Diagnosing Correctness of Semantic Workflow Models

Diana Borrego[a,*], Rik Eshuis[b], María Teresa Gómez-López[a], Rafael M. Gasca[a]

[a] *University of Seville, Department of Computer Languages and Systems, Av Reina Mercedes S/N, 41012 Seville, Spain*
[b] *Eindhoven University of Technology, School of Industrial Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*

Abstract

To model operational business processes in an accurate way, workflow models need to reference both the control flow and dataflow perspectives. Checking the correctness of such workflow models and giving precise feedback in case of errors is challenging due to the interplay between these different perspectives. In this paper, we propose a fully automated approach for diagnosing correctness of semantic workflow models in which the semantics of activities are specified with pre and postconditions. The control flow and dataflow perspectives of a semantic workflow are modeled in an integrated way using Artificial Intelligence techniques (Integer Programming and Constraint Programming). The approach has been implemented in the DiagFlow tool, which reads and diagnoses annotated XPDL models, using a state-of-the-art constraint solver as back end. Using this novel approach, complex semantic workflow models can be verified and diagnosed in an efficient way.

Keywords: workflow, business process management, diagnosis, constraint programming, integer programming

1. Introduction

Nowadays, organizations automate their business processes with workflow models that can be enacted using workflow management systems (WFMSs).

*Corresponding author. Tel. +34 954 556 234. Fax. +34 954 557 139
Email addresses: `dianabn@us.es` (Diana Borrego), `h.eshuis@tue.nl` (Rik Eshuis), `maytegomez@us.es` (María Teresa Gómez-López), `gasca@us.es` (Rafael M. Gasca)

For organizations it is essential to ensure the correct operation of workflow models at design time, before the workflow models get enacted. An incorrect operational workflow can dissatisfy customers and fixing the errors can be very costly, certainly compared to the costs of fixing the workflow model before it is deployed. Correctness of a workflow model can be verified by exhaustively checking all possible executions. Detected errors should be *diagnosed*, for instance by providing an error path that shows the cause of the error, such that errors can be repaired in a quick and effective way [1].

Workflow models can reference different perspectives [2]. Most workflow modeling and verification approaches only consider the control flow perspective [1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13], which is about the order in which the individual activities of a business process are executed. Another relevant perspective is the dataflow perspective [14], which details the flow of data among activities subject to certain constraints. The dataflow perspective is important because data constraints influence the possible executions of activities [14] and in turn, the execution of activities results in certain data constraints being enforced.

An effective means to express data constraints is to annotate activities in a workflow model with pre and postconditions that specify the effect on the data state for each activity. For instance, in the Sarbanes-Oxley Act of 2002, the internal audit department takes the lead and works alongside workflow owners for each process that has a direct effect on the data for the financial reporting. Annotating activities inside these processes with pre and postconditions facilitates compliance checking to ensure that workflows are properly designed.

Only recently, approaches for verifying workflow models with dataflows have been proposed [14, 15, 16, 17, 18]. However, these approaches do not consider diagnosis of dataflow errors. Diagnosing dataflow errors is complex due to the interplay between control flow and dataflow dependencies, as we explain in Section 2 with an example.

The goal of this paper is to develop an approach for *diagnosing the correctness of semantic workflow models*, containing activities whose effects are formally specified using pre and postconditions. An activity can start if the execution of the workflow model has reached the activity and its precondition is satisfied. Upon completion, the activity delivers data that satisfies its postcondition. An execution of the workflow can reach an activity whose precondition is not satisfied. In that case the execution gets stuck at the activity and fails.

We distinguish between two different notions of correctness to diagnose such dataflow errors:

- May-correctness. A workflow model is may-correct if every activity can be executed at least once, so there is an execution in which the activity is done.
- Must-correctness. A workflow model is must-correct if every possible execution that reaches an activity satisfies the precondition of the activity.

The diagnosis is performed at design-time, using Artificial Intelligence techniques to compute the execution instances allowed by a workflow model. For diagnosis, the workflow model is translated into two models: (1) an Integer Programming model (IP model), to determine the different instances of execution of the workflow, and (2) the preconditions and postconditions of the activities are modeled as constraints in a Constraint Satisfaction Problem (CSP) [19], following a BNF grammar in order to avoid any ambiguity.

This paper makes several contributions:

- *Workflow data graphs* are proposed as a formalism for modeling semantic workflows with pre and postconditions for the activities. These conditions are modeled as constraints according to a well-defined grammar in BNF.
- Two correctness notions for workflow data graphs, *may* and *must-correctness*, are proposed and novel diagnosis algorithms are developed for verifying may and must-correctness. The algorithms are complete: neither false positives nor false negatives are generated. Moreover, the algorithms offer precise diagnosis of the detected errors, indicating the execution causing the error where the workflow gets stuck.
- The approach has been implemented in the DiagFlow tool, presented in [1]. The tool reads XPDL models [20] in which the semantics of activities and the corresponding dataflow are specified using extended attributes.

This paper is organized as follows. Section 2 presents a motivating example to illustrate the concepts of may and must-correctness. Section 3 introduces workflow data graphs as a formal model for semantic workflows and defines may and must-correctness on workflow data graphs. Section 4 defines the IP and CSP formulations of a workflow data graph. The process of diagnosis is explained, and two algorithms are presented. The diagnosis of the motivating example is performed. Section 5 gives implementation details. Section 6 shows experimental results. Section 7 presents an overview of related work found in the literature. And finally, conclusions are drawn and future work is proposed in Section 8.

2. A Motivating Example

This section introduces an example of a semantic workflow model, shown in Figure 1. This example describes the handling of a conference for an organizing committee, and it is used to illustrate the concepts of may and must-correctness in semantic workflow models. We use BPMN 2.0 [21] to visualize workflow models.

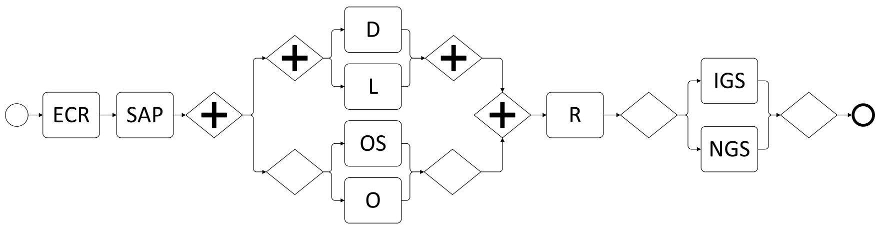


Figure 1: **Motivating Example**

Figure 1 shows a workflow that consists of nine activities (rectangles with rounded corners) and eight gateways or control nodes (diamonds), and a start and end event (circles). A gateway with one incoming edge and multiple outgoing edges is called a split; a gateway with multiple incoming edges and one outgoing edge is a join. Gateways with the +-symbol are AND: all incoming edges are required to pass the gateway, and the gateway activates all outgoing edges. The other gateways are XOR: one incoming edge can pass the gateway and one of the outgoing edges is activated as a result. In the figure, the activity labels are abbreviations of activity names that are listed in Table 1. The workflow performs the following steps:

1. The workflow starts with the establishment of the conference rate (ECR activity), in order to begin the registration period.

Table 1: Activities of the example

Abbreviation	Activity
ECR	Establishment of Conference Rate
SAP	Selection of Accepted Papers
D	Dinner
L	Lunch
OS	Other expenses + Social event
O	Other expenses
R	Registration
IGS	International Guest Speaker
NGS	National Guest Speaker

2. In the activity SAP, the process of acceptance of papers for the conference takes place. The number of final papers is determined.
3. The workflow is split into two branches. In the upper one, the cost of the gala dinner (D) and the lunches (L) to serve during the conference are calculated concurrently. On the lower branch, the workflow is routed according to the money spent in the social events during the conference (O or OS activities).
4. Next, the registration of the attendees of the conference takes place (R).
5. And finally, the workflow is routed depending on the available money to spend in the invitation of national or international guest speakers (NGS or IGS).

The activities in the example consume and produce data during the execution of the workflow by reading and writing variables. Those variables are listed in Table 2 with their corresponding domains and meanings. Table 3 shows how these variables are used by the activities of the example, indicating if they are read (*rd*) or written (*wt*).

Each activity in a workflow uses two types of condition over the dataflow which must be satisfied. A precondition must be satisfied prior to execution of the activity. If not, the workflow gets stuck at the activity and fails. A postcondition is satisfied immediately after the activity has finished. The preconditions and postconditions of the activities in the example in Figure 1

Table 2: Data input for the example in Figure 1

Variable	Domain	Meaning
regFee	{200..390}	Conference registration fee
sponsorship	{0..15000}	External contributions to support the event
numPapers	{50..80}	Number of accepted papers
dinner	{60..100}	Gala dinner cost
lunch	{10..30}	Cost of each lunch served during the conference
others	{30..185}	Money for other expenses, like social events
confAtt	{75..170}	Number of conference attendees
guestSpeaker	{0..10000}	Money to spend in inviting a guest speaker

are shown in Table 4. The notation is explained in the next section.

It is easy to check that the workflow is correct from the control flow perspective: each activity can be performed and there are no deadlocks, so the workflow can always complete. To assess the correctness of the example from the dataflow perspective, there are two important types of questions. Both questions test whether the precondition of an activity a can be satisfied by considering an arbitrary partial execution of the workflow in which a is to be executed next. The partial execution of the workflow results in a data state (assignment of values to variables) that has to satisfy the precondition of a.

One question is whether for each activity a there exists at least one partial execution of the workflow in which a can be done next *and* the resulting data state satisfies the precondition of a. In that case, activity a can become enabled and executed. Otherwise, the execution of the workflow may get stuck at a, if the current data state does not satisfy the precondition of a. If every activity can be executed, the workflow is *may-correct*.

The other relevant question is whether every possible partial execution of the workflow results in a data state that satisfies the precondition of the activity to be executed next. Phrased differently, can every possible partial execution always be continued such that eventually the end state is reached? If the answer is positive, the workflow is *must-correct*.

Note that must-correctness is stronger than may-correctness. It is straightforward to check that if a workflow is must-correct, it is also may-correct.

Table 3: Data read and written on each activity

	Activities								
Variables	ECR	SAP	D	L	OS	O	R	IGS	NGS
regFee	wt	-	rd	rd	rd	rd	rd	rd	rd
sponsorship	wt	-	rd	rd	rd	rd	-	rd	rd
numPapers	-	wt	rd	rd	rd	rd	rd	-	-
dinner	-	-	wt	-	-	-	rd	rd	rd
lunch	-	-	-	wt	-	-	rd	rd	rd
others	-	-	-	-	wt	wt	rd	rd	rd
confAtt	-	-	-	-	-	-	wt	rd	rd
guestSpeaker	-	-	-	-	-	-	-	wt	wt

May-correctness can be used as sanity check for each activity to see whether its precondition is not too strict. Must-correctness can be used to check the correctness of the entire workflow with all the activities.

In the case of the example in Figure 1, the workflow is may-correct since every activity is executable. On the other hand, it is not must-correct: for example, if the partial execution contains activities *ECR*, *SAP*, *D*, *L*, and *OS* then the resulting data state can assign the following values to the variables: *regFee*=200, *dinner*=100, *lunch*=30, *others*=30. But now the activity to be executed next, *R*, has a precondition that is false, since $3{*}30+100+30 \not< 200$. Therefore, with that assignment of the variables, the workflow gets stuck at activity *R*.

Note that all activities have correct pre and postconditions, and that the workflow model has a correct control flow definition: no deadlock occurs if the dataflow (pre and postconditions) is abstracted from. The error is caused by the interplay between the control flow, which specifies that *ECR*, *SAP*, *D*, *L*, and *OS* are performed before *R*, and the dataflow as specified by the pre and postcondition of each activity, which determines the possible data states just before *R*.

The error can be repaired in several ways, for instance by relaxing the precondition of *R*, by strengthening the postconditions of *D*, *L* and *OS*, or by rearranging the control flow. So finding an error at a precondition does not necessarily imply the precondition itself is flawed.

Table 4: Activities with their pre and postconditions

Activity	Precondition and Postcondition
ECR	**pre:** $true$ **post:** $true$
SAP	**pre:** $true$ **post:** $true$
D	**pre:** $sponsorship > 0 \vee numPapers > 60$ **post:** $regFee * 0.1 \leq dinner \wedge dinner \leq regFee * 0.35$
L	**pre:** $sponsorship > 0 \vee numPapers > 60$ **post:** $regFee * 0.1 \leq 3 * lunch \wedge 3 * lunch \leq regFee * 0.35$
OS	**pre:** $sponsorship > 0 \vee numPapers > 60$ **post:** $others \leq 0.2 * regFee + 0.05 * sponsorship \wedge$ $others \geq 0.05 * regFee + 0.05 * sponsorship$
O	**pre:** $sponsorship > 0 \vee numPapers > 60$ **post:** $others \leq 0.25 * regFee \wedge others \geq 0.05 * regFee$
R	**pre:** $3 * lunch + dinner + others < regFee$ **post:** $numPapers * 1.8 \geq confAtt$ $\wedge numPapers * 0.5 \leq confAtt$
NGS	**pre:** $confAtt * (3 * lunch + dinner + others) <$ $confAtt * regFee + sponsorship$ **post:** $guestSpeaker \geq 0.2 * sponsorship \wedge$ $guestSpeaker \leq sponsorship + 0.1 * regFee * confAtt$
IGS	**pre:** $confAtt * (3 * lunch + dinner + others) <$ $confAtt * regFee + sponsorship$ **post:** $guestSpeaker \geq 0.4 * sponsorship \wedge$ $guestSpeaker \leq sponsorship$

3. Workflow Data Graphs

In order to analyze may and must-correctness of semantic workflows, we propose an approach based on graph-theory and Artificial Intelligence techniques.

In this section, we define workflow data graphs, including the structural constraints that they should satisfy. Next, the notions of *data instance subgraph* and correctness for workflow data graphs are introduced. Finally, the concepts of *partial instance subgraph* and *border activity* are presented. The definitions extend earlier proposed definitions for the control flow perspective of workflow models [1, 22] by adding data.

3.1. Definition

A workflow data graph, such as the one shown in Figure 1, is a set of activities that is ordered to a set of procedural rules. The effect of each activity is specified with a pre and postcondition. The order of execution of the activities is specified by means of directed edges, with a unique start and a unique end node.

Definition 1. *A workflow data graph is a tuple $P = (Act, V, D, C, E, pre, post, wt)$ where:*

- *Act is a set of activities;*
- *V is a set of typed variables;*
- *D is a set of finite domains (types), that contains for each variable $v \in V$ a finite domain D_v;*
- *C is a set of control nodes (gateways), partitioned into disjoints sets of XOR splits S_{XOR}, AND splits S_{AND}, XOR joins J_{XOR}, AND joins J_{AND}, and $\{start, end\}$ where start is the unique start node and end the unique end node. Each split in S_{XOR} counts on condition expressions for each gate of the gateway in order to specify the flow depending on the data;*
- *$E \subseteq Act \times Act$ is a set of edges which determine precedence relation;*
- *$pre : Act \rightarrow Cs(V)$ assigns to each activity its precondition (a constraint $c \in Cs$ on the set of variables V);*
- *$post : Act \rightarrow Cs(V)$ assigns to each activity its postcondition (a constraint $c \in Cs$ on the set of variables V);*
- *$wt : Act \rightarrow V$ assigns to each activity its written variables;*

To simplify the exposition, OR gateways are not considered. We plan to consider OR gateways in future work. Also, we do not consider guard conditions. Other work considers verification of workflow models with dataflows and guard conditions but without preconditions [17]. In future work, we plan to consider guard conditions.

Let $a \in Act$ be an activity. We use the following rule. Variables in the precondition of a, so in $var(pre(a))$, are read by a. Variables in the postcondition of a, so in $var(post(a))$, are read and/or written by a.

Definition 2. *Let $v \in V$ be variable, let int_val be an integer value, let nat_val be a natural value, and let $float\_val$ be a float value. The set of constraints on V, denoted $Cs(V)$, is generated by the following grammar in BNF:*

$$\begin{array}{rcl}
constraint & ::= & Atomic\_Constraint\ BOOL\_OP\ Constraint \\
& \mid & Atomic\_Constraint \mid \neg Constraint \\
BOOL\_OP & ::= & \wedge \mid \vee \\
Atomic\_Constraint & ::= & Function\ \ PREDICATE\ \ Function \\
Function & ::= & v\ \ FUNCTION\ \ Function \\
& \mid & v \mid int\_val \mid nat\_val \mid float\_val \\
PREDICATE & ::= & < \mid \leq \mid = \mid > \mid \geq \\
FUNCTION & ::= & + \mid - \mid *
\end{array}$$

Next, each workflow data graph should satisfy the following structural constraints on its control flow [1]:

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 has one incoming and one outgoing edge;
4. each split node has one incoming and at least two outgoing edges;
5. each join node has at least 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.

As it was mentioned in earlier work [1], formalizations of these constraints are presented elsewhere [23]. The first five constraints are self-explanatory. Constraints 1 and 2 were also included by Sadiq and Orlowska [22]. 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 modelled by using a join before the end node. The sixth constraint rules out unconnected workflows because such workflow graphs contain unreachable parts and therefore are flawed by default.

The last constraint is also placed by other works on workflow verification [22, 3, 9]. However, workflow graphs are still sufficiently expressive to model loops that involve blocked iteration [22, 16]. Basically, any block in a correct workflow graph can be repeated multiple times without affecting control flow correctness, since a block has a single point of entry and a single point of exit. Since we do not consider guard conditions, to verify workflow models with loops a strong fairness constraint is required to ensure that loops are exited eventually [24].

As in [1], we also use auxiliary functions $inedge, outedge : N \to \mathcal{P}(E)$, 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, $inedge(n) = \{(x, y) \in E \mid y = n\}$ and $outedge(n) = \{(x, y) \in E \mid x = n\}$. 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 $inedge_2(n) = e_2$.

3.2. Analysis

In order to check the may and must-correctness of a workflow, it is necessary to check the executability of each activity. That executability depends on the precondition of the activity being checked, and on the pre and postconditions of the activities executed before it in a particular instance of the workflow. To define it formally, Sadiq and Orlowska [22] 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. The part of the workflow graph 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 elsewhere [23].

For this paper, we introduce two types of instance subgraphs: partial and complete. A *partial instance subgraph* of a workflow graph is generated 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 based on a guard condition;

- if an AND split node is visited, then all outgoing edges are visited [1];
- 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 [1];
- if an AND join node is visited, then its outgoing edge is visited only if all its incoming edges have been visited too [1].
- if an activity is visited, then its outgoing edge is visited too.

A *complete instance subgraph* is generated by traversing a workflow graph from the start node using the rules for partial instance subgraphs plus the additional rule:

- if the incoming edge of an activity is visited, then the activity is visited too.

Control flow verification [1, 22] only considers complete instance subgraphs. However, to verify dataflows, we also need to consider partial instance subgraphs, which contain the incoming edge of an activity but not the activity itself. To check whether the precondition of the activity is satisfied by the partial instance subgraph, we need to identify the possible assignments of variables written by the activities in the instance subgraph.

A *data instance subgraph* is an instance subgraph together with an assignment of values to the variables in V. The assignment must be feasible according to the postcondition of the activities visited last. To formalize this properly, we introduce the following notion: an activity a in an instance subgraph (partial or complete) is a *border activity* if there is no activity a' in the instance subgraph such that there is a directed path from a to a'. For the workflow in Figure 1, the subset of activities shadowed in Figure 2 ($\{ECR, SAP, D, L, OS\}$) induces a partial data instance subgraph, whose border activities are D, L, and OS. Note that due to the traversal rules the data instance subgraph also contains the successor (control) nodes of the border activities. In a data instance subgraph, the assignment of values to variables must be consistent with the postcondition of each border activity.

Definition 3. *A data instance subgraph of a workflow graph* $(Act, V, D, C, E, pre, post, wt)$ *is a tuple* $(Act', V', D', C', E', pre', post', wt', \nu)$ *where:*

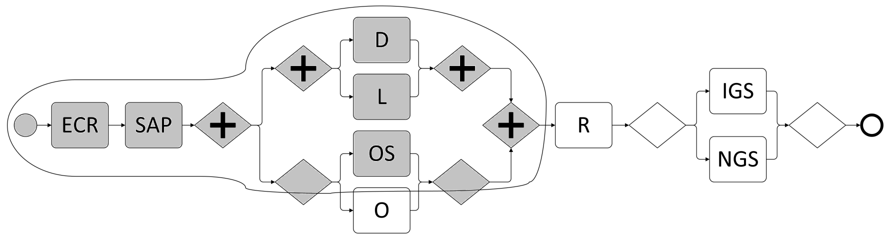


Figure 2: Example of instance subgraph (shadowed activities)

- $(Act', V', D', C', E', pre', post', wt')$ *is an instance subgraph with* $Act' \subseteq Act$*,* $V' = V$*,* $D' = D$*,* $C' \subseteq C$*,* $E' \subseteq E$*,* $pre' = pre \cap (Act' \to Cs(V))$*,* $post' = post \cap (Act' \to Cs(V))$*, and* $wt' = wt \cap (Act' \to V)$*, and*

- ν *is an assignment of values to the variables in* V'*, so that each variable* $v \in V'$ *is instantiated with a value* $\nu(v)$ *in its domain* $D'(v)$*. For each border activity* $a \in Act'$*, the valuation of variables in* $var(post(a))$ *should satisfy the postcondition* $post(a)$*. As an example, for the border activities D, L and OS in Figure 2, the valuation* ν *of var(post(D)) (i.e.,* ν*(regFee) and* ν*(dinner)), var(post(L)) (i.e.,* ν*(regFee) and* ν*(lunch)) and* $var(post(OS))$ *(i.e.,* ν*(others),* ν*(regFee) and* ν*(sponsorship)), satisfies the postconditions* $post(D)$*,* $post(L)$ *and* $post(OS)$ *respectively.*

There are two types of possible error for data instance subgraphs. First, a data instance subgraph can get stuck at an XOR or AND join. Then it contains the join but not the outgoing edge of the join [1]. This is a control flow error that can be detected using existing techniques [1, 22]. We therefore ignore such errors for the remainder of this paper.

The second error is a dataflow error. Different types of dataflow errors can occur, depending on the level of detail on which the dataflow is specified. At the minimal level, a workflow model specifies for each activity which variables it reads and writes, but the workflow model contains no pre and postconditions. Sun et al. [14] have analyzed which dataflow errors can occur in such workflow models. For the purpose of this paper, the following two errors are important:

- Data is *missing* if a variable is read by an activity, so referenced in its precondition, but not written in any preceding activity.

- Data is *conflicting* if the same variable is written in two parallel activities. In that case, one activity overwrites the value of the variable written earlier by the other activity.

These dataflow errors at the basic level can cause unexpected process interruptions and should therefore be avoided. In Section 4.2 we define an algorithm for detecting data conflicts.

At a more advanced level, a workflow model not only contains variables read and written by activities, but also contains pre and postconditions for activities. For such workflow models, dataflow errors can occur that are not considered by Sun et al. [14]. A precondition of an activity is *violated* if the precondition is not satisfied when the activity can be executed from a control flow point of view. In that case, the workflow gets stuck and fails.

To formalize precondition violation errors, we introduce the notion of trigger. A partial instance subgraph *triggers* activity a if it does not contain a but does contain the incoming edge of a, which is unique. So the instance subgraph "stops" just before a. In the example in Figure 2, the shadowed partial instance subgraph triggers the activity R.

Based on the notion of trigger, we define two new notions of dataflow correctness (cf. Def. 6 and Def. 7). First, we introduce two auxiliary notions in Definition 4 and Definition 5.

Definition 4. *An activity* a *is* may-executable *if there exists a data instance subgraph that triggers* a *and whose valuation of variables satisfies the precondition of* a*.*

Definition 5. *An activity* a *is* must-executable *if every data instance subgraph that triggers* a *has a valuation that satisfies the precondition of* a*.*

Next, we define the two new notions of dataflow correctness.

Definition 6. *A workflow data graph is* may-correct *if every activity is may-executable.*

Definition 7. *A workflow data graph is* must-correct *if every activity is must-executable.*

In a must-correct workflow data graph, no preconditions can be violated. But a may-correct workflow data graph might contain an activity whose precondition can be violated. Still may-correctness is useful, as explained in

Section 2: may-correctness can be used as sanity check for testing whether preconditions are not too strict while must-correctness can be used to check absence of precondition violations.

In the next section, we will formalize data instance subgraphs, including the valuations they allow, as Constraint Satisfaction Problems. We will define algorithms that use the IP and CSP formalizations to analyze the may and must-correctness of workflow data graphs.

4. Diagnosis of Workflow Data Graphs: May and Must-correctness

In this section, we explain how may and must-correctness of workflow data graphs can be diagnosed in a formal way. To formalize the control flow perspective of workflow data graphs, we use an Integer Programming (IP) formulation introduced in earlier work [1]. We formalize the dataflow perspective of workflow data graphs, i.e., pre and postconditions of activities, as Constraint Satisfaction Problems (CSPs). The combination of IP formulation and CSPs formalizes data instance subgraphs (cf. Definition 3). We introduce two diagnosis algorithms for checking may and must-correctness. If a workflow data graph is not may or must-correct, the algorithms identify which activities are responsible for the incorrectness.

We first explain the combined IP and CSP model created for each workflow data graph, explaining the preprocessing which is necessary to avoid conflicts among postconditions, and detect basic errors in the dataflow.

4.1. Combined IP and CSP model

We first explain the IP formulation that covers the control flow perspective of workflow data graph. Next we extend the IP model with CSP constraints that model the dataflow perspective of workflow data graph. Since each IP constraint can also be interpreted as a CSP constraint, we can view the entire model as a CSP model that we can check using CSP solvers.

IP formulation. For every activity $a \in Act$, we need to compute an instance subgraph (cf. Definition 3) that triggers a. For this, we use the IP formulation developed in earlier work [1], in which an IP variable is introduced for each node and each edge of the workflow graph. A solution to the IP formulation encodes an instance subgraph, where an IP variable has value 1 if and only if the corresponding node or edge is part of the instance subgraph. Complicating factor is that the existing IP formulation considers complete instance subgraphs, that can only get stuck at (faulty) AND or XOR joins. But an instance subgraph that triggers a is not complete.

To generate a partial instance subgraph that triggers a, we take the existing IP formulation [1] but replace one constraint. The existing IP formulation uses for each $a \in \mathit{Act}$ the constraint $\mathit{inedge}_1(a) - a = 0$, which states that if the incoming edge of a is activated, so $\mathit{inedge}_1(a) = 1$, then a is activated as well, so $a = 1$. To model that the incoming edge of a is activated but not a, we replace for every $a \in \mathit{Act}$ the constraint $\mathit{inedge}_1(a) - a = 0$ in the original IP formulation [1] with $\mathit{inedge}_1(a) >= a$. This constraint allows that the subgraph "stops" at a ($\mathit{inedge}_1(a) = 1$ and $a = 0$) but disallows that a is spontaneously activated, so $\mathit{inedge}_1(a) = 0$ and $a = 1$ is not allowed.

To generate partial instance subgraphs, as it was mentioned before, we use a slightly modified version of the basic IP formulation [1]. All constraints below, except IP4, are taken from the basic IP formulation [1]. For AND joins we use the relaxed IP formulation (IP4) to allow for partial instance subgraphs that stop at an AND join. In that case, one of the parallel branches synchronised by the AND join has completed, but the other ones have to complete. This behavior is disallowed by the basic IP formulation.

Definition 8. *For a workflow data graph $P = (\mathit{Act}, V, D, C, E, \mathit{pre}, \mathit{post}, \mathit{wt})$, the* ***Relaxed 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 P, so $x \in \{\mathit{Act} \cup C \cup E\}$, an IP variable $\mathbf{x}$ is created. The constraints, adapted from [1], are:*

IP0 `start = 1`

IP1 For $n \in (S_{AND} \cup S_{XOR} \cup \{\mathit{end}\})$: $\texttt{inedge}_1\texttt{(n) - n = 0}$

IP1a For $n \in \mathit{Act}$: $\texttt{inedge}_1\texttt{(n) >= n}$

IP2 For $n \in (\mathit{Act} \cup J_{AND} \cup J_{XOR} \cup \{\mathit{start}\})$: $\texttt{outedge}_1\texttt{(n) - n = 0}$

IP3 For $n \in S_{AND}$, being $|\mathit{outedge}(n)| = k$: $\sum_{i=1}^{k} \texttt{outedge}_i\texttt{(n) - k}\cdot\texttt{n = 0}$

IP4 For $n \in J_{AND}$, being $|\mathit{inedge}(n)| = k$: $\sum_{i=1}^{k} \texttt{inedge}_i\texttt{(n) - k}\cdot\texttt{n} \leq \texttt{1}$

$\forall i \in [1, k]$: $\texttt{n} \leq \texttt{inedge}_i\texttt{(n)}$

IP5 For $n \in S_{XOR}$, being $|outedge(n)| = k$: $\sum_{i=1}^{k} \texttt{outedge}_i\texttt{(n) - n = 0}$

IP6 For $n \in J_{XOR}$, being $|inedge(n)| = k$: $\sum_{i=1}^{k} \texttt{inedge}_i\texttt{(n) - n = 0}$

CSP formulation. Constraint programming is based on the algorithmic resolution of Constraint Satisfaction Problems, and is an Artificial Intelligence technique which provides us a way to model the semantic information of a workflow data graph. A CSP [19] consists of the triple $\langle V, D, Cs \rangle$, where V is a set of n variables $v_1, v_2, ..., v_n$ whose values are taken from finite domains $D_{v1}, D_{v2}, ..., D_{vn}$ respectively, and Cs is a set of constraints on their values. The constraint c_k $(x_{k1}, \ldots, x_{kn})$ is a predicate that is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the value assignment of these variables satisfies the constraint c_k.

The IP model encodes the control flow of a data instance subgraph. We now explain how the dataflow, so the pre and postconditions of the activities contained in the data instance subgraph, are translated into CSP constraints. For each activity $a \in Act$, a constraint of the form $a = 1 \Rightarrow (pre(a) \wedge post(a))$ is defined. That is, if a is part of the partial instance subgraph, then its pre and postcondition should be satisfied. We need the conjunction stipulating that $a = 1$ to ensure that the pre and postcondition are only enforced if a is activated, so a is in the instance subgraph.

Note that the pre and postcondition constraints hold for each activity in the data instance subgraph, not just for border activities (cf. Definition 3). This way, the constraints can be easily encoded in a CSP model. However, this encoding complicates finding a solution to the CSP model, since the postcondition of a border activity might conflict with the postcondition of an earlier executed activity, if both activities reference the same variable. For instance, a workflow data graph can contain two activities A and B that both write integer variable i, where the postcondition of A is $i < 10$ and the postcondition of B is $i > 10$. A data instance subgraph containing A and B can assign only one value to i, so either the postcondition of A or of B is violated. Therefore the postconditions of A and B conflict. To resolve conflicts, we put the CSP model in SSA form, explained next.

SSA form. In order to resolve conflicts among postconditions, we will convert the variables and constraints of the CSP model into Static Single Assignment

(SSA) form. The SSA form is used in compiler design as an intermediate representation for a program [25, 26]. If the workflow data graph is in SSA form, each variable is assigned a value by only one activity. To turn a workflow data graph into an SSA form, each variable v is separated into several variables v_i, each of which is assigned a value by only one activity.

A review of the literature reveals a highly cited algorithm to get the variables in a program in SSA form [26]. The algorithm computes the control flow properties of programs, like conditions (XOR) or loops. As an example of variables renaming, the workflow in Figure 3 uses 4 activities which read and/or write the variables w, x, y, and z. Table 5 shows the pre and post-conditions before and after renaming. Note that two new constraints have been added at activity D. They are known as Φ-functions in [26], and they indicate which assignment to the variable y reaches the join point. That is, the value for variable y depends on the activity which was executed (B or C).

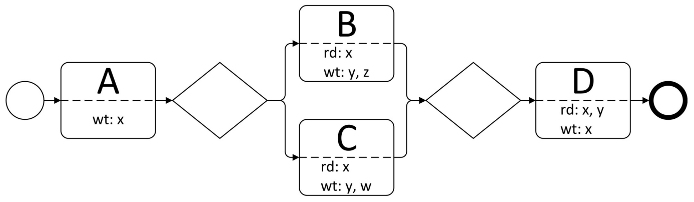


Figure 3: **Workflow Example**

However, the SSA form and the renaming algorithm is defined for sequential programs while workflow data graphs can contain parallelism. Due to parallelism, dataflow errors can arise. For instance, two parallel activities can assign the same variable a value (conflicting data, cf. Section 3). In that case, the assignment to the variable at a subsequent AND join may not be possible due to conflicting constraints in the CSP model. For instance, if in Figure 3 the XOR nodes are replaced with AND nodes, both B and C write variable y. The constraints encoding the Φ-function [26] for the subsequent AND join are now unsatisfiable for $y3$.

Such dataflow errors are at a more basic level than violations of may and must-correctness, as explained in Section 3. Therefore, these dataflow errors need to be detected and resolved before may and must-correctness can be diagnosed. The next section defines an algorithm for detecting basic dataflow

Table 5: Variables before and after the renaming

Activity	Before SSA	After SSA
A	**pre:** $true$	**pre:** $true$
	post: $x > 20$	**post:** $x1 > 20$
B	**pre:** $true$	**pre:** $true$
	post: $y = x + 10$	**post:** $y1 = x1 + 10$
	$z = x * 2$	$z1 = x1 * 2$
C	**pre:** $x < 100$	**pre:** $x1 < 100$
	post: $y = x + 50$	**post:** $y2 = x1 + 50$
	$w = y * 2$	$w1 = y2 * 2$
D		$B = 1 \Rightarrow y3 = y1$
		$C = 1 \Rightarrow y3 = y2$
	pre: $y > x$	**pre:** $y3 > x1$
	post: $x = x * y$	**post:** $x2 = x1 * y3$

errors.

4.2. Detecting basic dataflow errors

As explained in Section 3, two basic dataflow errors are missing data and conflicting data. We next discuss how each type of dataflow error can be detected.

To identify missing data, we use the following constraint. For each activity $a \in Act$ that reads a variable v, if there is a directed path from the start node to a where none of the activities in the path writes v, then missing data is identified.

To identify conflicting data, we use the algorithm presented in Figure 4. Variables in V are processed one by one in a while-loop. The current variable being processed is *current* (line 6). The algorithm iterates over all activities that write *current* in a nested for-loop. For each pair of distinct activities a_1 and a_2, the algorithm tests whether there exists a partial instance subgraph that triggers both a_1 and a_2. The pre and postconditions of the activities are not relevant for this check, so the CSP formulation is not used but only the IP formulation.

The next theorem asserts the correctness of the algorithm.

```
procedure DATAFLOW-NO-CONFLICT-CHECK($Act, V, D, C, E, pre, post, wt$)
    $error = false$
    $unmarked = V$
    $IP$ = make IP formulation for $(Act \cup C, E)$
    while $unmarked \neq \varnothing \wedge error = false$ do
        $current$ = a variable from $unmarked$
        for $a_1 \in Act$ such that $v \in wt(a_1)$ do
            for $a_2 \in Act$ such that $v \in wt(a_2)$ and $a_1 \neq a_2$ do
                $IP_1 = IP$ && $(inedge_1(a_1)$=1) && $(a_1$=0) && $(inedge(a_2)$=1) && $(a_2 = 0)$
                $sol$ = solve $IP_1$
                if $sol$ is not null then // $CSP'$ is satisfiable, so conflict
                    Print "Race between activities $a_1$ and $a_2$ for variable $current$"
                    $error = true$
                end if
            end for
        end for
        $unmarked = unmarked \setminus \{ current \}$
    end while
    if $error = false$ then
        Print "The workflow data graph is dataflow-correct"
    end if
end procedure
```

Figure 4: Algorithm for checking absence of conflicts

Theorem 1. *Let $(Act, V, D, C, E, pre, post, wt)$ be a workflow data graph. Algorithm Dataflow-No-Conflict-Check finds no error if and only if there is no conflicting data.*

Proof 1. *In the proof, we use the following lemma: two activities are triggered by the same instance subgraph if and only if they are in parallel. This lemma follows immediately from the definition of instance subgraph and the definition of trigger.*

$\Rightarrow$: Since algorithm Dataflow-No-Conflict-Check finds no error, for each variable there is no partial instance subgraph triggering two activities that write the same variable. Therefore, there are no two parallel activities writing the same variable. Therefore there is no conflicting data.

$\Leftarrow$: Suppose the algorithm finds an instance subgraph that triggers two

activities a_1 and a_2 that both write variable $v \in V$. Then by definition there is a data conflict.

4.3. Algorithm for May-correctness

To check may-correctness of a workflow data graph and provide proper feedback in case of an incorrectness, we developed an algorithm (Figure 5) that diagnoses whether each activity a is may-executable (cf. Def. 4). For each activity a, the algorithm tries to find a data instance subgraph that triggers a and whose valuation satisfies the precondition of a. The data instance subgraph that is searched for is a solution to the combined IP and CSP model defined in Section 4.1 plus additional constraints that encode that a is triggered and that the precondition of a is satisfied.

Looking at the algorithm in more detail, it begins with the IP formulation of the workflow data graph (line 4 in Figure 5), which states the control flow constraints for data instance subgraphs. This IP formulation is combined with the CSP formulation of the workflow (line 5), which states the pre and postcondition constraints for the activities in data instance subgraphs. This combined IP and CSP model is used in the sequel of the algorithm for every data instance subgraph. Next, the algorithm performs a loop to check if all the activities are may-executable (line 6). The activity being processed in the loop is stored in variable *current* (line 7). To test whether activity *current* is may-executable, the combined IP and CSP model is extended with constraints that are true if the data instance subgraph triggers *current* and satisfies the precondition of *current* (line 8). If no solution exists, there is no such data instance subgraph for *current*, so *current* is not may-executable (line 10). If all activities are may-executable, the workflow data graph is may-correct (line 17).

The next theorem asserts that the algorithm is correct.

Theorem 2. *Let $(Act, V, D, C, E, pre, post, wt)$ be a workflow data graph. Algorithm May-Correctness-Check finds no error if and only if $(Act, V, D, C, E, pre, post, wt)$ is may-correct.*

Proof 2. $\Rightarrow$: *If algorithm May-Correctness-Check finds no error, for each activity a, a data instance subgraph exists, represented by the solution to the CSP model (line 9), that triggers a (line 8) and whose valuation, represented by the assignment of variables to the CSP variables, satisfies the precondition $pre(a)$ of a (line 8). Therefore, each activity is may-executable, and therefore the workflow data graph is may-correct.*

```
 1: procedure MAY-CORRECTNESS-CHECK($Act, V, D, C, E, pre, post, wt$)
 2:     $error = false$
 3:     $unmarked = Act$
 4:     $IP$ = make IP formulation for $(Act, E)$
 5:     $CSP = IP$ + CSP formulation for $(Act, pre, post)$
 6:     while $unmarked \neq \varnothing$ do
 7:         $current$ = an activity from $unmarked$
 8:         $CSP' = CSP$ && $inedge_1(current) = 1$ && $current = 0$ && $pre(current)$
 9:         $sol$ = solve $CSP'$
10:         if $sol$ is null then // $CSP'$ is unsatisfiable
11:             Print "Activity $current$ is not may-executable"
12:             $error = true$
13:         end if
14:         $unmarked = unmarked \setminus \{ current \}$
15:     end while
16:     if $error = false$ then
17:         Print "The workflow graph is may-correct"
18:     end if
19: end procedure
```

Figure 5: Algorithm for checking may-correctness

$\Leftarrow$: *If $(Act, V, D, C, E, pre, post, wt)$ is not may-correct, then there is an activity a that is not may-executable. By Definition 4, every data instance subgraph that triggers a can only have a valuation that violates the precondition of a. Therefore, the CSP model (line 8) has no solution (line 10).* □

The performance of the algorithm is discussed in Section 6.

4.4. Algorithm for Must-correctness

To verify must-correctness of a workflow data graph, we develop an algorithm (Figure 6) that diagnoses whether each activity in the workflow data graph is must-executable (cf. Def. 5). If an activity a is not must-executable, the algorithm provides a counter example in the form of a data instance subgraph that triggers a and whose valuation violates the precondition of a. If every activity is must-executable, the workflow data graph is must-correct by definition.

First, the combined IP and CSP model is created (line 4 and line 5) as defined in Section 4.1. As in algorithm May-Correctness-Check, each data

```
1: procedure Must-Correctness-Check($Act, V, D, C, E, pre, post, wt$)
2:     $error = false$
3:     $unmarked = Act$
4:     $IP$ = make IP formulation for $(Act, E)$
5:     $CSP = IP$ + CSP formulation for $(Act, pre, post)$
6:     while $unmarked \neq \varnothing$ do
7:         $current$ = an activity from $unmarked$
8:         $CSP'$=$CSP$ && $inedge_1(current) = 1$ && $current = 0$ && $\neg pre(current)$
9:         $sol$ = solve $CSP'$
10:        if $sol$ is not null then // $CSP'$ is satisfiable
11:            Print "Activity $current$ is not must-executable"
12:            $error = true$
13:        end if
14:        $unmarked = unmarked \setminus \{ current \}$
15:    end while
16:    if $error = false$ then
17:        Print "The workflow graph is must-correct"
18:    end if
19: end procedure
```

Figure 6: Algorithm for checking must-correctness

instance subgraph is a solution to this CSP model extended with additional constraints. Next, the algorithm performs a loop that processes each activity of the input workflow data graph (line 6). Variable *current* stores the activity processed in the loop. The algorithm extends for *current* the combined IP and CSP model with constraints that state that the data instance subgraph triggers *current* and that the precondition of *current* is violated. If a solution to this extended CSP model exists (line 10) then there is a data instance subgraph that triggers *current* and whose precondition violates *current*. Therefore, *current* is not must-executable (line 11). Otherwise, *current* is must-executable and the next activity is processed. If every activity is must-executable, the workflow data graph is must-correct (line 17).

We next prove that the algorithm is correct.

Theorem 3. *Let $(Act, V, D, C, E, pre, post, wt)$ be a workflow data graph. Algorithm Must-Correctness-Check finds no error if and only if $(Act, V, D, C, E, pre, post, wt)$ is must-correct.*

Proof 3. $\Rightarrow$: *If algorithm Must-Correctness-Check finds no error, for each activity a no data instance subgraph exists that triggers a (line 8) and whose valuation, represented by the assignment of variables to the CSP variables, satisfies the negation of the precondition* $pre(a)$ *of a (line 8). Equivalently, each data instance subgraph that triggers a has a valuation that satisfies the precondition* $pre(a)$*. Therefore, each activity is must-executable, and therefore the workflow data graph is must-correct.*

$\Leftarrow$: *If* $(Act, V, D, C, E, pre, post, wt)$ *is not must-correct, then there is an activity a that is not must-executable. By Definition 5, there exists a data instance subgraph that triggers a and that has a valuation that violates the precondition* $pre(a)$ *of a. Therefore, the CSP model (line 8) has a solution and the activity is not must-executable (line 11), so the algorithm finds an error.* □

4.5. Correcting errors

From the feedback provided by the algorithms, the workflow designer should opt for a repair action in order to make the workflow correct. Among the possible repair actions when the activity a is non-executable and is identified as responsible of the incorrectness, the designer can decide, for example:

- to relax the constraint that defines the precondition of a. This option should be taken when the problem is caused by the strictness of the precondition.
- to strengthen the postcondition of some activities in the instance subgraph which triggers a. This may be a solution when the problem is caused by postconditions that are too weak, allowing valuations of the variables that cause the violation of the precondition of a.
- to modify the domain of values of some variables in the dataflow to avoid the conflicting valuations.

4.6. Diagnosing the Motivating Example

This section presents the results of applying the algorithms to diagnose may and must-correctness of the workflow model in Figure 1. The workflow model has no missing and no conflicting data. As explained in Section 4.1, the CSP model with the pre and postconditions for each activity needs to be in SSA form. Table 6 shows the SSA form of the pre and postconditions of the activities with the new names of the variables. Notice that for the activity R two new constraints are introduced because the variable $others$ has two new names ($others1$ and $others2$) assigned in two different branches of a XOR split. These two new constraints unify the name of the variable to $others3$ after the join.

Table 6: Activities with their pre and postconditions in SSA form

Activity	Precondition and Postcondition
ECR	**pre:** $true$ **post:** $true$
SAP	**pre:** $true$ **post:** $true$
D	**pre:** $sponsorship1 > 0 \vee numPapers1 > 60$ **post:** $regFee1 * 0.1 \leq dinner1 \wedge dinner1 \leq regFee1 * 0.35$
L	**pre:** $sponsorship1 > 0 \vee numPapers1 > 60$ **post:** $regFee1 * 0.1 \leq 3 * lunch1 \wedge 3 * lunch1 \leq regFee1 * 0.35$
OS	**pre:** $sponsorship1 > 0 \vee numPapers1 > 60$ **post:** $others1 \leq 0.2 * regFee1 + 0.05 * sponsorship1 \wedge$ $others1 \geq 0.05 * regFee1 + 0.05 * sponsorship1$
O	**pre:** $sponsorship1 > 0 \vee numPapers1 > 60$ **post:** $others2 \leq 0.25 * regFee1 \wedge others2 \geq 0.05 * regFee1$
R	$OS = 1 \Rightarrow others3 = others1$ $O = 1 \Rightarrow others3 = others2$ **pre:** $3 * lunch1 + dinner1 + others3 < regFee1$ **post:** $numPapers1 * 1.8 \geq confAtt1$ $\wedge numPapers1 * 0.5 \leq confAtt1$
NGS	**pre:** $confAtt1 * (3 * lunch1 + dinner1 + others3) <$ $confAtt1 * regFee1 + sponsorship1$ **post:** $guestSpeaker1 \geq 0.2 * sponsorship1 \wedge$ $guestSpeaker1 \leq sponsorship1 + 0.1 * regFee1 * confAtt1$
IGS	**pre:** $confAtt1 * (3 * lunch1 + dinner1 + others3) <$ $confAtt1 * regFee1 + sponsorship1$ **post:** $guestSpeaker2 \geq 0.4 * sponsorship1 \wedge$ $guestSpeaker2 \leq sponsorship1$

Next, we diagnose the workflow model for may-correctness by applying the algorithm May-Correctness-Check to the workflow model in Figure 1 with the renamed variables and pre and postconditions as shown in Table 6. The algorithm determines that the workflow in Figure 1 is may-correct, since for activity a with precondition $pre(a)$ it is always possible to find at least one data instance subgraph that triggers a and whose valuation of the variables is within the determined finite domains listed in Table 2 such that $pre(a)$ is satisfied.

We diagnose for must-correctness by applying the algorithm in Figure 6 to the workflow model in Figure 1 in SSA form. The algorithm finds for instance an error when activity R is processed, since a data instance subgraph exists that assigns the values 200, 100, 30 and 30 to the variables $regFee$, $dinner$, $lunch$ and $others$ respectively, which makes the precondition of activity R unsatisfiable. Therefore the workflow is not must-correct.

5. Implementation

We have implemented a tool that realizes the verification algorithms of the previous section by extending the tool DiagFlow [1]. The new tool takes XPDL 1.0/2.0 models [20] as input and translates them into CSPs according to the formalization presented in this paper. For solving these CSPs, the tool uses the COMET^{TM} solver by Dynadec [27]. COMET^{TM} combines the methodologies used for constraint programming, linear and integer programming, constraint-based local search, and dynamic stochastic combinatorial optimization and offers a comprehensive software platform for solving complex combinatorial optimization problems.

In order to carry out our new approach, we need to define some details about the format of the input files. The original XPDL schema does not model any semantic information of the workflow, so no pre and postconditions are considered. Therefore we propose the following extension of the XPDL schema, which conforms to the XPDL standard [20]:

1. input data of the activity (read operations)
2. output data of the activity (write operations)
3. precondition: constraint (or constraints) over the input data
4. postcondition: constraint (or constraints) over the input and output data

In order to include these data, the XPDL input file must contain several ExtendedAttributes to extend the XPDL node *Activity* [20]:

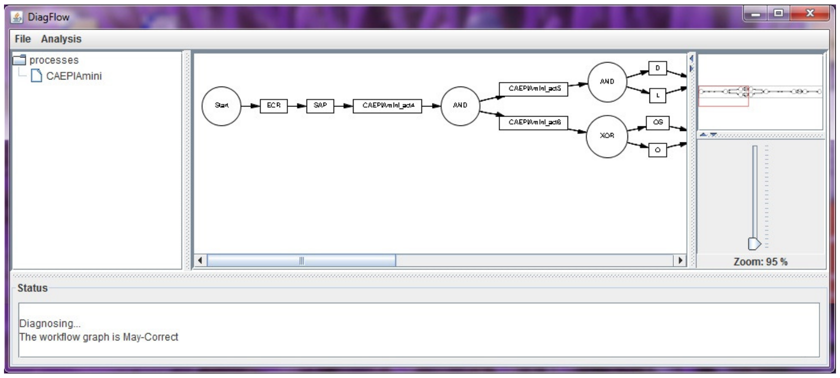
```
<xpdl:Activity ...>
```

```
  ...
  <xpdl:  ExtendedAttribute  Name="InputVariables">
    <variable>...</variable>
    <variable>...</variable>
    ...
  </xpdl:  ExtendedAttribute>
  <xpdl:  ExtendedAttribute  Name="OutputVariables">
    <variable>...</variable>
    <variable>...</variable>
    ...
  </xpdl:  ExtendedAttribute>
  <xpdl:  ExtendedAttribute  Name="Precondition"  Value="..."/>
  <xpdl:  ExtendedAttribute  Name="Postcondition"  Value="..."/>
  ...
</xpdl:Activity>
```

The variables that are referenced in the preconditions and postconditions of the activities use finite domains to define the ranges of values they can take. To define those domains in the XPDL file, it is necessary to add ExtendedAttributes within the node *WorkflowProcess*:

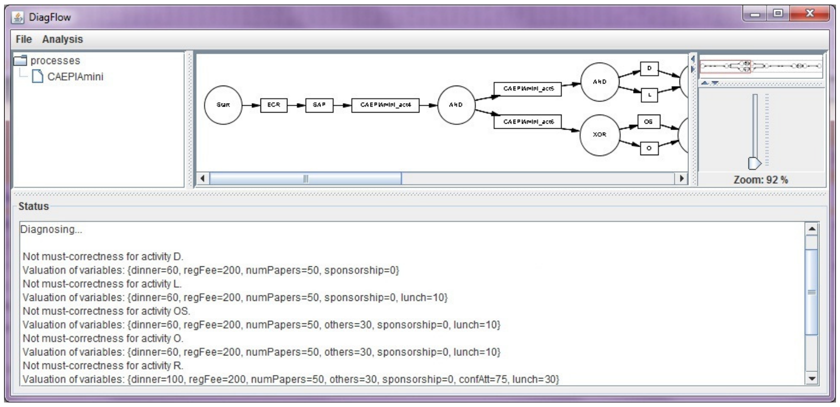
```
<xpdl:WorkflowProcess ...>
  ...
  <xpdl:ExtendedAttributes>
    <xpdl:ExtendedAttribute  Name="Domain">
      <variable>...</variable>
      <initialValue>...</initialValue>
      <finalValue>...</finalValue>
    </xpdl:  ExtendedAttribute>
    ...
  </xpdl:ExtendedAttributes>
  ...
</xpdl:WorkflowProcess>
```

Figure 7 shows a screenshot of the DiagFlow tool during the diagnosis of the may-correctness of the example discussed in this paper (Figure 1). According to the pre and postconditions of its activities and the domains of the variables, the DiagFlow tool determines it is may-correct. On the other hand, the workflow is not must-correct as can be seen in the screenshot in Figure 8.



```
Diagnosing...
The workflow graph is May-Correct
```

Figure 7: **Screenshot of DiagFlow indicating may-correctness**



```
Diagnosing...

Not must-correctness for activity D.
Valuation of variables: {dinner=60, regFee=200, numPapers=50, sponsorship=0}
Not must-correctness for activity L.
Valuation of variables: {dinner=60, regFee=200, numPapers=50, sponsorship=0, lunch=10}
Not must-correctness for activity OS.
Valuation of variables: {dinner=60, regFee=200, numPapers=50, others=30, sponsorship=0, lunch=10}
Not must-correctness for activity O.
Valuation of variables: {dinner=60, regFee=200, numPapers=50, others=30, sponsorship=0, lunch=10}
Not must-correctness for activity R.
Valuation of variables: {dinner=100, regFee=200, numPapers=50, others=30, sponsorship=0, confAtt=75, lunch=30}
```

Figure 8: **Screenshot of DiagFlow indicating no must-correctness**

6. Empirical Evaluation

The worst-case complexity of solving CSPs is high. Therefore we wish to empirically evaluate the performance of the developed algorithms. This way, we can assess whether in practice the time it takes to diagnose workflow data models with the algorithms is acceptable.

6.1. Experimental Design

The DiagFlow tool receives XPDL files as inputs, with the extensions explained in the previous section, and provides options to verify the different kinds of correctness.

The primary purpose of the experimental evaluation is to determine the execution time from start to completion of a correctness checking process over workflow data graphs with different control flows and dataflows.

With the aim of performing the execution time measurements, the algorithms are executed over different extended XPDL files, getting test cases with different number of activities, control nodes and data.

The test cases are measured using a Windows 7 machine, with an Intel Core I7 processor, 3.4GHz and 8.0GB RAM.

6.2. Performance results

In this subsection, the execution time of the correctness checking algorithms is measured.

Solving CSPs takes exponential time due to the dependency of their complexity on the number of values each variable can take [28][29][30]. However, in practice, since CSP solvers run very fast, this does not limit the applicability of our approach, as we will show next.

Since both algorithms for checking may and must-correctness have a similar structure, a while-loop which processes every activity by solving a combined IP and CSP model for the activity, they also have the same time complexity. Since the algorithms check all the activities in the workflow to find the ones which are not may or must-executable, the performance results depend on the size of the workflow data graph being analyzed.

Figure 9 shows the performance results of the may-correctness-check algorithm in terms of the workflow size (number of activities), including the range of the Control-flow Complexity metric (CFC, [31]) in brackets. That metric is used to quantify the presence of control nodes in a workflow W, defined as follows:

$$CFC(W) = \sum_{n \in S_{XOR}} CFC_{S_{XOR}}(n) + \sum_{n \in S_{AND}} CFC_{S_{AND}}(n)$$

where $CFC_{S_{XOR}}(n) = |outedge(n)|$ and $CFC_{S_{AND}}(n) = 1$. The results obtained for both the may-correctness-check and the must-correctness-check algorithms are shown in the same chart since they present the same execution time. For both algorithms, the execution time appears to scale linearly with respect to the number of activities checked by the algorithm. Therefore, the time it takes to diagnose even large workflow data models with the algorithms is acceptable.

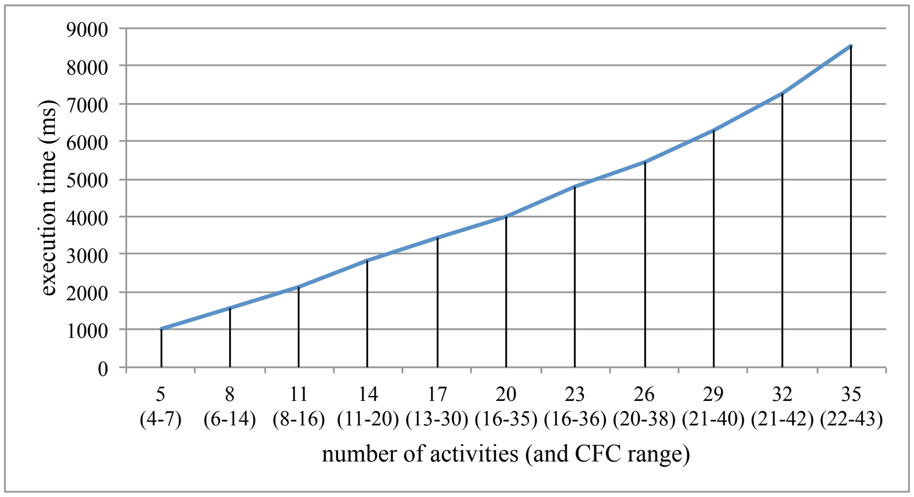


Figure 9: **Performance results**

7. Related work

As stated in the introduction, most of the previous works in the literature [1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13] only take into account the control flow to check the correctness of workflows. Nevertheless, to guarantee the correctness of a workflow model in practice, it is necessary to cover other perspectives as well, in particular the dataflow including the effects that the execution of the different activities has on the data (postconditions) as well as the conditions established at each activity which should be satisfied to make that execution possible (preconditions).

Only recently, researchers started considering verification of workflow models with dataflows. Sun et al. [14] define a dataflow perspective on workflows and identify several types of errors, based on earlier work by Sadiq et al [8]. Similarly, Trčka et al. [15] define some dataflow anti-patterns ir order to identify this kind of errors between data. The workflow models are abstract: they specify variables read and written per activity, but do not specify the effect of each activity by means of pre and postconditions.

Sidorova et al. [17] verify correctness for a subclass of Petri nets, workflow nets, extended with data operations. The workflow models they consider are abstract and need to be refined to be executable. The verification procedure checks whether the abstract workflow models can be refined into correct, concrete workflow models that have no deadlocks. The check sometimes results in a "yes, if ..." answer, indicating that only under certain conditions a correct refinement exists. Along the same lines, the work by van der Aalst et al. [18] performs the verification of the correctness of configurable process models, taking into account the data dependencies between activities.

Main difference with our approach is that their workflow models do not use pre and postconditions for the activities, not even the refined workflow models [17]. Consequently, according to their approach the example workflow in Figure 1 is correct, because it is always possible to reach a final state from any reachable state if pre and postconditions are abstracted from. On the other hand, our approach detects an error in the workflow, as explained in Section 4.6.

Borrego et al. [32] consider post-mortem diagnosis of business processes in which a particular execution of a business process is checked against compliance constraints. Also the execution of individual activities can be diagnosed. In this paper, we study diagnosis of business processes at design-time. Rather than considering one specific execution, we have to take into account all possible executions.

Weber et al. [16] consider verification of semantic business processes, in which activities are annotated with pre and postconditions. They detect conflicts between preconditions and postconditions of parallel activities and next study the reachability and executability of the activities, but only if the activities are conflict free. In their contribution, pre and postconditions are considered as CNF formulas with only boolean variables. Therefore, the approach by Weber et al. [16] cannot diagnose the correctness of workflows whose activities count on pre and postconditions involving other kinds of data (i.e., example in Figure 1). Moreover, they focus on analyzing the complexity of several verification tasks for semantic process models and do not focus on diagnosis of errors. Moreover, workflow graphs must be analyzed in two fixed sequential steps, while the approach we defined does not have this restriction. They study the general complexity of verifying semantic business process and present an algorithm for verifying whether an activity is executable. However, the algorithm does not consider the possible valuations of the data to determine an inconsistency. So, just as Sidorova et al. [17], the approach by Weber et al. [16] would diagnose example in Figure 1 as sound, without considering that under certain conditions (i.e., some valuations of the variables) the process may get stuck.

There exist modeling languages like Colored Petri nets [33] that combine control flow and data. However, these modeling languages are not targeted towards a specific application domain, while the approach in this paper is specific to workflows. Consequently, the notion of must and may correctness proposed in this paper is specific to workflows, but not used for these general purpose languages. For instance, using CPN Tools a state space graph of Colored Petri net can be constructed, but the resulting report only provides general statistics.

To the best of our knowledge, this paper presents the first verification approach for executable workflow models that integrates both process and numerical data verification. The approach can detect errors not detectable with other approaches. The approach builds on research done in the field of Constraint Programming and workflow verification, combining the best of both worlds to deliver advanced yet efficient verification and diagnosis of complex workflow models with dataflows.

8. Conclusions

To engineer workflow models with dataflows in a dependable way, diagnosis of correctness is of utmost importance. To that end, we have proposed workflow data graphs as formalization of semantic workflow models together with two correctness notions, may and must-correctness, that can be verified for workflow data graphs. Workflow data graphs model semantic workflows by extending workflow graphs with pre and postconditions for the activities. We also proposed two correctness notions, may and must-correctness, for workflow data graphs.

Next, we have presented a diagnosis approach to check may and must-correctness, which consists of several phases. First, preprocessing is applied to detect basic data anomalies. Then, the workflow data graph is translated into an IP formulation that models the executable instances, and into a CSP formulation that models the data states acceptable according to the pre and postconditions of the activities. The combined IP and CSP model can be efficiently solved using Constraint Programming techniques. In case of an error, feedback is provided in the form of an error path showing where the workflow gets stuck under certain conditions over the dataflow. Such feedback provides valid information for the workflow designer to fix future errors before the workflow is deployed. The approach is complete, so it always generates accurate feedback in case of an error.

The approach has been implemented by extending the DiagFlow tool [1]. The tool diagnoses workflow models in an extended XPDL format. The XPDL extension is needed to store the semantic information of each workflow, adding the dataflow with the pre and postconditions in the activities. Performance evaluation of the tool shows that the algorithms scale well for large workflow models with dataflows, despite the high worst-case complexity of solving constraints satisfaction programs.

As future work, we plan to extend the workflow data graph model with *OR* gateways. Likewise, we would also like to offer additional feedback to the end user in case of a violation, making easier the job of fixing the problem which causes the abnormal behavior. Another interesting extension is to consider stochastic behavior or timing behavior of activities, to improve the accuracy of the analysis.

References

[1] R. Eshuis, A. Kumar, An integer programming based approach for verification and diagnosis of workflows, Data Knowl. Eng. 69 (2010) 816–835.

[2] S. Jablonski, C. Bussler, Workflow management - modeling concepts, architecture and implementation, International Thomson, 1996.

[3] W. van der Aalst, A. Hirnschhall, H. M. W. Verbeek, An alternative way to analyze workflow graphs, in: Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE02), volume 2348 of Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 535–552.

[4] H. H. Bi, J. L. Zhao, Process logic for verifying the correctness of business process models, in: International Conference on Information Systems ICIS, Association for Information Systems, 2004, pp. 91–100.

[5] R. Eshuis, R. Wieringa, Verification support for workflow design with uml activity graphs, in: Proceedings of the 24th International Conference on Software Engineering, ACM, 2002, pp. 166–176.

[6] C. Karamanolis, D. Giannakopoulou, J. Magee, S. M. Wheater, Model checking of workflow schemas, in: Proceedings of IEEE EDOC, 2000, pp. 170–179.

[7] H. Lin, Z. Zhao, H. Li, Z. Chen, A novel graph reduction algorithm to identify structural conflicts, in: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02) - Volume 9, HICSS '02, IEEE Computer Society, Washington, DC, USA, 2002.

[8] S. W. Sadiq, M. E. Orlowska, W. Sadiq, C. Foulger, Data flow and validation in workflow modelling, in: ADC, volume 27 of *CRPIT*, Australian Computer Society, 2004, pp. 207–214.

[9] F. Touré, K. Baïna, K. Benali, An efficient algorithm for workflow graph structural verification, in: Proceedings of the OTM 2008 Confederated International Conferences. Part I on On the Move to Meaningful Internet Systems, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 392–408.

[10] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets with reset arcs, T. Petri Nets and Other Models of Concurrency 3 (2009) 50–70.

[11] M. T. Wynn, H. M. W. Verbeek, W. M. P. v. d. Aalst, A. t. Hofstede, D. Edmond, Business process verification - finally a reality!, Business Process Management Journal 15 (2009) 74–92.

[12] W. Van Der Aalst, K. Van Hee, A. Ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, M. Wynn, Soundness of workflow nets: Classification, decidability, and analysis, Formal Aspects of Computing 23 (2011) 333–363. Cited By (since 1996) 20.

[13] S. Patig, M. Stolz, A pattern-based approach for the verification of business process descriptions, Inf. Softw. Technol. 55 (2013) 58–87.

[14] S. X. Sun, J. L. Zhao, J. F. Nunamaker, O. R. L. Sheng, Formulating the data-flow perspective for business process management, Information Systems Research 17 (2006) pp. 374–391.

[15] N. Trčka, W. M. Aalst, N. Sidorova, Data-flow anti-patterns: Discovering data-flow errors in workflows, in: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, CAiSE '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 425–439.

[16] I. Weber, J. Hoffmann, J. Mendling, Beyond soundness: on the verification of semantic business process models, Distributed and Parallel Databases 27 (2010) pp. 271–343.

[17] N. Sidorova, C. Stahl, N. Trcka, Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible, Information Systems 36 (2011) pp. 1026–1043.

[18] W. M. P. van der Aalst, N. Lohmann, M. La Rosa, Ensuring correctness during process configuration via partner synthesis, Inf. Syst. 37 (2012) 574–592.

[19] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006.

[20] WFMC, Workflow Management Coalition Workflow Standard: Process Definition Interface – XML Process Definition Language, Technical Report WFMC-TC-1025, Workflow Management Coalition, 2005.

[21] OMG, Object Management Group, Business Process Model and Notation (BPMN), Version 2.0, OMG Standard, 2011.

[22] W. Sadiq, M. E. Orlowska, Analyzing process models using graph reduction techniques, Information Systems 25 (2000) pp. 117–134.

[23] R. Eshuis, A. Kumar, An Integer Programming based Approach for Diagnosing Workflows, Technical Report, Beta Working Paper Series, WP 264, Eindhoven University of Technology, 2008.

[24] R. Eshuis, R. Wieringa, Tool support for verifying uml activity diagrams, IEEE Transactions on Software Engineering 30 (2004) pp. 437–447.

[25] B. Alpern, M. N. Wegman, F. K. Zadeck, Detecting equality of variables in programs, in: Proceedings of the 15th ACM SIGPLAN-SIGACT, Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 1988, pp. 1–11.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (1991) pp. 451–490.

[27] Dynamic Decision Technologies, Dynadec web page, http://dynadec.com/, 2011.

[28] P. Traxler, The time complexity of constraint satisfaction, in: Proceedings of the 3rd international conference on Parameterized and exact computation, IWPEC'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 190–201.

[29] R. Dechter, Constraint networks, Encyclopedia of Artificial Intelligence, Second Edition (1992) pp. 276–285.

[30] V. Kumar, Algorithms for constraint satisfaction problems: A survey, AI Magazine 13 (1992) pp. 32–44.

[31] J. Cardoso, Evaluating the process control-flow complexity measure, in: Web Services, ICWS 2005. Proceedings. 2005 IEEE International Conference, volume 2, pp. 803–804.

[32] D. Borrego, R. M. Gasca, M. T. Gómez-López, L. Parody, Contract-based diagnosis for business process instances using business compliance rules, in: Proceedings of the 21st International Workshop on Principles of Diagnosis (DX-10), 2010, pp. 169–176.

[33] K. Jensen, L. M. Kristensen, L. Wells, Coloured petri nets and cpn tools for modelling and validation of concurrent systems, STTT 9 (2007) 213–254.