An integer programming based approach for verification and diagnosis of workflows

Rik Eshuis*, Akhil Kumarb,⁎

* Eindhoven University of Technology, School of Industrial Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
b Department of Supply Chain and Information Systems, Smeal College of Business, Penn State University, University Park, PA 16802, USA

ARTICLE INFO
Article history:
Received 29 July 2009
Received in revised form 7 March 2010
Accepted 10 March 2010
Available online 23 March 2010

Keywords:
Workflow
Business process management
Verification
Integer programming

ABSTRACT
Workflow analysis is indispensable to capture modeling errors in workflow designs. While several workflow analysis approaches have been defined previously, these approaches do not give precise feedback, thus making it hard for a designer to pinpoint the exact cause of modeling errors. In this paper we introduce a novel approach for analyzing and diagnosing workflows based on integer programming (IP). Each workflow model is translated into a set of IP constraints. Faulty control flow connectors can be easily detected using the approach by relaxing the corresponding constraints. We have implemented this diagnosis approach in a tool called DiagFlow which reads and diagnoses XPDL models using an existing open source IP solver as a backend. We show that the diagnosis approach is correct and illustrate it with realistic examples. Moreover, the approach is flexible and can be extended to handle a variety of new constraints, as well as to support new workflow patterns. Results of testing on large process models show that DiagFlow outperforms a state of the art tool like Workan in terms of the solution time.

1. Introduction

As businesses are starting to automate more complex and critical processes of their services and operations with workflow models [3], verification of these models assumes greater importance. Detecting and repairing errors at run time, when the workflow is deployed, is very expensive and time consuming. Therefore, a critical challenge in workflow modeling lies in the design–time verification of workflow models [1,5,6,11,15,19,24,32,33,36].

Workflows are often modeled as graphs, in which some special connector nodes are used to indicate splits and joins. Connector nodes can specify parallel branches (AND split/join) and exclusive branches (XOR split/join). Typically, workflow graphs are unstructured: splits and joins of type AND or XOR are used in arbitrary ways, leading to workflow models with goto like constructs and parallelism. Consequently, it is hard to detect errors in workflow designs by merely inspecting the syntax of the workflow.

Several authors have recognized this problem, and, starting with Sadiq and Orlowska [33], have defined approaches for analyzing workflow models [1,5,6,11,15,19,24,32,33,36]. The approaches typically detect two types of errors [33]. First, if only some branches of an AND join are activated, it leads to a deadlock at the join node. Second, lack of synchronization may arise at an XOR join if multiple branches of the XOR join are activated. This results in multiple instantiations of the join. Other approaches focus on analysis in terms of matched–unmatched pairs and nested–unnested patterns [24], but such analysis presupposes a certain degree of structuredness to be present in workflow models, which is not required by the approach of Sadiq and Orlowska.

Though these approaches help identify the presence of such types of error, the feedback they present in case of a found error is rather minimal. To illustrate this point consider the example in Fig. 1 which contains a flaw. Choosing the path towards A2 at XOR split node C3S will eventually result in deadlock at C2J′, since then edge A31 → C2J′ cannot get activated in the sequel, whereas
edge $C7S \rightarrow C2J'$ does get activated due to AND split $C2S$. Sadiq and Orlowska [33] propose a graph reduction approach to detect such errors. They define a set of rules to reduce a workflow graph; if a single node results after applying the rules repeatedly, the workflow graph is declared correct; otherwise, it is incorrect. Applying that approach to this example results in a workflow graph in which all activities are eliminated, and which cannot be reduced any further (see Fig. 2). Since the graph is irreducible and consists of more than one node, it is declared incorrect. However, then it is still not clear which particular control flow connector is causing the error.

To offer more detailed diagnosis of workflow errors, we present a workflow analysis approach that is based on Integer Programming (IP) [34]. Appendix A gives a short introduction to Integer Programming. Each workflow graph is translated into a set of 0/1 linear constraints to which a solution can be found by an IP solver.

The approach consists of three consecutive phases:

• First, the workflow graph is translated into an IP model in which each join is constrained to behave correctly. The IP model is checked for the existence of at least one solution. A solution corresponds to a correct execution of the workflow graph. In particular, an AND join is in the solution if and only if all its incoming edges are in the solution too, while an XOR join is in the solution if and only if one of its incoming edges is in the solution too. The IP formulation can be used to check relaxed soundness [7], i.e. that each node is covered by a correct execution.

• Second, the IP formulation is relaxed such that AND and XOR joins are allowed to express incorrect behavior, i.e., a solution contains some but not all incoming edges of an AND join. Thus, for an AND join with $e$ incoming edges, the number of active edges in the solution is strictly greater than 0, and less than $e$. Alternatively, an incorrect solution may allow strictly more than one incoming edge to be active at an XOR join. Then the join is not part of the solution. Thus, such a solution corresponds to an incorrect execution of the workflow graph, since the execution gets stuck at the join.

• Third, the relaxed IP formulation is used to diagnose the complete workflow graph by testing each join in turn, and adding a constraint that would force it to express incorrect behavior as just described above. If a feasible solution exists where such behavior is found, it would mean that there is an error in the workflow at this node. An example of “incorrect behavior” at an AND join node is reflected in an execution instance where only one incoming branch is activated. The workflow designer can use

Fig. 1. Example of workflow process from Sadiq and Orlowska [33].
This paper is organized as follows. Section 2 introduces workflow graphs and defines the IP formulation of a workflow graph for checking existence of a correct execution. We also give an algorithm for checking relaxed soundness [7] which uses the IP formulation. Section 3 defines a relaxation of the IP formulation which is used for diagnosis. The relaxed IP formulation can be used to check existence of an incorrect execution. Section 4 presents the third phase of the approach by defining a diagnosis algorithm which uses the relaxed IP formulation to test each join. Section 5 sketches extensions of our approach to advanced workflow patterns [2] such as inclusive-OR, discriminator and M-of-N parallel. Then, Section 6 describes the implementation of our approach in the DiagFlow tool [9] and also reports the performance of our tool for models of varying sizes. We show that for large models DiagFlow outperforms the state of the art version of the DiagFlow tool[9] and also reports the performance of our tool for models of varying sizes. We show that for large models.

Fig. 2. Diagnosis result of approach of [33] for Fig. 1.

the information about this node to repair the workflow graph. On the other hand, if no solution is found where any join behaves incorrectly, then each execution of the workflow graph is correct, and therefore the entire workflow graph is correct.

By applying this approach to the example in Fig. 1, we get the following outcomes for each phase:

• the workflow graph is not relaxed sound since A2 and A3 are not covered by any correct execution of the workflow graph;
• the workflow graph has an incorrect execution, for example one in which A2 and A3 are done and which gets stuck at AND join C2J';
• the workflow graph is not correct, since AND join C2J' and all subsequent AND joins on the path from C2J' to the end node are covered by incorrect executions. Section 4 explains the diagnosis for this particular example in more detail. Note that C2J' was eliminated in the approach of Sadiq and Orlowska [33] and is not part of the irreducible graph that their approach yields (cf. Fig. 2).

The IP approach has been implemented in a tool called DiagFlow [9], which reads XPDL [41] models and translates them into IP models using the formulations presented in this paper. To solve the IP models, the open source package LPsolve [25] is used. Feedback from the tool is presented graphically to the workflow designer by highlighting the correct or incorrect execution in the workflow graph itself. More details on the tool can be found in Section 6.

The IP approach to workflow verification makes several contributions compared to existing workflow graph verification approaches like [1,33]. First, it offers precise diagnostics for found errors. Other approaches offer either imprecise diagnostics or various conflicting diagnostics, as we show in Section 7. Second, the approach is flexible. In addition to checking for strict/weak correctness and producing detailed diagnosis, it can perform additional user-defined semantic analysis on the workflow as well as deal with workflows described with new routing constructs, beyond AND/XOR nodes (see Section 5). Other workflow verification approaches do not have such flexibility. Third, our approach is complete, in the sense that if a graph is incorrect, it will always detect the error and it will not report errors for a correct graph. In contrast, the original approach of Sadiq and Orlowska is not complete and reports errors for some correct workflow graphs [1,23]. Fourth, the approach can be combined with heuristic-based approaches, which use heuristics to either reduce a workflow graph [33] or to analyze decomposed regions of a workflow graph [38]. In a combined approach, first heuristics can be used to obtain a reduced workflow graph or smaller workflow subgraphs. Next, the IP approach can be used to efficiently analyze those remaining reduced graph or subgraphs. A detailed comparison with related work can be found in Section 7.

This paper is organized as follows. Section 2 introduces workflow graphs and defines the IP formulation of a workflow graph for checking existence of a correct execution. We also give an algorithm for checking relaxed soundness [7] which uses the IP formulation. Section 3 defines a relaxation of the IP formulation which is used for diagnosis. The relaxed IP formulation can be used to check existence of an incorrect execution. Section 4 presents the third phase of the approach by defining a diagnosis algorithm which uses the relaxed IP formulation to test each join. Section 5 sketches extensions of our approach to advanced workflow patterns [2] such as inclusive-OR, discriminator and M-of-N parallel. Then, Section 6 describes the implementation of our approach in the DiagFlow tool [9] and also reports the performance of our tool for models of varying sizes. We show that for large models DiagFlow outperforms the state of the art verification Woflan tool [40]. Finally, Section 7 presents related work and Section 8 ends with conclusions.

2. Basic IP model

In this section we develop the first phase of our formal approach for verifying workflow graphs using integer programming. First we introduce and define a syntax for workflow graphs and introduce the notions of instance subgraph and correctness for workflow graphs from Sadiq and Orlowska [33]. Based on the syntax, we define an IP formulation for workflow graphs. In the
formulation, each workflow graph is translated into a set of IP constraints. A solution to this formulation is a correct instance subgraph that represents a correct execution of the workflow graph. Next, we apply the IP formulation to two non-trivial examples from the literature. Finally, we present an algorithm for checking relaxed soundness [7] of a workflow graph using the IP formulation.

A short introduction to integer programming is presented in Appendix A.

### 2.1. Workflow graphs

A workflow graph like shown in Fig. 1 consists of nodes and edges. An edge from node A to B specifies that A is done before B. There are different types of nodes which convey different semantic meaning. An activity node models an atomic unit of work. An XOR split node models a choice, i.e. if the XOR split is entered at run time, then exactly one of the outgoing edges is chosen, so the different execution paths starting at the split are mutually exclusive. Correspondingly, an XOR join node merges mutually exclusive execution paths into one path. An AND split node models the concurrent, i.e. independent, execution of its outgoing paths. An AND join node models the synchronization of its incoming execution paths, which are independent. Finally, there is a unique start and a unique end node. This description constitutes a semantic basis for a workflow model.

We define a syntax for workflow graphs which can express models such as Fig. 1. The formalization is adapted from related work on workflow verification [21,33].

**Definition 1.** A workflow graph or workflow schema is a tuple \( P = (N, E) \) where:

- \( N \) is a set of nodes, partitioned into disjoints sets of XOR splits \( S_x \), AND splits \( S_a \), XOR joins \( J_x \), AND joins \( J_a \), activities (tasks) \( Act \), and (start, end) where start is the unique begin node and end the unique final node;
- \( E \subseteq N \times N \) is a precedence relation.

In Fig. 1, all nodes are part of \( N \) and all edges are in \( E \). Start, end, and activity nodes are depicted by a box, while circles are used to indicate control flow connectors, i.e., splits and joins. The type (AND/XOR) is written inside the connector.

Next, each workflow graph should satisfy the following structural constraints:

1. the start node has no incoming edge and one outgoing edge;
2. the end node has one incoming edge and no outgoing edge;
3. each activity node has one incoming edge and one outgoing edge;
4. each split node has one incoming and two outgoing edges;
5. each join node has two incoming edges and one outgoing edge;
6. each node is on a path from the start to the end node (connectedness);
7. the precedence relation is acyclic.

Formalizations of these constraints are presented in an accompanying technical report [10]. The first five constraints are self-explanatory. Constraints 1 and 2 are also put by Sadiq and Orlowska [33]. A workflow having more than one start point can be modeled by using immediately after the start node a split node that connects to the different start points. Similarly, a workflow with more than one end point can be modeled by using a join before the end node. Constraints 3–5 can be relaxed without problem; they are merely used to simplify the IP formulations in the sequel. The sixth constraint rules out unconnected workflows because such workflow graphs contain unreachable parts and therefore are flagged by default.

The last constraint is also placed by other works on workflow verification [1,33,36]. However, workflow graphs are still sufficiently expressive to model loops that involve blocked iteration [33]. Basically, any block in a correct workflow graph can be repeated multiple times without affecting correctness, since a block has a single point of entry and a single point of exit.

In the sequel, we also use auxiliary functions inedge, outedge: \( N \rightarrow P(E) \), which both map each node to a set of edges. For a node \( n \), inedge\( (n) \) is the set of edges entering \( n \), while outedge\( (n) \) is the set of edges leaving \( n \). Formally, inedge\( (n) = \{(x, y) \in E | y = n \} \) and outedge\( (n) = \{(x, y) \in E | x = n \} \). We use subscripts to identify the different elements of inedge\( (n) \) and outedge\( (n) \). For example, if inedge\( (n) = \{e_1, e_2\} \), then inedge\( _1(n) = e_1 \) and inedge\( _2(n) = e_2 \).

Sadiq and Orlowska [33] introduce the notion of an instance subgraph, which corresponds to a particular execution instance of a workflow graph. An instance subgraph represents a subset of workflow activities that may be executed for a particular instance of a workflow. An instance subgraph of a workflow graph can be generated in a naive way by traversing a workflow graph from the start node, using the following rules:

- if an XOR split node is visited, one of its outgoing edge is visited at random;
- if an AND split node is visited, then all outgoing edges are visited;
- if an XOR join node is visited, then its outgoing edge is visited only if one of its incoming edges has been visited too and all other incoming edges have not been visited;
- if an AND join node is visited, then its outgoing edge is visited only if all its incoming edges have been visited too.

The part of the subgraph that covers the visited nodes is an instance subgraph because it represents a specific execution instance based on the workflow graph. A formal definition of instance subgraphs is presented in an accompanying technical report [10].

---

1. An instance subgraph should not be confused with a workflow instance, which is a run-time instantiation of a workflow model.
An instance subgraph can get stuck at a join. In that case, the instance subgraph is not correct. By generating all instance subgraphs, the correctness of a workflow graph can be checked.

**Definition 2.** An instance subgraph is correct if and only if for each node $n$ in the subgraph:

- if $n$ is an AND join, then the subgraph contains all incoming nodes of $n$ (no deadlock)
- if $n$ is an XOR join, then the subgraph contains one incoming node of $n$ (no lack of synchronization)

A workflow graph is correct if and only if each generated instance subgraph is correct.

To illustrate this definition, Fig. 3 shows a workflow graph that has an incorrect instance subgraph. The AND join is part of the instance subgraph, but the incoming node A1 is not. So the instance subgraph deadlocks at the AND join. Since there is an incorrect instance subgraph, the workflow graph is not correct.

In the next subsection, we will formalize correct instance subgraphs as Integer Programming problems.

### 2.2. Basic integer programming (IP) formulation

The IP formulation consists of a set of constraints for a given workflow graph $(N, E)$. The constraints encode a correct execution of the workflow graph so that an instance subgraph has no deadlock and no lack of synchronization. For each node and edge of the workflow graph, an IP variable is created. A solution to the IP formulation assigns a 0–1 integer value to each node and edge variable in the graph. The nodes and edges that are assigned a 1 value are present in the instance subgraph, and the others are not.

In the IP formulation below, we use the definition of inedge and outedge given in Section 2.1. To avoid confusion between an IP variable and the corresponding node or edge, each IP variable is listed in teletype font. Each edge and node is given a variable name in the IP formulation. For a node $n$, inedge$_1(n)$ and inedge$_2(n)$ represent the names of IP variables corresponding to the two incoming edges of node $n$. Similarly, outedge$_1(n)$ and outedge$_2(n)$ represent the names of IP variables corresponding to the two outgoing edges of node $n$. The formulation is given first and a description follows. A generalization of the definition for splits (joins) with more than two outgoing (incoming) edges is presented in Appendix B.

**Definition 3.** For a workflow graph $(N, E)$, the Basic IP formulation maximizes the value at the end node subject to the following constraints at each node in the workflow graph. For each node and edge $x$ of $(N, E)$, so $x \in N \cup E$, an IP variable $x$ is created. The constraints are:

\[
\begin{align*}
\text{IP0} & \quad \text{start} = 1 \\
\text{IP1For } n \in \text{Act} \cup \text{S}_{A} \cup \text{S}_{X} \cup \{\text{end}\} : & \quad \text{inedge}_1(n) - n = 0 \\
\text{IP2For } n \in \text{Act} \cup \text{J}_{A} \cup \text{J}_{X} \cup \{\text{start}\} : & \quad \text{outedge}_1(n) - n = 0 \\
\text{IP3For } n \in \text{S}_{A} : & \quad \text{outedge}_1(n) + \text{outedge}_2(n) - 2n = 0 \\
\text{IP4For } n \in \text{J}_{A} : & \quad \text{inedge}_1(n) + \text{inedge}_2(n) - 2n = 0 \\
\text{IP5For } n \in \text{S}_{X} : & \quad \text{outedge}_1(n) + \text{outedge}_2(n) - n = 0 \\
\text{IP6For } n \in \text{J}_{X} : & \quad \text{inedge}_1(n) + \text{inedge}_2(n) - n = 0
\end{align*}
\]

The instance subgraph is a solution to the IP formulation. It is built inductively, starting from the start node (IP0). The remaining constraints are symmetrical for in/out edges and split/join nodes. Constraints IP1 and IP2 state that a node with a single incoming (IP1) or a single outgoing edge (IP2) is part of the instance subgraph if and only if the edge is part too.
Constraint IP3 (IP4) states that an AND split (join) is part of the instance subgraph if and only if all its outgoing (incoming) edges are part too. Constraint IP5 (IP6) states that an XOR split (join) is part of the instance subgraph if and only if one of its outgoing (incoming) edges is part too. Together, constraints IP4 and IP6 ensure that the instance subgraph is deadlock free (IP4) and has no lack of synchronization (IP6). Thus, a solution to the Basic IP formulation specifies a correct instance subgraph.

As stated in the previous subsection, to simplify the presentation we assume that each split has two outgoing edges, and each join has two incoming edges. However, this can be easily generalized to more incoming and outgoing edges; see Appendix B. This generalization was actually realized in the verification tool we developed (see Section 6).

2.3. Examples

We tested our approach on two non-trivial examples from the literature. First, consider the example shown in Fig. 1, taken from [33]. We translated the workflow into an IP and solved the IP using the DiagFlow tool (discussed in Section 6) that incorporates the IP solver LPsolve [25]. The solution produced by the tool corresponds to an instance subgraph, and it is shown in Fig. 4 by solid lines. The dashed lines belong to the workflow graph but not to the subgraph.

A second example we ran was taken from [23]. This process is reproduced in Fig. 5 and it was chosen because according to Lin et al. [23], this correct workflow graph cannot be verified by the approach of Sadiq and Orlowska [33]. In this figure the activity nodes have been omitted. This workflow graph was also converted into an IP formulation as above. The solution of the IP formulation, i.e., a correct instance subgraph, is shown in the figure by the solid lines.

2.4. Relaxed soundness

Above we showed how to check for the existence of one correct instance subgraph in the workflow graph. Now, we show how it is also possible to check for a stronger notion of correctness called relaxed soundness. Our definition for relaxed soundness is derived from [7], and is stated as follows.
Definition 4. A workflow graph is relaxed sound \((N, E)\) if every node \(n \in N\) appears in at least one correct instance subgraph of \((N, E)\).

To check a workflow graph for relaxed soundness, we can run the algorithm in Fig. 6. The precondition for the algorithm is that a solution exists for the Basic IP formulation. Initially, all nodes of the workflow graph are unmarked (l. 3). In the while loop, one of the unmarked nodes is picked and put in variable current (l. 6) and the corresponding IP variable is forced to be 1 by adding that constraint to the Basic IP formulation (l. 7). If a solution does not exist, then the workflow graph is not relaxed sound due to node current (l. 10). Otherwise, all nodes that part of the solution, so the corresponding IP variables have value 1, are removed from unmarked (l. 13), since the instance subgraph corresponding to sol contains these nodes. Since the IP variable for current is forced to be 1, node current will also be removed in line 13. If each node has been processed and no error has been found, the workflow graph is relaxed sound (l. 17). Otherwise, it is not relaxed sound and a message is printed along with the name of the node that causes failure of the test.

The next theorem asserts the correctness of the algorithm.

Theorem 1. Let \((N, E)\) be a workflow graph.
Algorithm Relaxed-Soundness-Check finds no error if and only if \((N, E)\) is relaxed sound.

Proof. \(\Rightarrow\) Since algorithm Relaxed-Soundness-Check finds no error, for each node there is a correct instance subgraph which contains the node. Thus, \((N, E)\) is relaxed sound by Definition 4.

\(\Leftarrow\) Suppose the algorithm finds a node \(n\) such that IP\(_1\) does not have a feasible solution. Then there is no correct instance subgraph which contains \(n\), so \((N, E)\) is not relaxed sound.
Above, we have shown how a workflow graph can be modeled and solved formally using an IP approach. While the above formulation is elegant, it can only tell us if a correct instance subgraph exists for the workflow graph, but not if all instance subgraphs are correct. Moreover, the Basic IP formulation only allows correct instance subgraphs. If an XOR join or AND join belongs to an incorrect instance subgraph, then the corresponding Basic IP model has no solution (in terms of linear programming, the solution is unfeasible). Since no feasible solution is found, there is also no feedback provided in the form of an execution path that leads to the error. This makes it difficult for a workflow designer to diagnose and correct the design.

Here, we relax (or unconstrain) the IP formulation slightly to a new formulation ($IP_{Relax}$) in order to test each join for errors. Each solution to a relaxed IP model corresponds to an instance subgraph, which can be either correct or incorrect. If the instance subgraph is incorrect, it contains a join whose outgoing edge is not in the instance subgraph. Note that this was not allowed in the Basic IP formulation due to the strict equality constraints $IP4$ and $IP6$; therefore, these are the constraints that will be relaxed in this section. In the presentation, we treat the cases for AND joins and XOR joins separately. In the next section, $IP_{Relax}$ is used to diagnose workflow graphs (by testing each join for errors) and to do correctness checking.

A generalization of the constraints for joins with more than two incoming edges is presented in Appendix B.

3.1. AND joins

At every AND join $n \in J_A$, we relax constraint $IP4$ to:

$$in_{edge}_1(n) + in_{edge}_2(n) - n \leq 1$$

This constraint forces the value of $n$ to be 1 when both its incoming edges are 1. However, it does allow $n$ to be 1 if one of its incoming edges is 1, which would mean the join is activated if only one incoming edge is active. To avoid this undesired behavior, we need to add two more constraints for the AND join:

$$n \leq in_{edge}_1(n)$$
$$n \leq in_{edge}_2(n)$$

With these constraints, if both incoming edges are 0, or only one incoming edge is 1, then $n$ is forced to be 0. Therefore, for all combinations of values for incoming edges, a correct value for $n$ is determined. It is straightforward to extend this to the case of more than two incoming edges; see Appendix B.

3.2. XOR joins

In a similar way, we also relax the strict equality constraint at the XOR join nodes ($IP6$) to allow for multiple incoming edges to be active. For each XOR join $n \in J_X$, we need to introduce an auxiliary variable $a$ that becomes 1 if and only if some incoming edge of
n is active. Using a, node n can be forced to become 1 if and only if one incoming edge is active, so having value 1. Without using an auxiliary variable, this latter constraint cannot be enforced.

\[
\begin{align*}
\text{(XOR1)} & \quad \text{inedge}_1(n) + \text{inedge}_2(n) \geq a \\
\text{(XOR2)} & \quad \text{inedge}_1(n) + \text{inedge}_2(n) \leq 2a \\
\text{(XOR3)} & \quad \text{inedge}_1(n) + \text{inedge}_2(n) \geq n \\
\text{(XOR4)} & \quad \text{inedge}_1(n) + \text{inedge}_2(n) + n \geq 2a \\
\text{(XOR5)} & \quad \text{inedge}_1(n) + \text{inedge}_2(n) + n \leq 2
\end{align*}
\]

Constraint XOR1 and XOR3 force a and n to be 0 if both incoming edges are 0. Constraint XOR2 forces a to be 1 if one or both of the incoming edges is 1. Combining XOR1 and XOR2, a is 1 if and only if one or more of the incoming edges of n are 1, so they are active.

Next, constraint XOR4 forces n to be 1 if a is 1 and one of the incoming edges is 1. If both incoming edges are 1, XOR4 allows n to be either 0 or 1. Note that XOR4 also allows that both a and n are 0. XOR4 cannot be expressed without using an auxiliary variable like a. The next constraint, XOR5, forces n to be 0 if both incoming edges are 0 to reflect that there is a problem at the XOR join node.

3.3. Example

To illustrate the IPRelax formulation in this section, we give an example of an incorrect workflow graph, i.e. a workflow graph containing an incorrect instance subgraph (see Fig. 7). The IPRelax formulation for this example workflow graph has a solution. The corresponding instance subgraph is shown with solid lines in Fig. 7. In the solution, the XOR node x that succeeds A3 is 1, since only one of its incoming edges is 1. The auxiliary variable a corresponding to x is also 1. It can be checked that indeed XOR1–XOR5 are satisfied by the solution that corresponds to Fig. 7. Next we will show how such erroneous workflows can be diagnosed.

4. Diagnosis algorithm and results

In this section, we discuss the third phase of our approach in which we do detailed diagnosis of workflow graphs as well as checking for correctness. Section 4.1 describes our diagnosis algorithm, Section 4.2 gives the results from running the algorithm, and Section 4.3 provides a discussion to highlight a few specific features of the IP approach.
4.1. Algorithm description

In Section 3, we showed how one can check if a workflow graph is correct, i.e. there is no instance subgraph that corresponds to an erroneous execution instance of the process. If there is even one such execution instance then it is necessary to identify it and notify the user. The algorithm in Fig. 8 performs this procedure.

This algorithm first creates an IPRelax formulation for the workflow graph (l. 2) and the error flag is initialized to false (l. 3). Next, the algorithm checks each join node in turn (l. 6 and l. 14). For each join, it adds a constraint to the IPRelax formulation to force an execution instance that would generate an error at this join (l. 4 and l. 13). For an AND join, an erroneous situation corresponds to only one incoming edge being activated, resulting in deadlock. The constraint $\text{inedge}_1(n) + \text{inedge}_2(n) = 1$ expresses this (l. 5). For an XOR join, an error corresponds to multiple incoming edges being activated, resulting in multiple instances. Constraint $\text{inedge}_1(n) + \text{inedge}_2(n) \geq 2$ at line 14 reflects this. After adding the appropriate constraint, we solve the extended IPRelax formulation (l. 6 and l. 15). If a feasible solution is not found, it means that it is not possible to create an execution instance in which an error will occur at this node. On the other hand, if a solution is found, it is reported as part of the diagnosis, and the error flag is raised (l. 8–10 and l. 17–19). The solution found represents an actual instance subgraph that contains an error arising at this specific node. This information tells a user exactly where an error can occur and the exact execution path that leads to the error. Finally, if the error flag has not been raised (l. 22), it means that no error at any join node was found, and that the workflow graph is correct.

The theorem below asserts the correctness of the algorithm for the correctness check.

**Theorem 2.** Let $(N, E)$ be a workflow graph. Algorithm Diagnosis finds no error if and only if $(N, E)$ is correct.

**Proof.** $\Rightarrow$: Since algorithm Diagnosis finds no error, for each join there is no instance subgraph in which the join behaves incorrectly, so the conditions in l. 7 and l. 16 never become true. Thus, $(N, E)$ is correct by Definition 2.

$\Leftarrow$: Suppose the algorithm finds a join $j$ such that $\text{IP}_j$ produces a feasible solution (l. 7 and l. 16). We only consider the case that $j$ is an AND join, the case where $j$ is an XOR join is treated by similar reasoning. The solution found satisfies $\text{inedge}_1(j) + \text{inedge}_2(j) = 1$ (l. 5), so not all incoming branches of $j$ are activated. But then the corresponding instance subgraph is not correct, so then $(N, E)$ is not correct by Definition 2.

Thus, algorithm Diagnosis implements both a detailed diagnosis procedure as well as a correctness check. Next, we discuss the results of applying this algorithm.

4.2. Results

We ran algorithm Diagnosis for the two examples discussed above in Section 2.3. For the first example (Fig. 4), the results are shown in Table 1. These results show that there are five AND joins (C2J’, C2J, C6J, C7J, C1J) where a deadlock can occur because only one incoming edge is activated. All these deadlocks however occur because of the same problem which is the AND join node C2J’.
The subsequent join nodes lie on a path from C2J' to end. By inspecting the solution found for C2J', we can trace this deadlock back to node C2S', which is not activated if the left branch is taken at XOR split node C3S. Hence, the problem can be isolated to the (right) branch from C3S to C2S'. At the XOR joins, no problems were found.

We did similar testing on the second example (Fig. 5). Here the diagnosis results showed that every join node was correct. Hence, the entire process was correct. Returning to the example of Fig. 3, the diagnosis algorithm will show that the problem lies at the AND join node that synchronizes A1 and A2 in this process. This is because only one inedge of this AND join is activated and the other edge is dead.

4.3. Discussion

Our approach has two distinguishing features. First, it allows a precise feedback of modeling errors. We illustrate this by revisiting the diagnosis of the example of Section 1, as listed in Table 1. AND join C2J' is diagnosed as being flawed: an instance subgraph exists in which the AND join C2J' deadlocks since the left path from C3S is taken. In addition, there are four other AND joins C2J, C6J, C7J and C1J, which are deadlocked as a result, because the nodes and edges on the path from C2J' to end all have values of 0 in our solution and fail to get activated. This clearly suggests that the main problem lies at node textsfC2J' and creates a cascading effect resulting in the subsequent problems. The workflow designer must fix this problem first and then check the workflow model again.

This additional diagnosis information is more useful to a user in resolving the problem in the process than the feedback produced by the approach of Sadiq and Orlowska [33], which is the reduced process in Fig. 2. This process is the result from repeatedly applying a set of reduction rules to the example. The reduction rules can be combined with our approach. First the reduction rules can be applied to the workflow graph, and next the reduced workflow graph can be converted into an IP model. This way, a reduced IP model is obtained which can be verified more efficiently than the original IP model. In that sense, our approach complements the approach of Sadiq and Orlowska.

Another feature of our approach is that the complexity only grows if the number of join nodes increases. Basically the diagnosis procedure requires that we solve the IP formulation separately for each join node. Thus, for the example of [33] there are 13 join nodes in Fig. 1, and as shown in Table 1, we solved the IP formulation 13 times and reported the results in order to produce a complete diagnosis. Although, theoretically, solving an IP has exponential complexity, in practice the algorithm runs very fast for two reasons. First, most IP software algorithms first solve the relaxed linear programming version of the problem. If it is infeasible, so no solution exists, then there is no need to solve the IP formulation. Secondly, if the relaxed formulation gives an integer solution then it is not necessary to run a more expensive algorithm like branch and bound [34] in order to get the IP solution.

5. Extensions

In this section we discuss two key extensions of our approach. Both serve to highlight how this approach provides more flexibility than the other ones such as the one discussed in [33]. The first extension shows how it is possible to perform semantic checking of the workflow graph. The second extension shows how to apply our approach to workflow graphs that contain additional control patterns such as Inclusive-OR, Discriminator and M-of-N AND patterns. Other approaches (notably of [1,33,40]) do not lend themselves so easily to such extensions.

5.1. Semantic checking and analysis

A workflow graph may be structurally correct, but it may violate certain basic (business) rules regarding relationships between activities. This means it is semantically incorrect. For example, in an insurance claim process, an application must be received before the claim can be reviewed. Therefore, the activity “application received” must always occur in every instance. Another

<table>
<thead>
<tr>
<th>Node</th>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3J</td>
<td>XOR join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C2J'</td>
<td>AND join</td>
<td>Solution found, A31 = 0, C7S = 1</td>
</tr>
<tr>
<td>C6J</td>
<td>AND join</td>
<td>Solution found, A12 = 1, A17 = 0</td>
</tr>
<tr>
<td>C4J</td>
<td>AND join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C5J</td>
<td>AND join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C2J</td>
<td>AND join</td>
<td>Solution found, A13 = 1, A16 = 0</td>
</tr>
<tr>
<td>C7J</td>
<td>AND join</td>
<td>Solution found, A21 = 0, C8J = 1</td>
</tr>
<tr>
<td>C1J</td>
<td>AND join</td>
<td>Solution found, C7J = 0, C11J = 1</td>
</tr>
<tr>
<td>C8J</td>
<td>XOR join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C9J</td>
<td>XOR join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C9J'</td>
<td>XOR join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C10J</td>
<td>AND join</td>
<td>Infeasible</td>
</tr>
<tr>
<td>C11J</td>
<td>XOR join</td>
<td>Infeasible</td>
</tr>
</tbody>
</table>

The subsequent join nodes lie on a path from C2J' to end. By inspecting the solution found for C2J', we can trace this deadlock back to node C2S', which is not activated if the left branch is taken at XOR split node C3S. Hence, the problem can be isolated to the (right) branch from C3S to C2S'. At the XOR joins, no problems were found.
simple rule is that a client’s application cannot both be accepted and denied for the same process instance. Thus, the two activities “application accepted” and “application rejected” are exclusive. Similarly, another possible rule is that if an application is received, then one of two activities “application accepted” or “application rejected” must occur. We show here how a variety of such rules can be checked easily using our approach. Such constraints fall into generic categories [17]:

- (C1) Occurrence/Non-occurrence of an activity: does an activity always (never) appear in at least one execution instance?
- (C2) Co-occurrence of activities in an activity group: do two activities always (never) appear together or not at all in all execution instances?
- (C3) Dependency relationships between activities: does one task always depend upon another?

The general idea is that to check for each rule or constraint we can first add additional constraints to our IP formulation, and then solve it. Thus, a C1 type constraint simply checks if an activity a (never) appears in an execution instance of a process. To perform this check, we can add a constraint \( a = 1 \) or \( a = 0 \) to the formulation IPRelax and solve it. If a solution is found the answer is true, else it is not. Instead of the activity we can use the incoming or outgoing edge of the activity because their IP variables will have the same value by the IP constraints. A type C2 constraint checks if a group of \( n \) activities \( \{a_1, a_2, .., a_n\} \) always occur together. We can add an IP constraint to the formulation IPRelax as follows:

\[ a_1 + a_2 + ... + a_n = n \]

If no solution is found then the constraint is true; else, it is false. In a similar way it is possible to check for other kinds of relationships by adding simple IP constraints. Thus, one can analyze the workflow process in considerable detail, and get a better understanding of its behavior. Moreover, we can also show counterexamples where the desired conditions are violated, and this can help to make corrections to the workflow process.

5.2. Adding new modeling constructs

The IP formulations in Sections 2.2 and 3 assume that workflows are designed using standard AND and XOR patterns. However, the IP formulations can also be extended for verification of workflows containing advanced control flow patterns [2]. We illustrate our idea with three patterns.

5.2.1. Inclusive-OR pattern

The (inclusive-)OR is a special variant of the OR split pattern (see Fig. 9a). In this pattern, the OR split node is not treated as an exclusive OR split; rather, it is possible to activate one or more outgoing branches at this node. Thus, the semantics of an OR split is that if its incoming edge is activated, then one or more outgoing edges may be activated. Similarly, an OR join (also known as synchronizing merge [2]) is activated if one or more of its incoming edges are activated. Thus, the number of outgoing edges activated by an OR split and the number of incoming edges required to activate an OR join is non-deterministic, but it should be at least one. To model this pattern, we modify the constraints for the two nodes. For the OR split node (n1), the constraint is:

\[
\text{outedge}_1(n1) + \text{outedge}_2(n1) - n1 \geq 0
\]

\[
n1 \geq \text{outedge}_1(n1)
\]

\[
n1 \geq \text{outedge}_2(n1)
\]

Fig. 9. Three new patterns.
Now, it is possible for one or more outgoing branches to be activated, instead of strictly one being activated. Similarly, for the OR join node \( n_2 \), the new constraints are:

\[
\begin{align*}
\text{inedge}_1(n_2) + \text{inedge}_2(n_2) - n_2 & \geq 0 \\
n_2 & \geq \text{inedge}_1(n_2) \\
n_2 & \geq \text{inedge}_2(n_2)
\end{align*}
\]

Here, the OR join at node \( n_2 \) is activated by any one of its incoming edges. All these constraints can be used in combination with both the exact and the relaxed IP formulation, since for OR joins there is no strict distinction between an incorrect and a correct use of an OR join.

5.2.2. Discriminator (DIS) pattern

The discriminator pattern (also, called multiple instances) matches an AND split node with an OR join node (see Fig. 9b). The idea behind this pattern is that it can activate tasks along multiple outgoing paths at the AND split node, and the OR join node is activated by the first path that reaches it. This is also called a multiple instance pattern. From the point of view of diagnosis using the IP approach, the AND split node is treated as discussed in Section 2.2, i.e. all the outgoing branches are activated. The OR join node is treated as explained above. Note that the activation behavior for the OR join as explained above cannot be captured exactly in the IP approach. Whether the first path that reach the OR join activates it or the last path, the corresponding IP solutions are all identical.

5.2.3. M-of-N AND split pattern

Next, consider the \( m \)-of-\( n \) AND split pattern in which a split \( n_1 \) activates only \( m \) out of its \( n \) outgoing branches \( (m \leq n) \) (see Fig. 9c). Similarly, the \( m \)-of-\( n \) AND join pattern specifies that a join \( n_2 \) is activated by exactly \( m \) out of its \( n \) incoming branches. These patterns can be modeled with constraints like:

\[
\begin{align*}
\text{outedge}_1(n_1) + \text{outedge}_2(n_1) + \ldots + \text{outedge}_n(n_1) & = m n_1 \\
\text{inedge}_1(n_2) + \text{inedge}_2(n_2) + \ldots + \text{inedge}_n(n_2) & = m n_2
\end{align*}
\]

Note that these constraints generalize IP3 and IP4, respectively, of Definition 3, and they ensure that the number of outgoing edges at \( n_1 \) is equal to the number of incoming edges at \( n_2 \). For diagnosis purposes, the equality in the above two constraints can be relaxed to “\( \leq \)” and “\( \geq \)” respectively. In this way it is possible to check if a solution exists where the constraint is violated. If no feasible solution is found then it would mean that the constraint is never violated.

5.3. Example: diagnosis of processes containing OR joins

Fig. 10 shows examples of processes containing OR splits and joins on which our diagnosis approach was tried. For the OR split and joins we used the new constraints described in Section 5.2.1 above, and for other control constructs the constraints described previously were used. Table 2 summarizes the results of the experiments.
In addition to illustrating the use of OR splits and joins, these examples also study the effect of changing an OR node to an AND or XOR node on correctness of the process. Table 2 shows that for example 1 one correct solution exists, but it is possible to find a solution in which the AND2 node is activated improperly because only one inedge is active. However, there are no solutions in which the OR joins or splits are activated improperly. Example 2 is a modified version of Example 1, with two OR join nodes replaced by AND joins. Here we found a correct solution but also found that incorrect solutions exist that cause problems at nodes AND2, AND3 and AND4. Finally, example 3 is a further modification of example 2 with the introduction of two XOR join nodes. Here again a correct solution is found, but incorrect solutions are also found at nodes AND2, XOR3 and XOR4.

These experiments show that the OR join operates correctly as expected. Moreover, because of its non-deterministic behavior, an OR join generates more correct execution instances than an AND or XOR join. By replacing the OR join with more restrictive constructs like AND and XOR joins, flexibility in the process flow is reduced and more incorrect execution instances are actually found. Of course, the overarching goal in process design is not flexibility, but to ensure that the process model captures the real world process as accurately as possible. The constructs that are best suited to achieve it should be used.

5.4. Discussion

We showed in this section that it is possible to create IP formulations to check correctness of workflow graphs that use additional patterns. In a similar manner even more control flow patterns can also be added by describing the IP constraints that relate to them. As we illustrated, the verification and diagnosis of process graphs with these additional patterns is similar to the approach in previous sections for graphs without these patterns.

However, we also noticed some limitations of the IP approach. Sometimes different patterns (OR join, discriminator join pattern) are modeled using the same IP formulation, even though their exact execution behavior differs. This is because the IP formulation encodes a complete execution instance and does not allow the expression of behavior in intermediate states.

6. Implementation and performance results

In this section we discuss our prototype tool called DiagFlow and also give results of testing models of various size with this tool.

6.1. XPDL DiagFlow tool

The verification approach has been implemented in DiagFlow [9], a publicly available Java-based tool for verifying the control flow of XPDL (XML Process Definition Language) workflow models. XPDL [41] is a standard language endorsed by the Workflow Management Coalition for exchanging workflow definitions among workflow management systems. In addition to the control flow, XPDL also supports modeling of the data flow.

The DiagFlow tool reads XPDL 1.0/2.0 models and translates them into an internal workflow graph representation which is visualized on the screen. The tool uses several publicly available libraries for this. The JDom library [18] is used for parsing the XML-based XPDL models. Using JDom, the tool translates an XPDL model into an internal graph structure that implements a workflow graph (cf. Definition 1). The workflow graph is visualized on the screen using the Graphviz/dot [12] library.

For solving integer programming problems, the tool uses the open source library LPsolve [25]. Each type of IP formulation (basic or relaxed from Section 2) has its own translation in the tool. Each translation is implemented as a method that converts the internal workflow graph representation into an IP formulation, which is an internal LPsolve data structure that is created by invoking LPsolve routines for defining IP variables and constraints on these IP variables. The translations implemented by the tool can deal with splits with more than two outgoing edges and joins with more than two incoming edges, as defined in Appendix B.

<table>
<thead>
<tr>
<th>Example</th>
<th>Constraint</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>None</td>
<td>and1=and2=or1=or2=or3=or4=1; e1, e2, ... e8=1</td>
</tr>
<tr>
<td>1</td>
<td>and2=0</td>
<td>and1=or1=or2=or3=1; e1, e2, e3, e5, e7=1</td>
</tr>
<tr>
<td>1</td>
<td>or1=0</td>
<td>Infeasible</td>
</tr>
<tr>
<td>1</td>
<td>or2=0</td>
<td>Infeasible</td>
</tr>
<tr>
<td>1</td>
<td>or3=0</td>
<td>Infeasible</td>
</tr>
<tr>
<td>1</td>
<td>or4=0</td>
<td>Infeasible</td>
</tr>
<tr>
<td>1</td>
<td>None</td>
<td>and1=and2=or1=or2=and3=and4=1; e1, e2, e8=1</td>
</tr>
<tr>
<td>2</td>
<td>and2=0</td>
<td>and1=or1=or2=and3=1; e1, e2, e3, e5, e6, e7=1</td>
</tr>
<tr>
<td>2</td>
<td>e3 + e5 = 1</td>
<td>and1=or1=or2=and4=1; e1, e2, e4, e5, e6, e8=1</td>
</tr>
<tr>
<td>2</td>
<td>e4 + e6 = 1</td>
<td>and1=or1=or2=and3=1; e1, e2, e3, e5, e6, e7=1</td>
</tr>
<tr>
<td>3</td>
<td>None</td>
<td>and1=and2=or1=or2=xor3=xor4=1; e1, e2, e4, e5, e7, e8=1</td>
</tr>
<tr>
<td>3</td>
<td>and2=0</td>
<td>and1=or1=or2=xor3=1; e1, e2, e3, e4, e6, e7=1</td>
</tr>
<tr>
<td>3</td>
<td>e3 + e5 = 2</td>
<td>and1=or1=or2=1; e1, e2, e3, e5=1</td>
</tr>
<tr>
<td>3</td>
<td>e4 + e6 = 2</td>
<td>and1=or1=or2=1; e1, e2, e4, e6=1</td>
</tr>
</tbody>
</table>

Table 2
Diagnosis results on examples of Fig. 10.
Having constructed an IP formulation, the tool calls LPsolve routines to solve the formulation. LPsolve returns whether an optimal solution is obtained or whether the model is infeasible. In the first case, the actual solution is obtained by querying the values of the IP variables in the internal LPsolve data structure. This solution corresponds to an instance subgraph (explained in Section 2.2), which is shown to the user by highlighting corresponding parts of the workflow graph on the screen, as explained next. In the second case, there is no solution, so no corresponding instance subgraph needs to be shown to the user.

DiagFlow analyzes XPDL models by providing three options for analysis in a menu. The first option uses the IP formulation from Section 2.2 to find a correct instance subgraph. This subgraph is shown to the user by highlighting the nodes and edges that participate in the subgraph in the figure on the screen. The nodes and edges that are not a part of the instance subgraph are shown by dashed lines. The second analysis option in DiagFlow tests for relaxed soundness. Recall from Section 2.4 that relaxed soundness requires that every node must appear in at least one correct instance subgraph. Nodes that are not contained in a correct instance subgraph are listed in the lower pane just below the figure. Third, DiagFlow tests each join using the relaxed IP formulation from Section 3 and the diagnosis procedure described in Section 4. When this option is selected for analysis, all the incorrect joins are shown in the lower pane. If the user clicks on an incorrect join, an incorrect instance subgraph that contains the join is highlighted. This instance subgraph shows why the join is incorrect and thus serves as a visual proof for the analysis. In this way, designers can easily see what causes the problem at the join and correct the XPDL model.

We have applied the DiagFlow tool to the examples discussed in this paper and to some larger case studies. To illustrate the practical applicability, Fig. 11 shows a screenshot of the diagnosis of an XPDL sample that comes with the JaWE/Together XPDL workflow engine. The XPDL model contains a flawed AND join; an incorrect instance subgraph leading to the flawed join is highlighted in the XPDL model shown in Fig. 11.

6.2. Performance results

Solving IP problems usually takes exponential time in the number of variables in the problem formulation. Solving its relaxation, i.e. solving its LP variant, has an exponential worst-case complexity, but in practice it can be done in polynomial time. Nevertheless, in general, the worst-case complexity for solving an integer programming formulation using the branch-and-bound method is exponential. Our experiments show that this exponential increase in time does also occur with our tool, but as we demonstrate it does not limit the applicability of our approach, since IP solvers run very fast in practice.

Table 3 details the performance results for experiments we conducted with models of various sizes with our tool, which in turn uses the LPsolve [25] software for solving the IP formulation. All the experiments were run on a Windows XP machine, with an Intel Core 2 Duo processor, 2.33 GHz, and 2.0 Gb RAM. In order to generate models of various sizes we combined the base example.

![Fig. 11. Incorrect instance subgraph in XPDL model of JaWE/Together.](image)
from Sadiq and Orlowska [33] shown in Fig. 1 with multiple copies of itself in different configurations. In this way, we were able to generate successively larger models. The models we considered were as follows:

Sadiq1: model of Fig. 1 from Sadiq and Orlowska [33]
Sadiq2: model Sadiq1 in parallel with itself
Sadiq3: models Sadiq2 and Sadiq1 in sequence
Sadiq4: model Sadiq2 in sequence with itself
Sadiq5: model Sadiq4 in sequence with itself

The DiagFlow tool implements the Diagnosis algorithm, and for each join, a relaxed IP model was solved as described above. The results do show that the running time increases exponentially with the size of the model. However, for comparison, we also translated the models into Petri nets and ran the Woflan tool [40] on these nets. In all cases, we found that DiagFlow outperforms Woflan. For Sadiq1, Woflan takes 11 s running time, for Sadiq4 1047 s, and for Sadiq5 there was no result (the Woflan program was terminated after 4.50 h).

The main bottleneck in Woflan appears to be the analysis of the occurrence graph for computing locking scenarios to provide feedback for the end user. This step takes an exponential time with a higher order exponent than for DiagFlow, since it requires computation of the state space of the Petri net that underlies the workflow graph; the number of states is exponential in the size of the net. Note that DiagFlow does not compute the state space of the workflow graph. Instead, it computes a subgraph of the workflow graph, which is much less expensive to compute than a state space, since a subgraph contains less details than a state space. Still, however, feedback provided by DiagFlow is at least as useful as the feedback provided by Woflan, as we show in the next section.

7. Discussion and related work

We defined two IP formulations, an exact one to check for existence of a correct instance subgraph, and a relaxed one to check correctness and do detailed diagnosis. Using the example in Fig. 1, we compare the IP approach with the reduction algorithm of [33] and the Woflan-based approach of [1]. Next, we discuss other process verification approaches.

The solution found by the reduction algorithm of [33] for Fig. 1 is given in Fig. 2, already discussed in Section 1. The algorithm stops here because none of the reduction rules can be applied anymore. But with our approach, we can diagnose specific joins, as explained in Section 4.3. This additional diagnosis information is more useful to a user in pinpointing the problem in the process than the information provided by the reduction based algorithm of [33]. Nevertheless, the reduction rules of Sadiq and Orlowska can be integrated with our approach: first the reduction rules can be applied to the workflow graph, and next the reduced workflow graph can be converted into an IP model. In this way, a reduced IP model is obtained, and therefore, our approach complements that of [33].

Based on Lin et al. [23], van der Aalst et al. [1] point out that the approach of [33] is not complete, and they propose to use Petri net-based analysis techniques to diagnose workflows. These techniques have been implemented in the Woflan tool [40] that offers several diagnostics in case of errors. However, these diagnostics are sometimes contradictory and do not always pinpoint the actual error as we show next. The three main types of Woflan diagnostics are mismatches, locking scenarios, and coverability of threads of control. A mismatch occurs if there is a pair of split and join nodes, connected by two directed disjoint paths, that have different types. Such pairs can be detected from the syntax of the workflow model. van der Aalst et al. [1] suggest that a deadlock corresponds to an XOR split that has a corresponding AND join while a lack of synchronization corresponds to an AND split that has a corresponding XOR join. However, applying Woflan to the example of Fig. 1 yields 3 mismatched pairs: (C3S,C6J), (C3S,C2J), and (C9S,C10J). None of these pairs identify that C2J′ is the main cause of the problem. Moreover, pair (C9S,C10J) belongs to a part of the workflow which is correct.

Another diagnostic is a locking scenario, which is a series of activities that lead to a point of execution from which no proper termination is possible. For Fig. 1, Woflan returns 126 of these scenarios, all of which stop after activity A2 has been done. Again, this does not indicate that C2J′ is causing the problem. Finally, Woflan computes whether each element of the workflow can be covered by a thread of control, i.e. a sequential state machine. Woflan points out that the left incoming edge of C2J′ is not covered by any thread, but it is not clear how the workflow can be adjusted to repair this. The IP diagnosis also identifies C2J′ as cause of the error, but in addition provides an incorrect instance subgraph that illustrates the flaw at C2J′. In sum, for the example in Fig. 1, the three Woflan diagnostics provide different, contradictory clues which are not very accurate. In contrast, the detailed diagnosis provided by the IP approach gives exact feedback to the workflow designer.

Several alternative approaches for the verification of graph-based workflow models are discussed in [5,6,11,19,20,30,36]. Other research has looked at verification in the context of non-graph-based methods of workflow modeling; some notable examples are: data-centric business processes [8,35] and processes based on event-condition-action (ECA) rules [4]. Since the graph-based verification approaches are most closely related to our work, we discuss these in more detail. The verification approaches by Lin et al. [23] and Touré et al. [36] extend the approach of Sadiq and Orlowska [33]. Consequently, like [33] these approaches can detect structural conflicts, but do not give details on the causes of found conflicts.

A verification approach based on workflow decomposition is given in [6]. This approach cannot verify unstructured workflows that are not decomposable. Kiepuszewski et al. [22] address the possibility that an unstructured workflow can be mapped to a structured one through equivalence preserving transformations, but the discussion is mainly through examples, and lacks generality. Logic-based approaches for workflow verification are discussed by Bi and Zhao [5]. While a propositional logic program
can also be represented as an integer program, the latter offers greater ease of representation for verification. For example, a constraint to allow only one (out of n) active incoming edges at an XOR join can be written as one IP constraint while it would require n logic constraints. While the logic formulation does constraint satisfaction, the IP can also optimize an objective. Techniques for verification of metagraph based workflows are discussed in [30]. Next, there are approaches for analyzing workflow designs that use model checking [11,19], but there only one error trace (corresponding to one flawed instance subgraph) is returned, so the feedback is less detailed than in the IP approach (cf. the algorithm in Fig. 6).

Work by Vanhatalo et al. [38] describes faster algorithms for process verification based on decomposition of a processes into Single-Entry-Single-Exit (SESE) fragments. The main idea is to apply local deadlock and lack of synchronization checking rules to each SESE fragment. The heuristics proposed by the authors work fast on well-structured graphs and some types of unstructured graphs. However, there is still a large percentage of graphs (about 30% in the authors’ sample) for which soundness cannot be decided by their method, so their method is not complete. Our approach can be integrated with their method and it can be applied to such complex graphs and their fragments. A related paper [39] shows how to decompose a workflow graph into R fragments (with unique entry and exit nodes) and N fragments (with unique entry and exit edges). The properties of these concepts are explored and then applied to develop methods for the completion of a process graph with multiple end nodes into one with a single end node. This completion approach can be used as preprocessing step for the IP approach: a workflow graph violating constraint 2 (single exit node) can be preprocessed and transformed into a graph that does satisfy constraint 2. Thus, the SESE approach [38,39] is complementary to the IP approach. Related work by the same authors focuses on region trees [14] and refined process trees [37] as techniques for workflow analysis, decomposition and transformation.

It should also be noted that there is a branch of related work that addresses modeling support for users [13,16,26,27,31]. This research, though not directly related to process verification, aims at providing systematic modeling support methods to users for developing process models. By applying these methods, the number of modeling errors can be reduced, thus simplifying the verification task. Work on metrics for process models also has implications for verification, since an empirical study by Mendling et al. [28,29] shows that these metrics are useful to estimate the probability of errors in a process model. This error probability can be reduced by ensuring that the process model performs well on the process model metrics. Still, verification is needed to check that a process model is free of errors.

8. Conclusions

Recognizing the importance and continuing need for effective and efficient techniques for analysis of workflows, we developed and tested an IP based approach for verification of workflow models represented as workflow graphs. The approach consists of several phases in which a workflow graph is converted into an IP formulation. The approach has been implemented in the DiagFlow tool that diagnoses XPDL workflow models, and provides the verification feedback visually in a graphical representation of the XPDL model. While other methods for verification already exist, we showed that our approach has some unique features that complement the other approaches very well.

In particular, the proposed approach gives precise diagnostic information that helps a workflow designer correct flaws in workflow graphs. It can also be easily adapted for checking arbitrary semantic constraints and for verifying new workflow patterns by adding new constraints to capture their behavior. Finally, the running time with the DiagFlow tool on models of various sizes was very impressive as compared with Woflan. A limitation is that at present this approach can check correctness only for workflows containing block structured loops, i.e. loops that contain single-entry–single-exit blocks, but not arbitrary cycles in the workflow graph. However, many common workflow processes can be designed to satisfy this restriction.

For future work, we envision an extension of our tool that allows a user to specify arbitrary constraints, and interactively check if they are true, through the tool. We would also like to extend the tool with the ability to verify advanced workflow patterns, as well as make it flexible by giving users the ability to add their own new patterns in a convenient manner.

Acknowledgment

Some part of the work on this paper was done while the second author was visiting the Information Systems Department (now part of the School of Industrial Engineering) at the Technical University of Eindhoven. He appreciates the hospitality of the hosts. His research was funded in part by the Smeal College of Business.

Appendix A. Introduction to integer programming

Integer programming is a versatile technique that has applications in a variety of areas such as train scheduling, network design, vehicle routing, transportation, facility location, production planning, etc. An integer program consists of an objective function defined in terms of variables so as to represent a quantity of interest such as profit, revenue, cost, etc. This objective function is to be maximized or minimized subject to constraints expressed in terms of the variables. The solution to the integer program is the values of the variables that produce an optimal solution, and the value of the objective function for these variable values. To illustrate, we consider an example from production planning where a manufacturer is planning to produce new widgets. The setup cost of the production facilities and the unit profit for each widget are given in Table 4.

The company has two factories that are capable of producing these widgets (see Table 5). Factories 1 and 2, respectively, have 480 and 720 h of production time available for the production of these widgets.
In order to avoid doubling the setup cost, only one factory could be used. Therefore, the manufacturer wants to know which of the new widgets to produce, where and how many of each (if any) should be produced so as to maximize the total profit. We need to introduce 0–1 decision variables to formulate the above problem as an integer program as follows:

- \( f_{ij} = 1 \) if factory \( i \) (\( i = 1, 2 \)) is setup to produce widgets of type \( j \) (\( j = 1, 2 \)), \( f_{ij} = 0 \) otherwise.
- \( x_{ij} \) is the number of widgets of type \( j \) (\( j = 1, 2 \)) produced in factory \( i \) (\( i = 1, 2 \)), where \( x_{ij} \geq 0 \) and integer.

We can next write the objective function in terms of these variables and the profit margins and costs as:

\[
\text{Maximize } 12(x_{11} + x_{21}) + 16(x_{12} + x_{22}) - 45,000(f_{11} + f_{21}) - 76,000(f_{12} + f_{22})
\]

Next we describe the constraints. There are two types of constraints:

**Type 1 constraints.** At each factory we cannot exceed the production time available. This is expressed by means of two constraints, one for each factory, as:

- **Constraint 1:** \( x_{11}/52 + x_{12}/38 \leq 480 \)
- **Constraint 2:** \( x_{21}/42 + x_{22}/23 \leq 720 \)

**Type 2 constraints.** We cannot produce a widget \( j \) at a factory \( i \) unless that factory is setup to do so, so \( f_i = 1 \). In that case, the total number of produced widgets \( j \) is limited by the hourly production rate for \( j \) and the available hours. We need four constraints because there are two possible setups at each factory. Thus:

- **Constraint 3:** \( x_{11} \leq 52 \times 480 \times f_{11} \)
- **Constraint 4:** \( x_{12} \leq 38 \times 480 \times f_{12} \)
- **Constraint 5:** \( x_{21} \leq 42 \times 720 \times f_{21} \)
- **Constraint 6:** \( x_{22} \leq 23 \times 720 \times f_{22} \)

The complete formulation of this integer programming problem consists of the objective function and the six constraints. Next, we can store the objective function and the constraints in a file and after minor reformatting give it as input to a standard solver like LPSolve or CPLEX. The solution for this problem is:

- **Objective value:** 572,400
- **Variable \( x_{11} \):** 24,960
- **Variable \( x_{21} \):** 30,240
- **Variable \( f_{11} \):** 1
- **Variable \( f_{21} \):** 1

This result is quite intuitive. It shows that both factories should only produce widget 1. The quantity to be produced in factory 1 is 24,960 and in factory 2 is 30,240. After subtracting the setup cost the total profit is 572,400.

**Appendix B. Generalization of IP formulation to polyadic splits and joins**

To simplify the presentation, we have restricted ourselves in Sections 2 and 3 to binary splits and joins, so each split has two outgoing edges and each join has two incoming edges. In this section, we generalize the definitions of these sections by presenting the exact and the relaxed IP formulations for polyadic splits and joins, so each split (join) has \( k \) outgoing (incoming) edges, where \( k \geq 2 \).
We first present the revised version of the Basic IP formulation (Definition 3). Considering polyadic splits and joins only affects constraints IP3, IP4, IP5, and IP6, so we present generalizations for these four constraints:

\[
\text{(IP3) For } n \in S_h : \sum_{i=1}^{k} \text{outedge}(n)_i - kn = 0, \text{where } k = |\text{outedge}(n)|
\]

\[
\text{(IP4) For } n \in J_A : \sum_{i=1}^{k} \text{inedge}(n)_i - kn = 0, \text{where } k = |\text{inedge}(n)|
\]

\[
\text{(IP5) For } n \in S_k : \sum_{i=1}^{k} \text{outedge}(n)_i - n = 0, \text{where } k = |\text{outedge}(n)|
\]

\[
\text{(IP6) For } n \in J_k : \sum_{i=1}^{k} \text{inedge}(n)_i - n = 0, \text{where } k = |\text{inedge}(n)|
\]

Next, we discuss how the constraints for polyadic AND and XOR joins can be relaxed (cf. Section 3). For each AND join \( n \in J_h \), generalized constraint IP4 is relaxed into

\[
\sum_{i=1}^{k} \text{inedge}(n)_i - n \leq k - 1.
\]

where \( k = |\text{inedge}(n)| \). This constraint forces \( n \) to be 1 if all its incoming edges are 1. The remaining constraints for relaxation of AND joins do not change, so for each \( i : 1 \leq k, n \leq |\text{inedge}(n)| \) as before.

For each XOR join \( n \in J_k \), generalized constraint IP6 is relaxed into the following constraints

\[
\text{(XORJ1) } \sum_{i=1}^{k} \text{inedge}(n)_i \geq a
\]

\[
\text{(XORJ2) } \sum_{i=1}^{k} \text{inedge}(n)_i \leq ka
\]

\[
\text{(XORJ3) } \sum_{i=1}^{k} \text{inedge}(n)_i \geq n
\]

\[
\text{(XORJ4) } n + \sum_{i=1}^{k} \text{inedge}(n)_i \geq 2a
\]

\[
\text{(XORJ5) } n + \sum_{i=1}^{k} \text{inedge}(n)_i \leq 2,
\]

where as before \( k = |\text{inedge}(n)| \).

Constraint XORJ1 and XORJ3 force \( a \) and \( n \) to be 0 if all incoming edges are 0. Constraint XORJ2 forces \( a \) to be 1 if at least one incoming edge is 1. Combining XORJ1 and XORJ2, \( a \) is 1 if and only if one or more incoming edges of \( n \) are 1, so they are active.

Next, constraint XORJ4 forces \( n \) to be 1 if \( a \) is 1 and one of the incoming edges is 1. If two or more incoming edges are 1, XORJ4 allows \( n \) to be either 0 or 1. Note that XORJ4 also allows that both \( a \) and \( n \) are 0: if \( a = 0 \) then \( \sum_{i=1}^{k} \text{inedge}(n)_i = 0 \) by XORJ1 and XORJ2, and consequently \( n = 0 \) by XORJ3.

Constraint XORJ4 cannot be expressed without using an auxiliary variable like \( a \). The next constraint, XORJ5, forces \( n \) to be 0 if two or more incoming edges are 1 to reflect that there is a problem at the XOR join node.

References


Rik Eshuis is an assistant professor in information systems at Eindhoven University of Technology, The Netherlands. He received an MSc degree (1998) and a PhD degree (2002) in Computer Science from the University of Twente. Afterwards, he spent one year as a visiting postdoctoral researcher at the CRP Henri Tudor and LIASIT in 2002 in Computer Science from the University of Twente. Afterwards, he spent one year as a visiting postdoctoral researcher at the CRP Henri Tudor and LIASIT in Luxembourg. His main research interest is in model-driven engineering of process-oriented information systems. Concrete topics include service-oriented process composition, process views, process modeling and process analysis. He has published papers on these topics in leading academic journals and international conferences. Dr. Eshuis is a member of IEEE and ACM.

Akhil Kumar is a professor of information systems at the Smeal College of Business at Penn State University. He received his Ph.D. from the University of California, Berkeley, and has previously been on the faculties at Cornell University and the University of Colorado, and also spent one year at Bell Labs, Murray Hill, NJ. His research interests are in workflow systems, Web services, distributed information systems and intelligent systems. He has done pioneering work in data replication techniques and in advancement of XML-based workflows. He has published more than 80 papers in academic journals and international conferences. He has served on several editorial boards and program committees. Currently he is an Associate editor for IEEE Transactions on Services Computing and Information and Technology Management journals.