A Novel Graph Reduction Algorithm to Identify Structural Conflicts *

Hao Lin
Department of Electronics
Engineering, Tsinghua
University
linhao@sdp.ee.tsinghua.edu.cn

Zhibiao Zhao
Department of Electronics
Engineering, Tsinghua
University
zhaozb@sdp.ee.tsinghua.edu.cn

Hongchen Li
School of Computer
Science, Henan University
lhchen@cs.net4.cs.tsinghua.edu.cn

Zhiguo Chen
School of Computer
Science, Henan University
chenzg@mail.henu.edu.cn

Abstract

The algorithm [1] is based on a set of graph reduction rules to identify the deadlock and lack of synchronization conflicts that could compromise the correct execution of a workflow. However, an example, which is apparently correct but cannot be reduced by the rules, shows the incompleteness of the algorithm. In this paper, we present a complete and minimal set of rules and a novel algorithm to implement the identification of structural conflicts in process models. The correctness and completeness of the algorithm is strictly proved. Finally, the complexity of the algorithm is analyzed.

Keywords:
Business Process Modeling, Verification of Process Models, Workflow Management Systems

1. Introduction

A process model is the formal definition of business processes and the correctness of a process model directly affects the business objective. There are several aspects in a process model including structure, data flow, roles, application interface, temporal constrains, and others, of which the structure of a workflow plays a primary and important role in the execution of the scheduling and coordinating workflow tasks and represents the building block for a workflow management. Thus, it is important to ensure the correctness of the structure of a workflow.

Various approaches to workflow modeling can be found in the literature [1, 4, 5, 6, 7, 8, 9]. The Workflow Management Coalition [2, 3] is developing a standard process definition language and an interface specification that could be used to transfer process models between products. However, we have found limited works on the verification of workflow modeling languages. In [1, 8, 10] several verification issues of workflow structures have been examined and complexity of certain verification problems has been shown. However, no strict proof of completeness has been shown. In [11, 12] the reduction and synthesis techniques in Petri nets have been explored for workflow verification.

The work presented in this paper focuses on the verification of a workflow graph. It proposes a complete and minimal set of graph reduction rules and an effective algorithm for workflow verification. This paper is the first one to present a strict proof of the algorithm correctness and completeness.

In this paper we first describe a problem in the algorithm [1] in section 2. A complete and minimal set of rules and a novel algorithm to identify two kinds of structural conflicts are presented in Section 3. Section 4 presents a strict proof of the algorithm correctness and completeness and the analysis of the algorithm complexity. Section 5 offers some conclusions.

2. Problem Description

2.1. Fundamentals of Process Modeling Language

We shall first quote some contributions and concepts in [1] including a typical workflow modeling language and its graphical representation, instance subgraphs of a workflow, two kinds of structural conflicts and the correctness criteria for workflow graphs.

Two types of objects, nodes and edges, are used in this graphical language. A transition graphically represented by a directed edge links two nodes in the graph and shows the execution order and flow between its head and tail nodes. A node is classified into two subclasses: task and choice/merge coordinator. A task represents the work done to achieve some objectives. In this paper, it is used to build sequence and fork (represented by a square) node, and synchronizer (represented by a rectangle) node. A choice/merge coordinator, graphically represented by a circle, is used to build choice and merge node.

In this paper, it is important to point out that if a task has two or more incoming edges and more than one outgoing edges, we will represent the task with a
synchronizer node connecting with a fork node, and if a choice/merge coordinator has two or more incoming edges and more than one outgoing edges, we will represent the choice/merge coordinator with a merge node connecting with a choice node. The five kinds of nodes are shown in Figure 1.

![Process modeling nodes](image)

**Figure 1: Process modeling nodes**

A sequence node is the most basic modeling node and defines the ordering of task execution. It has at most one incoming and one outgoing edge.

A fork (AND-split) node has two or more outgoing edges and is used to represent concurrent paths within a workflow graph.

A synchronizer (AND-join) node with more than one incoming edges is applied to synchronize such concurrent paths. A task waits until all incoming transitions (edges) have been triggered.

A choice (XOR-split) node has two or more outgoing edges and is used to model mutually exclusive alternative paths. At run-time, the workflow selects one of the alternative paths for a given instance of the business process by activating one of the transitions (edges) originating from the choice node. The choice node is exclusive and complete. The exclusive characteristic ensures that only one of the alternative paths is selected. The complete characteristic ensures that one of its outgoing flows will always be triggered.

A merge (XOR-join) node is “opposite” to the choice node. It has two or more incoming edges and joins mutually exclusive alternative paths into one path.

By connecting nodes with edges, directed acyclic graph (DAG) called workflow graph where vertices represent nodes and directed edges represent transitions is built. Therefore, a workflow graph has at least one node that has no incoming edge (source, namely begin) and at least one node that has no outgoing edge (sink, namely end).

There are two kinds of structural conflicts in process models: deadlock and lack of synchronization conflicts defined as follows:

- **Deadlock** - Joining exclusive choice paths with a synchronizer results into a deadlock conflict. A deadlock at a synchronizer node blocks the continuation of a workflow path since one or more of the proceeding edges of the synchronizer are not triggered.

- **Lack of Synchronization** - Joining fork concurrent paths with a merge node introduces lack of synchronization conflict in the process model. A lack of synchronization at a merge node results into unintentional multiple activation of nodes that follow the merge node.

It is also important to define the concept of instance subgraphs. An instance subgraph represents a subset of workflow tasks that may be executed for a particular instance of a workflow. It can be generated by visiting only one outgoing edge of a choice node and all outgoing edges of a fork node from the begin node to the end node.

The correctness criteria for instance subgraphs are defined as follows:

- Correctness Criteria 1 - deadlock free instance subgraphs: An instance subgraph is free of deadlock structural conflicts if it does not contain only a proper subset of the incoming nodes of a synchronizer node.

- Correctness Criteria 2 - lack of synchronization free instance subgraphs: An instance subgraph is free of lack of synchronization structural conflicts if it does not contain more than one incoming nodes of a merge node.

If and only if all instance subgraphs of a workflow graph meet Correctness Criteria 1 and 2 above, the workflow graph is structurally correct. We should point out here that structural conflicts are not the only type of errors possible in process models. However, they do present the primary source of errors in structural specifications. Other modeling aspects may also affect the correct execution of a workflow. However the existence of other errors does not introduce or remove structural conflicts. Therefore, the identification of structural conflicts in a workflow model can be performed independently from other types of verification analysis. In this paper, we will only discuss the problem of structural conflicts.

The verification process for workflow structures can be partitioned into two phases. In the first phase, basic syntax checking is performed to ensure that the model conforms to the modeling language syntax and that all necessary properties of its components have been defined. The verification of basic syntax is easy to facilitate and requires local analysis of workflow modeling objects and structures. The second phase of verification requires a rigorous analysis of the workflow model based on the criteria mentioned above.

### 2.2. The Problem of the Algorithm in [1]

Based on the fundamentals above, literature [1] presents a set of graph reduction rules to identify deadlock and lack of synchronization conflicts in workflows. However, the rules in [1] are not a complete set of operations to detect structural conflicts. Just take Figure 2(a) as an example, it is sure to be a correct workflow graph because its subgraphs as shown in Figure 2(b), (c), (d), (e) conform to Correctness Criteria 1 and 2. However, none of the graph reduction rules in [1] can be applied to reduce the workflow graph any further. Therefore, those rules are not a complete set of operations.
3. Graph Reduction Rules and Algorithm

In this section, we first present the graph reduction rules and an algorithm based on these rules to identify deadlock and lack of synchronization conflicts. Finally, we will show the reduction process of the workflow graph shown in Figure 2(a) and that the set of rules is minimal.

3.1. Reduction Rules

We will present a complete and minimal set of conflict-preserving rules to remove all nodes from the workflow graph that are definitely correct. Iteratively applying these rules will eventually reduce a structurally correct workflow graph to an empty graph. However, a workflow graph with structural conflicts is not completely reduced. This can be achieved by using the algorithm in Section 3.2.

We should point out here that the first four rules are quoted from [1].

**Rule 1: Terminal Reduction Rule.** The terminal reduction rule removes begin or end nodes from a workflow graph only if the number of edges attached to it is less than or equal to one.

Figure 3 shows an example of applying terminal reduction rule where begin and end nodes are removed to get a reduced graph from (a) to (b).

**Rule 2: Sequential Reduction Rule.** If a node with exactly one incoming and one outgoing edge, we can remove the node by changing the incoming edge to the outgoing edge. This is called sequential reduction rule.

Figure 3 shows an example of applying sequential reduction rule where we remove tasks $T_1, T_2, T_3, T_4, T_5$ and $T_6$ to get a reduced graph from (b) to (c).

**Rule 3: Adjacent Reduction Rule.** After applying the reduction rules mentioned above, the remaining nodes are either split or join nodes since they would either have out degree or in degree that is more than one. In this case, there are two kinds of nodes: one with the same type as the proceeding node, which has a single incoming edge and several outgoing edges, and the other with the same type as the succeeding node, which has a single outgoing edge and several incoming edges. Such nodes and their edges can be removed by connecting their proceeding node(s) with their succeeding node(s).

Figure 3 shows an example of applying adjacent reduction rule where we remove fork node $F_2$ and synchronizer node $S_1$ to get a reduced graph from (c) to (e).

**Rule 4: Closed Reduction Rule.** In general, the application of sequential and adjacent rules introduces closed components in workflow graphs. A closed component comprises of two nodes of the same type that have more than one edges between them. The closed reduction deletes all but one edge between such nodes.

Figure 3 shows an example of applying sequential reduction rule where we get a reduced graph from (e) to (f).

**Rule 5: Choice-Convergence Reduction Rule.** We call the structure shown in Figure 4(a) choice-convergence structure.

In this case, all outgoing edges of a choice are the incoming edges of the fork nodes, at least one outgoing edge of which reaches the same merge node. Then, we can transform this kind of graph into graph shown in Figure 4(b), where all edges from $F_i (i = 1, 2, ... n)$ to $M_1$ are...
removed and a new fork node $F_x$ is added with one of its outgoing edge reaching $M_1$ and the other reaching $C_1$.

**Rule 6: Synchronizer-Convergence Reduction Rule.**

We call the structure shown in Figure 5(a) synchronizer-convergence structure. In this case, all incoming edges of a synchronizer are the outgoing edges of merge nodes, at least one incoming edge of which is from the same fork node. Then, we can transform this kind of graph into graph shown in Figure 5(b), where all edges from $F_1$ to $M_i$ ($i = 1, 2, \ldots, n$) are removed and a new merge node $M_x$ is added with one of its incoming edge from $F_1$ and the other from $S_1$.

![Figure 5: Synchronizer-Convergence Reduction Rule](image)

**Rule 7: Merge-Fork Reduction Rule.** In this rule, if the workflow graph has the structure as shown in Figure 6(a), we call it merge-fork structure, which has a merge node connecting with a fork node. Then we can transform this kind of graph into graph shown in Figure 6(b), which has two layers of nodes. Every fork node at Layer 1 has an outgoing edge to each merge node at Layer 2.

![Figure 6: Merge-Fork Reduction Rule](image)

### 3.2. The Algorithm

The seven reduction rules will be applied in the following algorithm, which takes a workflow graph as input. By iteratively applying the transformation rules according to the orders in the algorithm, we reduce a correct workflow graph into an empty graph or show which workflow graph has structural conflicts.

**Algorithm: Reduce a workflow graph.**

**STEP 1:** Apply the reduction Rules 1 ~ 6 iteratively to the workflow graph until they cannot reduce the graph any further.

**STEP 2:** If the reduced graph after STEP 1 is empty, then the original graph is correct and the reduction process ends.

**STEP 3:** Apply the reduction Rules 1 ~ 4 and 7 iteratively to the workflow graph until they cannot reduce the graph any further.

**STEP 4:** If the graph does not change during STEP 1 ~ 3, then the original graph is incorrect and the reduction process ends. Otherwise, go to STEP 1.

We should note that applying Rules 5, 6 and Rule 7 in different steps is just for the optimization of algorithm. If we investigate the relationship of all the reduction rules, there may be more effective algorithms.

For the counterexample shown in Figure 2(a), we have two ways to reduce the workflow graph to an empty graph as shown in Figure 7. We can use Rules 1 ~ 4 and 6 first to reduce the original workflow graph to the graph as shown in Figure 7(b), and then by Rules 1 ~ 5 reduce it to the graph as shown in Figure 7(d) and finally into an empty one. We can also use Rules 1 ~ 5 first to reduce the original workflow graph to the graph as shown in Figure 7(c), and then by Rules 1 ~ 4 and 6 reduce it to the graph as shown in Figure 7(d) and finally an empty one. This shows that the performing sequence of these rules is arbitrary.

When we replace the dashed rectangles as shown in Figure 8(a) with the graph as shown in Figure 8(b), we shall find that the only rule we can apply first on this graph is Rule 7. From using Rule 7, we can finally reduce the graph to an empty graph. This example shows that Rules 1 ~ 7 are a minimal set of rules for workflow reduction processes.
4. Verification of the Algorithm Correctness And Completeness

In this section, we first present some useful definitions and the properties of them. Then, by introducing some lemmas, we present the proof of the algorithm correctness and completeness. Finally, we analyze the algorithm complexity.

4.1. Basic Definitions

There is a formal notation of workflow graphs that will be used in the algorithm verification and in the algorithm complexity analysis as follows:

**Definition 1.** The workflow graph $G = (N, E)$ is a simple directed acyclic graph (DAG) where

1. $N$ is a finite set of nodes;
2. $E$ is a finite set of directed edges representing transitions between two nodes.
3. $|G| = |N| + |E|$ represents the total number of nodes and edges in $G$.

The graph $G$ meets the following syntactical correctness properties:
- It uses only core modeling nodes, namely, sequence, choice, merge, fork and synchronizer;
- It does not contain any cycles, i.e., $\forall n, n \in N$, a path from $n$ to $n$ implies that $n \neq n$ (no self-loops) and no path exists from $n$ to $n$ (no cycles);
- It has exactly a single begin node $n_{\text{begin}}$;
- It has exactly a single end node $n_{\text{end}}$.

**Definition 2.** For each path $p$ from $n_i$ to $n_j$, where $n_i, n_j \in N$, we define:

\[ \text{pathNodes}[p] = \{ n_i, \ldots, n_j \} \]

represents a set of nodes contained within $p$.

**Definition 3.** For each node $n \in N$:

1. $\text{nodeType}[n] \in \{ \text{SEQUENCE}, \text{CHOICE}, \text{MERGE}, \text{FORK}, \text{SYNCHRONIZER} \}$ represents the node type of $n$;
2. $\text{outNodes}[n]$ represents a set of succeeding nodes that are adjacent to $n$;
3. $\text{inNodes}[n]$ represents a set of preceding nodes that are adjacent to $n$;
4. $\text{forwardNodes}[n] = \{ m : m \in N \text{ and } \exists p \text{ from } n \text{ to } n_{\text{end}} \text{ where } m \in \text{pathNodes}[p] \}$, i.e., a set consisting of $n$ and its succeeding nodes. For example, in Figure 2(a), forwardNodes[C1] = $\{ C2, F3, F4, M1, M2, M3, S1, S2, S3 \}$;
5. $\text{backwardNodes}[n] = \{ m : m \in N \text{ and } \exists p \text{ from } n_{\text{begin}} \text{ to } m \text{ where } m \in \text{pathNodes}[p] \}$, i.e., a set consisting of $n$ and its proceeding nodes. For example, in Figure 2(a), backwardNodes[C2] = $\{ C2, F1, C1 \}$ and backwardNodes[S1] = $\{ S1, M1, M2, F4, C2, F1, F3, F6, C3, F2, C1 \}$;

**Definition 4.** For each node $C_i \in N$ where nodeType[C] = CHOICE:

1. $\text{confirmedNodes}[C_i] = \{ m : m \in N \text{ and } \neg \exists n \in \text{outNodes}[C_i] \}$ where $m \notin \text{forwardNodes}[n]$ i.e., a set of nodes that are contained in forwardNodes[] of each node in outNodes[C]. For example, in Figure 2(a), confirmedNodes[C1] = $\{ M5, M1, M2, M3, M4, M6, S1, S2, S3 \}$, confirmedNodes[C2] = $\{ M5, M3, S2, M6, S3 \}$;
2. $\text{chosenNodes}[C_i] = \{ m : m \neq C_i, m \in \text{forwardNodes}[C_i] \}$ and $\exists n \in \text{outNodes}[C_i]$ where $m \notin \text{forwardNodes}[n]$. For example, in Figure 2(a), chosenNodes[C1] = $\{ F1, F2, C2, C3, F3, F4, F5, F6 \}$ and chosenNodes[C2] = $\{ F3, F4, M1, M2, S1 \}$;
3. $\text{mergedNodes}[C_i] = \{ m : m \in \text{confirmedNodes}[C_i] \}$ and $\exists n \in \text{inNodes}[m]$ where $n \notin \text{chosenNodes}[C_i] \cup \{ C_i \}$. For example, in Figure 2(a), mergedNodes[C1] = $\{ M5, M1, M2, M3, M4, M6 \}$ and mergedNodes[C2] = $\{ M3, M5 \}$;
4. $\text{insertedNodes}[C_i] = \{ m : m \in \text{chosenNodes}[C_i] \}$ and $\exists n \in \text{inNodes}[m]$ where $n \notin \text{forwardNodes}[C_i]$; For example, in Figure 2(a), insertedNodes[C1] = NULL and insertedNodes[C2] = $\{ M1, M2 \}$;
5. $\text{Exponent}[C_i]$ represents the number of the synchronizer nodes in chosenNodes[C]. For example, in Figure 2(a), Exponent[C1] = 0 and Exponent[C2] = 1;
6. $\text{extendedExponent}[C_i]$ represents the number of the synchronizer nodes in forwardNodes[C]. For example, in Figure 2(a), extendedExponent[C1] = 3 and extendedExponent[C2] = 3;

**Definition 5.** A reducible choice node is a choice node $C_i$ where the number of the choice nodes in chosenNodes[C] is 0, i.e., $\neg \exists n \in \text{chosenNodes}[C_i]$ with
nodeType[n] = CHOICE. For example, in Figure 2(a), C2 and C3 are reducible choice nodes.

Definition 6. A bottom choice node is a choice node Ci where the number of the choice nodes in forwardNodes[Ci] is 0, i.e., \( \neg \exists n \in \) forwardNodes[Ci] with nodeType[n] = CHOICE. For example, in Figure 2(a), C2 and C3 are bottom choice nodes.

Definition 7. A removable synchronizer node Sj of a choice node Ci is a synchronizer node where Sj \( \not\in \) chosenNodes[Ci] and \( \exists \, n \in \) chosenNodes[Ci] \( \cap \) backwardNodes[Ci] and \( n \neq S_j \) with nodeType[n] \( \in \{ \text{CHOICE, SYNCHRONIZER} \} \). For example, in Figure 2(a), S1 is a removable synchronizer of C2, and S2 is a removable synchronizer of C3.

Definition 8. The eigen of a workflow graph is defined by:

\[
\text{Eig} [G] = \begin{cases} 
-1 & \text{G is empty} \\
1 - nS(nS + 1) + E_{\text{min}} & \text{otherwise}
\end{cases}
\]

where \( n_c \) is the number of choice nodes in G, \( n_S \) is the number of synchronizer nodes in G, and

\[
E_{\text{min}} = \min_{C, C' \in N} \{ \text{extendExponent}[C, C'] \}
\]

For example, in the workflow graph shown in Figure 2(a), there are three choice nodes, C1, C2, and C3, and there synchronizer S1, S2, and S3. It can be obtained that:

\[
\text{extendExponent}[C1]=3,
\text{extendExponent}[C2]=3,
\text{extendExponent}[C3]=3.
\]

and then \( \text{Eig}[G] = 3(3 + 1) + 3 = 15 \).

It is easy here to establish certain elementary properties based on the definitions above:

Property 1. If and only if \( n_f \in \) forwardNodes[n] and \( n_f \neq n \), then \( \exists \, n_\ell \in \) outNodes[n] where \( n_\ell \in \) forwardNodes[n].

Property 2. If and only if \( n_\ell \in \) forwardNodes[n] and \( n_\ell \neq n \), then \( \exists \, n_\ell \in \) inNodes[n] where \( n_\ell \in \) forwardNodes[n].

Property 3. If and only if \( n_\ell \in \) backwardNodes[n] and \( n_\ell \neq n \), then \( \exists \, n_\ell \in \) inNodes[n] where \( n_\ell \in \) backwardNodes[n].

Property 4. If and only if \( n_\ell \in \) backwardNodes[n] and \( n_\ell \neq n \), then \( \exists \, n_\ell \in \) outNodes[n] where \( n_\ell \in \) backwardNodes[n].

Property 5. If and only if \( n_\ell \in \) forwardNodes[n], then \( n_\ell \in \) backwardNodes[n].

Property 6. If nodeType[C] = CHOICE, then forwardNodes[C] \( \in \) {C} \( \cup \) chosenNodes[C] \( \cup \) confirmedNodes[C].

Property 7. If \( n \in \) confirmedNodes[C] where nodeType[C] = CHOICE, and \( m \in \) forwardNodes[n], then \( m \in \) confirmedNodes[C].

Property 8. If \( n \in \) chosenNodes[C] where nodeType[C] = CHOICE, and \( m \in \) backwardNodes[n] where \( m \in \) forwardNodes[C], then \( m \in \) chosenNodes[C] or \( m = C \).

Property 9. If \( C_i \) is a choice node, then outNodes[C_i] \( \subset \) chosenNodes[C_i] \( \cup \) mergedNodes[C_i].

Property 10. If \( n \in \) chosenNodes[C] where nodeType[C] = CHOICE, then \( n \in \) chosenNodes[C] \( \cup \) mergedNodes[C].

4.2. Verification of algorithm correctness and completeness

Assume that the workflow graph \( G^{(k)} \) is the reduced graph after k times of using the reduction rules on G. We shall prove that:

Proposition 1. The reduction rules do not generate structural conflicts, i.e., if \( G^{(k)} \) is correct, then \( G^{(k+1)} \) is correct.

Proposition 2. The reduction rules do not create structural conflicts, i.e., if \( G^{(k)} \) is incorrect, then \( G^{(k+1)} \) is incorrect.

Proposition 3 (Correctness) If the algorithm reduces G to an empty graph, then G is correct.

Proposition 4 (Completeness) If G is correct, then the algorithm will always reduce G to an empty graph.

Because graph reduction Rules 1 and 2 are relative to logic relationships, and Rules 3 to 7 do not change the logic relationship, it is evident that using these rules cannot contribute toward a structural conflict or remove a structural conflict. Therefore, Proposition 1 and 2 are correct. Furthermore, it is clear that Proposition 2 \( \Rightarrow \) Proposition 3. Before proving Proposition 4, we need to prove some lemmas as follows.

Lemma 1. If the structure shown in Figure 9(a) exists in a workflow graph, then the workflow graph is incorrect.

Proof. From Figure 9(a), we know that e1 and e2 must be concurrently in the same instance subgraphs, i.e., if an instance subgraph contains e1 or e2, then it contains the other. Let I0 be an instance subgraph containing e4. To meet Criteria 2, I0 must not contain e2 and e1. Since I0 contains e3, we know that there is a deadlock conflict at S1 in I0. Consequently, there is at least a structural conflict -
deadlock or lack of synchronization- in the workflow graph, if the structure shown in Figure 9(a) exists.

**Lemma 2.** If the structure shown in Figure 9(b) is contained in a workflow graph, then the workflow graph is incorrect.

Proof. Let $I_1$ be an instance subgraph containing $C_1$ and choosing the path with the edge $e_1$. There is another instance subgraph $I_2$, which contains all nodes and edges in $I_1$ except the edge $e_1$ and chooses the path with the edge $e_2$ at the choice node $C_1$. Therefore, $I_2$ contains, at least, another path $p_0$ started with the node $F1$ and the edge $e_4$ besides those of $I_1$. Moreover, the extra path $p_0$ and the instance subgraph $I_1$ should meet at some nodes as shown in Figure 9(b). Since the meeting node $M_1$ has at least one incoming edge not belonging to $I_1$, to meet Criteria 1, $M_1$ cannot be a synchronizer node. Since $I_2$ contains at least two incoming edges of $M_1$, to meet Criteria 2, $M_1$ cannot be a merge node. Consequently, there is at least a structural conflict - deadlock or lack of synchronization- in the workflow graph, if the structure shown in Figure 9(b) exists.

**Lemma 3.** If the structure shown in Figure 9(c) is contained in a workflow graph, then the workflow graph is incorrect.

Proof. Let Suppose $I_1$ be a correct instance subgraph, which contains $C_1$ and chooses the path with the edge $e_2$. To meet Criteria 1, $I_1$ must contain the edge $e_3$. There is another instance subgraph $I_2$, which contains all nodes and edges in $I_1$ except the edge $e_2$ but chooses the path with the edge $e_1$ at the choice node $C_1$. Since $I_2$ contains $e_3$, by Criteria 1, we obtain that there is a deadlock conflict at $SI$ in the instance subgraph $I_1$. Consequently, there is at least a deadlock conflict in the workflow graph, if the structure shown in Figure 9(c) exists.

**Lemma 4.** Consider a correct workflow graph $G$, and let $n_0 \in N$, and $n \in forwardNodes[n_0]$. If an instance subgraph $I_1$ contains $n_0$ and $n$, then there exists a path from $n_0$ to $n$ in the instance subgraph $I_1$.

Proof. Suppose that there is an instance subgraph $I_1$, which contains $n_0$ and $n$, but does not contain any paths from $n_0$ to $n$. Let $p_1$ represent a path from $n_{begin}$ to $n$ contained in $I_1$. Since $n \in forwardNodes[n_0]$, there is another instance subgraph $I_2$, which contains all nodes contained in $I_1$ except the nodes in $forwardNodes[n_0]$. $I_2$ also contains $p_1$, but contains a path $p_2$ from $n_0$ to $n$. The paths $p_1$ and $p_2$ meet at a node $M_1 \in backwardNodes[n] \cap forwardNodes[n_0]$. Since the meeting node $M_1$ has at least one incoming edge not belonging to $I_1$, to meet Criteria 1, $M_1$ cannot be a synchronizer node. Since $I_2$ contains at least two incoming edges of $M_1$, to meet Criteria 2, $M_1$ cannot be a merge node. Therefore, we know that the supposition is not true, if the workflow graph is correct.

**Lemma 5.** Consider a correct workflow graph $G$, and let $C_i \in N$ where $nodeType[C_i] = CHOICE$. If $n \in insertedNodes[C_i]$, then $nodeType[n] = MERGE$.

Proof. Suppose that $nodeType[n] = SYNCHRONIZER$.

Let $I_1$ be a correct instance subgraph, which contains $C_i$ and $n$. To meet Criteria 1, $I_1$ contains all nodes in $inNodes[n]$.

By Definition 4(4), we deduce that $n \in chosenNodes[C_i]$ and $\exists m \in inNodes[m]$ where $m \in forwardNodes[C_i]$. Therefore, $I_1$ contains a path $p_1$ from $n_{begin}$ to $n$ containing $m$, and then $pathNodes[p_1] \cap forwardNodes[C_i] = \{ n \}$.

Since $p_1$ is independent on the choice of $C_i$, there should exist an instance subgraph $I_2$, which also contains $p_1$ and chooses the path with $n_1$ at $C_i$. Therefore, every path from $C_i$ to $n_{end}$ in $I_2$ must contain $n_1$.

Additionally, by Definition 4(2), we deduce that $\exists n_1 \in outNodes[C_i]$ where $n \notin forwardNodes[n_1]$. Consequently, there is no path from $C_i$ to $n$ in the instance subgraph $I_2$.

By Lemma 4, we know that the supposition is not true, and then $nodeType[n] = MERGE$.

**Lemma 6.** Consider a correct workflow graph $G$. Let $C_i \in N$ where $nodeType[C_i] = CHOICE$, $n \in outNodes[C_i]$ and $m \in chosenNodes[C_i]$ where $nodeType[m] = SYNCHRONIZER$. If $m \in forwardNodes[n]$, then $inNodes[m] \subseteq forwardNodes[n]$.

Proof. By Lemma 5, we deduce that $m \notin insertedNodes[C_i]$, and then $inNodes[m] \subseteq forwardNodes[C_i]$. By Lemma 3 and Property 8, we obtain that $C_i \notin inNodes[m]$, and then $inNodes[m] \subseteq chosenNodes[C_i]$.

Consider a correct instance subgraph $I_1$ having a path from $C_i$ to $m$ containing $n$. Since $C_i$ is a choice node, every path from $C_i$ to $n_{end}$ in $I_1$ must contain $n$.

To meet Criteria 1, $I_1$ contains all nodes in $inNodes[m]$. By Lemma 4, there exists a path from $C_i$ to each node in $inNodes[m]$, which contains $n$. Consequently, $inNodes[m] \subseteq forwardNodes[n]$.

**Lemma 7.** Consider a correct workflow graph $G$, and let $C_i \in N$ where $nodeType[C_i] = CHOICE$. If $n \in mergedNodes[C_i]$, then $nodeType[n] = MERGE$.

Proof. By Definition 4(3), we obtain that $\exists m \in inNodes[n]$ where $m \in chosenNodes[C_i] \cup \{ C_i \}$. Then we only have the following two cases:

Case 1. $m = C_i$. By Lemma 3, we deduce that $nodeType[n] \neq SYNCHRONIZER$, and then $nodeType[n] = MERGE$.

Case 2. $m \in chosenNodes[C_i]$. By Definition 4(2), we know that $\exists n_1 \in outNodes[C_i]$ where $m \notin forwardNodes[n_1]$, therefore there is no path from $C_i$ to $m$ containing $n_1$ in the workflow graph.

By Definition 4(3), we know that $n \in forwardNodes[n_1]$. Therefore, there is a correct instance subgraph $I_0$, which has a path from $C_i$ to $n$ containing $n_1$. Therefore, every path in $I_0$ containing $C_i$ must also contain $n_1$.

Consequently, there is no path from $C_i$ to $m$ in $I_0$. By Lemma 4, the instance subgraph $I_0$ cannot contain $m$. To
meet Criteria 1, \( \text{nodeType}[n] \neq \text{SYNCHRONIZER} \), therefore \( \text{nodeType}[n] = \text{MERGE} \).

**Lemma 8.** Consider a correct workflow graph \( G \), and let \( C \) is a reducible choice node and \( M_i \in \text{mergedNodes}[C] \), then \( \text{backwardNodes}[M_i] \cap \text{mergedNodes}[C] = \{ M_i \} \), i.e., there is no node preceding \( M_i \) in \( \text{mergedNodes}[C] \).

**Proof.** Suppose that \( \text{backwardNodes}[M_i] \cap \text{mergedNodes}[C] = \{ M_{i1}, M_{i2}, \ldots, M_{in} \} \) where \( n \geq 1 \). Since \( G \) is an acyclic graph, then \( \exists M_k \in \{ M_{j1}, M_{j2}, \ldots, M_{kn} \} \) where \( \text{backwardNodes}[M_k] \cap \{ M_{i1}, M_{i2}, \ldots, M_{in} \} = \{ M_i \} \). It is evident that \( M_i \neq M_k \), and \( \text{backwardNodes}[M_k] \cap \text{mergedNodes}[C] = \{ M_k \} \).

Additionally, by Definition 5, we know that every path from \( C \) to \( M_i \) does not contain any choice nodes besides \( C \). Therefore, by Definition 4(1), we obtain that every instance subgraph, which contains \( C \), must also contain \( M_i \).

By Definition 4(3), we know that \( \exists n_i \in \text{inNodes}[M_i] \) where \( n_i \in \text{chosenNodes}[C] \cup \{ C \} \). Therefore, there is an instance subgraph \( I_0 \), which contains a path from \( C \) to \( M_i \) containing \( n_i \). Since \( I_0 \) also has \( M_i \), by Lemma 4, \( I_0 \) must contain a path \( p_i \) from \( M_i \) to \( M_i \).

By Property 7, we have that \( \text{pathNodes}[p_i] \subseteq \text{confirmedNodes}[C] \), and then \( n_i \notin \text{pathNodes}[p_i] \). Consequently, this instance subgraph \( I_0 \) contains at least two incoming edges of \( M_i \). However, by Lemma 7, it follows that \( \text{nodeType}[M_i] = \text{CHOICE} \). Therefore, \( I_0 \) do not meet Criterion 2.

Therefore, we know that the supposition is not true, and then the proof is completed.

**Lemma 9.** Consider a correct workflow graph \( G \), which cannot be reduced by Rules 1 ~ 4 and 7 any further. Let \( S \) be a removable synchronizer of a choice \( C \). By STEP 1, i.e., iteratively applying rules 1 ~ 6, \( S \) will be removed out of \( \text{forwardNodes}[C] \) or completely be removed from the graph.

![Figure 10: Structures of Lemma 9](image)

**Proof.** Since Rules 1 and 2 cannot reduce the graph \( G \) any further, we can obtain that there is no sequence node in \( G \). Let \( N_o = \text{chosenNodes}[C] \cap \text{backwardNodes}[S] \). By Definition 7, we deduce that if \( n \in N_o \) and \( n \neq S \), then \( \text{nodeType}[n] \in \{ \text{MERGE}, \text{FORK} \} \).

Let \( N_i = \text{outNodes}[C] \cap \text{backwardNodes}[S] \). It is evident that \( N_i \neq \text{NULL} \), and \( N_i \subseteq N_o \).

We shall prove that if \( n \in N_i \), then there exists a synchronizer-convergence structure between \( n \) and \( S \). By Lemma 3, \( \text{nodeType}[n] = \text{SYNCHRONIZER} \), and then \( n \neq S \).

Suppose that \( \text{nodeType}[n] = \text{MERGE} \), and let \( n \in N_o \) where \( n \notin \text{outNodes}[n] \), as shown in Figure 10(a). Since Rules 3 and 7 cannot reduce the graph \( G \) any further, we deduce that \( \text{nodeType}[n] \notin \{ \text{MERGE}, \text{FORK} \} \).

However, from Figure 10(a), by Lemma 6, we deduce that \( \text{nodeType}[n] \neq \text{SYNCHRONIZER} \). Since \( n \notin N_o \), the supposition is impossible. Therefore, we know that \( \text{nodeType}[n] = \text{FORK} \).

It follows that the number of the nodes in \( \text{outNodes}[n] \) is more than one. Let \( n \notin \text{outNodes}[n] \cap \text{backwardNodes}[S] \) where \( n \neq S \), as shown in Figure 10(b). By Property 8, we deduce that \( n \notin \text{chosenNodes}[C] \), and then \( n \notin N_o \). Since Rules 3 cannot reduce the graph \( G \) any further, we deduce that \( \text{nodeType}[n] = \text{MERGE} \).

Since there is only one node in \( \text{outNodes}[n] \), by Property 7, we deduce that \( \text{outNodes}[n] \subseteq N_o \). Since Rules 3 and 7 cannot reduce the graph \( G \) any further, the node in \( \text{outNodes}[n] \) must be a synchronizer node. By Definition 7, we deduce that \( \text{outNodes}[n] = \{ S \} \).

From Figure 10(b), by Lemma 1, we know that there is no edge from \( n \) to \( S \). By Lemma 6, we know that \( \text{inNodes}[S] \subseteq \text{forwardNodes}[n] \). Therefore, the structure existing between \( n \) and \( S \) should be a synchronizer-convergence structure, as shown in Figure 10(c).

By applying Rule 6, \( S \) can be removed out of \( \text{forwardNodes}[n] \), as shown in Figure 10(d). Thus, by iteratively applying Rules 1 ~ 6, \( S \) can be removed out of \( \text{forwardNodes}[C] \) or completely be removed from the graph.

**Lemma 10.** Consider a correct workflow graph \( G \), which cannot be reduced by Rules 1 ~ 4 and 7 any further. If \( C \) is a reducible choice with \( \text{Exponent}[C] = 0 \), then the structure shown in Figure 11(c) exists, and \( C \) will be reduced by STEP 1, i.e., iteratively applying rules 1 ~ 6, as shown in Figure 11(d).

![Figure 11: Structures of Lemma 10](image)

**Proof.** Since Rules 1 and 2 cannot reduce the graph \( G \) any further, we can obtain that there is no sequence node in \( G \). By Definition 5 and 4(5), we obtain that if \( n \in \...
chosenNodes[C1], then nodeType[n] ∈ {MERGE, FORK}.

Since Rules 1 ~ 4 cannot reduce the graph G any further, it is clear that chosenNodes[C1] ≠ NULL. Firstly, we shall prove that if n1 ∈ chosenNodes[C1], then nodeType[n] = FORK.

Suppose that ∃ n1 ∈ chosenNodes[C1], where nodeType[n1] = MERGE, and let n2 ∈ outNodes[n1], as shown in Figure 11(a). Since Rules 3 and 7 cannot reduce the graph G any further, we can obtain that nodeType[n2] ≠ {MERGE, FORKR}. It follows that n2 ∉ chosenNodes[C1]. However, by Lemma 7, we can deduce that n2 ∉ mergedNodes[C1]. Therefore, by Property 10, we know that the supposition is not true.

Let Fj ∈ chosenNodes[C1] (i = 1, 2, ..., n) where nodeType[Fj] = FORK. Let n2 ∈ outNodes[Fj] and m ∈ intNodes[Fj], as shown in Figure 11(b).

Since Rules 3 cannot reduce the graph G any further, we deduce that nodeType[n2] ≠ FORK. It follows that m, n2 ∉ chosenNodes[C1]. By Properties 8 and 10, we obtain that m = C1 and n2 ∈ mergedNodes[C1], respectively. By Lemma 7, we can obtain that nodeType[n2] = MERGE.

Let Mj ∈ mergedNodes[C1] (j = 1, 2, ..., k). By Lemma 8 and Definition 4(3), there exists an edge from Fj to Mj, as shown in Figure 11(b).

Consequently, the structure shown in Figure 11(c) exists, and iteratively applying Rules 1 ~ 5 will remove C1 from the graph, as shown in Figure 11(d).

Now let us describe the final theorems.

**Theorem 1.** Let G(k) be a correct workflow graph, in which there is no choice node or no synchronizer node, i.e., n_c(k) = 0 or n_s(k) = 0. Then, by Rules 1 ~ 4, the graph G(k) can be reduced to an empty graph.

**Proof.** Case 1: n_c(k) = 0, and then Eigen[G(k)] = 0. Since G(k) is correct, there is no merge node in G(k). Then, Rules 1 ~ 4 can reduce G(k) to an empty graph.

Case 2: n_s(k) = 0, and then Eigen[G(k)] = n_c(k). Since G(k) is correct, there is no fork node in G(k). Then, Rules 1 ~ 4 can reduce G(k) to an empty graph.

**Theorem 2.** Let G(k) be a correct workflow graph, which is not empty and cannot be reduced by Rules 1 ~ 4 and 7 any further such as an outcome of STEP 3. Then, by STEP 1, the eigen of the graph will decrease, i.e., Eigen[G(k+1)] < Eigen[G(k)].

**Proof.** By Theorem 1, it can be derived that n_c(k) ≠ 0, n_s(k) ≠ 0. For convenience, we shall divide the problem into two cases:

Case 1: there is, at least, a reducible choice node C_j with zero exponent in the graph G(k).

By Lemma 10, we know that STEP 1 will remove C_j from the graph. Since Rules 1 ~ 6 do not increase the number of synchronizer nodes in the graph. We obtain that:

\[ n_c(k) \leq n_c(k-1) \quad \text{and} \quad n_s(k+1) \leq n_s(k). \]

It follows that:

\[ \text{Eigen}[G(k+1)] = (n_c(k+1) + 1)(n_s(k+1) + 1) \leq (n_c(k) - 1)(n_s(k) + 1) = \text{Eigen}[G(k)]. \]

Case 2: there is no reducible choice node with zero exponent in the graph G(k).

Let C_j be a bottom choice node. By definition, C_j is also a reducible choice node and there is at least a removable synchronizer node S_i ∈ chosenNodes[C_j]. By Lemma 9, we know that S_i will be removed from forwardNodes[C_j] or completely removed from the graph by STEP 1. Since Rules 1 ~ 6 do not increase the number of synchronizer nodes in forwardNodes[C_j], then the exponent and the extended exponent of C_j decrease after STEP 1. Note that the choice node with minimum extended exponent must be a bottom choice node. We obtain that:

\[ n_c(k+1) \leq n_c(k), \quad n_s(k+1) \leq n_s(k), \quad \text{and} \quad E_{min}^{(k+1)} \leq E_{min}^{(k)} - 1. \]

It is clear that: \[ \text{Eigen}[G(k+1)] \leq \text{Eigen}[G(k)] - 1. \]

**Proof of Proposition 4:**

From Theorems 1 and 2, it follows that if the workflow graph is correct, then the eigen of the graph in the reduction process will decrease monotonically to -1, which means the graph is empty.

Consequently, the graph reduction algorithm is correct and complete.

### 4.3. Analysis of algorithm complexity

First of all, we will analyze the nodes added by Rules 5, 6 and 7.

In Figure 12(a), outNodes[F1] = {CI, M2}, nodeType[M2] = CHOICE. Rule 7 removes the merge-fork structure and adds two merge nodes Mv and M2, as shown in Figure 12(b). But Rule 3 removes the node Mv, as shown in Figure 12(c). If F1 or F2 is still in a merge-fork structure, then we can continue to apply Rules 7 and 3 to remove them, but do not add any new merge nodes.

![Figure 12: Merge Nodes Added by Rules 7 and 3](image-url)

From the example above, we can derive that by combining with Rule 3, Rule 7 adds a new merge node.
only when there is an incoming edge from the fork node to a non-merge node. Moreover, by applying Rules 7 and 3, the number of such edges in the graph decreases. Thus, the number of the merge nodes added by Rules 7 and 3 is limited. Similarly, the number of the fork nodes added by Rules 7 and 3 is also limited.

Meantime, the application of Rule 5 and 3 to a choice-convergence structure will add at most one fork node. The application of Rule 6 and 3 to a synchronizer-convergence structure will add at most one merge node.

Therefore, for the original graph $G$, the number of nodes in the reduced graph $G^{(k)}$ is:

$$\text{size}(N^{(k)}) < \text{size}(N) + \text{size}(E) + \text{size}(N) < 2\text{size}(G).$$

The worst case of STEP 1 is for a completely reducible workflow graph and that each iteration of applying Rules 1 ~ 6 is able to reduce at the most one node. The time complexity of STEP 1 is:

$$o(\text{size}^2(N^{(i)}) < o(\text{size}^2(G)).$$

The time complexity of STEP 3 is less than that of STEP 1. The number of cycles of STEP 1 ~ 3 is less than $E_{G}(G)$. Therefore, the time complexity of the algorithm is:

$$o((\text{size}(G))^2) \bullet o((\text{size}(N))^2)$$

5. Conclusion

In this paper we introduce a typical workflow modeling language that makes use of five kinds of modeling nodes – sequence, fork, choice, merge, synchronizer – to build structural specifications and give a review of two correctness criteria to identify two kinds of structural conflicts – deadlock and lack of synchronization – based on instance subgraphs of a workflow in [1]. Furthermore, we show the incompleteness of the algorithm in [1] when those rules are applied to the counterexample above.

A complete set of graph reduction rules is presented to reduce the workflow graph and an algorithm is introduced to implement the identification of structural conflicts in process models. The graph reduction algorithm can remove all nodes from workflow graphs that are definitely correct. However, a workflow graph with structural conflicts is not completely reduced to an empty graph. With the introduction of a variable $Eigen$, the paper presents the strict proof of algorithm correctness and completeness. If a workflow graph is free of structural conflicts, then $Eigen$ will decrease monotonously and finally to $-1$, which means that the final reduced graph is empty. Otherwise, $Eigen$ cannot reach $-1$. Finally, the complexity of the algorithm is analyzed.

References