_{\mathit{Task}} \cup N_{\mathit{Split}} \cup N_{\mathit{Merge}} \cup N_{\mathit{Fork}} \cup N_{\mathit{Join}} \tag{2}$$

All nodes of the same set have the same label:

$$\begin{aligned} l(n_{\mathit{Start}}) &= \mathit{Start}, \text{ the } \textit{start node} & l(n_{\mathit{End}}) &= \mathit{End}, \text{ the } \textit{end node} \\ \forall n \in N_{\mathit{Task}}\colon l(n) &= \mathit{Task}, \text{ the } \textit{task nodes} & \forall n \in N_{\mathit{Split}}\colon l(n) &= \mathit{Split}, \text{ the } \textit{split nodes} \\ \forall n \in N_{\mathit{Merge}}\colon l(n) &= \mathit{Merge}, \text{ the } \textit{merge nodes} & \forall n \in N_{\mathit{Fork}}\colon l(n) &= \mathit{Fork}, \text{ the } \textit{fork nodes} \\ \forall n \in N_{\mathit{Join}}\colon l(n) &= \mathit{Join}, \text{ the } \textit{join nodes.} \end{aligned}$$

Furthermore, *WFG* has the following properties:

1. Each task node of $N_{\mathit{Task}}$ has exactly one incoming and exactly one outgoing edge.
2. Each split and fork node of $(N_{\mathit{Split}} \cup N_{\mathit{Fork}})$ has exactly one incoming and at least two outgoing edges.
3. Each merge and join node of $(N_{\mathit{Merge}} \cup N_{\mathit{Join}})$ has at least two incoming edges and exactly one outgoing edge.

Each kind of node has a special symbol if the workflow graph is visualized (cf. Figure 4). Start and end nodes are illustrated as (bold) circles. Unfilled rectangles mark task nodes. Split and merge nodes are (bold) diamonds. Filled rectangles indicate fork and join nodes. They can be distinguished by their numbers of incoming and outgoing edges. Although both split and merge nodes and fork and join nodes have similarities, they differ in their behavior, i. e., split and merge nodes process decisions in workflow graphs and fork and join nodes create and consume parallelism.

The behavior of workflow graphs is defined by the labels of the nodes. It is a common standard to use Petri net semantics to mathematically define the semantics of workflow graphs. The following definitions (Definition 2 to Definition 5) are based on the work of Vanhatalo et al. [18] and Völzer [19]. In the first place, the semantics consider execution situations — *states* — and the transitions between these situations.

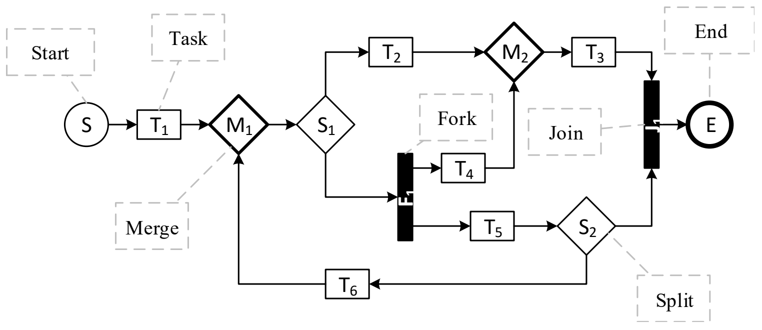


**Figure 4.** A workflow graph with its different node visualizations

**Definition 2 (State).** A *state* of a workflow graph $(N, E, l)$ is a multiset $S$ over the set of edges $E$. It assigns to each edge $e \in E$ a natural number $m$ of *tokens*, $m = S(e)$. The set of all (arbitrary) states over $E$ is designated with $\mathcal{S}(E)$.

An edge $e$ of a workflow graph *carries*, *owns*, or simply *has* a token in a state $S$ if $S(e) \geq 1$. Tokens are represented as filled black circles at the edges. For instance, the workflow graph in Figure 5 is currently in the state $S = \langle (T_4, M_2), (S_2, T_6) \rangle$. Special cases of states are the *initial state*, in which only the start edge $e_{Start}$ carries exactly one token, and the *termination state*, in which only the end edge $e_{End}$ has exactly one token. If nodes have tokens at their incoming edges, they can be *executable*:

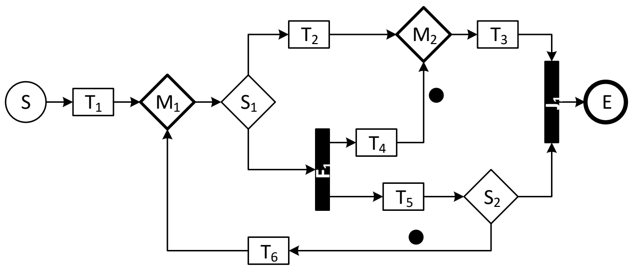


**Figure 5.** A workflow graph in a state with tokens at edges $(T_4, M_2)$ and $(S_2, T_6)$

**Definition 3 (Executability).** Let $(N, E, l)$ be a workflow graph in a state $S$. The start and end nodes are never executable. All other nodes $n \in N \setminus \{n_{Start}, n_{End}\}$ are *active* if they have at least one incoming edge that carries a token. They are *executable* in $S$ iff either 1) $n$ is active and not a join node, or 2) all of $n$'s incoming edges have at least one token:

$$n \text{ is executable in } S \iff \left(n \notin N_{Join} \wedge \triangleright n \cap S \neq \emptyset\right) \vee \left(\triangleright n \subseteq S\right).$$

$\mathcal{E}xec(S)$ is the set of all executable nodes in state $S$:

$$\mathcal{E}xec(S) = \{n \in N \colon n \text{ is executable in } S\}.$$

The nodes $T_6$ and $M_2$ of the workflow graph in Figure 5 are currently executable in the state $S$, $\mathcal{E}xec(S) = \{M_2, T_6\}$. The join node $J_1$, however, can only be executed in any state $S' \supseteq \{(T_3, J_1), (S_2, J_1)\}$. Therefore, the executability of join nodes is specific for workflow graphs. If a node is executable, the state can transit to another state:

**Definition 4 (State transitions).** Let $S$ be a state of a workflow graph $WFG = (N, E, l)$ and $n \in N$ be an executable node. After executing $n$, *WFG changes* into $S'$, written $S \xrightarrow{n} S'$. The state $S'$ is defined for the node $n$ as follows:

$n \in (N_{Task} \cup N_{Fork} \cup N_{Join})$**:** From each incoming edge of $n$, one token is removed, and at each outgoing edge of $n$, an additional token is placed: $S' = (S \setminus \triangleright n) \cup n\triangleleft$

$n \in (N_{Split} \cup N_{Merge})$**:** From exactly one incoming edge *in* of $n$ with a token, one token is removed, and at exactly one randomly chosen outgoing edge *out*, an additional token is placed: $S' = (S \setminus \{in\}) \cup \{out\}, in \in (\triangleright n \cap S), out \in n\triangleleft$

For instance, either $T_6$ or $M_2$ can be executed in state $S$ of the workflow graph in Figure 5 non-deterministically. If the node $T_6$ is executed, $S$ changes to the state $S'$, $S \xrightarrow{T_6} S'$, $S' = \langle (T_4, M_2), (T_6, M_1) \rangle$.

Definition 3 and Definition 4 specify the semantics of the different kinds of nodes and thus of workflow graphs. In summary, it can be said that the start and end node have no semantics. They only mark where the execution begins and ends. With the exception of join nodes, all nodes can be executed if at least one incoming edge has at least one token. A task node would take a token from its incoming edge and place a token on its outgoing edge. Split and merge nodes make non-deterministic decisions. A split node takes a token from its incoming edge and puts it randomly on one of its outgoing edges. Merge nodes randomly take one token from one of their incoming edges (with at least one token) and place it on their outgoing edge. Fork nodes also take a token from their incoming edge, but set a token on each of their outgoing edges. Therefore, there is a parallelism after the execution of a fork. This parallelism can be synchronized by a join node. Such a join node can only be executed if each of its incoming edges has at least one token. During its execution, it picks a token from each of its incoming edges. It then places an additional single token on its outgoing edge.

The semantics of the different kinds of nodes follows the common semantics of workflow graphs in the literature (defined by Sadiq and Orlowska [5]). To simplify notions and proofs, the notion *reachability* is useful:

**Definition 5 (Reachability).** Let $(N, E, l)$ be a workflow graph. A state $S_{to}$ is *strictly reachable* from a state $S_{from}$ (written $S_{from} \to S_{to}$) if there is an executable node $n$ in $S_{from}$ whose execution ends in state $S_{to}$. $S_{to}$ is *reachable* from $S_{from}$ (written $S_{from} \to^* S_{to}$) if there is a sequence of states $S_0, \ldots, S_m$, $m \geq 1$, such that $S_0 \to S_1 \to \ldots \to S_{m-1} \to S_m$ and $S_0 = S_{from}$, $S_m = S_{to}$. The *state-space* of a workflow graph is defined by the strictly reachable relation.

## 2.4 Soundness

The *soundness* property defines a correctness criterion for workflow graphs [20]. It describes the absence of (local) *deadlocks* and *synchronization lacks* [5].

**Definition 6 (Deadlock).** A state $S_{dead} \neq S_{End}, S_{dead} \in \mathcal{S}(E)$, of a workflow graph $(N, E, l)$ is a *deadlock* if there is an active node $n$ in $S_{dead}$ that is not executable in each reachable state. That means, $n$ can only be a join node. We also say that $n$ has a deadlock in $S_{dead}$. This is called a *local deadlock* in Fahland et al. [4].

For instance, the workflow graph in Figure 6 *a)* is in a deadlock $\langle (T_3, J_1) \rangle$ because the join node $J_1$ is not executable.

**Definition 7 (Abundance).** A lack of synchronization describes a state $S_{abu} \in \mathcal{S}(E)$ of a workflow graph $(N, E, l)$, in which at least one edge carries more than one token. Since the term *lack of synchronization* describes the origin of the error situation and not the situation itself, we prefer the term *abundance* and will use it throughout the rest of this article.

The workflow graph in Figure 6 *b)* is present in an abundance since the edge $(M_2, T_3)$ has two tokens.

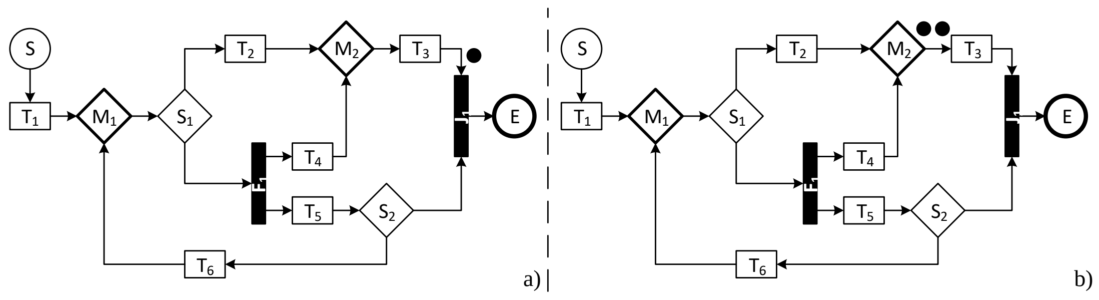


**Figure 6.** A workflow graph in a deadlock *a)* and an abundance *b)*

**Definition 8 (Soundness).** A workflow graph is *sound* if neither a deadlock nor an abundance is reachable from the initial state.

Our example workflow graph is *not* sound. The definition of soundness is valid only with the assumption of *fairness*. Fairness means, in our context, that non-deterministic decisions are made non-deterministically and, therefore, loops terminate in sound workflow graphs [6], [21], [22], [23]. Throughout the rest of this article, fairness is assumed.

## 3 State of the Art

The behavioral analysis of workflows has a long tradition and is meanwhile applicable to real workflows [24]. One of the best known behavioral notions is *soundness*, which was firstly introduced by Van der Aalst [25]. Soundness has evolved to this day leading to many variants, e. g., the *relaxed* or *weak* soundness (the reader can find a good overview of the variants in Puhlmann [26] and Van der Aalst et al. [27]). As already mentioned, this article considers the classical definition of soundness by Van der Aalst [6]. Van der Aalst defined the soundness on special Petri nets with exactly one start and exactly one end place — *workflow nets*. A workflow net is *sound* if and only if for each reachable state the termination state is reachable (there is no deadlock), the end place carries a token only in the termination state (there is no abundance) and no transition is dead.

With the *rank theorem* [28], the soundness property of workflow nets can be checked by considering the properties *liveness* and *boundedness* [6]. Liveness corresponds to the absence of deadlocks and boundedness belongs to the absence of abundances. The advantage of this approach is the application of well-known Petri net theory to workflows. Furthermore, the runtime behavior is cubic $O(N^3)$ with $N$ being the maximum number of places, transitions, and edges [29]. However, its disadvantage is the lack of diagnostic information to describe the violations of soundness [30]. For this reason, new models and approaches for the description and verification of workflows have been developed so far. They are briefly presented further in this section.

### 3.1 Formal Models

Workflow graphs [5] are the most noted formal model for describing workflows alongside workflow nets. They are similar to control-flow graphs, but allow explicit parallelism. Like workflow nets, workflow graphs abstract from the real workflow and contain only those parts that are important for describing the control-flow.

Van der Aalst has shown that every acyclic workflow graph has a semantically equivalent Free Choice Petri net [31]. Therefore, the rank theorem can also be applied to workflow graphs. Favre et al. later showed the equivalence between arbitrary workflow graphs and workflow nets [32]. However, workflow graphs are easier to extend by new control-flow elements (such as *OR*-join nodes) [32], whose semantics cannot be formalized by a token game that easy. This disadvantage of workflow nets motivated Van der Aalst and Ter Hofstede to develop the workflow modeling language Yet Another Workflow Language (YAWL) [33]. Starting from different business workflow modeling languages, they extended workflow nets with missing but often used control-flow elements such as cancellation patterns.

Chrzastowski-Wachtel et al. proposed a semantically similar way of cancellation patterns [34]. In their work, they also introduced a new method to describe workflows: their description as *trees*. In these trees, children of a node describe refinements similar to sub-processes. They create a hierarchical order of the nodes of a workflow. Chrzastowski-Wachtels et al. propose to construct a workflow by a stepwise refinement. If the developer applies sound structures in each step, then the resulting workflow is also sound. However, Chrzastowski-Wachtel et al. missed to show how an already existing workflow can be decomposed.

### 3.2 Soundness Check With Decomposition

A decomposition of a workflow into a tree was examined by Vanhatalo et al. [18] with the introduction of the Process Structure Tree (PST) as a result of decomposition into Single-Entry-Single-Exits (SESE). The advantage is that developers can create the workflows as usual and a tool derives the PST. The derived PST is hidden from the developer, but can be used for analysis. Vanhatalo et al. use simple rules and heuristics as analyses, which are unfortunately incomplete when parts of the tree (the SESE fragments) are unstructured [18]. Besides this disadvantage, SESE decomposition has many advantages: applied analyses lead to a high quality of the diagnostic information, arbitrary analyses are possible, and the decomposition is fast. It is possible to detect causes of the errors (faults) for each fragment instead of just the error. This is also possible for faults, which are never reached at runtime (e. g., because an earlier deadlock hinders them). Since each fragment is a workflow again, any analyses can be applied to it. Therefore, it is possible to use other approaches for unstructured fragments.

The asymptotic runtime behavior of the SESE decomposition depends on the time in which the workflow is decomposed and the PST is built. Both steps have their origin in the works of Johnson et al. [35], [36] about *Program Structure Trees* for ordinary computer programs. Ananian improved the methods of Johnson et al. for his *Single Static Information Form* [37]. For this reason, the approach of Vanhatalo et al. [18] is the first to consider workflows as control-flow graphs. They later refined the approach by finding more detailed fragments based on the 3-connectedness [38]. Such 3-connected fragments have exactly one incoming and one outgoing edge. They can be detected with the efficient linear-time algorithm of Hopcroft and Tarjan [39] for control-flow graphs. Since the detected fragments are more detailed than the original SESE fragments, a more detailed failure diagnosis is possible. It is also possible to rearrange workflows using these detailed fragments [40]. In addition, tools can automatically close incomplete fragments with a matching convergent node. The approach of Chrzastowski-Wachtel et al. [34] can have a revival. Kühne et al. [41] showed its practical benefit. It was possible to obtain diagnostic information during the construction of workflow graphs.

### 3.3 Model-Checking and Rank Theorem

The disadvantage of decomposition is its incompleteness. Model-checking is certainly the most popular alternative approach. For verification of workflows it was first used by Van der Aalst [42]. Model-checking describes an efficient exploration of the state-space of the workflow. The state-space contains all reachable states and the transitions between them. The fundamental problem of state-space exploration is its complexity. Even small graphs can have an unbounded state-space — a state-space explosion [4], [43]. There are approaches to limit the state-space. For instance, explorations replace unbounded growing numbers of tokens with an identity element. The resulting state-space is a *Coverability Tree* [23], whose construction is EXPSPACE-hard [44]. However, it halts and therefore always leads to diagnostic information. The diagnostic information consists of the state transitions to the first erroneous state. Lohmann and Fahland also tried to reduce the state-space [45]. Their goal was to explore the decisions of split nodes that lead to erroneous behaviors. Unfortunately, this approach was not pursued to the end.

For state-space explorations, there are two well-known tools: Woflan [46] and LoLA [47]. Verbeek et al. have developed Woflan and it is today the most complete tool for checking workflow nets. In addition to other quality criteria, it also checks soundness. For that, it tries to minimize the workflow net. If the resulting net is trivial, then it is sound. Otherwise, Woflan decides on the *S-coverability* whether there can be diagnostic information or not. S-coverability describes the ability to decompose the workflow net into S-components. S-components are minimal sets that contain all direct predecessors and successors of each place [28]. If decomposition is possible, the workflow net is free of abundances. Otherwise, the workflow net is unsound, however it is not known whether there is a deadlock or an abundance. Deadlocks are determined with the help of a state-space exploration for free-choice workflow nets. For this reason, Woflan also uses an EXPSPACE-hard algorithm. The work of Fahland et al. underlines this fact in a comparison of three approaches for soundness checking [4]: state-space exploration with LoLA, SESE decomposition with the *IBM WebSphere Business Modeler*, and S-coverability and state-space exploration with Woflan. Their main finding was that for most real workflows, soundness checking is possible in a short period of time. But the state-space-based approaches also showed that they cannot consider each of the 1,386 workflows because they took too much time or did not finished.

The mentioned tool LoLA is an analyzer for simple Petri nets. It is a general tool for model-checking and has not been developed specifically for soundness. LoLA accepts the properties to be tested as formal equations, e. g., the equation for finding abundances contains the condition that in at least one state at least one place has more than one token. Formal equations are checked by various reduction methods of the entire net and subsequent state-space explorations.

There are other approaches to soundness checking based on reduction methods. For instance, Sadiq and Orlowska [5] used reduction rules to find deadlocks and abundances. These rules eliminate start and end nodes, combine sequences of nodes to a single edge, etc. If the resulting graph has only one node, the workflow is sound. Otherwise, it is unsound. Although the asymptotic runtime behavior $O(E^2)$ seems promising, Van der Aalst showed that they are incomplete [31].

Eshuis and Kumar criticize the state-space exploration and rank theorem approaches for their poor fault localization [48]. Instead, they use *instance graphs* to show soundness, with instance graphs representing possible executions of a workflow graph [5]. The instance graphs are derived with Integer Programming and represent errors if the graph does not contain all of a join node's direct predecessors — a deadlock — or more than one direct predecessor of a merge node — an abundance. However, this approach is bound to acyclic workflow graphs, cannot find faults behind other faults, and has an exponential runtime. Although the implementation in *DiagFlow*[3] is faster than Woflan [46], the time spent is not acceptable for immediate fault feedback.

[3] http://is.ieis.tue.nl/staff/heshuis/DiagFlow/ (Mai 2019)

### 3.4 Pattern- and Compiler-Based Approaches

Approaches that take into account the structure of workflows such as compilers are alternatives to model-checking. The most famous approach is the SESE decomposition. However, the literature also knows other compiler-based approaches, as will be shown in the following.

For instance, Van Dongen et al. have defined two relations (referred to as *causal footprints*) [49], the first claiming for an executed node that each node in relation is executed too; and the second claiming that an executed node was executed after at least one node in relation. Based on these causal footprints, Dongen et al. defined three erroneous patterns for deadlocks, abundances, and non-terminating loops. Unfortunately, only for non-terminating loops was it shown that the workflow is unsound. For both other patterns, it is uncertain whether there is a deadlock or abundance.

Favre created another algorithm for soundness checking in acyclic workflow graphs on patterns and relations [30]. He defined an always-parallel-relation between the edges for deadlocks, which is determined by a kind of data-flow analysis that propagates the information throughout the workflow graph. The relation helps to know whether all incoming edges of a join node are always in parallel or not. If they are not always in parallel, a deadlock is possible. For abundances, he defines a perhaps-parallel-relation. If an abundance is impossible, all two incoming edges of merge nodes cannot be in that relation. Although the approach provides good diagnostic information and is possible in polynomial runtime complexity, limiting it to acyclic graphs is not practicable and the finding of faults behind other faults is not always possible.

Favre et al. proposed another approach based on anti-patterns [50]. The patterns are similar to those of Dongens et al. [49], however apply to cyclic workflows. Favre et al. show that the existence of such an anti-pattern makes a workflow unsound. If a workflow is unsound, a failure diagnostic starts. This diagnosis determines the subgraph, which belongs to the failure, using the rank theorem and other approaches. The runtime behavior of the approach is quintic, but the approach leads to very good diagnostic information. But it only finds one error per workflow.

## 4 Partial Analysis and Entry Points

The aim of this research is to find methods that provide diagnostic information about deadlocks and abundances in high quality and quantity. Figure 7 shows a workflow graph that can easily be divided into two subgraphs (visualized by the dashed boxes). Each execution of the left subgraph ends in a deadlock in the join node $J$. On the contrary, each execution of the right subgraph ends in an abundance at an edge after the merge node $M$. Any consideration of the workflow graph from the initial state, however, would only end in a deadlock, so that the abundance is not reachable. Since the abundance is reachable immediately after the repair of the left part of the workflow graph, it would be useful to know the abundance already at repair time. Classical state-space exploration fails in this task since it discovers only *reachable* states.

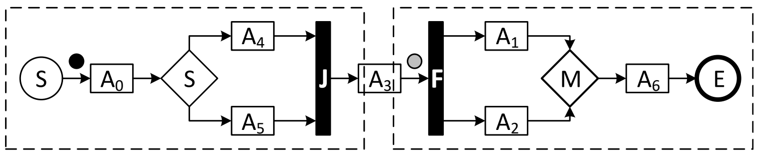


**Figure 7.** A workflow graph whose execution is blocked in the join node $J$

Our basic approach is to start the examination of workflow graphs (similarly to the SESE approach) from different points of the graph — the *entry points*. Unlike the SESE approach, an entry point could be any edge of the workflow. The advantage is that the entry point approach also applies to unstructured workflows. However, the search for good entry points is much more difficult than for SESE.

### 4.1 Computations and Control-Flows

Before considering different entry points, some terms should be clarified. For instance, different *executions* of a workflow were discussed. This commonly used term can be refined in the notion of *computations* (based on Kindler and Van der Aalst [22]):

**Definition 9 (Computation).** Let $(N, E, l)$ be a workflow graph in a state $S_0$. Any long sequence of strictly reachable states

$$S_0 \to S_1 \to S_2 \to \dots \qquad S_0, S_1, S_2, \dots \in \mathcal{S}(E) \tag{3}$$

is a *computation* $c_{S_0}$ *starting in* $S_0$ iff for each $S_i$, $i \geq 0$, there is a strictly reachable $S_{i+1}$ or this sequence is maximal. If the length of computation is bounded, then the computation is *finite*. Otherwise, it is *infinite*. All possible computations starting in $S_0$, $c^0_{S_0}$, $c^1_{S_0}$, …, comprises the set $\mathcal{C}_{S_0} = \{c^0_{S_0}, c^1_{S_0} \dots\}$. We write $S \in c_{S_0}$ if $S$ is a state in the sequence of $c_{S_0}$.

For instance, starting in the state $\Big\langle (A_1, M), (M, A_6) \Big\rangle$ in Figure 7, $\Big( \Big\langle (A_1, M), (M, A_6) \Big\rangle, \Big\langle (A_1, M), (A_6, E) \Big\rangle, \Big\langle (M, A_6), (A_6, E) \Big\rangle, \Big\langle (A_6, E), (A_6, E) \Big\rangle \Big)$ is a valid computation. However, $\Big( \Big\langle (A_1, M), (M, A_6) \Big\rangle, \Big\langle (A_1, M), (A_6, E) \Big\rangle, \Big\langle (M, A_6), (A_6, E) \Big\rangle \Big)$ is not a valid computation since the state $\Big\langle (A_6, E), (A_6, E) \Big\rangle$ is reachable from the last state and, therefore, the computation is not maximal.

Since computations start in individual states of a workflow graph, the behavior of the workflow can be considered in different situations. However, not every computation start considered can be reached from the initial state. This would lead to false positive and false negative analysis results. Consideration of computations that begin in states with a single token avoids this problem. In the following, we are therefore interested in computations that start with a single token. We call them *control-flows*:

**Definition 10 (Control-flow).** A *control-flow* $f_e$ from an edge $e$ of a workflow graph $(N, E, l)$ is a computation $c_{\langle e \rangle}$ starting in a single-token state $\langle e \rangle$. Let all possible control-flows $f^0_e$, $f^1_e$, …, of the edge $e$ be in the set $\mathcal{F}_e = \{f^0_e, f^1_e, \dots\} = \mathcal{C}_{\langle e \rangle}$.

$\Big( \Big\langle (A_1, M) \Big\rangle, \Big\langle (M, A_6) \Big\rangle, \Big\langle (A_6, E) \Big\rangle \Big)$ is a valid control-flow of the edge $(A_1, M)$ in Figure 7. As can be seen in this example, a control-flow of an edge $e$ describes possible states that result from a single token on $e$. Therefore, it also describes at which other edges a token could get to when starting at $e$.

*Remark 1 (Path on control-flow).* Let $(N, E, l)$ be a workflow graph. If an edge $a$ gets a token in at least one control-flow $f_e \in \mathcal{F}_e$ of an edge $e$, then there is a path from $e$ to $a$.

**Theorem 1 (Control-flow on a path).** Let $e, a \in E, e \neq a$, be two edges of a workflow graph $(N, E, l)$.

$$\text{There is a path } P \text{ from } e \text{ to } a \tag{4}$$

$$\Longrightarrow$$

$$\exists f_e \in \mathcal{F}_e \colon \Big( \forall b \in P \colon b \text{ gets a token in } f_e \ \vee \ \exists S \in f_e \colon S \text{ is a deadlock} \Big) \tag{5}$$

## 4.2 Partial Analysis

Control-flows are suitable for *local*, i. e., *partially*, considerations about the behavior of workflow graphs. Due to the explicit assumption that exactly one token lies on exactly one edge, it is possible to simulate what can happen if this edge receives a token in a computation. The consideration of single edges with their control-flows is so isolated that each edge can be regarded as entry point for an analysis. Therefore, an *entry point* is any edge at which a single token is assumed. The advantages of considering single edges as entry points are:

1. It is simple and not as complex as determining a set of edges as entry points.
2. Each control-flow of an entry point is part of at least one computation where a token reaches that entry point.

The following two theorems describe the second advantage:

**Theorem 2.** Let $e \in E$ be an edge (an entry point) of a workflow graph $(N, E, l)$ in a state $S, e \in S$.

$$\text{let } f_e \in \mathcal{F}_e \tag{6}$$

$$\Longrightarrow$$

$$\exists c_S \in \mathcal{C}_S \colon\ \forall S_e \in f_e \colon\ \exists S_c \in c_S \colon S_e \subseteq S_c \tag{7}$$

Figure 8 shows an example of a workflow graph in the state $\langle k, o\rangle = S$. There is a control-flow $(\langle k\rangle\ ,\ \langle f\rangle\ ,\ \langle g\rangle) = f_k$ from edge $k$. On the basis of Theorem 2, there is at least one computation $c_S \in \mathcal{C}_S$, which contains this control-flow. An example of $c_S$ is: $(\langle \boldsymbol{k}, o\rangle\ ,\ \langle \boldsymbol{f}, o\rangle\ ,\ \langle \boldsymbol{g}, o\rangle\ ,\ \langle \boldsymbol{g}, p\rangle\ ,\ \langle \boldsymbol{g}, c\rangle\ ,\ \langle \boldsymbol{g}, d\rangle\ ,\ \langle \boldsymbol{g}, e\rangle\ ,\ \langle \boldsymbol{g}, f\rangle\ ,\ \langle \boldsymbol{g}^2\rangle) = c_S$.

If a computation $c_S \in \mathcal{C}_S$ starting in state $S$ *contains* a control-flow $f_e \in \mathcal{F}_e, e \in E$, then for each state of $f_e$, there is a state in $c_S$ that is a superset of it, formally: $\forall S_f \in f_e \colon\ \exists S_c \in c_S \colon S_f \subseteq S_c$. And for each edge with a token in a state $S$ exists a control-flow that is contained in a computation starting in $S$:

**Theorem 3.** Let $(N, E, l)$ be a workflow graph in a state $S \in \mathcal{S}(E)$.

$$\text{let } c_S \in \mathcal{C}_S \tag{8}$$

$$\Longrightarrow$$

$$\forall e \in S \colon\ \exists f_e \in \mathcal{F}_e \colon\ \forall S_f \in f_e \colon\ \exists S_c \in c_S \colon S_f \subseteq S_c \tag{9}$$

The control-flow $(\langle k\rangle\ ,\ \langle f\rangle\ ,\ \langle g\rangle) = f_k$ of the edge $k$ of Figure 8 is part of the following computation, as previously mentioned: $(\langle k, o\rangle\ ,\ \langle f, o\rangle\ ,\ \langle g, o\rangle\ ,\ \langle g, p\rangle\ ,\ \langle g, c\rangle\ ,\ \langle g, d\rangle\ ,\ \langle g, e\rangle\ ,\ \langle g, f\rangle\ ,\ \langle g^2\rangle)$. The control-flow of edge $o$ within this computation is: $(\langle o\rangle\ ,\ \langle p\rangle\ ,\ \langle c\rangle\ ,\ \langle d\rangle\ ,\ \langle e\rangle\ ,\ \langle f\rangle\ ,\ \langle g\rangle)$.

Theorem 2 and Theorem 3 are the formal basis of our approach of using different entry points as starting points for analyses. They can be used to identify errors behind other errors. For instance, if there is an abundance in a control-flow of an entry point, then that abundance is possible for any computation that puts a token at this entry point. If a control-flow of the start edge puts a token at this entry point, an abundance is also possible. If there is no such a control-flow of the start edge that puts a token at that entry point, then the workflow graph has a reachable deadlock (because of Theorem 1). Furthermore, in this case, the approach finds a possible error behind the deadlock. In both cases, the workflow graph is not sound anyway. Since earlier errors can hide (mask) later errors at runtime, the latter are called *potential errors*. However, if there is a deadlock in a control-flow of an entry point, then this deadlock must not manifest itself in any computation. This is possible because the token at the entry point may always appear with other tokens at other edges, which can avoid the deadlock. In this case, the quality of deadlock analysis depends on well selected entry points.

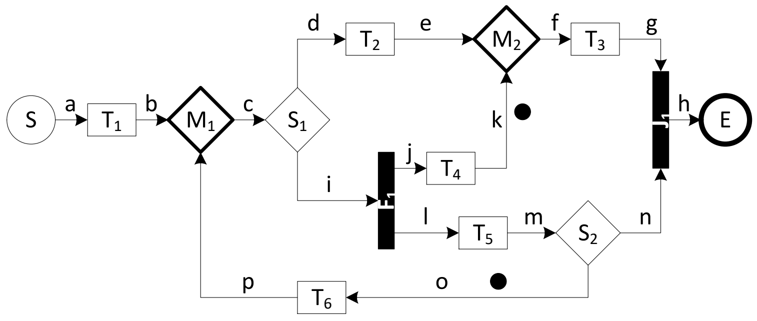


**Figure 8.** The example workflow graph in state $\langle k, o \rangle$

## 5 Causes of Deadlocks

A workflow graph is unsound if a deadlock is reachable from the initial state. The determination of all these deadlocks is possible, e. g., taking into account the entire state-space of the workflow graph. However, such a state-space exploration is very time-consuming. Furthermore, the quality and quantity of fault information can be improved, as they only contain the errors instead of faults as explained before and highlighted as a problem in previous work [11].

It is our goal to provide immediate diagnostic information about deadlocks. Therefore, the following approach renounces from finding all *reachable deadlocks* from the initial state. Instead, the approach considers the *causes* of the deadlocks, since many deadlocks can be reached from a single fault. The advantage is that workflows can be repaired by eliminating the causes of deadlocks.

Two basic cases separate causes of deadlocks: A join node receives either 1) not enough tokens or 2) too many tokens. In the second case, the join node executes, however, blocks subsequently since tokens remain at its incoming edges. This kind of deadlock has a preceding abundance. In the first case, in which a join node does not receive enough tokens, there are two subcases: Either 1a) there are not enough tokens during the execution of the workflow graph, or 1b) another deadlock prevents the arrival of the required tokens. Note that in the case of 1b) there must be a preceding deadlock. Therefore, a deadlock is only independent of other errors in case 1a). As a consequence, a workflow graph is sound if it neither has deadlocks of type 1a) nor an abundance: It is sufficient to find all causes of type 1a) deadlocks.

Figure 9 shows a simple and sound workflow graph. It can be observed that the join node *J never blocks*. This happens since the fork *F guarantees* the execution of this join node. Each node that guarantees the execution of a join node is called an *activation point*. The incoming edges of activation points are similarly called *activation edges*. The basic idea is that a single join node never gets into a deadlock if there is such an activation edge on every path to that join node. Otherwise, the potential of a deadlock exists because execution is not guaranteed.

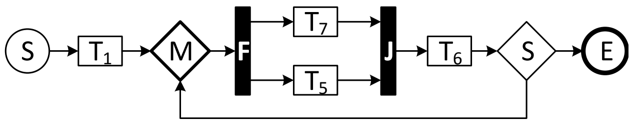


**Figure 9.** A simple (sound) workflow graph

The notation 'each path to a join node' is an inaccurate description, since the starting points of these paths are not specified. The starting points should be the entry points for deadlock analysis. As shown in the following, it is sufficient to derive only two entry points for each join node to decide whether there is a cause of a deadlock for a join node.

## 5.1 Entry Points and Entry Graph

In acyclic, sound workflow graphs, join nodes are executed only once. However, if sound graphs contain loops, join nodes may be executed more than once. In the latter case, the state-space can be separated into two sets for a single join node: The set of states that can be reached *before* its first execution and the set of states that can be reached *after* its first execution.

**Theorem 4 (Two independent entry points).** Let $WFG = (N, E, l)$ be a workflow graph with its start edge *entry* ($= e_{Start}$) and $j$ be a join node with its outgoing edge *out*. If there is a control-flow of *entry* or *out* with a deadlock in $j$, then $WFG$ is not sound:

$$\exists f_{entry} \in \mathcal{F}_{entry}\colon\ \exists S \in f_{entry}\colon j \text{ has a deadlock in } S \tag{10}$$
$$\vee \exists f_{out} \in \mathcal{F}_{out}\colon\ \exists S \in f_{out}\colon j \text{ has a deadlock in } S \tag{11}$$
$$\Longrightarrow$$
$$WFG \text{ is not sound} \tag{12}$$

In short, entry points for deadlock analysis are the start edge and the outgoing edges of join nodes. If a deadlock analysis that starts at these entry points detects a deadlock, then the workflow graph is unsound. However, the detected deadlock must never occur at runtime due to previous faults. This inaccuracy is accepted for fast analysis.

To enable a straight-forward analysis without really focusing on whether the states *before* or *after* a join node are being examined, we separate the workflow graph based on these states. This is achieved by duplicating the join node. The old join node retains its incoming edges and the "new" join node gets its outgoing edge. The workflow now has two start edges and two end nodes. The start edges are also the entry points. Figure 10 visualizes this. An analysis that starts with the original start edge considers all states before the join node's execution, while an analysis that starts with the new start edge examines the states after execution.

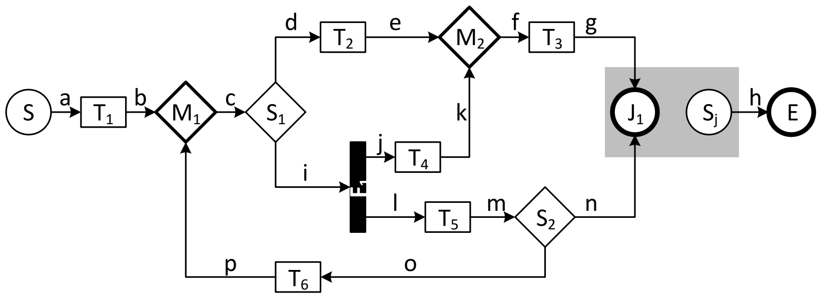


**Figure 10.** Separation of the join node $J_1$ from its outgoing edge

In the introduction of this section, we have argued that we recognize deadlocks independently of other errors. Due to the separation of the join node, deadlocks in the join caused by abundances are no longer possible. We further avoid a deadlock in this join caused by other deadlocks by assuming the other join nodes as deadlock-free. They get the label *Safe*, whose semantics avoid deadlocks in these join nodes.

Assume that two join nodes mutually prevent each other from detecting deadlocks in the other, because it is assumed that the other is deadlock-free. Then, these join nodes must have a path to each other, i. e., there is a cycle (compare Fig. 11). Since there is a path from each node to the end node in a workflow graph, this cycle needs at least one exit node to the outside (see also Fig. 11). If these loop exits are all fork nodes, then there is the possibility for an abundance [51]. Otherwise, if one of these exit nodes is a split node, then this split node has at least one outgoing edge that no longer has a path to the join nodes. As we will soon show in this section, in this case, it is irrelevant whether one of the join nodes is assumed to be deadlock-free, i. e., one of the join nodes does not cause the other to be deadlock-free. In other words: If we find no abundance and no deadlock in the workflow graph, our detection is safe. Otherwise, if we find an abundance, there could be also a deadlock in the workflow graph, however, the graph is still unsound.

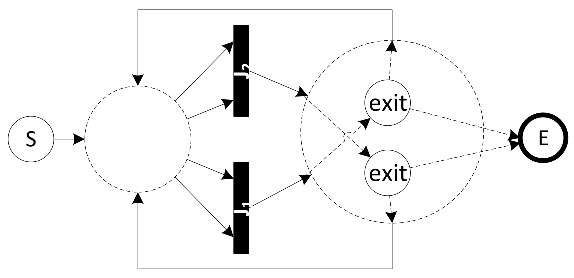


**Figure 11.** Abstract illustration of two join nodes that may prevent the deadlock detection due to deadlock-free assumption

After the separation and label assignment of *Safe*, the resulting graph is called the *entry graph* of the examined join node.

**Definition 11 (Entry graph).** Let $\mathit{WFG} = (N_W, E_W, l_W)$ be a workflow graph and $j \in N_{\mathit{Join}}$ a join node with its outgoing edge *out*. An *entry graph* of $j$ is a labelled graph $\mathcal{EG}(j) = (N, E, l)$. It is based on $\mathit{WFG}$ with some modifications:

1. The set of nodes $N = N_W \cup \{S\}$
2. The set of edges $E = \Big(E_W \setminus \{out\}\Big) \cup \Big\{\Big(S, tgt(out)\Big)\Big\}$
3. The labels $l = l_W \setminus \Big\{(j', \mathit{Join})\colon j' \in N_{\mathit{Join}}\Big\} \cup \Big\{(j', \mathit{Safe})\colon j' \in \Big(N_{\mathit{Join}} \setminus \{j\}\Big)\Big\} \cup \Big\{(S, \mathit{Start}), (j, \mathit{End})\Big\}$

Figure 10 illustrates the entry graph of the join node $J_1$ of the exemplary workflow graph. The entry graphs are very similar to the original workflow graph. Although an entry graph is *not* a workflow graph, its semantics are assumed. Under these circumstances, a deadlock of a join node $j$ occurs in the entry graph if and only if a control-flow of its start edges delivers tokens on at least one but not at all incoming edges of the new end node $j$. This is an *immediate deadlock*.

**Definition 12 (Immediate deadlock).** Let $j$ be a join node of a workflow graph $(N, E, l)$ with its entry graph $\mathcal{EG}(j)$. Furthermore, let *entry* be a start edge of $\mathcal{EG}(j)$ and $f_{entry}$ a control-flow.

$$j \text{ has an immediate deadlock in } f_{entry} \tag{13}$$

$$\Longleftrightarrow$$

$$1 \leq \Big|\{in \in \triangleright j \colon in \text{ gets a token in } f_{entry}\}\Big| < |\triangleright j| \tag{14}$$

In the entry graph of Figure 10, the control-flow $\big(\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle, \langle e\rangle, \langle f\rangle, \langle g\rangle\big)$ ends in an immediate deadlock of $J_1$. The more complex control-flow $\big(\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle i\rangle, \langle j, l\rangle, \langle k, l\rangle, \langle f, l\rangle, \langle g, l\rangle, \langle g, m\rangle, \langle g, o\rangle, \langle g, p\rangle, \langle g, c\rangle, \langle g, i\rangle, \langle g, j, l\rangle, \langle g, k, l\rangle, \langle g, f, l\rangle, \langle g, g, l\rangle, \langle g, g, m\rangle, \langle g, g, n\rangle\big)$ does not end in an immediate deadlock, but with an abundance at edge $g$. In the original workflow graph, a token would remain on $g$ and causes a deadlock. But this deadlock results from an abundance and is ignored by our analysis.

Immediate deadlocks can appear at execution time if there are no previous errors. If they never occur at runtime, it is only because an earlier error prevents them. Either way, the workflow graph is unsound.

## 5.2 Activation Edges and Causes of Deadlocks

If it is guaranteed that every time a join node receives a token, all other incoming edges also receive tokens, an immediate deadlock is impossible. In the following, we introduce special edges — which we call *activation edges* of a join node — whose execution guarantees tokens on all incoming edges of a join node.

**Definition 13 (Activation edges).** Let $\mathcal{EG}(j) = (N, E, l)$ be an entry graph of a join node $j$ of a workflow graph.

An edge $a \in E$ is an *activation edge* of a single incoming edge *in* of $j$ if in each control-flow of $a$ there is a state where *in* has a token. We write $a \curvearrowright in$:

$$(a, in) \in \curvearrowright \iff \forall f_a \in \mathcal{F}_a\colon\ \exists S \in f_a\colon in \in S \tag{15}$$

If $a$ is an activation edge of each incoming edge of $j$, then $a$ is also an activation edge of $j$, $a \overset{all}{\curvearrowright} j$.

Reconsider the entry graph of Figure 10. The incoming edge $g$ of the join node $J_1$ has the activation edges $a$, $b$, $c$, $d$, $e$, $f$, $i$, $j$, $k$, $o$, $p$, and $g$ itself; $\forall x \in \{a, b, c, d, e, f, g, i, j, k, o, p\}\colon x \curvearrowright g$. This is understandable when we look at the graph: Regardless of how the split node $S_1$ decides, a token reaches with guarantee the edge $g$. In this example, no activation edge of $J_1$ exists since there is no edge, which guarantees a token on edge $n$ (except $n$ itself).

If on each path from an entry point to the join node lies an activation edge of that join node, then an immediate deadlock is impossible. Otherwise, if the join has an immediate deadlock, there must be a path to that join without an activation edge. This states the following theorem:

**Theorem 5 (Deadlock theorem).** Let $j$ be a join node with its entry graph $\mathcal{EG}(j)$, and *entry* be a start edge of the entry graph.

$$\forall f_{entry} \in \mathcal{F}_{entry}\colon j \text{ does not have an immediate deadlock in } f_{entry} \tag{16}$$

$$\iff$$

$$\forall in \in \triangleright j\colon \text{ all paths from } entry \text{ to } in \text{ contain an activation edge of } j \tag{17}$$

Figure 12 shows the entry graph of a join node $J_1$ of a sound workflow graph. For this graph, $(S, T_1)$ is a start edge (an entry point). On all paths from $(S, T_1)$ to the incoming edges $(T_7, J_1)$ and $(T_5, J_1)$ of $J_1$ an activation edge of both edges lies: $(S_1, F_1)$. The same applies to all paths from $(S_j, S_2)$ to these incoming edges. Theorem 5 states that an immediate deadlock is impossible. This is easy to understand regarding the graph in Figure 12 — each control-flow of $(S, T_1)$ and $(S_j, S_2)$, respectively, either reaches $J_1$ completely or not at all. This is in contrast to the entry graph of Figure 10, where the join node $J_1$ has no activation edge. Therefore, an immediate deadlock is possible in Figure 10.

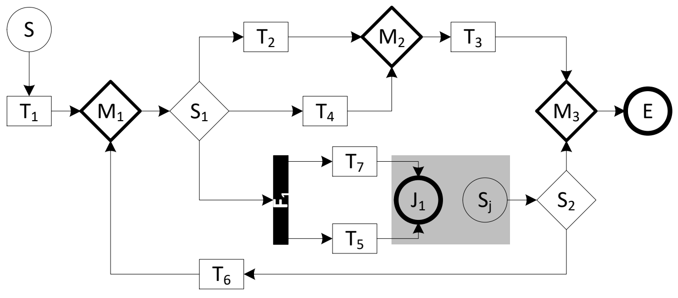


**Figure 12.** Entry graph of a join node $J_1$ of a sound workflow graph

Basically, we can use Theorem 5 to determine potential, immediate deadlocks for each join node. The *causes* of these potential, immediate deadlocks are structural failures in workflow graphs, i. e., the existence of computations that reach join nodes but do not guarantee their execution. As we argued earlier, if we have neither immediate deadlocks nor abundances, the workflow graph contains no deadlock and is, therefore, sound. If it contains an immediate deadlock, the workflow is unsound.

## 5.3 Algorithmic Derivation

So far, the theory of finding causes of deadlocks shows that the absence of activation edges causes immediate deadlocks. In the following, Algorithms 1 and 2 explain how to find activation edges and immediate deadlocks in any workflow graph.

The determination of activation edges requires the consideration of control-flows by definition. However, the consideration of control-flows is a partial consideration of the state-space and therefore inefficient. In order to obtain an efficient determination, a relationship between edges and their direct successors helps.

*Remark 2.* Let $j$ be a join node with its entry graph $\mathcal{EG}(j)$ and one of its incoming edges $in$. In addition, let $a$ be any edge with its direct successors $Succ = tgt(a)\triangleleft$.
$a$ is an activation edge of $in$ iff

1. $a = in$, or
2. $tgt(a)$ is a task, fork, merge, or safe node (former a join node) and $\exists b \in Succ\colon b \curvearrowright in$, Figure 13 *a)* to *d)*, or
3. $tgt(a)$ is a split node and $\forall b \in Succ\colon b \curvearrowright in$, Figure 13 *e)*.

Therefore, each activation edge ($\neq in$) has at least one activation edge as a direct successor.

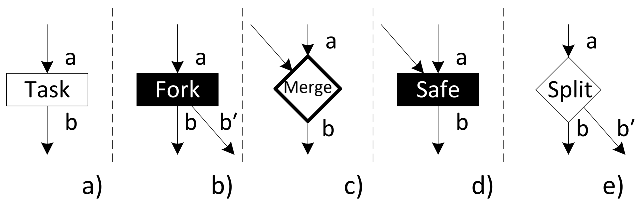


**Figure 13.** The direct predecessor of an activation edge is also an activation edge, except the node in between is a split node

Activation edges and their adjacent nodes build a connected graph [52]. For this reason, each activation edge of an incoming edge $in$ of a join node $j$ is $in$'s predecessor — the set of all predecessors is a superset of all activation edges of $in$. The predecessor superset results directly from an inverse depth-first search at the edges of the entry graph starting at $in$ (an *inverse* depth-first search visits the edges in opposite direction). The approach of the algorithm is to filter the set of activation edges, $\mathit{Activation}(in)$, iteratively from this superset. In each iteration, $\mathit{Activation}(in)$ gets an update by eliminating the edges that are not $in$'s activation edges. If the filtering finds a fixed point, the algorithm terminates. $\mathit{Activation}(in)$ then contains the set of activation edges of $in$.

Each edge ($\neq in$) in $\mathit{Activation}(in)$ has a direct successor in $\mathit{Activation}(in)$. With respect to Remark 2, the algorithm only has to eliminate incoming edges of task, fork, merge, and safe nodes if it has already eliminated all their direct successors. In contrast, the incoming edges of split nodes are not activation edges if at least one direct successor is not in $\mathit{Activation}(in)$. As a consequence, there are two steps for each iteration: 1) eliminating incoming edges of split nodes, and 2) eliminating incoming edges of task, fork, merge, and safe nodes. Since the latter have no direct successors in $\mathit{Activation}(in)$, they are no longer reachable from $in$ and can be easily removed by a new inverse depth-first search.

**Algorithm 1** Determination of the activation edges

```
Require: The entry graph $EG = \mathcal{EG}(join)$ of a join node $join$.
Ensure: The sets of activation edges $Activation(in)$ for each incoming edge $in$, and $ActivationJoin$ for $join$ itself.
  $ActivationJoin \leftarrow E(EG)$
  for all $in \in \triangleright join$ do
      // Initial position: All predecessors of $in$ are $Activation(in)$. Determination by an inverse depth-first search from $in$.
      $Activation(in) \leftarrow InverseDepthFirstSearch\big(E(EG), in\big)$
      repeat
          $OldActivation \leftarrow Activation(in)$
          // Eliminate incoming edge of split node from $Activation(in)$ if not all its outgoing edges are in $Activation(in)$.
          for all $split \in N_{Split}$ do
              if $split\triangleleft \not\subseteq Activation(in)$ then
                  $Activation(in) \leftarrow Activation(in) \setminus \triangleright split$
              end if
          end for
          // Inverse depth-first search starting from $in$ on the edges of $Activation(in)$. Visited edges are the new $Activation(in)$.
          $Activation(in) \leftarrow InverseDepthFirstSearch\big(Activation(in), in\big)$
          // Compute as long as $Activation(in)$ changes.
      until $Activation(in) = OldActivation$
      $ActivationJoin \leftarrow ActivationJoin \cap Activation(in)$
  end for
```

Algorithm 1 shows the determination of the activation relation. Considering the entry graph of Figure 10 as an example, the initial set $\mathit{Activation}(g)$ from the incoming edge $g$ of the join node $J_1$ is $\{a, b, c, d, e, f, g, i, j, k, l, m, o, p\}$. Now, the algorithm determines those split nodes that have outgoing edges outside of $\mathit{Activation}(g)$. The split node $S_2$ satisfies this property; the algorithm eliminates its incoming edge $m$. Then, an inverse depth-first search updates $\mathit{Activation}(g)$ to $\{a, b, c, d, e, f, g, i, j, k, o, p\}$. Since no edge can be longer eliminated, the algorithm terminates. Each edge of $\mathit{Activation}(g)$ guarantees a token on $g$ if it carries a token.

Two loops characterize the asymptotic runtime behavior of Algorithm 1 for a single incoming edge $in$ of a join node: the outer repeat-until-loop and the inner for-all-loop. The repeat-until-loop is executed at most as it eliminates edges: $O(E)$. The time-consuming steps of this loop are the inverse depth-first search, $O(E)$, and the inner for-all-loop. The for-all-loop examines each split node, $O(N_{\mathit{Split}})$ or simplified $O(E)$. Therefore, each iteration of the repeat-until-loop requires $O(E)$. In total, the repeat-until-loop has a quadratic runtime, $O(E^2)$. To determine the activation edges of all incoming edges of all join nodes, the total asymptotic runtime is cubic, $O(E^3)$.

Now, all activation edges are available. Algorithm 2 for finding immediate deadlocks uses them. It is based on the deadlock Theorem 5, which takes into account all paths from each start edge of the entry graph of $j$ to $j$ itself. If there is a path without an activation edge of $j$, then an immediate deadlock is possible at $j$. An inverse depth-first search from the incoming edges of $j$ considers all of those paths. If the search reaches an activation edge of $j$, the search does not consider subsequent edges. However, if the search reaches one of the start edges of the entry graph, then there is at least one path from that start edge to at least one incoming edge of $j$ without an activation edge.

Three loops characterize the asymptotic runtime behavior of Algorithm 2: 1) the construction of the entry graphs, 2) the determination of the activation edges, and 3) the application of the inverse depth-first search including the determination of immediate deadlocks. The construction of a single entry graph (1), is linear, $O(E)$. It only has to replace the labels of all join nodes and to separate the considered join node. The construction of all entry graphs is quadratic. The computation of all activation edges (2), has a cubic complexity as already mentioned. For each incoming edge of all join nodes, the algorithm applies an inverse depth-first search and a determination of immediate deadlocks (3). The inverse depth-first search has a linear complexity, $O(E)$. The immediate deadlock check is constant. Step 3 therefore has a quadratic runtime complexity considering the number of edges. The complete algorithm for determining all immediate deadlocks is cubic, $O(E^3)$.

**Algorithm 2** Determination of immediate deadlocks

```
Require: A workflow graph $(N, E, l)$
Ensure: A set $\mathcal{D}ead \subseteq N_{Join}$ of all join nodes with potential immediate deadlocks.
  Compute the entry graphs for each join node join with start edges $Start(join)$.
  Compute the activation edges for each join node.
  for all $join \in N_{Join}$ do
      for all $in \in \triangleright join$ do
          // Define the edges where the search stops. That are the activation edges of join.
          $StopSearch \leftarrow \{a \in E : a \overset{all}{\curvearrowright} join\}$
          // An empty set defines visited edges during the search.
          $Visited \leftarrow \emptyset$
          // Apply an inverse depth-first search starting from in.
          InverseStoppableDepthFirstSearch(in, StopSearch, Visited)
          // Determine the immediate deadlocks.
          if $\big(Start(join) \subseteq Visited\big)$ then
              $\mathcal{D}ead \leftarrow \mathcal{D}ead \cup \{join\}$
          end if
      end for
  end for
  procedure InverseStoppableDepthFirstSearch(current, StopSearch, Visited)
      $Visited \leftarrow Visited \cup \{current\}$
      for all $pred \in \big(\triangleright src(current) \setminus (StopSearch \cup Visited)\big)$ do
          InverseStoppableDepthFirstSearch(pred, StopSearch, Visited)
      end for
  end procedure
```

## 6 Causes of Abundances

In workflow graphs, semantics bind deadlocks to join nodes, while abundances can occur at any edge with any number of tokens. A small change in the order of execution leads to an earlier or later abundance or hides it completely. Finding all these abundances (e. g., with a state-space exploration) is impossible or at least very time-intensive in practice.

As with deadlocks, the aim of this work is to find a fast approach. This is possible by not determining all *reachable* abundances. Instead, the *causes* of abundances are the focus since, without causes of abundances, there are no abundances at runtime. Knowing the cause further helps to avoid it during construction.

Basically, since there is parallelism with at least two tokens, the origin of each abundance lies in the execution of a single fork node. For this reason, fork nodes are suitable entry points for an abundance analysis. An abundance occurs when no join node synchronizes at least two control-flows of a fork node. Our approach is very simple: Two control-flows, which start in a single fork node, can reach exactly those edges *at first* simultaneously, to which the fork has two disjoint paths. These first edges are called *intersections*. If the sources of these intersections are join nodes, there are no problem. Otherwise, abundances are possible.

In the following, it is assumed that fork nodes have exactly two outgoing edges. Although this is a limitation, any workflow graph can be transformed without changing the semantics by simply adding additional fork nodes. For each fork node *fork* with more than two outgoing edges, the transformation requires a maximum of $|fork\triangleleft| - 1$ additional fork nodes. Since the transformation doubles the number of edges at maximum, the graph grows only linearly.

## 6.1 Intersections of Control-Flows

In an abundance, at least two tokens lie on at least one edge. Since an abundance has at least *two* tokens, a fork node must be executed before to enable this parallelism. After the execution of a fork node *fork*, each of its outgoing edges carries at least one token. These tokens move almost independently from each other from edge to edge. They pass places where they can meet for the *first time*. In other words, in such places, two tokens of *fork* can reach the same edge and cause an abundance for the first time. These places are *intersections*.

**Definition 14 (Intersection).** Let *fork* be a fork node of a workflow graph $(N, E, l)$. An edge $\iota \in E$ is an *intersection* for the two outgoing edges $a, b \in fork\triangleleft$ of *fork* if there are two paths $P_a$ from $a$ and $P_b$ from $b$ to $\iota$ and $P_a \cap P_b = \{\iota\}$. The paths $P_a$ and $P_b$ are *routes* from $a$ and $b$ to $\iota$. Also, $\iota$ is an intersection of *fork*.

In Figure 14, the edge $f$ is an intersection of the fork node $F_1$. It can be reached via the routes $(j, k, f)$ and $(l, m, o, p, c, d, e, f)$ from the two outgoing edges of $F_1$. Further intersections of $F_1$ are the edges $h$ and $j$.

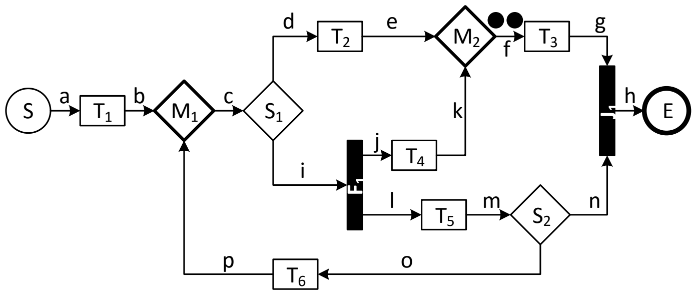


**Figure 14.** An abundance on the edge $f$

If all intersections of all fork nodes have join nodes as source, then all tokens meet in join nodes at first — and are synchronized. An abundance is *impossible*.

*Remark 3 (Exclusion of abundances).* Let $(N, E, l)$ be a workflow graph.

$$\forall fork \in N_{Fork}\colon \text{ each intersection of } fork \text{ has a join node as source} \tag{18}$$

$$\Longrightarrow$$

$$\forall e \in E\colon\ \forall S \in \mathcal{S}(E), \langle e\rangle \to^* S\colon\ S \text{ is not an abundance} \tag{19}$$

After using the contraposition $(a \to b \Leftrightarrow \neg b \to \neg a)$ of Remark 3, the necessity of abundances results:

*Remark 4 (Condition of abundances).* Let $(N, E, l)$ be a workflow graph. From the contraposition of Proposition 3 follows:

$$\exists e \in E\colon\ \exists S \in \mathcal{S}(E), \langle e\rangle \to^* S\colon\ S \text{ is an abundance} \tag{20}$$

$$\Longrightarrow$$

$$\exists fork \in N_{Fork}\colon \text{ there is an intersection of } fork \text{ whose source is not a join node} \tag{21}$$

For instance, the source of the edge $f$ of Figure 14 is *no* join node, it is the merge node $M_2$. The abundance on $f$ results from the missing synchronization in $M_2$.

## 6.2 Superfluous Intersections

Remark 4 states that if an abundance has occurred, then at least one intersection of a fork node has not a join node as source. This condition is necessary for an abundance. We must, however, check that it is *sufficient.* In the following, *fork* should be a fork node with one of its intersections $\iota$.

The basic idea of checking for sufficiency is to assume a *sound* workflow graph. In a sound workflow graph, a non-join intersection $\iota$ is unnecessary ("*superfluous*"), because it can never be reached by two tokens at the same time, i. e., it never synchronizes tokens. Otherwise, the workflow would not be sound. We can formulate the following sufficient condition for sound workflow graphs informally without knowing superfluous intersections in detail yet:

*Remark 5.* Let $\mathit{WFG} = (N, E, l)$ be a workflow graph.

$$\mathit{WFG} \text{ is sound} \tag{22}$$

$$\Longrightarrow$$

$$\forall \mathit{fork} \in N_{\mathit{Fork}}: \text{ each intersection of } \mathit{fork} \text{ has a join node as source or is "superfluous"} \tag{23}$$

According to Theorem 1, the tokens of *fork* can arrive at $\iota$ via any routes. It is of interest what exactly prevents the tokens that they arrive at $\iota$ at the same time. The following explains that unnecessary ("superfluous") intersections are activation edges of some incoming edges of join nodes on the routes:

**Theorem 6.** Let $\mathit{WFG} = (N, E, l)$ be a workflow graph with a $\mathit{fork} \in N_{\mathit{Fork}}$, $\mathit{fork}\triangleleft = \{a, b\}$. An intersection of *fork* is $\iota$.

$$\mathit{src}(\iota) \notin N_{\mathit{Join}} \ \wedge \ \mathit{WFG} \text{ is sound} \tag{24}$$

$$\Longrightarrow$$

$$\begin{gathered}\text{On all routes } (P_a, P_b) \text{ from } a \text{ and } b \text{ to } \iota \text{ lies a join node on } P_a \text{ or } P_b\\ \text{and } \iota \text{ is an activation edge of a real non-empty subset of the join's incoming edges}\end{gathered} \tag{25}$$

The preceding theorem describes the necessary condition for unnecessary ("superfluous") intersections in sound workflow graphs. Instead of directly showing that the condition is sufficient, we build the contraposition for checking the sufficient condition. This is done in the following theorem:

**Theorem 7.** Let $\mathit{WFG} = (N, E, l)$ be a workflow graph with a $\mathit{fork} \in N_{\mathit{Fork}}$, $\mathit{fork}\triangleleft = \{a, b\}$. *fork* has an intersection $\iota$.

$$\begin{gathered}\text{There is a route } (P_a, P_b) \text{ from } a \text{ and } b \text{ to } \iota\\ \text{where } \iota \text{ is } \textbf{not} \text{ an activation edge}\\ \text{for a real non-empty subset of the incoming edges of any join node on } P_a \text{ and } P_b\end{gathered} \tag{26}$$

$$\Longrightarrow$$

$$\mathit{src}(\iota) \in N_{\mathit{Join}} \ \vee \ \mathit{WFG} \text{ is unsound} \tag{27}$$

It is now possible to define superfluous intersections formally:

**Definition 15 (Superfluous intersections).** Let *fork* be a fork node in a sound workflow graph. An intersection $\iota$ of *fork* with $\mathit{src}(\iota) \notin N_{\mathit{Join}}$ is *superfluous* if on all routes $A$ and $B$ to $\iota$ is a join node on the path $B$, where $\iota$ is an activation edge for a non-empty proper subset of this join's incoming edges.

From the contraposition of Remark 5 follows our abundance criterion:

*Remark 6.* Let $WFG = (N, E, l)$ be a workflow graph.

$$\begin{gathered}\exists \mathit{fork} \in N_{\mathit{Fork}}\colon \text{ There is an intersection of } \mathit{fork} \\ \text{that has not a join node as source and is not superfluous} \qquad (28) \\ \Longrightarrow \\ \mathit{WFG} \text{ is unsound} \qquad (29)\end{gathered}$$

Remark 6 can be used to identify abundances in workflow graphs. Although the remark only claims that the workflow is unsound, the proofs of Theorems 6 and 7 in the appendix explain that there is an abundance except a previous deadlock avoids it. In other words, Remark 6 is able to find abundances behind other errors. As the evaluation later shows, the detected causes of abundances are very accurate. **With Theorem 5 to detect immediate deadlocks and Remark 6 to detect causes of abundances, we are now complete in terms of soundness.**

*Remark 7 (Completeness).* A workflow graph is sound iff it neither has an immediate deadlock according to Theorem 5 nor an abundance according to Remark 6.

**Justification:** Suppose we have found an immediate deadlock by Theorem 5. This deadlock does not occur during execution only if another error prevents it (as explained before). Either way, the workflow graph is unsound if we find any immediate deadlock. If we did *not* find any immediate deadlock, then the workflow graph can still have deadlocks, but only if the workflow graph contains an abundance.

Suppose we have found a cause of an abundance by Remark 6. This abundance does not occur during execution only if some other error prevents it. Nevertheless, the workflow graph is still unsound. If we did *not* find any cause of an abundance, then no abundance is possible at all. In summary, the workflow graph is sound if and only if we find neither immediate deadlock nor abundance.

## 6.3 Algorithmic Derivation

The determination of intersections of a fork node is similar to $\phi$-placement in the minimal Static Single Assignment Form (SSA form) — a traditional compiler construction problem [53], [54]. In the SSA form each variable gets statically only once a value. To achieve this, the transformation inserts a new variable (called *definition*) for each variable assignment. If two definitions of the same variable are valid in one place, a $\phi$-function combines both values. The $\phi$-function selects the correct definition dependent on the actual control-flow at runtime. It has the form of $d_n = \phi(d_1, \ldots, d_m)$, $m \geq 2$, where $d_1, \ldots, d_m$ are the different definitions whose resulting definition is $d_n$.

The simplest transformation inserts $\phi$-functions at all merging points resulting in unnecessary $\phi$-functions. In the *minimal* SSA form, $\phi$-functions are only located at those positions where they are statically necessary (by static placement of $\phi$-functions fairness is assumed, i. e., each path can be reached by control-flows). Cytron et al. describe those positions as nodes where two paths starting in two different definitions of the same variable $v$ first meet [55]. If we apply this description to edges, it is equal to our definition of intersections of fork nodes. Cytron et al. derived an algorithm for constructing a minimal SSA form. For any two paths starting at any nodes, the algorithm detects the first meeting nodes. This algorithm also applies to edges.

If we want to find intersections of a fork node *fork*, a *fork*-specific variable $v$ in the form $v \leftarrow \phi(x)$ is placed at each outgoing edge of *fork*, where $x$ is an arbitrary value (see Figure 15, left picture). The algorithm of Cytron et al. computes the $\phi$-functions necessary at minimum for this variable $v$. Either the algorithm of Cytron et al. or a faster version is applicable (e. g., by Lengauer and Tarjan [56] or Cooper et al. [57]). Most of those algorithms assume a variable assignment on the start edge of the control-flow graph. But this is unnecessary for our purpose. After the algorithm's application, all $\phi$-functions are available in the workflow graph with respect to *fork* $(v)$ (see the positions of the $\phi$-functions in Figure 15, right picture). Each $\phi$-function with at least two parameters is a meeting point of two disjoint paths and, therefore, an intersection of *fork*. We can find all intersections of *fork*.

In Figure 15 (right picture), the upper outgoing edge of the fork node is an intersection. Two paths meet firstly in this edge — the path consisting of the edge itself and the path from the lower outgoing edge to this edge. During runtime the token may pause on that intersection and the token at the other edge reaches it and causes an abundance.

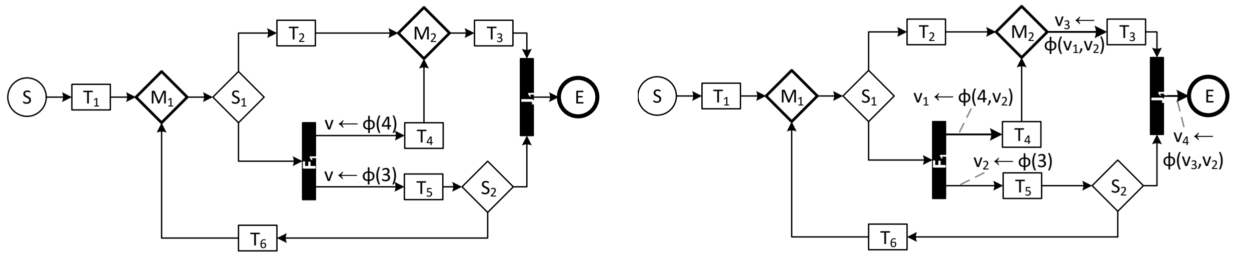


**Figure 15.** Two assignments of the *fork*-specific variable $v$ (left workflow graph) and $\phi$-placements for the *fork*-specific variable $v$ (right workflow graph)

Algorithm 3 shows an implementation to find all intersections of the fork nodes. Its asymptotic runtime complexity depends on the construction of the minimal SSA form. This is cubic, $O(X^3)$, according to Cytron et al. [55] concerning the maximum $X$ of the number of nodes, edges, and assignments. Since the number of nodes and assignments is smaller than the number of edges, $X = E$ follows. So the worst-case complexity is $O(E^3)$.

**Algorithm 3** Determination of all intersections of all fork nodes

```
Require: A workflow graph $(N, E, l)$
Ensure: The set $inter(fork)$ of all intersections of each fork node $fork$
  // Initialization: Insertion of all fork-specific variables.
  for all $fork \in N_{Fork}$ do
      for all $out \in fork\triangleleft$ do
          Insert $\phi$-function $out \leftarrow \phi(x)$, $x$ is an arbitrary value
      end for
  end for
  Compute the minimal SSA form (e. g. following Cytron et al.)
  // Detection of the intersections.
  for all $fork \in N_{Fork}$ do
      for all $\phi$-functions $\phi$ of $fork$ do
          Let $t$ be the edge at which $\phi$ is placed
          if Number of parameters of $\phi \geq 2$ then
              $inter(fork) \leftarrow inter(fork) \cup \{t\}$
          end if
      end for
  end for
```

We have the intersections of the fork nodes. We have to eliminate superfluous intersections to be exact. To check the superfluousness of an intersection $\iota$ of a fork node *fork*, we need to check whether two routes from *fork* to $\iota$ do not contain join nodes, for which $\iota$ is an activation edge of at least one of their incoming edges.

The proposed approach uses the theory of *flow networks* [15] to solve the problem. A flow network is a digraph $G = (N, E)$ with exactly one *flow source*, *source* $\in N(G)$, and exactly one *flow sink*, $sink \in N(G)$. The flow source has no incoming edge. Each edge $e$ of the flow network has a *capacity* $cap(e)$ on possible *flows*. Figure 16 *a)* shows a simple flow network.

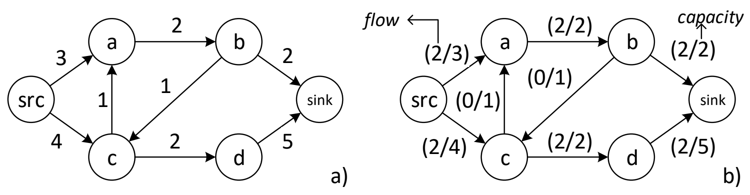


**Figure 16.** A simple flow network, *a)*, and its maximum flow, *b)*

The most prominent problem with flow networks is the calculation of the *maximum flow* (max-flow problem) [15], [58]. In this problem, as many flows as possible should start in the flow source without exceeding the capacities of the edges (cf. Figure 16 *b)*). The upper limit is the sum of the capacities of the incoming edges of the flow sink, $\sum\limits_{e \in \triangleright sink} cap(e)$.

We can apply the max-flow problem to check the superfluousness of an intersection $\iota$ of a fork node *fork*. In other words, it helps to decide whether there are at least two disjoint paths (except $\iota$) from *fork* to $\iota$ without join nodes, for whose incoming edges $\iota$ is an activation edge. To achieve this, we reformulate our problem into the max-flow problem by transforming the workflow graph into a flow network concerning *fork* and $\iota$.

A flow in the flow network represents the “travel” of a token from an outgoing edge of *fork* to $\iota$. At each edge, there is a space for a single flow/token. If an edge has two flows, both paths are not disjoint. Therefore, the capacity of each edge $e$ is 1 at this moment, $cap(e) = 1$. The flow source is the node *fork* since it “produces” the tokens. The flow sink is the source of $\iota$, $src(\iota)$.

To prevent a flow/token from travel over a join node, for which $\iota$ is an activation edge of at least one of its incoming edges, the capacity of all outgoing edges of all these join nodes is set to 0. As a result, no flow can use these edges. We can simply determine the join nodes, whose outgoing edges are set to 0, with the information about the activation edges from Section 5. A recalculation is not necessary.

At last, we must eliminate the incoming edge *in* from *fork* to match the definition of flow networks. But this would also destroy potential paths from an outgoing edge of *fork* to $\iota$. To avoid this, we do *not* eliminate *in*, but detach *in* from *fork* and insert a new node *fork'*. *in* is the new incoming edge of *fork'* (see Figure 17, step 1). In the next step, we detach each outgoing edge *out* of *fork* and insert a new merge node $M_{out}$ for each edge. *Out* is the new outgoing edge of $M_{out}$ (see Figure 17, step 2). Then for each edge *out* we add a new edge from *fork* to $M_{out}$ and from *fork'* to $M_{out}$. These new edges all have capacity of 1. Figure 17, step 3, illustrates the resulting flow network cut-out. It is easy to verify that each original outgoing edge of *fork* can host a maximum one flow.

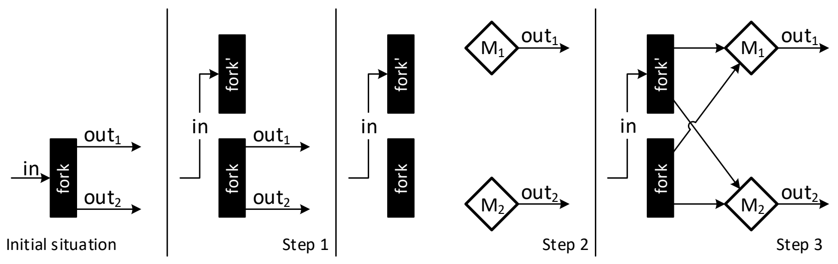


**Figure 17.** Steps for duplicating a fork node to preserve paths via the incoming edge *in*

The transformation of the workflow graph into a flow network regarding *fork* and $\iota$ is completed. Figure 18 illustrates our example workflow graph transformed into a flow network concerning the fork node $F_1$ and the intersection $(M_2, T_3)$.

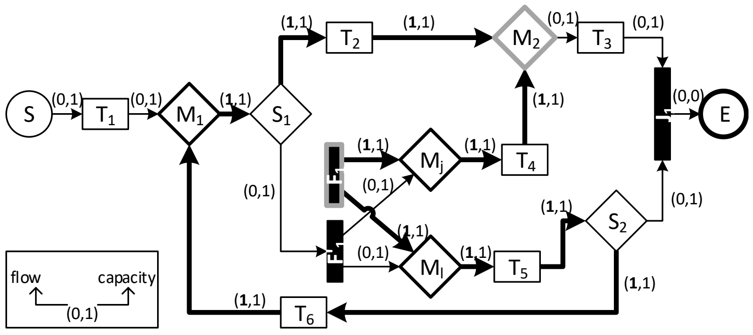


**Figure 18.** The flow network of our example workflow graph concerning the fork node $F_1$ and intersection $(M_2, T_3)$

On the created flow network, we compute the maximum flow with respect to *fork* (see Figure 18). If the incoming flow of the sink (i. e., $src(\iota)$, $M_2$ in the example) is greater than or equal to 2, then there are two disjoint paths $P_a$ and $P_b$ from *fork* to $\iota$. They avoid paths with join nodes for which $\iota$ is an activation edge of a subset of their incoming edges, because the capacities on these edges are 0. In this case, $\iota$ is a non-superfluous intersection of *fork*. In the example, $(M_2, T_3)$ is such an intersection of $F_1$, because two flows reach it. But its source is a merge node and, therefore, there is a possibility of abundance. On the contrary, if the incoming flow is less than or equal to 1, $\iota$ is superfluous. It can be eliminated as an intersection of *fork*.

Figure 19 shows a sound workflow graph and Figure 20 its flow network regarding the fork node *fork*. The intersection $(M_2, join) = \iota$ is superfluous since $(M_2, join)$ is an activation edge of the incoming edge $(= \iota)$ of the join. Figure 20 illustrates the maximum incoming flow of 1 of $(M_2, join)$. Another flow does not reach it because the outgoing edge of the join node has a capacity of 0. Our algorithm identifies it correctly.

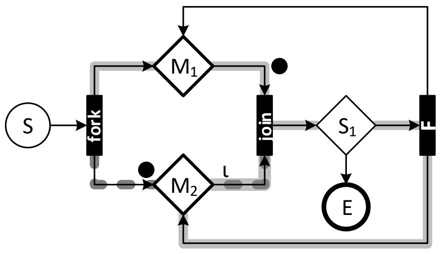


**Figure 19.** A sound workflow graph with the intersection $\iota$ of the fork node *fork* with two routes in grey and dashed lines

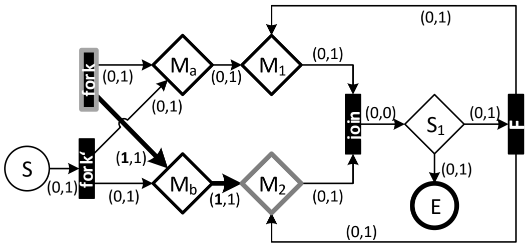


**Figure 20.** The flow network with respect to the fork node *fork* and the superfluous intersection $(M_2, join)$

The transformations of the workflow graph in a flow network and the calculation of the maximum flow solve our problem of eliminating superfluous intersections. The algorithm of Ford and Fulkerson [15], [59] can be used to calculate the maximum flow, for instance. It calculates the maximum flow $f$ in an asymptotic runtime of $O(fE)$. Since $f$ is at maximum as high as the intersection's $\phi$-function has parameters, the maximum flow is a constant in our case. Therefore, it is possible to calculate the maximum flow in linear time behavior $O(E)$.

The transformation of the workflow graph into a flow network regarding a single fork node and a single intersection is possible in linear time. The information about activation edges can either be taken from the deadlock analysis (see Section 5) or be recomputed using the algorithm in that section. As mentioned there, the asymptotic runtime complexity is cubic, $O(E^3)$. Since the number of fork nodes and the number of intersections of each fork node is linear to $E$ in the worst case, the superfluous intersection elimination is cubic $O(E^3)$. Finally, Algorithm 4 shows the complete algorithm to determine causes of abundances. This algorithm has a cubic total runtime complexity.

**Algorithm 4** Detection of abundances' causes

```
Require: A workflow graph $WFG = (N, E, l)$
Ensure: The set $Causes \subseteq N_{Fork} \times E$ of all causes of abundances
  // Initialization
  For each fork node $fork$ detect its intersections $inter(fork)$
  For each intersection determine the join nodes, for which it is an activation edge for a subset of incoming edges
  // Check all non-join intersections
  for all $fork \in N_{Fork}$ do
      for all $\iota \in inter(fork)$ do
          if $src(\iota) \notin N_{Join}$ then
              // Check whether $\iota$ is superfluous
              Transform $WFG$ into a flow network regarding $fork$ and $\iota$
              Compute the maximum flow of the flow network $maxFlow$
              if $maxFlow >= 2$ then
                  $Causes \leftarrow Causes \cup \{(fork, \iota)\}$
              end if
          end if
      end for
  end for
```

## 7 Implementation

We have developed a tool *Mojo* to evaluate the previously introduced algorithms and approaches [60]. The tool is part of our concept of a system for developing and executing workflows [61], shown in Figure 21. Currently, *Mojo* covers parts of the *producer side*. On the producer side (*the compiler*), a *parser* reads the workflow. During parsing, the structure of the workflow is checked. Then, a *transformer* translates the workflow into an *Intermediate Representation* (IR). As IR, Mojo uses workflow graphs. In general, an IR abstracts from language-dependent properties of the entire modeling language.

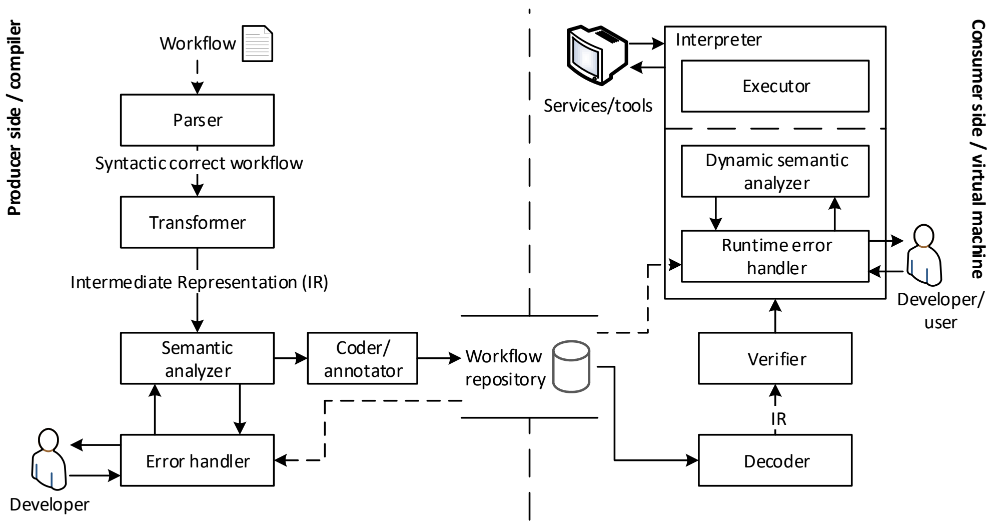


**Figure 21.** Map of a system for developing and executing workflows (adapted from [61])

After production of the IR, the workflow is checked semantically in a *semantic analyzer*. An example for such a semantic check is the application of the algorithms presented in this article. If semantic faults occur, an *error handler* informs the developer about the faults, so that the developer receives immediate feedback. Otherwise, if the workflow is semantically correct, it is encoded and stored into a file or workflow repository.

Besides the producer side, there is a *consumer side*. The consumer side is a virtual machine. It reads the encoded IR from the files or workflow repositories, a *decoder* decodes the IR, and a *verifier* verifies it against the same semantic properties as on the producer side. The verification can be accelerated by adding annotations to the IR that contain previous analysis results. The goal of the verifier is to detect IR manipulations and avoid the execution of malformed workflows. After verification, the virtual machine executes the workflow in an *interpreter* and performs dynamic and semantic runtime analyses to avoid errors at runtime as early as possible. The virtual machine illustrates the errors to the user or developer.

Further information about the system and its concepts can be found in Prinz et al. [61]. An overview of the compiler and the virtual machine is given by Prinz et al. [62], [63].

Mojo has an expandable software architecture that uses the concept of extension points. The extension points allow the adaptation of new modeling languages and new analyses. For this reason, other researchers can extend Mojo on their purpose.

*Analysis plans* describe the order in which Mojo applies analyses. They define the necessary phases to obtain correct analysis results. Such phases can be bigger analysis plans again and can include other phases. Many compilers use such an approach.

Figure 22 illustrates a conceptual picture of the Mojo architecture. In the current state of development, inputs are possible either via files in Petri Net Markup Language (PNML, only free-choice nets) [64] and BPMN or by direct programming of workflow graphs. There are predefined plug-ins for PNML and BPMN. These plug-ins consist of a parser that disassembles the files in their components, and a transformer that translates the entire workflows to semantically equivalent workflow graphs by abstracting language-dependent properties of the modeling language. For instance, the transformers combine different start nodes into a single one or they combine multiple end nodes by the algorithm of Kiepuszewksi [65]. Some of the features of BPMN are yet simplified and not fully supported. The current state of implementation is a prototype.

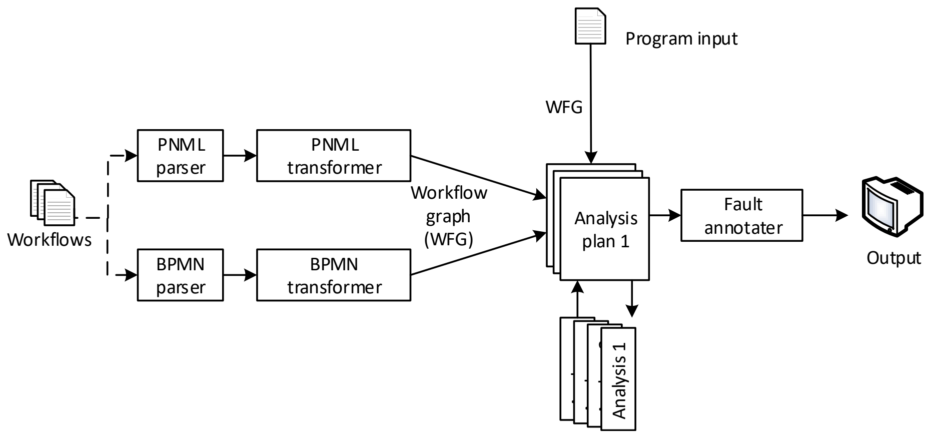
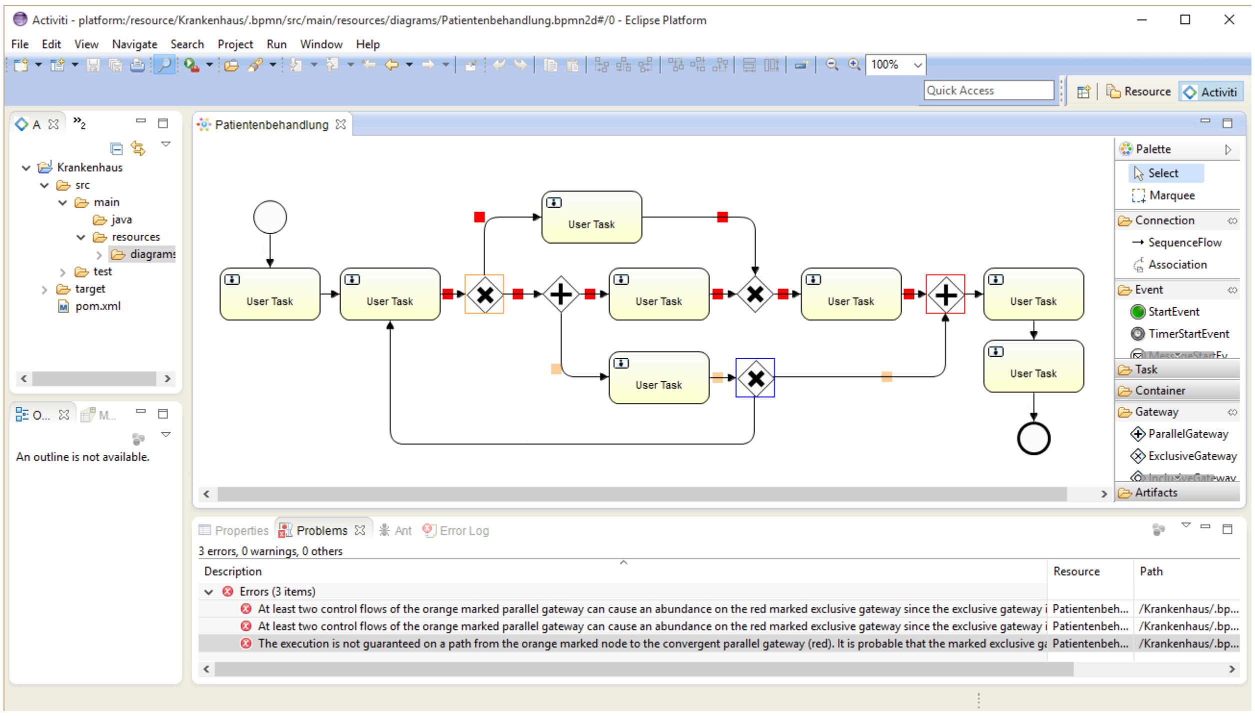


**Figure 22.** Conceptual architecture of Mojo

The (transformed) workflow graphs are explored by the analysis plans. Typical components of such analysis plans are the finding of causes of deadlocks and abundances. Which analysis plan is to be used is defined in the tool. Each analysis plan has a unique number for easy identification. For instance, the analysis plan with the number 0 evaluates the algorithms of this work and defines a general plan for the analysis of workflow graphs, which consists of two phases: 1) preparation and 2) analysis of causes of deadlocks and abundances.

The first phase (preparation) modifies the workflow graph to simplify further steps. For instance, it inserts additional task nodes to make multiple edges between nodes unique. Based on the information collected, the analysis of causes (the second phase) performs the algorithms introduced in this article. Any fault that has been detected is registered and annotated to the workflow graph. The faults are described with a textual label and a graphical highlighting within the development tool. To see the fault highlighting, Mojo must be integrated into a graphical workflow designer.

We have integrated Mojo into the *Activiti BPMN 2.0 Designer* (http://activiti.org/, last visited April 2021). This is easy since Mojo is a standalone Java library and an Eclipse plug-in. Figure 23 shows the integration. In *each* modification step of a workflow, Mojo performs analyses without visible delay. In this way, Mojo transforms the internal workflow model of *Activiti* into at least one workflow graph. An appropriate analysis plan is then applied. Our *Activiti* extensions visualize the collected analysis information for the developer. The developer can view the faults in two different modes. The *overview* mode illustrates *all* faults with reduced textual and graphic information. It provides an overview of the faults within the workflow. The *detail* mode visualizes exactly one fault (selected by the developer) with all available diagnostic information. This mode can be used to correct the workflow.


Activiti - platform:/resource/Krankenhaus/.bpmn/src/main/resources/diagrams/Patientenbehandlung.bpmn2d#/0 - Eclipse Platform

3 errors, 0 warnings, 0 others

| Description | Resource | Path |
|---|---|---|
| Errors (3 items) | | |
| At least two control flows of the orange marked parallel gateway can cause an abundance on the red marked exclusive gateway since the exclusive gateway i | Patientenbeh... | /Krankenhaus/.bp... |
| At least two control flows of the orange marked parallel gateway can cause an abundance on the red marked exclusive gateway since the exclusive gateway i | Patientenbeh... | /Krankenhaus/.bp... |
| The execution is not guaranteed on a path from the orange marked node to the convergent parallel gateway (red). It is probable that the marked exclusive g | Patientenbeh... | /Krankenhaus/.bp... |

**Figure 23.** Integration of Mojo in the Activiti BPMN 2.0 Designer

## 8 Evaluation

In this section, the introduced algorithms are evaluated with regard to their quality and quantity of the errors found. In addition, the runtime behavior is evaluated in a real context. We used the previously introduced tool Mojo to check the soundness property for a process library with about 1,000 real workflows.

### 8.1 Test Settings

The test environment for Mojo was a commercially available personal computer with a 64-Bit *Debian GNU/Linux 9.0 (stretch)* operating system. The version of the Linux kernels was *4.8.0-2-amd64 x86_64*. The PC used a *4-Core Intel©Core™ i5-4570* processor with *8 GB* main memory. *Mojo* ran on an OpenJDK in version *1.8.0_111*. The committed memory was 2.048 MB.

For quantitative evaluations, we used a library of real process models of the *IBM WebSphere Business Modeler*. This library contains 1,368 workflows separated into five benchmarks $A$ (282 workflows), $B1$ (288 workflows), $B2$ (363 workflows), $B3$ (421 workflows), and $C$ (32 workflows). The library contains workflows in different sizes, structures, and behaviors and was made freely available on the internet by IBM Zurich in the original XML file format of the WebSphere modeler [4]. The usage without high effort is possible by using the PNML file format [64]. PNML describes Petri nets in a simple syntax. The reader was able to find the workflows of the library as PNML files in the context of the work of Fahland et al. [4] [5]. These PNML files were also used in this evaluation.

We considered the workflows of the library with three tools: 1) *Mojo*, 2) *LoLA* [47], and 3) *jBPT* [66]. The tool LoLA uses state-space exploration and requires Petri nets in a special, proprietary file format. The necessary files can be found on the internet (see reference to URLs above). The project *Business Process Technologies 4 Java* provides its tool jBPT. It is open source and also available on the internet [6]. It contains an implementation of the SESE decomposition of Vanhatalo et al. [38], [18]. Based on the simple rules of Vanhatalo et al., we created a correctness validation for well-structured fragments — so-called *bonds* being produced by the construction of the refined process structure tree [18]. Such a bond is a typical structure of an opening and closing node as it is known from programming (e. g., if-then-else constructs). Our implementation of the correctness validation checks the fitting of the opening to the closing node. If the opening and closing nodes do not fit (e. g., a split node is closed by a join node), there is a fault. The heuristic rules of Vanhatalo et al. were *not* considered. In the following, SESE refers to the analysis with the tool jBPT since jBPT's algorithms are based on the SESE decomposition. We used the SESE approach since it was recognized as the most efficient technique available [4] in 2011. Although the approach is attested as incomplete, a pre-computation of the structured workflow parts is promising.

## 8.2 Detailed Comparison of the Tools

First, the evaluation compares in detail the results of the application of all three tools/techniques with the example BPMN workflow in Figure 24.

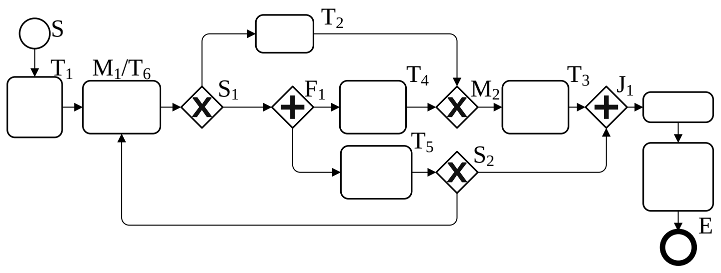


**Figure 24.** Example workflow for the comparison of the analysis tools

**Mojo.** Applying the tool Mojo to the example workflow produces a similar overview of problems as shown in Figure 25. A developer will find clues to each cause of an error: 1) a

[4] https://web.archive.org/web/20130513144952/http://www.zurich.ibm.com/csc/bit/downloads.html, last visited April 2021

[5] https://web.archive.org/web/20131208132841/http://service-technology.org/publications/fahlandfjklvw_2009_bpm, last visited April 2021

[6] https://code.google.com/archive/p/jbpt/, last visited April 2021

potential deadlock in a join node, 2) a possible abundance in a misaligned synchronization, and 3) a potential production of any number of control-flows. The latter appears when the outgoing edge of a fork node is an intersection. In this case, the control-flows of the fork are not synchronized before the fork's next execution. This can lead to the production of an arbitrary number of tokens. In the following, we call it an abundance in a *loop*.

Each cause of error can be considered in detail. To obtain detailed diagnostic information, Mojo applies further simple techniques. In most cases, these techniques use a simple depth-first search. Their runtime behavior is at least as efficient as the approaches described in this article.

Figure 26 shows a detailed view of the cause of a potential deadlock in the example workflow. The additional information is obtained by an inverse depth-first search. This information helps the developer to see on which path the join node can block and which node is causing this blocking.

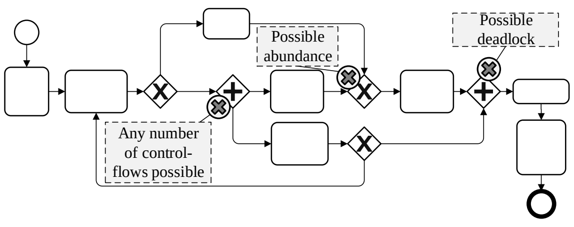


**Figure 25.** An overview of faults with Mojo

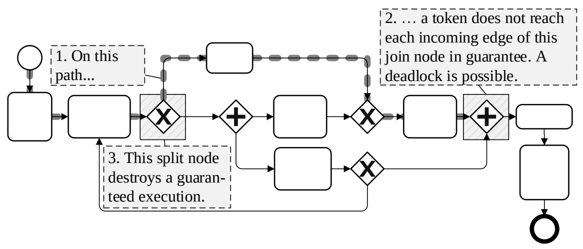

1. On this path...
2. … a token does not reach each incoming edge of this join node in guarantee. A deadlock is possible.
3. This split node destroys a guaranteed execution.

**Figure 26.** Detailed fault feedback of the cause of the potential deadlock in the join node $J_1$

Mojo generates similarly detailed information for the potential abundance in the merge node, Figure 27. A developer of the workflow sees the paths where two control-flows can collide for the first time. Additional information is derived directly from the maximum flow. The same is true for the fork node $F_1$, which produces any number of control-flows, i. e., an abundance in a loop, Figure 28. In Figure 28, one path marks the route without intersection, so that the corresponding fork node can be executed arbitrarily often.

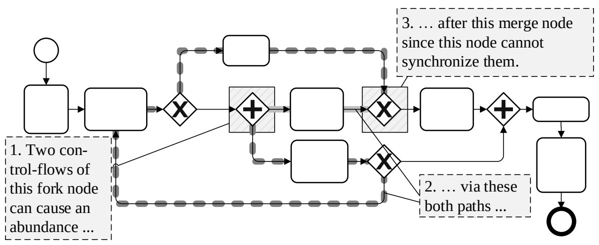

3. … after this merge node since this node cannot synchronize them.
1. Two control-flows of this fork node can cause an abundance ...
2. … via these both paths ...

**Figure 27.** Detailed fault feedback for the potential abundance in the merge node $M_2$

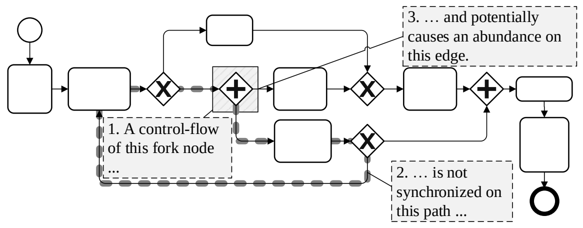

3. … and potentially causes an abundance on this edge.
1. A control-flow of this fork node ...
2. … is not synchronized on this path ...

**Figure 28.** Detailed feedback for a fault where a fork node can generate any number of control-flows

In summary, Mojo offers a detailed failure diagnostic regarding the causes of errors.

**LoLA.** LoLA's state-space exploration can determine exactly one error. In addition to the kind of error (deadlock or abundance), the approach provides a *failure trace*. Such a failure trace is like a control-flow from the start edge containing the erroneous state (cf. Figure 29). One way to visualize a failure trace is a simulation. In this simulation, the developer sees step by step how the error is reached. It seems good that the developer receives error feedback for a failure, which may actually occur at runtime. However, our previous work [10], [11] shows that finding the cause of the error is very difficult due to fault distance, masking, illusion, and blocking [9], [7]. Furthermore, most state-based approaches provide only a single error since full exploration of the state-space often ends in a state-space explosion as mentioned earlier.

The result of a state-based approach is sufficient, but does not provide much diagnostic information for the developer.

**SESE.** The SESE decomposition leads to a separation of the workflow into smaller fragments (cf. Figure 30). However, Figure 30 shows that most nodes of the workflow are located in an unstructured fragment. Unfortunately, all faults lie within this unstructured subgraph and for these only heuristics exist that can determine errors. For this example, the heuristics of Vanhatalo et al. [18] cannot be applied. As a result, the application of the SESE decomposition cannot give error feedback to the developer. However, SESE decomposition has its advantages for simple, well-structured workflows since the fault is easily recognizable in illustrations.

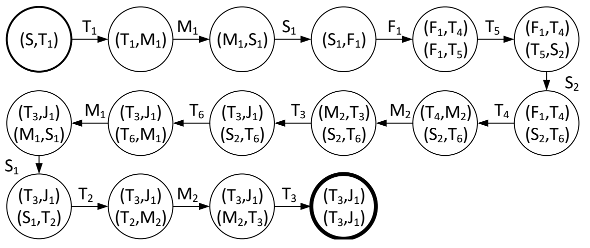


**Figure 29.** Failure trace up to a deadlock and abundance in the example workflow

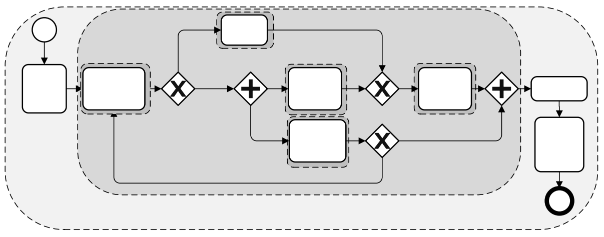


**Figure 30.** SESE decomposition of the example workflow

*Conclusion.* In summary, Mojo offers the best overview and the best diagnostic details for the exemplary workflow.

## 8.3 Causes of Errors

In addition to the detailed consideration of a single example, we have applied all three tools to the process library mentioned above. First, we considered which workflows of the library are marked as unsound by the tools and how many faults/errors the tools find.

**Mojo.** Mojo identified 742 workflows as unsound and 644 as sound. Table 1 summarizes the number of different kinds of faults (deadlocks and abundances) of the different benchmarks. Abundances seem to occur more frequently than deadlocks. Table 2 (column *Mojo*), p. 34, summarizes these results and allows their comparison with the other tools. The total number of identified faults is high at 4 007. Each workflow therefore contains an average of 2 to 3 faults.

**Table 1.** Number (of causes) of deadlocks and abundances

| | **Mojo** | | **LoLA** | | **SESE** | |
|---|---|---|---|---|---|---|
| | **Deadlocks** | **Abundances** | **Deadlocks** | **Abundances** | **Deadlocks** | **Abundances** |
| A | 140 | 170 | 97 | 68 | 0 | 0 |
| B1 | 273 | 720 | 81 | 200 | 30 | 98 |
| B2 | 326 | 948 | 84 | 238 | 36 | 113 |
| B3 | 289 | 1 056 | 83 | 262 | 18 | 118 |
| C | 24 | 61 | 10 | 14 | 8 | 1 |
| Sum | 1 052 | 2 955 | 355 | 782 | 92 | 330 |
| Total | 4 007 | | 1 137 | | 422 | |

**LoLA.** Table 1, column *LoLA*, shows the number of deadlocks and abundances identified by the tool LoLA. LoLA identifies only a quarter of the number of errors (not faults!) compared to Mojo. Since LoLA uses a state-space exploration, it stops the analysis at the first identified error. In the standard configuration of LoLA, it performs one analysis for deadlocks and one for abundances. Therefore, LoLA can detect a maximum of 2 errors. This explains the small number of identified errors.

In a detailed comparison of all workflows, we found the following list of differences between the results of both tools, Mojo and LoLA:

1. LoLA identifies an abundance where Mojo identifies nothing.
2. LoLA identifies an abundance where Mojo only identifies causes for deadlocks.
3. LoLA identifies an abundance and a deadlock where Mojo only identifies causes for abundances.
4. LoLA identifies an abundance where Mojo identifies causes for abundances and deadlocks.
5. LoLA identifies a deadlock where Mojo identifies causes for abundances and deadlocks.

The worst case is that LoLA identifies an abundance where Mojo identifies nothing (1.). This happens twice in the library. We have looked at both workflows in detail. The "fault" seems to be that both workflows are unconnected. Since Mojo was developed to support the development process of workflows, it can also handle unconnected graphs. It automatically constructs a connected workflow based on the semantics of BPMN. In the resulting two workflows, there is no fault. This might be the cause of the divergence but we do not know how LoLA handles unconnected Petri nets.

Similar happens in the case that LoLA identifies an abundance where Mojo only identifies causes for deadlocks (point 2 in the list of differences) — the cause of a deadlock cannot lead to an abundance. This case occurs only once in the library and also that workflow is unconnected. It contains a cause of deadlock that is easy to check. But LoLA identifies an abundance due to its different strategy.

The last three mentioned workflows are the only ones in the library that are unconnected. The library contains further 3 workflows in which LoLA reaches both an abundance and a deadlock, but Mojo only identifies causes for abundances (point 3 in the list of differences). In a detailed examination of these workflows, the deadlocks identified by LoLA are the result of abundances in the workflow — these are cases of fault illusion [11]. In 171 workflows, LoLA leads to abundances only where Mojo finds causes of deadlocks and abundances (point 4 in the list of differences). Each time LoLA finds an abundance, it interrupts its execution for both abundance and deadlock analysis — these are cases of fault blocking [11]. Finally, there are 205 workflows in which Mojo finds causes for deadlocks and abundances, but LoLA identifies only deadlocks (point 5 in the list of differences). This happens since a state-based approach like the one used in LoLA cannot find errors behind deadlocks having more cases of fault blocking.

As an intermediate result, it can be emphasized that Mojo has identified more faults and gave more detailed and accurate diagnostic information.

**SESE.** We also applied SESE to the process library. Table 1, column *SESE*, shows the results for SESE. For the benchmark $A$, SESE provides less faults and diagnostic information, since most faults are in unstructured subgraphs. The total number of identified faults is also low. This is surprising because SESE finds faults behind other faults like Mojo. But faults seem to occur more often in unstructured than in well-formed subgraphs. For this reason, our implementation of SESE is not able to find these faults.

*Conclusion.* Table 2 emphasizes the differences between Mojo, LoLA, and SESE. The derivation for these numbers in the case of SESE was more difficult. First, we identified well-formed workflows. If SESE did not identified a fault, these workflows are sound. If a fault was detected, then the workflows are unsound, regardless of whether it is well-formed or not. For all other workflows, it is unknown whether they are sound or not. Overall, Mojo uses the best approach to detect faults in workflows.

**Table 2.** The number of sound, unsound, and unknown workflows identified by the different tools

| | | Mojo | | LoLA | | SESE | | |
|---|---|---|---|---|---|---|---|---|
| | Total | Sound | Unsound | Sound | Unsound | Sound | Unsound | Unknown |
| A | 282 | 152 | 130 | 152 | 130 | 65 | 0 | 217 |
| B1 | 288 | 107 | 181 | 107 | 181 | 75 | 92 | 121 |
| B2 | 363 | 161 | 202 | 161 | 202 | 121 | 103 | 139 |
| B3 | 421 | 207 | 214 | 207 | 214 | 144 | 107 | 170 |
| C | 32 | 17 | 15 | 15 | 17 | 9 | 7 | 16 |
| Sum | 1 386 | *644 | *742 | *642 | *744 | 414 | 309 | 663 |
| * Please see the text for the explanation of the reasons for the differences | | | | | | | | |

## 8.4 Time Behavior

Finally, the evaluation examines how much time Mojo spends on its analyses. For its application in practice, the time required is an important quality criterion. In 2009, a soundness analysis about 5 seconds for a single workflow was described as efficient [24]. But a latency of 5 seconds is too slow to apply for soundness analysis at development time.

The consideration of the time behavior is difficult, because there are side effects of hardware, operating system, and Java runtime environment. More detailed information about how the following times were measured can be found in the appendix of the previous work [10].

Figure 31 shows the distribution of the analysis times spent for the different workflows with Mojo. The main subset of the workflows (about 87%) needs less than 1 and 95% less than 2 milliseconds. Mojo is fast and should be able to analyze a workflow without visible delay in a workflow designer. It requires a minimum of $0.03ms$ per workflow, a median $0.25ms$, and a maximum of $23.25ms$.

Figure 32 summarizes the total analysis times. The deadlock and abundance analysis runs in parallel. Therefore about half of the total analysis time is needed for input, transformations, and others. In total, Mojo required $817.68ms$.

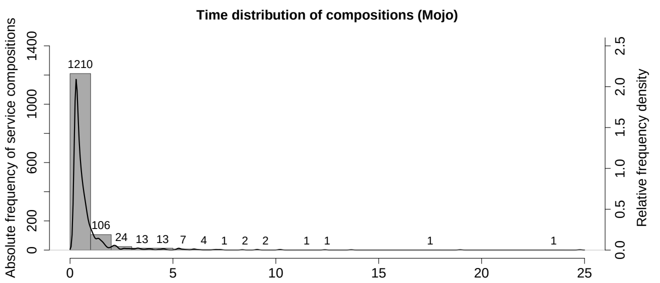


**Figure 31.** Distribution of the analysis times for Mojo

| | Mojo | | |
|---|---|---|---|
| | DL *ms* | AB *ms* | Total *ms* |
| A | 75.55 | 89.73 | 169.53 |
| B1 | 70.62 | 92.92 | 165.92 |
| B2 | 79.46 | 93.45 | 196.60 |
| B3 | 129.10 | 157.13 | 258.04 |
| C | 8.53 | 12.28 | 27.60 |
| Sum | 363.27 | 445.50 | 817.68 |
| DL = deadlock analysis, AB = abundance analysis | | | |

**Figure 32.** Analysis times for the library with Mojo

In addition to the total runtimes, the asymptotic runtime behavior of Mojo was also considered as a function of the size of the entire workflow graph. An important indicator for the size of a workflow is the number of edges, $|E|$. Figures 33 and 34 show the analysis times concerning the number of edges. The former takes into account the time in milliseconds; the latter shows the asymptotic runtime behavior. The asymptotic behavior is determined by the exponent $e$ of $|E|^e$. The line was interpolated for a smooth line. Otherwise, it would be difficult to detect the trend. The time spent increases with the number of edges, obviously (Figure 33). The asymptotic runtime behavior tends to a value of 2.0 (Figure 34). In practice, Mojo seems to have a quadratic asymptotic runtime $O(E^2)$.

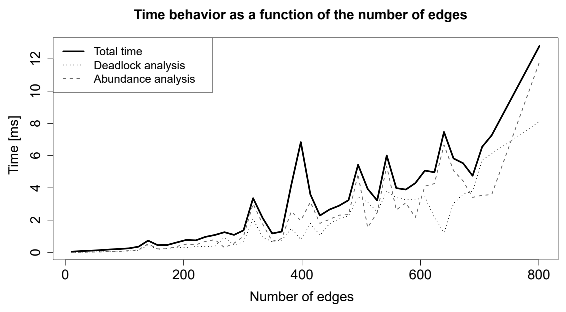


**Figure 33.** Runtime as a function of the number of edges

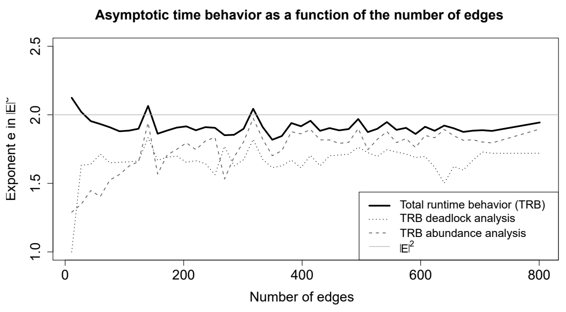


**Figure 34.** Asymptotic runtime behavior as a function of the number of edges

In summary, Mojo's better diagnostic information is not the result of time-consuming analyses. In practice, the runtime complexity is quadratic.

## 9 Conclusion

The *soundness* notion describes the absence of errors such as deadlocks and abundances instead of their faults. Therefore, the usual analysis techniques for soundness checking try to find only errors. However, there is a gap between the error and its causes (the faults), the so-called fault distance. This gap becomes larger in cases of fault blocking, masking, and illusion [10], [11]. Our motivation for this article was to develop new techniques that directly find causes of deadlocks and abundances.

This article reviewed the basics of workflow verification. It recapitulated the existing techniques in the state of the art. These techniques are either incomplete, have sparse diagnostic information, are restricted by finding only errors, or are too complex in terms of their asymptotic runtime complexity. We showed that it is possible to define soundness through the absence of the *causes* of deadlocks and abundances. This definition can be used to find qualitative fault analysis results. In the worst case, our approaches have a cubic asymptotic runtime complexity, although the approaches provide accurate diagnostic information. Our approaches are also able to find faults without visible delays during construction. The algorithms are based on a partial analysis, which allows a partial consideration of the workflow behavior.

In short, deadlocks occur when at least one, but not all, incoming edges of a join node receive tokens. To avoid deadlocks in join nodes, either none or all of their incoming edges must get tokens. This is ensured by the activation edges of the join nodes. A token on an activation edge follows the execution of the join node in the future. If a path to a join node does not pass an activation edge, then a token can reach the join node, but it is not guaranteed that the join node will be executed — the workflow can lock. This article presented algorithms, how the activation edges of a join node and how the deadlock faults can be found.

Abundances occur when two control-flows are not synchronized. Such a synchronization should happen when two control-flows meet for the first time. These locations are called intersections. Finding causes of abundances is to check whether it is possible to reach an intersection with two control-flows at the same time. This is possible when two disjoint paths from a fork node to an intersection are executable by two independent control-flows. Independence is ensured if there is no join node on both paths where the intersection is an activation edge. The resulting algorithm is based on the SSA form and the maximum flow problem.

We validated that the algorithms convince in practice. All algorithms are implemented in our tool Mojo. An evaluation with a workflow library and two further soundness detection approaches leads to more faults, more diagnostic information, and fast analysis times for Mojo. Regarding this tool comparison, our algorithms seem to be the best currently used in research.

In the future, we want to investigate in a study how novices and domain experts use Mojo in a business process modeler like Activiti, how helpful they find the diagnostic information, and how well Mojo influences the design process. Furthermore, we want to extend the algorithms to handle workflows with Or-join nodes, as they occur frequently in practice. Our first complete semantics might be helpful [67]. Another research topic is the overall system for the development, storage, and execution of workflows [61]. So far Mojo only accepts the structure of workflows without data information. As mentioned in earlier work [61], defining a clean intermediate representation for workflows that allows both control-flow and data-flow analyses is profitable. One possibility is the extended workflow graph [68]. Another option is a fold-out graphs [63] based on the concepts of SafeTSA [69]. Besides the pure intermediate representation, the algorithms have to be extended to process the data information for soundness analysis. For instance, after the application of the restructuring and enfolding of Heinze et al. [70], [71], [72], [73], our algorithms can be applied. All these algorithms can also have an application in a runtime environment. Such a runtime environment is a virtual machine that reads, verifies, and executes a workflow [62]. The verification step can be accelerated by annotating diagnostic information. The diagnostic information can be also used to correct the workflow during a simulation.

## References

[1] T. H. Davenport, *Process Innovation: Reengineering Work Through Information Technology.* Boston, MA, USA: Harvard Business School Press, 1993.

[2] W. Kirk, *Public Management: Gestaltung von Dienstleistungen im allgemeinen Interesse - Prozessmanagement (Public Management: Designing services of general interest - Process management) [Series Die öffentliche Verwaltung der Bundesrepublik Deutschland auf dem Weg zum Verwaltungsbetrieb (The public administration of the Federal Republic of Germany on its way to administrative operation), Volume 8]*, 1st ed. Norderstedt, Germany: Books on Demand, Dec. 2010.

[3] Object Management Group (OMG), "Business Process Model and Notation (BPMN) Version 2.0. formal/2011-01-03. http://www.omg.org/spec/BPMN/2.0." OMG, Jan. 2011, standard. [Online]. Available: http://www.omg.org/spec/BPMN/2.0

[4] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, "Analysis on demand: Instantaneous soundness checking of industrial business process models," *Data Knowl. Eng.*, vol. 70, no. 5, pp. 448–466, 2011. [Online]. Available: https://doi.org/10.1016/j.datak.2011.01.004

[5] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Inf. Syst.*, vol. 25, no. 2, pp. 117–134, 2000. [Online]. Available: https://doi.org/10.1016/S0306-4379(00)00012-0

[6] W. M. P. van der Aalst, "The application of petri nets to workflow management," *Journal of Circuits, Systems, and Computers*, vol. 8, no. 1, pp. 21–66, 1998. [Online]. Available: https://doi.org/10.1142/S0218126698000043

[7] IEEE Computer Society, "IEEE Standard Glossary of Software Engineering Terminology. 610.12-1990," Dec 1990.

[8] B. Marick, *The Craft of Software Testing: Subsystem Testing. Including Object-Based and Object-Oriented Testing*, 1st ed. New Jersey, USA: Prentice Hall PTR, Jan. 1995.

[9] M. A. Friedman and J. M. Voas, *Software Assessment: Reliability, Safety, Testability (Series New Dimensions In Engineering Series, Book 16)*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., Aug. 1995.

[10] T. M. Prinz and W. Amme, "Why We Need Advanced Analyses of Service Compositions," in *SERVICE COMPUTATION 2017: The Ninth International Conferences on Advanced Service Computing, Athens, Greece, February 19–23, 2017. Proceedings*, M. de Barros, J. Klink, T. Uhl, and T. M. Prinz, Eds. ThinkMind Digital Library, 2017, pp. 48–54.

[11] T. M. Prinz and W. Amme, "Why We Need Static Analyses of Service Compositions — Fault vs. Error Analysis of Soundness," *International Journal on Advances in Intelligent Systems*, vol. 10, no. 3 & 4, pp. 458–473, Dec. 2017, ISSN 1942-2679.

[12] W. Reisig, "The linear theory of multiset based dynamic systems," in *Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View [Workshop on Multiset Processing, WMP 2000, Curtea de Arges, Romania, August 21-25, 2000]*, C. Calude, G. Puaun, G. Rozenberg, and A. Salomaa, Eds., vol. 2235, Lecture Notes in Computer Science. Springer, 2000, pp. 287–298. [Online]. Available: https://doi.org/10.1007/3-540-45523-X_15

[13] K. H. Rosen, *Discrete Mathematics and Its Applications*, 7th ed. New York, USA: Mcgraw-Hill Education Ltd, Aug. 2012.

[14] G. Chartrand and P. Zhang, *Discrete Mathematics*, 1st ed. Long Grove, Illinois, USA: Waveland Press, Inc., Mar. 2011, iSBN 978-1577667308.

[15] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Massachusetts, USA: The MIT Press, Dec. 2013.

[16] M. Bossert and M. Breitbach, *Digitale Netze [Serie Informationstechnik] (Digital networks [Information technology series])*, 1st ed. Stuttgart, Leipzig: Teubner Verlag, Mar. 1999.

[17] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York, NY, USA: ACM Press Frontier Series, Jan. 1991.

[18] J. Vanhatalo, H. Völzer, and F. Leymann, "Faster and more focused control-flow analysis for business process models through SESE decomposition," in *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, B. J. Krämer, K. Lin, and P. Narasimhan, Eds., vol. 4749, Lecture Notes in Computer Science. Springer, 2007, pp. 43–55. [Online]. Available: https://doi.org/10.1007/978-3-540-74974-5_4

[19] H. Völzer, "A new semantics for the inclusive converging gateway in safe processes," in *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings*, R. Hull, J. Mendling, and S. Tai, Eds., vol. 6336, Lecture Notes in Computer Science. Springer, 2010, pp. 294–309. [Online]. Available: https://doi.org/10.1007/978-3-642-15618-2_21

[20] W. M. P. van der Aalst, "Interval timed coloured petri nets and their analysis," in *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, M. A. Marsan, Ed., vol. 691, Lecture Notes in Computer Science. Springer, 1993, pp. 453–472. [Online]. Available: https://doi.org/10.1007/3-540-56863-8_61

[21] K. R. Apt, N. Francez, and S. Katz, "Appraising fairness in languages for distributed programming," *Distributed Computing*, vol. 2, no. 4, pp. 226–241, 1988. [Online]. Available: https://doi.org/10.1007/BF01872848

[22] E. Kindler and W. M. P. van der Aalst, “Liveness, fairness, and recurrence in petri nets,” *Inf. Process. Lett.*, vol. 70, no. 6, pp. 269–274, 1999. [Online]. Available: https://doi.org/10.1016/S0020-0190(99)00074-5

[23] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989.

[24] M. T. Wynn, H. M. W. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, and D. Edmond, “Business process verification - finally a reality!” *Business Proc. Manag. Journal*, vol. 15, no. 1, pp. 74–92, 2009. [Online]. Available: https://doi.org/10.1108/14637150910931479

[25] W. M. P. van der Aalst, “A class of Petri nets for modeling and analyzing business processes,” Eindhoven University of Technology, Eindhoven, Netherlands, Computing Science Reports 95/26, 1995, technical Report.

[26] F. Puhlmann, “Soundness verification of business processes specified in the pi-calculus,” in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, R. Meersman and Z. Tari, Eds., vol. 4803, Lecture Notes in Computer Science. Springer, 2007, pp. 6–23. [Online]. Available: https://doi.org/10.1007/978-3-540-76848-7_3

[27] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn, “Soundness of workflow nets: classification, decidability, and analysis,” *Formal Asp. Comput.*, vol. 23, no. 3, pp. 333–363, 2011. [Online]. Available: https://doi.org/10.1007/s00165-010-0161-4

[28] J. Desel and J. Esparza, *Free Choice Petri Nets (Cambridge Tracts in Theoretical Computer Science 40).* Cambridge, Great Britain: Cambridge University Press, 1995, iSBN 0-521-46519-2.

[29] P. Kemper and F. Bause, “An efficient polynomial-time algorithm to decide liveness and boundedness of free-choice nets,” in *Application and Theory of Petri Nets 1992, 13th International Conference, Sheffield, UK, June 22-26, 1992, Proceedings*, K. Jensen, Ed., vol. 616, Lecture Notes in Computer Science. Springer, 1992, pp. 263–278. [Online]. Available: https://doi.org/10.1007/3-540-55676-1_15

[30] C. Favre and H. Völzer, “Symbolic execution of acyclic workflow graphs,” in *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings*, R. Hull, J. Mendling, and S. Tai, Eds., vol. 6336, Lecture Notes in Computer Science. Springer, 2010, pp. 260–275. [Online]. Available: https://doi.org/10.1007/978-3-642-15618-2_19

[31] W. M. P. van der Aalst, A. Hirnschall, and H. M. W. Verbeek, “An alternative way to analyze workflow graphs,” in *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Özsu, Eds., vol. 2348, Lecture Notes in Computer Science. Springer, 2002, pp. 535–552. [Online]. Available: https://doi.org/10.1007/3-540-47961-9_37

[32] C. Favre, D. Fahland, and H. Völzer, “The relationship between workflow graphs and free-choice workflow nets,” *Inf. Syst.*, vol. 47, pp. 197–219, 2015. [Online]. Available: https://doi.org/10.1016/j.is.2013.12.004

[33] W. M. P. van der Aalst and A. H. M. ter Hofstede, “YAWL: yet another workflow language,” *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, 2005. [Online]. Available: https://doi.org/10.1016/j.is.2004.02.002

[34] P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto, "A top-down petri net-based approach for dynamic workflow modeling," in *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*, W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, Eds., vol. 2678, Lecture Notes in Computer Science. Springer, 2003, pp. 336–353. [Online]. Available: https://doi.org/10.1007/3-540-44895-0_23

[35] R. Johnson, D. Pearson, and K. Pingali, "Finding regions fast: Single entry single exit and control regions in linear time. tr 93-1365." Cornell University, Ithaca, NY, USA, Technical Report, Jul. 1993.

[36] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, V. Sarkar, B. G. Ryder, and M. L. Soffa, Eds. ACM, 1994, pp. 171–185. [Online]. Available: https://doi.org/10.1145/178243.178258

[37] C. S. Ananian, "The Static Single Information Form. mit-lcs-tr-801. technical report," Massachusetts Institute of Technology (MIT), Tech. Rep., Sep. 1999, [Online]. Available on http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-801.pdf.

[38] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," *Data Knowl. Eng.*, vol. 68, no. 9, pp. 793–818, 2009. [Online]. Available: https://doi.org/10.1016/j.datak.2009.02.015

[39] J. E. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM J. Comput.*, vol. 2, no. 3, pp. 135–158, 1973. [Online]. Available: https://doi.org/10.1137/0202012

[40] J. Vanhatalo, H. Völzer, F. Leymann, and S. Moser, "Automatic workflow graph refactoring and completion," in *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings*, A. Bouguettaya, I. Krüger, and T. Margaria, Eds., vol. 5364, Lecture Notes in Computer Science, 2008, pp. 100–115. [Online]. Available: https://doi.org/10.1007/978-3-540-89652-4_11

[41] S. Kühne, H. Kern, V. Gruhn, and R. Laue, "Business process modeling with continuous validation," *Journal of Software Maintenance and Evolution*, vol. 22, no. 6-7, pp. 547–566, 2010. [Online]. Available: https://doi.org/10.1002/smr.517

[42] W. M. P. van der Aalst, "Verification of workflow nets," in *Application and Theory of Petri Nets 1997, 18th International Conference, ICATPN '97, Toulouse, France, June 23-27, 1997, Proceedings*, P. Azéma and G. Balbo, Eds., vol. 1248, Lecture Notes in Computer Science. Springer, 1997, pp. 407–426. [Online]. Available: https://doi.org/10.1007/3-540-63139-9_48

[43] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, W. Reisig and G. Rozenberg, Eds., vol. 1491, Lecture Notes in Computer Science. Springer, 1996, pp. 429–528. [Online]. Available: https://doi.org/10.1007/3-540-65306-6_21

[44] A. Cheng, J. Esparza, and J. Palsberg, "Complexity results for 1-safe nets," *Theor. Comput. Sci.*, vol. 147, no. 1–2, pp. 117–136, 1995. [Online]. Available: https://doi.org/10.1016/0304-3975(94)00231-7

[45] N. Lohmann and D. Fahland, "Where did I go wrong? - explaining errors in business process models," in *Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings*, S. W. Sadiq, P. Soffer, and H. Völzer, Eds., vol. 8659, Lecture Notes in Computer Science. Springer, 2014, pp. 283–300. [Online]. Available: https://doi.org/10.1007/978-3-319-10172-9_18

[46] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst, “Diagnosing workflow processes using woflan,” *Comput. J.*, vol. 44, no. 4, pp. 246–279, 2001. [Online]. Available: https://doi.org/10.1093/comjnl/44.4.246

[47] K. Schmidt, “Lola: A low level analyser,” in *Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000, Aarhus, Denmark, June 26-30, 2000, Proceeding*, M. Nielsen and D. Simpson, Eds., vol. 1825, Lecture Notes in Computer Science. Springer, 2000, pp. 465–474. [Online]. Available: https://doi.org/10.1007/3-540-44988-4_27

[48] R. Eshuis and A. Kumar, “An integer programming based approach for verification and diagnosis of workflows,” *Data Knowl. Eng.*, vol. 69, no. 8, pp. 816–835, 2010. [Online]. Available: https://doi.org/10.1016/j.datak.2010.03.003

[49] B. F. van Dongen, J. Mendling, and W. M. P. van der Aalst, “Structural patterns for soundness of business process models,” in *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China.* IEEE Computer Society, 2006, pp. 116–128. [Online]. Available: https://doi.org/10.1109/EDOC.2006.56

[50] C. Favre, H. Völzer, and P. Müller, “Diagnostic information for control-flow analysis of workflow graphs (a.k.a. free-choice workflow nets),” in *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, M. Chechik and J. Raskin, Eds., vol. 9636, Lecture Notes in Computer Science. Springer, 2016, pp. 463–479. [Online]. Available: https://doi.org/10.1007/978-3-662-49674-9_27

[51] Y. Choi, P. Kongsuwan, C. M. Joo, and J. L. Zhao, “Stepwise structural verification of cyclic workflow models with acyclic decomposition and reduction of loops,” *Data Knowl. Eng.*, vol. 95, pp. 39–65, 2015. [Online]. Available: https://doi.org/10.1016/j.datak.2014.11.003

[52] P. J. Pahl and R. Damrath, *Mathematical Foundations of Computational Engineering: A Handbook*, 1st ed. Berlin, Germany: Springer, Jul. 2001.

[53] B. Alpern, M. N. Wegman, and F. K. Zadeck, “Detecting equality of variables in programs,” in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, J. Ferrante and P. Mager, Eds. ACM Press, 1988, pp. 1–11. [Online]. Available: https://doi.org/10.1145/73560.73561

[54] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Global value numbers and redundant computations,” in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, J. Ferrante and P. Mager, Eds. ACM Press, 1988, pp. 12–27. [Online]. Available: https://doi.org/10.1145/73560.73562

[55] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[56] T. Lengauer and R. E. Tarjan, “A fast algorithm for finding dominators in a flowgraph,” *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, 1979. [Online]. Available: https://doi.org/10.1145/357062.357071

[57] K. D. Cooper, T. J. Harvey, and K. Kennedy, “A Simple, Fast Dominance Algorithm. tr-06-33870. technical report,” Department of Computer Science, Rice University, , Houston, Texas, USA, Tech. Rep., 2001, [Online]. Available on https://www.cs.rice.edu/~keith/EMBED/dom.pdf.

[58] T. E. Harris and F. S. Ross, *Fundamentals of a Method for Evaluating Rail Net Capacities*, 1st ed. Santa Monica, California, USA: Armed Services Technical Information Agency, Oct. 1955, rM-1573. A report prepared for United States Air Force Project RAND.

[59] J. Lestor R. Ford and D. R. Fulkerson, *Flows in Networks*, 1st ed. Santa Monica, California, USA: Princeton University Press, Aug. 1962, r-375-PR. A report prepared for United States Air Force Project RAND.

[60] T. M. Prinz, N. Spieß, and W. Amme, "A first step towards a compiler for business processes," in *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, A. Cohen, Ed., vol. 8409, Lecture Notes in Computer Science. Springer, 2014, pp. 238–243. [Online]. Available: https://doi.org/10.1007/978-3-642-54807-9_14

[61] T. M. Prinz, T. S. Heinze, W. Amme, J. Kretzschmar, and C. Beckstein, "Towards a Compiler for Business Processes - A Research Agenda," in *SERVICE COMPUTATION 2015: The Seventh International Conferences on Advanced Service Computing, Nice, France, March 22–27, 2015. Proceedings*, M. de Barros and C.-P. Rückemann, Eds., 2015, pp. 49–54.

[62] T. M. Prinz, "Proposals for a virtual machine for business processes," in *Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015, Jena, Germany, February 19-20, 2015.*, ser. CEUR Workshop Proceedings, T. S. Heinze and T. M. Prinz, Eds., vol. 1360. CEUR-WS.org, 2015, pp. 10–17. [Online]. Available: http://ceur-ws.org/Vol-1360/paper2.pdf

[63] T. M. Prinz, R. Charrondière, and W. Amme, "Geschäftsprozesse kompiliert - Wichtige Unterstützung für die Modellierung (business processes compiled — important support for the construction)," in *Proceedings 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach am Wörthersee, Austria, October 5–7, 2015*, J. Knoop and M. A. Ertl, Eds., 2015, pp. 476–491.

[64] M. Weber and E. Kindler, "The petri net markup language," in *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets*, H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, Eds., vol. 2472, Lecture Notes in Computer Science. Springer, 2003, pp. 124–144. [Online]. Available: https://doi.org/10.1007/978-3-540-40022-6_7

[65] B. Kiepuszewski, A. H. M. ter Hofstede, and W. M. P. van der Aalst, "Fundamentals of control flow in workflows," *Acta Inf.*, vol. 39, no. 3, pp. 143–209, 2003. [Online]. Available: https://doi.org/10.1007/s00236-002-0105-4

[66] A. Polyvyanyy and M. Weidlich, "Towards a compendium of process technologies - the jbpt library for process model analysis," in *Proceedings of the CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering (CAiSE), Valencia, Spain, June 20th, 2013*, ser. CEUR Workshop Proceedings, R. Deneckère and H. A. Proper, Eds., vol. 998. CEUR-WS.org, 2013, pp. 106–113. [Online]. Available: http://ceur-ws.org/Vol-998/Paper14.pdf

[67] T. M. Prinz and W. Amme, "A complete and the most liberal semantics for converging OR gateways in sound processes," *CSIMQ*, vol. 4, pp. 32–49, 2015. [Online]. Available: https://doi.org/10.7250/csimq.2015-4.03

[68] W. Amme, A. Martens, and S. Moser, "Advanced verification of distributed WS-BPEL business processes incorporating cssa-based data flow analysis," *IJBPIM*, vol. 4, no. 1, pp. 47–59, 2009. [Online]. Available: https://doi.org/10.1504/IJBPIM.2009.026985

[69] W. Amme, J. von Ronne, and M. Franz, “Ssa-based mobile code: Implementation and empirical evaluation,” *TACO*, vol. 4, no. 2, p. 13, 2007. [Online]. Available: https://doi.org/10.1145/1250727.1250733

[70] T. S. Heinze, “Eine methode zur kontrollierten kontrollflussentfaltung und ihre anwendung zur präzisierung petrinetzbasierter verifikationsmodelle (a method for controlled control-flow-unfolding and its application to the refinement of petri-net-based verification models),” Ph.D. dissertation, Friedrich Schiller University of Jena, Germany, 2013. [Online]. Available: http://d-nb.info/104689952X

[71] T. S. Heinze, W. Amme, and S. Moser, “A restructuring method for WS-BPEL business processes based on extended workflow graphs,” in *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009. Proceedings*, U. Dayal, J. Eder, J. Koehler, and H. A. Reijers, Eds., vol. 5701, Lecture Notes in Computer Science. Springer, 2009, pp. 211–228. [Online]. Available: https://doi.org/10.1007/978-3-642-03848-8_15

[72] T. S. Heinze, W. Amme, and S. Moser, “Compiling more precise petri net models for an improved verification of service implementations,” in *7th IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2014, Matsue, Japan, November 17-19, 2014*. IEEE Computer Society, 2014, pp. 25–32. [Online]. Available: https://doi.org/10.1109/SOCA.2014.8

[73] T. S. Heinze, W. Amme, and S. Moser, “Process restructuring in the presence of message-dependent variables,” in *Service-Oriented Computing - ICSOC 2010 International Workshops, PAASC, WESOA, SEE, and SOC-LOG, San Francisco, CA, USA, December 7-10, 2010, Revised Selected Papers*, E. M. Maximilien, G. Rossi, S. Yuan, H. Ludwig, and M. Fantinato, Eds., vol. 6568, Lecture Notes in Computer Science. Springer, 2010, pp. 121–132. [Online]. Available: https://doi.org/10.1007/978-3-642-19394-1_13

[74] R. Hull, J. Mendling, and S. Tai, Eds., *Business Process Management - 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13-16, 2010. Proceedings*, vol. 6336, Lecture Notes in Computer Science. Springer, 2010. [Online]. Available: https://doi.org/10.1007/978-3-642-15618-2

[75] J. Ferrante and P. Mager, Eds., *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988.* ACM Press, 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=73560

# Appendix

*Proof (Theorem 1).* This proof is done by mathematical induction. A path $P_l = (e_0, \ldots, e_l), l \geq 1$, is assumed and induced over its length $l$. The basis of the induction, $l = 1$, is trivial: Either there is a strictly reachable state of $\langle e_0 \rangle$, in which $e_1$ has a token (and thus such a control-flow exists), or, there is no such state, but then $\langle e_0 \rangle$ is already a deadlock.

In the step case, $l$, it is assumed that this theorem is valid for all paths of length $l$. It is checked whether it is also valid for all paths of length $l + 1$. There are two cases for the path $P_l$ and a matching control-flow $f_{e_0}$:

**1. Case** $f_{e_0}$ has a deadlock. Therefore, it also has a deadlock for $l + 1$. ✓

**2. Case** $f_{e_0}$ puts a token at each edge of path $P_l$. Let $S_l \in f_{e_0}$ be the state that contains $e_l$. For $S_l$ is valid:

1. There is a reachable state $S_{l+1}$ from $S_l$ in which $e_{l+1}$ has a token. So there is a control-flow of $e_0$ which of course puts a token at each edge of $P_{l+1}$. ✓
2. There is no reachable state from $S_l$ containing $e_{l+1}$. Therefore, $f_{e_0}$ ends in a deadlock since $\mathit{tgt}(e_l)$ is active but never executed. ✓

*Proof (Theorem 2).* Let $(S_0, S_1, S_2, \ldots) = f_e \in \mathcal{F}_e$ be a control-flow of edge $e$ and $S' = S \setminus \langle e \rangle$ be the state without one token on $e$. There are two cases:

**Case** 1 $f_e$ is *infinite*. A computation that contains $f_e$, can be made by the set union of each state of the control-flow with the state $S'$: $(S_0 \cup S', S_1 \cup S', S_2 \cup S', \ldots) \in \mathcal{C}_S$.

**Case** 2 $f_e$ is *finite*. Let $f_e$ end in a state $S_l$, $l \in \mathbb{N}$. A resulting computation is similar to the infinite case. However, another computation that starts in the state $S_l \cup S'$ must be extended (depicted with the concatenation symbol $+$) to meet Definition 9:

$$\big((S_0 \cup S', S_1 \cup S', S_2 \cup S', \ldots, S_l \cup S') + c_{S_l \cup S'}\big) \in \mathcal{C}_S, \quad c_{S_l \cup S'} \in \mathcal{C}_{S_l \cup S'}$$

*Proof (Theorem 3).* Suppose $(S_0, S_1, \ldots)$, $S_0 = \langle e \rangle$, $e \in S$, is a longest (maybe infinite) sequence of strictly reachable states that is "part of" $c_S$ ($\forall i \in \{0, 1, \ldots\}\colon\ \exists S \in c_S\colon S_i \subseteq S$). If this sequence is not a control-flow of $e$, then by definition the sequence must be unfinished, i. e., there is a directly reachable state from the sequence's last state $S_l$ (the sequence must be finite). Therefore, $S_l$ is part of $c_S$, but each direct reachable state $S_{l+1}$ is not. In other words, in $S_l$ is an executable node $n$, whose execution leads to $S_{l+1}$. This node $n$, however, is obviously executable in each state $S' \in c_S$, $S_l \subseteq S'$, and the sequence above could be extended. But in this case, however, our sequence is not a longest one $\lightning$. I. e., this sequence *must be* a control-flow.

*Proof (Theorem 4).* Considering the theorem, there are two independent cases: The join node $j$ blocks in a control-flow of 1) the start edge *entry* or 2) the outgoing edge *out* of $j$. The theorem claims that the workflow graph *WFG* is unsound in both cases.

**Case 1):** Each control-flow of *entry* is equal to a computation from the initial state. If it contains a deadlock, then *WFG* is unsound by definition. ✓

**Case 2):** Proof by contradiction: *WFG* is sound, but $j$ has a deadlock in a state $S_{dead}$ in a control-flow $f_{out}$ of *out*.

In $S_{dead}$, there are some incoming edges of $j$ with tokens, and there are some incoming edges of $j$ without token. Let $\mathit{Missing} \subset \triangleright j$ be the set of edges that have no tokens in $S_{dead} \cap \mathit{Missing} = \emptyset$. Also $\mathbb{S}_{missing}$ should be states, $\mathbb{S}_{missing} \subseteq \mathcal{S}(E)$, where each computation of these states guarantees at least one token at each edge of *Missing* (*without* executing $j$).

Since *WFG* is assumed as sound by contradiction, there is a control-flow from the start edge *entry*, where the edge *out* of $j$ gets a token. Let $S_{out}$ be such a state. $S_{out}$ consists of the edge *out* and a multiset *Add* of edges, $S_{out} = \langle \mathit{out} \rangle \cup \mathit{Add}$. Based on the assumption, the following holds:

$$\langle \mathit{entry} \rangle \rightarrow^* \big(\langle \mathit{out} \rangle \cup \mathit{Add}\big) \rightarrow^* \big(S_{dead} \cup \mathit{Add}\big) \quad \Longrightarrow \quad \mathit{Add} \neq \emptyset \qquad \text{otherwise: } \langle \mathit{entry} \rangle \rightarrow^* S_{dead} \ \lightning$$

Let $\mathcal{A}\mathit{dd}$ be the set of all sets of edges *Add* that are (combined with *out*) reachable from the initial state: $\mathcal{A}\mathit{dd} = \{\mathit{Add} \subseteq E\colon \langle \mathit{entry} \rangle \rightarrow^* \langle \mathit{out} \rangle \cup \mathit{Add}\}$.

$\langle \mathit{entry} \rangle \rightarrow^* \big(\langle \mathit{out} \rangle \cup \mathit{Add}\big) \rightarrow^* \big(S_{dead} \cup \mathit{Add}\big)$ is valid for each $\mathit{Add} \in \mathcal{A}\mathit{dd}$. If $\mathcal{A}\mathit{dd} \nsubseteq \mathbb{S}_{missing}$, then the execution of $j$ is not guaranteed from $S_{dead} \cup \mathit{Add}$. *WFG* can end in a deadlock $\lightning$. Therefore, it must be valid that $\mathcal{A}\mathit{dd} \subseteq \mathbb{S}_{missing}$.

In other words, every time *out* gets a token, it is in a state $\langle \mathit{out} \rangle \cup S_{missing}, S_{missing} \in \mathbb{S}_{missing}$. From this state, it is guaranteed that all edges of *Missing* get tokens in a subsequent state $S_{sub}, \triangleright j \subseteq S_{sub}, \langle \mathit{out} \rangle \cup S_{missing} \rightarrow^* S_{sub}$. Let $\mathbb{S}_{sub}$ be all such subsequent states $S_{sub}$, $\mathbb{S}_{sub} = \{S_{sub} \subseteq \mathcal{S}(E)\colon \triangleright j \subseteq S_{sub}, \langle \mathit{out} \rangle \cup S_{missing} \rightarrow^* S_{sub}, S_{missing} \in \mathbb{S}_{missing}\}$. Since such a $S_{sub}$ is reached every time, we depict it with $\rightarrow^!$ for simplification: $\langle \mathit{out} \rangle \cup S_{missing} \rightarrow^! S_{sub}$.

Each time *out* gets a token, it is in a state $\langle \mathit{out} \rangle \cup S_{missing}, S_{missing} \in \mathbb{S}_{missing}$. And: $\langle \mathit{out} \rangle \cup S_{missing} \rightarrow^! S_{sub}, S_{sub} \in \mathbb{S}_{sub}$. It must hold:

$$\langle \mathit{entry} \rangle \rightarrow^* \big(\langle \mathit{out} \rangle \cup S_{missing}\big) \rightarrow^! S_{sub} \rightarrow^! \big(\langle \mathit{out} \rangle \cup S'\big),\ S' \in \mathcal{S}(E) \quad \Longrightarrow \quad S' \in \mathbb{S}_{missing}$$

In other words, in any computation where *out* gets a token, the computation is infinite:

$$\langle \mathit{entry} \rangle \rightarrow^* \big(\langle \mathit{out} \rangle \cup S_{missing}\big) \rightarrow^! S_{sub} \rightarrow^! \big(\langle \mathit{out} \rangle \cup S'_{missing}\big) \rightarrow^! S'_{sub} \rightarrow^! \ldots$$

Then, however, the execution of *WFG* never ends in the termination state — *WFG* is not sound or not fair $\lightning$.

*Proof (Theorem 5).* We prove both directions $\Longleftarrow$ and $\Longrightarrow$.

$\Longleftarrow$ $\forall \mathit{in} \in \triangleright j\colon\ \forall P \in \mathcal{P}_{entry \to in}\colon P \cap \curvearrowright(j) \neq \emptyset \implies \forall f_{entry} \in \mathcal{F}_{entry}\colon j$ does not have an immediate deadlock in $f_{entry}$.

This is a constructive proof. There are two complete cases for each control-flow $f_{entry} \in \mathcal{F}_{entry}$ in $\mathcal{EG}(j)$:

**Case 1:** $f_{entry}$ does *not* reach an activation edge of $\curvearrowright(j)$. Since each path to an incoming edge of $j$ has an activation edge, no token reaches an incoming edge of $j$ (otherwise, the control-flow $f_{entry}$ would pass an activation edge of $j$). This means that no token reaches an incoming edge of $j$, why $j$ does not have an immediate deadlock in $f_{entry}$. ✓

**Case 2:** $f_{entry}$ reaches an activation edge of $\curvearrowright(j)$. Therefore, $f_{entry}$ delivers all incoming edges of $j$ with token and no immediate deadlock is possible at $j$ in $f_e$. ✓

$\Longrightarrow$ Proof by contradiction:

$$\begin{aligned} & \forall f_{entry} \in \mathcal{F}_{entry} \colon j \text{ does not have an immediate deadlock in } f_{entry} \\ \wedge \quad & \exists in \in \triangleright j \colon \; \exists P \in \mathcal{P}_{entry \to in} \colon P \cap \curvearrowright (j) = \emptyset \end{aligned}$$

Let $P$ be a such a path without activation edges of $j$ from *entry* to an incoming edge *in* of $j$. Also, let $a \in P$ be the first edge that is an activation edge of at least one incoming edge of $j$. There is at least *in* on that path, which is an activation edge of itself, then $a = in$. Remember, $a \overset{all}{\not\curvearrowright} j$. Since the label of all join nodes in the entry graph is *Safe*, there is no deadlock in the entry graph. As a consequence of Theorem 1, at least one control-flow $f_{entry}$ can put a token at each edge of $P$ to $a$, which successively leads to a state $S$ with a token on $a$. In each reachable state of $S$, in which no token can travel anymore, there is at least one incoming edge of $j$ with a token. But not in all such reachable states *all* incoming edges of $j$ have tokens, otherwise $a$ would be an activation edge for all incoming edges; $j$ blocks immediately in these states. $\lightning$

*Proof (Theorem 6).* This is a constructive proof. Let $P_a$ and $P_b$ be two routes to $\iota$. Theorem 1 states that tokens can traverse each path from one edge to another edge in a sound workflow graph. In the following proof, we will only consider those computations where the tokens of *fork* strictly follow paths $P_a$ and $P_b$. For this reason, tokens on $a$ and $b$ can travel on $P_a$ and $P_b$. But since *WFG* is sound, they cannot reach $\iota$ at the same time. Without loss of generality, there is a computation from the start edge that executed *fork* and where a token travelled from $a$ via $P_a$ to $\iota$, but the token of $b$ on $P_b$ could yet have reached $\iota$. Suppose the former token of $b$ is now at edge $e$ on $P_b$ in a state $S$. There are two cases:

Case 1 After $e$, path $P_b$ does not pass a join node that prevents the token from arriving at $\iota$ as long as the former token of $a$ is on it. Regarding Theorem 1, the token can arrive at $\iota$ at the same time as the former token of $a$. There is an abundance. $\lightning$

Case 2 After $e$, the path $P_b$ passes at least one join node that prevents the token from arriving at $\iota$ as long as the former token of $a$ is on it. Let $join \in N_{Join}$, $join\triangleleft = \{out\}$, $out \in P_b$, be the last join node that hinders the token on $P_b$ as long as the former token of $a$ is on $\iota$. There are exactly two cases with the former token of $a$ on $\iota$:

Case 2a There is at least one computation from $S$ where the tokens strictly follow $P_a$ and $P_b$ and where *join* can*not* be executed *after* the former token of $a$ leaves $\iota$. Since *join* has the former token of $b$ at one of its incoming edges and it also cannot be executed before the former token of $a$ leaves $\iota$, there is a deadlock. $\lightning$

Case 2b In all computations from $S$ where the tokens strictly follow $P_a$ and $P_b$, *join* can be executed *after* the former token of $a$ leaves $\iota$. Since *join* cannot be executed until the former token of $a$ has left $\iota$, this leaving of $\iota$ must always lead to tokens on a non-empty set of incoming edges of *join* in all computations. $\iota$ must be an activation edge for at least one incoming edge of *join*. But $\iota$ cannot be an activation edge of *join* since *join* would receive tokens for all its incoming edges, i. e., also for the incoming edge that already carries the former token of $b$. ✓

*Proof (Theorem 7).* The proof is done by contradiction:

$$\begin{gathered} \text{There is a route } (P_a, P_b) \text{ from } a \text{ and } b \text{ to } \iota \\ \text{where } \iota \text{ is } \textbf{not} \text{ an activation edge for a real non-empty subset of the incoming edges of any join node on } P_a \text{ and } P_b \\ src(\iota) \notin N_{Join} \;\wedge\; WFG \text{ is sound} \end{gathered}$$

Or in other words:

$$\begin{gathered} \text{There is a route } (P_a, P_b) \text{ from } a \text{ and } b \text{ to } \iota \\ \text{where } \iota \textbf{ is} \text{ or } \textbf{is not} \text{ an activation edge of any incoming edge of any join node on } P_a \text{ and } P_b \\ src(\iota) \notin N_{Join} \;\wedge\; WFG \text{ is sound} \end{gathered}$$

Let $P_a$ and $P_b$ be two routes to $\iota$ for which the above equation applies. Theorem 1 states that any path can be traversed by a token in a sound workflow graph. Otherwise, there would be a deadlock. Therefore, in at least one computation starting in the initial state, a token of *fork* can arrive at $\iota$ via the path $P_a$ without loss of generality. In the following proof, we will only consider those computations where the tokens of *fork* follow strictly the paths $P_a$ and $P_b$. Well, there are two cases:

Case 1 There is at least one computation in which the former token of $a$ remains at $\iota$ until the other former token of $b$ reaches $\iota$ via $P_b$. Since $src(\iota) \notin N_{Join}$, an abundance is on $\iota$. $\lightning$

Case 2 There is no computation where the former token of $a$ remains on $\iota$ until the other former token of $b$ reaches $\iota$ via $P_b$. In other words, in *each* computation (where the tokens strictly follow $P_a$ and $P_b$), the token via $P_a$ must not remain on $\iota$ until the other token via $P_b$ reaches $\iota$. I. e., in each such computation the former token of $a$ must leave $\iota$ *before* the former token of $b$ reaches $\iota$ via $P_b$. In fact, there is no node semantics that can force the former token of $a$ at $\iota$ to leave $\iota$ before the former token of $b$ reaches $\iota$ via $P_b$. The only way to force the former token of $a$ to leave $\iota$ is to require that the former token of $b$ reaches $\iota$ via $P_b$. And this is again only possible if there is a node on $P_b$ whose execution depends on a token at $\iota$. The only node that cannot be executed by the token via $P_b$ alone is a join node. I. e., there is a join node on $P_b$ whose execution depends on a token on $\iota$. Let *join* be a join node on $P_b$, $\{out\} = join\triangleleft$, $out \in P_b$, whose execution depends on $\iota$. Let the workflow graph be in a state in which the token via $P_b$ lies on the incoming edge *in* of *join*, $in \in P_b$. There are two subcases:

Case 2a $\iota \overset{all}{\curvearrowright} join$. From the former token of $a$ on $\iota$ follows in the guarantee at least one token on each incoming edge of *join*. Therefore, *in* of *join* also gets an additional token. An abundance on *in* is possible. $\lightning$

Case 2b $\forall in \in \triangleright join \colon \iota \not\curvearrowright in$. If the former token of $a$ leaves $\iota$ and since $\iota$ does not guarantee a token on any incoming edge of *join*, there is at least one reachable state in which at least one incoming edge $in' \in \triangleright join$, $in' \neq in$, gets no token. I. e., there is a reachable state in which *join* is not executed, but has a token on *in*: a deadlock. $\lightning$

All cases have contradictions. The theorem holds.