INDIAN INSTITUTE OF MANAGEMENT CALCUTTA

WORKING PAPER SERIES

WPS No. 621/ February 2008

Workflow Graph Verification Using Graph Search Techniques

by

Ambuj Mahanti and Sinnakkrishnan Perumal

IIM Calcutta, Diamond Harbour Road, Joka P.O., Kolkata 700104, India.

Workflow Graph Verification Using Graph Search Techniques

Mahanti Ambuj and Sinnakkrishnan Perumal

Indian Institute of Management Calcutta, Joka, D. H. Road

Kolkata 700104, India, email: {am,krish}@iimcal.ac.in

ABSTRACT

Workflow management systems provide a flexible way of implementing business processes. Structural conflicts such as deadlock and lack of synchronization are commonly occurring errors in workflow processes. Workflows with structural conflicts may lead to error-prone and undesirable results in business processes, which may in turn affect customer satisfaction, employee productivity, and integrity of data, and may also cause legal issues. Workflow verification is meant for detecting structural conflicts in workflow processes. Workflow management systems do not have the functionality for workflow verification except through simulation which does not detect the error completely. In this paper, we present a simple workflow verification method based on the principle of depth-first search. This method is meant for verifying acyclic workflow graphs. We illustrate our method with detailed workouts using business examples. We also present a detailed theoretical analysis and empirical evaluation of the proposed method. We compare our method with the well-known graph reduction based method. We observe that our method provides significantly better results. Workflow verification is crucial as workflows with structural conflicts when deployed will cause malfunctioning of workflow management systems. Moreover, our method has worst-case time complexity of $O(E^2)$ as against $O((E+N)^2.N^2)$ for the graph reduction method. We believe that our method will make the workflow verification task simpler and efficient.

Keywords: Workflow verification, Workflow management, Graph search techniques, Depth-first search.

1 INTRODUCTION

Workflow management corresponds to "everything from modeling processes up to synchronizing the activities of information systems and humans that perform the processes" (Georgakopolous et al. 1995). Information required for modelling and executing a business process in a workflow management system has different perspectives such as functional, process, organizational, informational and operational as given in (van der Aalst et al. 2003b). Here, functional perspective deals with the actual work corresponding to each task of the workflow, organizational perspective provides the mapping between the tasks and the roles/organizational groups who can execute the tasks, informational perspective provides the flow of business data and execution data in the workflow, and operational perspective presents various elementary operations (say, applications executing scripts, or human resources following procedures). (van der Aalst and Jablonski 2000) details various perspectives of workflows as process, organization, information, operation and integration, where the last perspective depicts the dependencies between the first four perspectives. Process perspective provides the structure for the workflow. Through this structure, execution dependencies between various tasks of the workflow are depicted. Rules, constraints, and/or graphical constructs are used to depict the execution dependencies between the tasks in a workflow in commercial workflow management systems (Georgakopolous et al. 1995). This paper deals with the workflow verification, i.e., detection of structural errors in the process perspective of the workflows. If a workflow containing structural conflicts is executed, it may lead to undesirable effects such as business loss, negative brand image, increased overload of employees, and customer frustration. Hence, structural conflicts have to be identified and eliminated before the workflows are deployed in the business environment.

Workflows are specified using a workflow specification language (Georgakopolous et al. 1995). Some of the workflow specification languages are Workflow Graphs (as given in (Sadiq and Orlowska 2000), (Choi and Zhao 2002) and (Bi and Zhao 2004)), WF Nets (as given in (van der Aalst 1998)), Unified Modeling Language (UML) Activity Diagrams (as given in (Dumas and ter Hofstede 2001)), Object Coordination Nets (OCoNs) (as given in (Wirtz et al. 2001)), Workflow Evolution Approach (as given in (Casati et al. 1998)), Adjacency Matrix (as given in (Choi and Zhao 2002)), Metagraphs (as given in (Basu and Blanning 2000)), YAWL (as given in (van der Aalst and ter Hofstede 2005)), and Event-driven Process Chain (as given in (van der Aalst 1999)). Also, commercial workflow management systems such as Verve, Visual Workflo, Forté, MQSeries/Workflow, Staffware, COSA, InConcert, Changeengine, I-Flow, and SAP/R3, and laboratory workflow management systems such as Meteor and Mobile use various workflow specification languages (van der Aalst et al. 2003a). A workflow specification language is useful in communicating the various aspects of a workflow to designers, users, knowledge engineers, managers and technical personnel.

A workflow is executed for a case. Various examples of a case are an insurance claim, a student admission, and a loan application. Only a subset of tasks of a workflow process are executed for a case based on the customer data, environment data, execution data and business domain data. This subset of tasks together with the control flow between them is called an instance. So far, most Workflow Management Systems (WfMSs) provide only simulation tools for validating workflow models using the trial-and-error method(Bi and Zhao 2004). An instance has to be specified in a simulation tool to verify it. However, a typical large workflow would have many instances, and it will be tedious to identify and verify each of them.

Workflow verification is a computationally complex problem, and the method adopted to solve this should correspond to the workflow specification language used. Various methods for workflow verification are available for various workflow specification languages such as method given in (Verbeek et al. 2001) for verifying WF Nets, method given in (Verbeek et al. 2007) for verifying YAWL, method given in (Eshuis and Wieringa 2002) for verifying UML Activity Diagrams, methods given in (Sadiq and Orlowska 2000), (Choi and Zhao 2002), (Choi and Zhao 2005), (Lin et al. 2002), (Perumal and Mahanti 2007) and (Bi and Zhao 2004) for verifying workflow graphs, and methods given in (van Dongen et al. 2005) and (van der Aalst 1999) for verifying Event-driven Process Chains. Certain workflow languages are such that any process modeled in these languages will be structurally correct. This is because these workflow languages follow a specific structure to ensure the structural correctness of the resulting processes. IBM MQSeries Workflow is one such example (van Dongen et al. 2005). However, modeling a process in such a language will be complex and the resulting process will not be intuitive.

Two well known methods available in literature for workflow graph verification are: (a) Graph reduction techniques (Lin et al. 2002; Sadiq and Orlowska 1999, 2000), and (b) Conversion of Workflow graphs to WF nets (van der Aalst et al. 2002) and then identifying the structural conflicts through well established Petri-net theory(Verbeek and van der Aalst 2000). However, the graph reduction technique (as given in (Lin et al. 2002)) is quite complicated, and has worst case time complexity as $O((E+N)^2.N^2))$ - where E is the number of edges and N is the number of nodes in the workflow graph (O denotes the "Big-O notation"). Value of E could range between O(N) and O(N^2). Workflow-nets, on the other hand, lose the intuitive graphical understanding associated with the Workflow graphs. In addition, Workflow-net algorithm has the worst case time complexity as O(E^3).

This paper presents an algorithm for workflow verification called Mahanti-Sinnakkrishnan (MS) algorithm. MS algorithm is simple and it is explained through business examples. Being based on the simple and efficient depth first search principle, MS algorithm is able to have the worst case time complexity as O(E^2). Compared to the existing algorithms in the literature, our algorithm is simpler to understand, easier to process and, more importantly, easier to debug during implementation. Thus, it is felt that MS algorithm will be quite useful for the workflow verification process.

Detailed theoretical analysis on the correctness of MS algorithm is presented with adequate illustrations. Also, in this paper, we present a comparative study based on empirical evaluation of the performance of MS algorithm with the graph reduction based method. This was tested using various randomly generated workflow graphs based on varying set of factors, and it was found that MS algorithm provides significantly better results than the graph reduction algorithm.

Advantage of MS algorithm are: (a) it is simple, (b) it takes lesser time to execute compared to other methods, (c) it is based on well known graph search methods, and (d) the original graph structure is not changed during verification. Although MS algorithm is for workflow graph representation, it can be extended for UML activity graphs as well.

This paper is organized as follows. Section 2 introduces the workflow graph representation used in this paper. Section 3 presents further details about the graph reduction method. Section 4 explains MS algorithm for workflow graph verification with detailed description, workout using a business example, proofs and trace of the algorithm for various workflow graphs. Finally, section 5 presents the implementation of MS algorithm and graph reduction algorithm, comparison of performance of these algorithms and analysis of the results.

2 WORKFLOW GRAPH REPRESENTATION

A simple directed graph representation is used to represent workflow graphs that is comprised of a set of nodes called V (we use N to denote the total number of nodes in the workflow graph) and a set of edges called E. Nodes are of two types, condition (denoted as C) and task nodes (denoted as T). Condition nodes can be further divided into OR-split and OR-merge nodes. Similarly, task nodes can be further divided into AND-split, AND-merge and sequence nodes. AND-split and OR-split nodes are together called as split nodes. Similarly, AND-merge and OR-merge nodes are together called as merge nodes or join nodes. Sequence nodes have one incoming edge and one outgoing edge. Split nodes have one incoming edge, and more than one outgoing edge. Merge nodes have one outgoing edge, and more than one incoming edge. Without loss of generality, we can assume that workflow graphs can have only one start node and only one end node as given for the definition of WF nets as in (van der Aalst 1998). Start node and end node of the workflow are special nodes in that start node does not have any incoming edge, and end node does not have any outgoing edge.

Task nodes (i.e., sequence, AND-split and AND-join nodes) are used to represent various tasks of the workflow. Apart from that, an AND-split node triggers a set of concurrent paths from it. Hence, if an AND-split node is executed for an instance, all the concurrent paths emanating from it should also be executed. An AND-merge node is used to merge such concurrent paths. An OR-split node is used to create a set of mutually exclusive alternative paths.

So, if an OR-split node is executed for an instance, exactly one of the paths emerging from it will be executed. An OR-merge node is used to merge such mutually exclusive alternative paths in the workflow graph. “OR” in the OR-split/OR-join nodes is a misnomer, as they correspond to “exactly one” of the paths that are splitting/merging from/in a node. We use these terms as the workflow literature uses similar terms. However, in recent workflow literature as in (Workflow Management Coalition 2005), XOR-split and XOR-join terms are used for nodes that split/merge exactly one of the mutually exclusive alternative paths, and OR-split and OR-join terms are used for nodes that split/merge more than one alternative path.

For executing an instance of a workflow process, the workflow graph corresponding to the workflow process is traversed from its start node. This start node is activated and executed. Further on, for any OR-split node that is activated and executed, exactly one of its outgoing paths is chosen and the child node through that path is activated and executed. For any other type of node that is activated and executed, all child nodes through all its outgoing edges are activated and executed. If an OR-merge node is activated more than once for an instance of the workflow process, then it leads to Lack of Synchronization structural conflict. This structural conflict leads to undesirable multiple executions of the nodes following the OR-merge node. Similarly, if an AND-merge node is not activated through all its incoming edges for an instance of the workflow process, then it leads to Deadlock structural conflict. This structural conflict leads to infinite waiting for the AND-merge node to get activated by its other incoming edges. Deadlock and Lack of Synchronization are the major structural conflicts verified by workflow verification algorithms. There are other structural conflicts such as dangling nodes, etc., but these are considered as minor structural conflicts and very easily solvable problems in the literature.

Subgraph comprising all the nodes and edges that are activated and executed for an instance of a workflow process is called an instance subgraph.

For representing the workflow graphs, we use process language constructs as given in (Sadiq and Orlowska 2000) based on constructs in (Workflow Management Coalition 1996). This is similar to the generic modeling concepts as given in WfMC standard Workflow Process Definition Interface XPDL report as in (Workflow Management Coalition 2002). It could be noted that more advanced constructs are used in the recent version of the WfMC standard Workflow Process Definition Interface XPDL report as in (Workflow Management Coalition 2005).

3 MORE ABOUT GRAPH REDUCTION METHOD

3.1 Graph Reduction Algorithm - I

Graph reduction based verification of workflow graphs was first given in (Sadiq and Orlowska 1999) and (Sadiq and Orlowska 2000). We call this algorithm as Graph Reduction Algorithm - I. This method used a simple representation of workflows using sequence, AND-split, AND-join, OR-split and OR-join nodes. Graph reduction method uses a set of graph reduction rules which are applied repeatedly on the workflow graph. If the given workflow graph eventually reduces to an empty graph, then the workflow graph is a correct graph. However, if none of the graph reduction rules can reduce the workflow graph any further, then the algorithm stops and the given workflow graph has some structural conflict. Graph reduction rules have a property that they do not introduce or remove structural conflicts, but they only transform a workflow graph from one structure to another structure.

Figure 1 shows application of various graph reduction rules, and these rules are explained below.

R1: Adjacent Reduction Rule:

This rule reduces four different patterns, and comprises four different sub-rules as follows:

(a) If a node is a terminal node and is connected to the rest of the workflow graph through a single edge, then the node and the edge can be removed from the workflow graph.

(b) If a node is a sequence node, then the source node of the outgoing edge from it is changed to its parent node. After this, the node and its incoming edge are deleted from the workflow graph.

If the above two patterns cannot be applied on a node, then the node has to be either a split node or a merge node.

(c) If the node is a split node and its parent node is also a split node of the same type (type refers to AND or OR), then source node of all its outgoing edges is changed as the parent node. After this, the node and its incoming edge are deleted.

(d) If the node is a merge node and its child node is also a merge node of the same type (type refers to AND or OR), then destination node of all its incoming edges is changed as the child node. After this, the node and its outgoing edge are deleted.

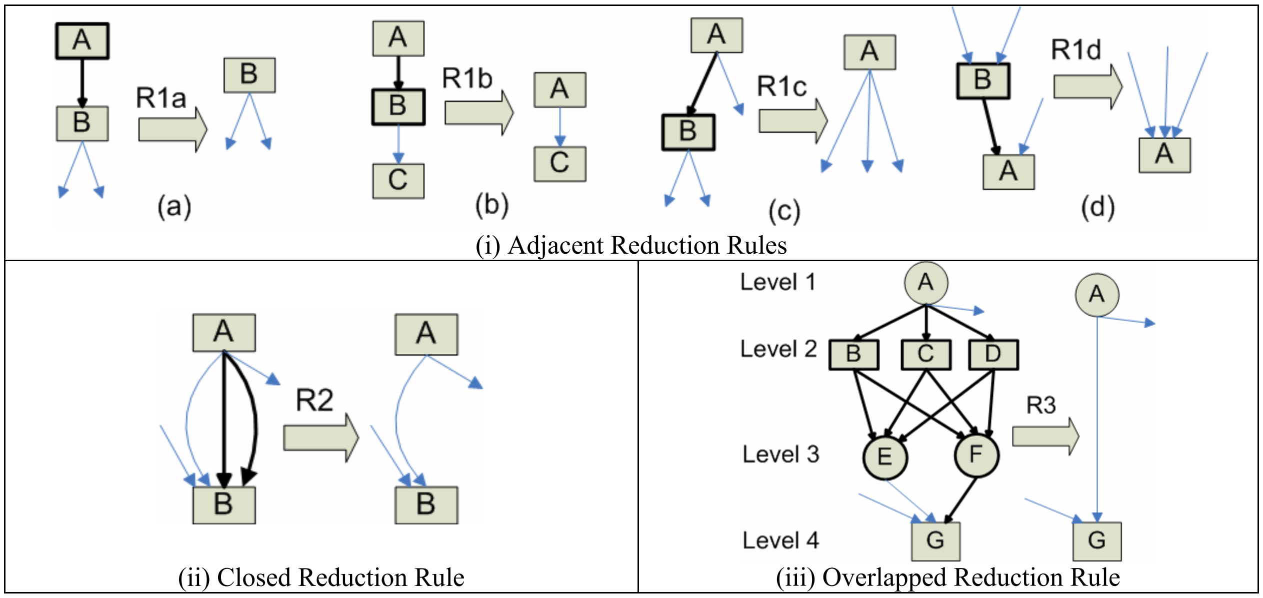
R2: Closed Reduction Rule:

During graph reduction, it is possible that some nodes may end up having more than one direct edge between them. For such pairs of nodes, additional edges between them are removed.

R3: Overlapped Reduction Rule:

Overlapped reduction rule reduces an overlapped structure in the workflow graph, which has four layers. First layer has an OR-split node. Second layer has a set of AND-split nodes which have only the first layer OR-split node as the parent node. Fourth layer has an AND-merge node. Third layer has a set of OR-merge nodes each of which has all the second layer AND-split nodes as its parents, and the fourth layer AND-merge node as its only child node.

Figure 1: Graph Reduction rules are shown individually


A
B
R1a
(a)
C
R1b
(b)
R1c
(c)
R1d
(d)
(i) Adjacent Reduction Rules
R2
(ii) Closed Reduction Rule
Level 1
Level 2
Level 3
Level 4
D
E
F
G
R3
(iii) Overlapped Reduction Rule

This algorithm has complexity as O(N^2). This algorithm does not reduce all correct workflow graphs into an empty graph as intended. This is proved through counterexamples in (Lin et al. 2002) and (van der Aalst et al. 2002).

3.2 Graph Reduction Algorithm II

To correct the problems in Graph Reduction Algorithm I, a new set of graph reduction rules and a new algorithm was introduced in (Lin et al. 2002). We call this algorithm as Graph Reduction Algorithm II. Thus, this algorithm was able to reduce all correct workflow graphs into an empty graph, and wrong workflow graphs will not be reduced to empty graph by applying these rules. The new rules are complex, and it is difficult to comprehend visually. This algorithm has worst-case time complexity of $O((N+E)^2.N^2)$. This algorithm is only for verifying acyclic workflow graphs.

4 MAHANTI-SINNAKKRISHNAN (MS) ALGORITHM FOR WORKFLOW GRAPH VERIFICATION

MS algorithm is given in Figure 2. This algorithm uses graph search techniques like AO* and Depth-First Search (more details about these graph search techniques can be found in (Nilsson 1982) and (Mahanti and Bagchi 1985)). In this algorithm, for the sake of uniformity, a sequence node is considered an AND-split node with a single child node.

Figure 2: Mahanti-Sinnakkrishnan (MS) algorithm for workflow verification

```
Algorithm Verify_Workflow(Graph G)
  Initialization:
    Initialize a stack Z containing only the start node.
    Initialize a stack called OR_Split_Stack to NIL.
    Initialize the explicit graph G' by installing the start node in it. Label start node as
not expanded in G'.
  Do
    Call the procedure Create_Instance_Subgraph(G, G', Z, OR_Split_Stack).
    Call the procedure Verify_Instance_Subgraph(G').
    Call the procedure Prepare_for_Next_Instance(G, G', Z, OR_Split_Stack).
  While OR_Split_Stack is not empty
Procedure Create_Instance_Subgraph(G, G', Z, OR_Split_Stack)
  While Z is not empty do
    Pop the top node from Z. Let this node be called "q".
    If q is not already expanded in G' then
      In G', label q as expanded
      If q is OR-split node then
        Install the first child node of q in G' if it is not already present in G'.
        Install the edge to this child node of q in G' and mark this edge.
        Push this child node to the top of Z.
        Push q to OR_Split_Stack.
      Else
        Install all the child nodes of q in G' if they are not already present in G'.
        Install the edges to these child nodes of q in G' and mark these edges.
        Push these child nodes to the top of Z in, say, left-to-right order such that the
right-most child node is on the top of Z.
  End while
End Procedure
Procedure Verify_Instance_Subgraph(G')
  Label all nodes of G' as "not visited".
  Set VisitCount to zero for all AND-join nodes in G'.
  Initialize a stack Y containing only the start node.
  While Y is not empty do
    Pop the top node q from Y
    If q is not visited already then
      If q is an OR-split node then
        Push the marked child node of q to the top of Y.
      Else
        Push the marked child nodes of q to the top of Y in, say, left-to-right order such
that the right-most child node is on the top of Y.
    If q is an already visited OR-join node then
      Report "Structural Conflict: Lack of Synchro-nization" Error and Exit
    If q is an AND-join node then
      Increment the VisitCount of q.
    Label q as "visited"
  End while
  If number of parents(i.e., MergeCount) did not match with the VisitCount for any
visited AND-merge node in G' then
    Report "Structural Conflict Error: Deadlock" and Exit.
End Procedure
Procedure Prepare_for_Next_Instance(G, G', Z, OR_Split_Stack)
  While all child nodes of the top node of OR_Split_Stack have already been
considered for creating instance sub-graph
    Pop the top node from OR_Split_Stack.
  If OR_Split_Stack is not empty then
    For the top node p of OR_Split_Stack, generate the next child node.
    Install this generated child node in G' if it is not already present in G'.
    Install the edge from p to this child node in G'.
    Shift the marking below p to the edge connecting this child node.
    Push this child node to the top of Z.
End Procedure
```

An instance subgraph is defined as follows:

- Start node of the workflow graph is included in the instance subgraph.
- For any included OR-split node, exactly one child node and the edge to it are included.
- For any other type of node that is included in the instance subgraph, all child nodes and the edges to them are included.

Instance subgraphs of a workflow graph vary from each other due to the choice taken while choosing an outgoing edge (and the corresponding child node) from an OR-split node.

If a workflow graph does not have any OR-split node, then it will have only one instance subgraph. Then, the workflow graph will be structurally correct if and only if this instance subgraph is correct. However, if there are OR-split nodes in the workflow graph, then it will have many instance subgraphs. A brute force method to verify the workflow graph would be to identify and verify all the instance subgraphs of it. However, this would be cumbersome and very time consuming when there are many instance subgraphs in the workflow graph. Also, this shows the need for comparing workflow verification algorithms based on the time consumed for verifying a workflow process. If there is infinite amount of time available to verify a workflow process, then it can be simply verified by verifying all its instance subgraphs. MS algorithm identifies and verifies a subset of instance subgraphs of the workflow graph to verify it completely. Even though MS algorithm verifies only a subset of instance subgraphs, it chooses this in a systematic order that verifying these instance subgraphs would be sufficient to verify the complete workflow graph.

Definitions:

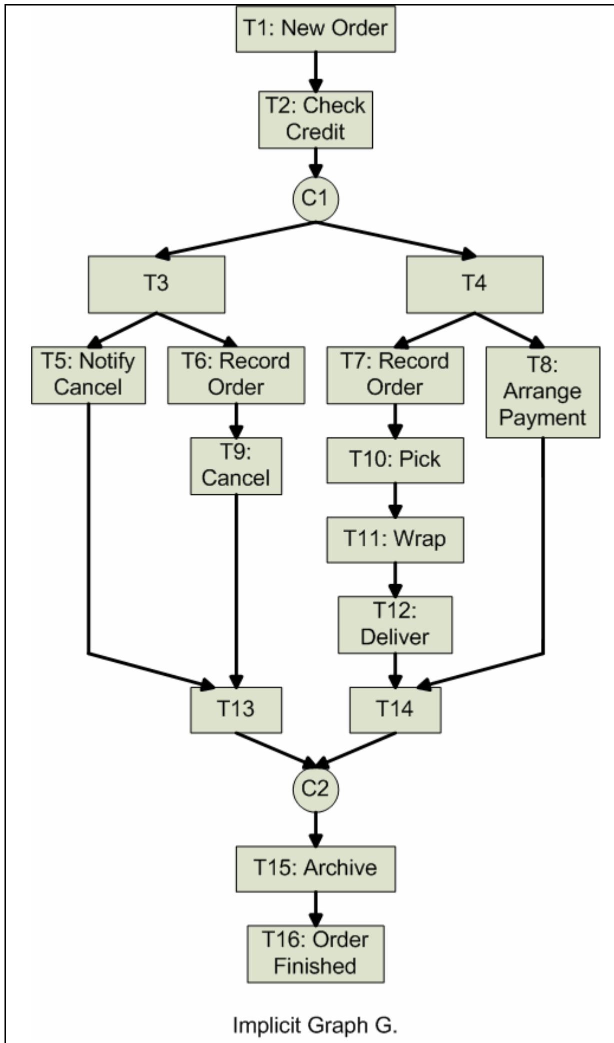
G : Implicit Graph, i.e., the original complete workflow graph

G' : At any moment during the execution of the algorithm, the explicit graph *G'* is defined as the portion of implicit graph *G* that has been traversed so far.

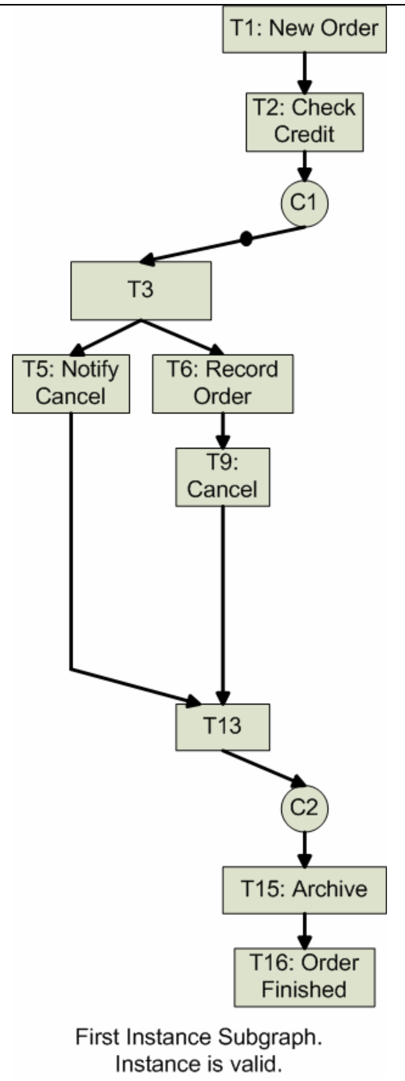
MS algorithm is iterative and each iteration comprises two phases, *CIS* and *VIS*, where *CIS* is for creating an instance subgraph and *VIS* is for verifying an instance subgraph. *CIS* stands for *Create_Instance_Subgraph* and *VIS* stands for *Verify_Instance_Subgraph*. *CIS* and *VIS* traverse the graph in depth-first manner. *CIS* marks one outgoing edge for every OR-split node that it expands and all outgoing edges for the expansion of any other type of node. Thus, it creates an instance subgraph. When illustrating using figures, if a node included in an instance subgraph is not an OR-split node, then marking of outgoing edges from it are not shown as it is obvious that all outgoing edges from it have to be included in the instance subgraph. **An instance subgraph can be derived from an explicit graph by beginning from the start node and traversing along its marked edges.** *VIS* traverses the marked edges of the explicit graph obtained after *CIS* was executed, to verify the instance subgraph created in this iteration. If structural conflict is found, then *VIS* reports the structural conflict and the algorithm stops. If *VIS* finds that the instance subgraph is structurally correct, then, *Prepare_for_next_instance* is called to prepare data structures for the next iteration.

In the first iteration, MS algorithm chooses the left-most outgoing edge (and the corresponding child node) from any OR-split node for creating the first instance subgraph. For any subsequent iteration, choices of outgoing edges from OR-split nodes are same as that of the instance subgraph in the previous iteration except for a special OR-split node. This special OR-split node is called PED-OR node, which stands for "Partially-Explored, Deepest OR-split node". PED-OR node would have been traversed through at least one of its outgoing edges in one of the previous iterations. Hence, in this iteration, a new unexplored outgoing edge (and the corresponding child node) is chosen from the PED-OR node for creating a new instance subgraph.

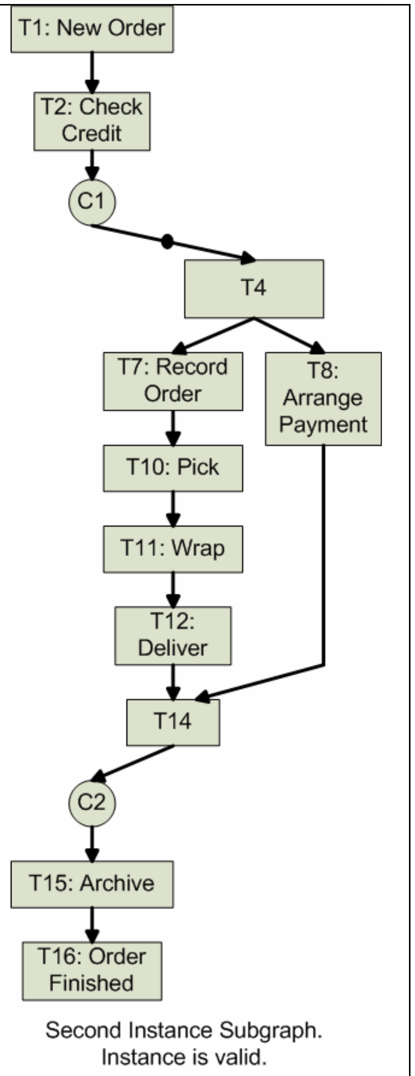
Figure 3: Order processing process showing trace of MS algorithm



(a) Workflow graph adopted from (Dehnert and Aalst 2004)



(b)



(c)

Table 1: Table showing the detailed trace of MS algorithm for the workflow graph given in (Figure 3)

Iteration no.	Instance Creation and Verification				
	Create Instance Subgraph			Verify Instance Subgraph	
1	**Node expanded**	**Z**	**OR_Split_Stack**	**Node visited**	**Y**
	T1	T2		T1	T2
	T2	C1		T2	C1
	C1	T3	C1	C1	T3
	T3	T6, T5	C1	T3	T6, T5
	T6	T9, T5	C1	T6	T9, T5
	T9	T13, T5	C1	T9	T13, T5
	T13	C2, T5	C1	T13	C2, T5
	C2	T15, T5	C1	C2	T15, T5
	T15	T16, T5	C1	T15	T16, T5
	T16	T5	C1	T16	T5
	T5	< T13 >	C1	T5	< T13 >
2	**Node expanded**	**Z**	**OR_Split_Stack**	**Node visited**	**Y**
	C1	T4	C1	T1	T2
	T4	T8, T7	C1	T2	C1
	T8	T14, T7	C1	C1	T4
	T14	<C2>, T7	C1	T4	T8, T7
	T7	T10	C1	T8	T14, T7
	T10	T11	C1	T14	C2, T7
	T11	T12	C1	C2	T15, 7
	T12	<T14>	C1	T15	T16, 7
				T16	T7
				T7	T10
				T10	T11
				T11	T12
				T12	<T14>

Data structures used in the algorithm:

Z : Stack for storing the nodes that have to be expanded for creating the instance subgraph corresponding to an iteration

OR_split_stack : Stack for storing the partially explored OR-split nodes such that the top node of the stack has the PED-OR node for the next iteration

Y : Stack used in *VIS* for storing the nodes that have to be visited for verifying the instance subgraph created by *CIS*

We refer to (Figure 3) and (Table 1) for pictorially illustrating various steps of MS algorithm (detailed workout for this example is given in section 4.1). For the first iteration, *CIS* begins expanding nodes from the start node of the workflow graph (Refer to Figure 3 part (b) for the first instance subgraph created by *CIS*). During subsequent iterations, *CIS* begins expanding from the PED-OR node (PED-OR node for the second iteration in the example is C1 and this is italicized in (Table 1)). During any iteration, *CIS* does not traverse beyond and does not expand any node that was expanded earlier (such nodes are shown in angled brackets in (Table 1)). In any iteration, *VIS* visits the instance subgraph created by *CIS* in that iteration by following the marked edges beginning from the start node. If *VIS* finds that an OR-merge node in the instance subgraph is visited more than once, then it reports *"Lack of Synchronization"* structural conflict and the algorithm stops. After visiting all the nodes of the instance subgraph, *VIS* checks if an AND-merge node is visited through all its incoming edges. If any AND-merge node is not visited through all its incoming edges, then *VIS* reports *"Deadlock"* structural conflict and the algorithm stops. During any iteration, *VIS* does not traverse beyond any node that was visited earlier (such nodes are shown in angled brackets in (Table 1)). As a final step in the iteration, *Prepare_for_next_instance* pops any OR-split node from the top of the OR_split_stack until it finds an OR-split node that has unexplored outgoing edges from that node. If such an OR-split node is found, then it is used as PED-OR node for the next iteration. If the OR_split_stack becomes empty, then *Prepare_for_next_instance* reports that the workflow graph is structurally correct and the algorithm stops. It could be noted that the purpose of OR_split_stack is to just find the PED-OR node for the next iteration. We have also designed and implemented a modified version of MS algorithm in which PED-OR node is obtained through an explicit search while

VIS traverses the instance subgraph created by *CIS*. This modified version will not require OR_split_stack.

4.1 Work-out of the Proposed Algorithm

An *Order processing* business process is used to trace the execution of the algorithm and the corresponding workflow graph is given in Figure 3 part (a). This business process is adopted from the Event-driven Process Chain representation of it given in (Dehnert and Aalst 2004) and adapted to meet the needs of this paper. Instance subgraphs obtained during various iterations of the algorithm for the workflow graph is shown in Figure 3 part (b) and Figure 3 part (c). Instance subgraphs are obtained from explicit graphs by starting from the start node and by following marked edges for an OR-split node and all other edges from any other node. Trace of the algorithm showing the iteration by iteration content of stack Z and OR_Split_Stack after each node expansion in *CIS*, and content of stack Y after each node is visited in *VIS*, in each iteration is given in Table 1.

4.2 Proof:

4.2.1 Completeness Proof

Theorem A: Let *G* be a workflow graph and let *n* be an AND-merge node in *G* such that:

(i) there exists an instance subgraph $\mathcal{J}$ such that *n* belongs to $\mathcal{J}$ and there is a deadlock at *n* in $\mathcal{J}$, and

(ii) there does not exist any predecessor *q* of *n* in *G* where a deadlock or lack of synchronization error occurs in $\mathcal{J}$ or any other instance subgraph of *G*.

Now, if MS algorithm does not verify $\mathcal{J}$, then there must exist another erroneous instance subgraph $\mathcal{J}^*$ that is verified by MS algorithm.

Proof: See AND-merge proof section in Theoretical Analysis Section. □

Theorem B: Let G be a workflow graph and let n be an OR-merge node in G such that:

(i) there exists an instance subgraph $\mathcal{I}$ such that n belongs to $\mathcal{I}$ and there is a lack of synchronization at n in $\mathcal{I}$, and

(ii) there does not exist any predecessor q of n in G where a deadlock or lack of synchronization error occurs in $\mathcal{I}$ or any other instance subgraph of G.

Now, if MS algorithm does not verify $\mathcal{I}$, then there must exist another erroneous instance subgraph $\mathcal{I}^*$ that is verified by MS algorithm.

Proof: See OR-merge proof section in Theoretical Analysis Section. □

Theorem C: In MS algorithm, if one erroneous instance subgraph $\mathcal{I}$ is not verified then there must exist another erroneous instance subgraph $\mathcal{I}^*$ that is verified by MS algorithm. We call $\mathcal{I}^*$ as the dual of $\mathcal{I}$.

Proof: Structural conflict could occur in an instance subgraph as either deadlock in an AND-merge node, or as lack of synchronization in an OR-merge node. In Theorem A, we have proved that deadlock will be detected by MS algorithm. Similarly, in theorem B, we have proved that lack of synchronization will be detected by MS algorithm. Hence, proved. This theorem provides the completeness proof for MS algorithm. □

4.2.2 Termination Proof

For creating an instance subgraph, at least one new edge will be traversed. Exactly one instance is created for any iteration. Thus, the algorithm will terminate after finitely many iterations, since there are only E edges in the workflow graph G.

4.2.3 Complexity Proof

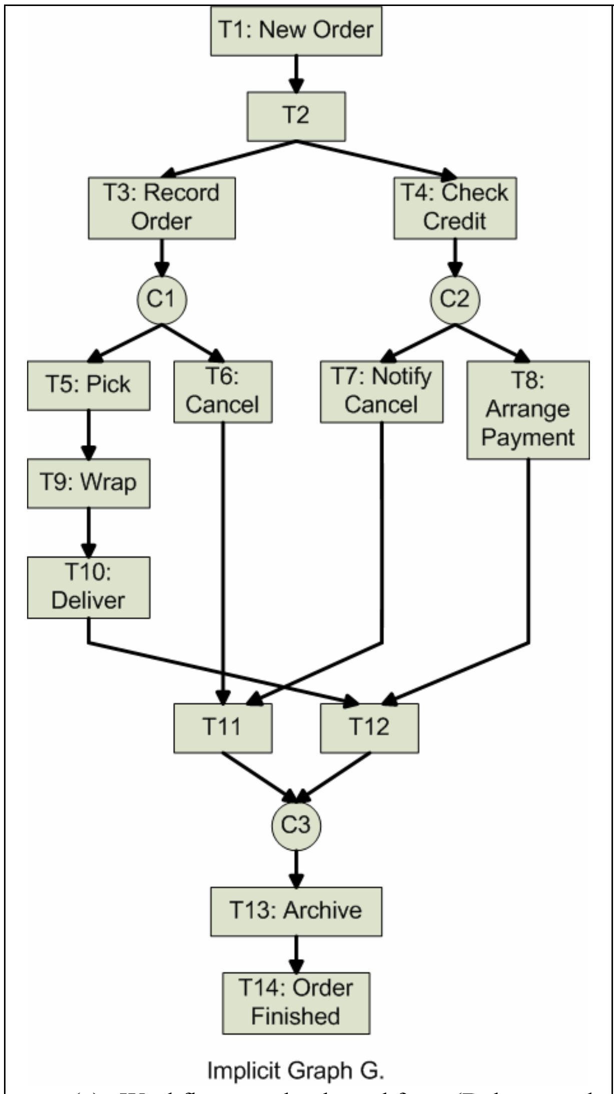
For creating each instance subgraph, a maximum of O(*E*) computations will be made in *CIS* as no edge of G is traversed more than once while expanding the nodes for that instance subgraph. For verifying each instance subgraph, a maximum of O(*E*) computations will be made in *VIS* as no edge of the instance subgraph is traversed more than once while visiting the nodes in *VIS* for that instance subgraph. For creating each instance subgraph, a new child node of the PED-OR node (which is an OR-split node) is chosen as the starting node for *CIS*. Hence, number of instances generated is less than the sum of number of child nodes of all OR-split nodes in G. Let E_{OSi} denote the number of child nodes for the ith OR-split node and let N_{OS} be the number of OR-split nodes in the workflow graph G. Then, complexity for verifying workflow graphs using MS algorithm is,

$$\mathrm{O(E)}*\sum_{i=1}^{N_{OS}} \mathrm{E_{OSi}} \le \mathrm{O(E)*O(E)=O(E^2)}.$$

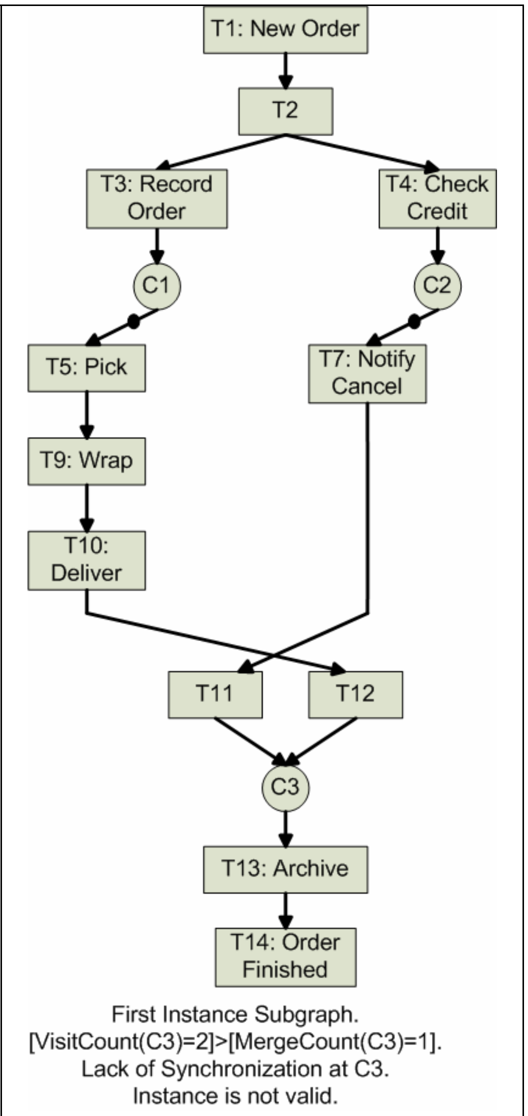
4.3 Trace of MS Algorithm for Various Workflow Graphs

Figure 4 part (a) depicts a workflow graph for *Order Processing* process originally given in (Dehnert and Aalst 2004) with a Event-driven Process Chain representation. This process was modified to meet the needs of this paper. This process is for handling orders of mobile phones in a telephone company, and consists of two parts: (a) distribution processing, and (b) payment processing. For distribution processing, the order is recorded, and if the item is available then the item is picked, wrapped and delivered. If in case, the item is not available, then the distribution is cancelled. For payment processing, if the credit is ok, then the payment is arranged for the order and if otherwise the payment is cancelled. Finally the order is archived and finished. This workflow process works fine when both the item is available and credit is ok. It also works fine when both the item is not available and the credit is not ok. However, if either the item is not available or the credit is not ok, then the workflow process will have lack of synchronization structural conflict at the OR-merge node *"C3"*. First instance subgraph generated is shown in Figure 4 part (b). Since the error is detected in the first instance subgraph itself, the algorithm stops after the first iteration.

Figure 4: Order processing process showing lack of synchronization structural conflict



(a) Workflow graph adopted from (Dehnert and Aalst 2004)



(b)

Figure 5 gives an example workflow graph with multiple levels of overlapping. An overlapped structure has an AND-split node connected directly to an OR-join node in a peculiar way that does not introduce structural conflicts. Trace of executing MS algorithm for this workflow graph is given in Table 2. This table gives the PED-OR node for each iteration along with the sequence of node expansion in *CIS* and sequence of node visit in *VIS*. Node *"C1"* is italicized in the second, third and fourth rows because the expansion of PED-OR node happens in the procedure *Prepare_for_next_instance* and not in *CIS* as for other nodes.

Figure 5: Toy problem with multiple level overlapping and no structural conflicts

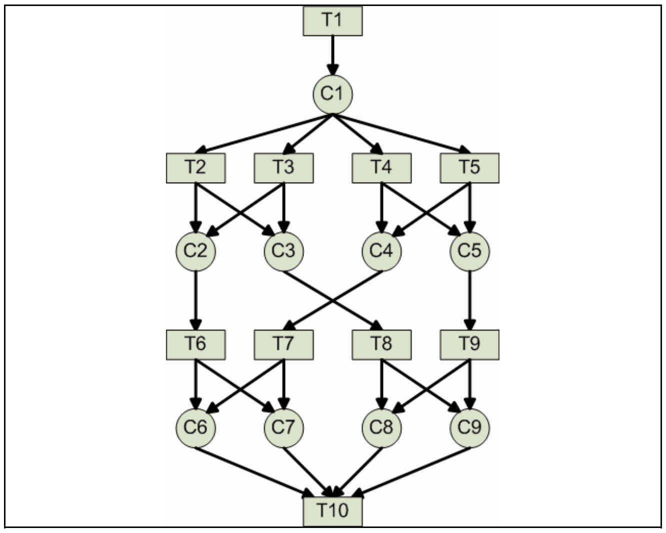


Table 2: Table showing the trace of MS algorithm for (Figure 5)

Iteration no.	**Instance Creation and Verification**		
	PED-OR node	**Create Instance Subgraph: Sequence of node expansion**	**Verify Instance Subgraph: Sequence of node visit**
1	-	T1, C1, T2, C3, T8, C9, T10, C8, C2, T6, C7, C6	T1, C1, T2, C3, T8, C9, T10, C8, C2, T6, C7, C6
2	C1	*C1*, T3	T1, C1, T3, C3, T8, C9, T10, C8, C2, T6, C7, C6
3	C1	*C1*, T4, C5, T9, C4, T7	T1, C1, T4, C5, T9, C9, T10, C8, C4, T7, C7, C6
4	C1	*C1*, T5	T1, C1, T5, C5, T9, C9, T10, C8, C4, T7, C7, C6

5 IMPLEMENTATION, RESULTS AND ANALYSIS

Both MS algorithm and Graph Reduction Algorithm II were implemented in C language on Linux platform. For MS algorithm, each node was represented through node number, node type, number of child nodes, child node numbers, number of parent nodes, parent node numbers, a true/false boolean for checking if the node is expanded, a true/false token boolean for checking if the node is visited, a marking indicator which specifies the outgoing edge chosen for an expanded OR-split node, and "visit count" which indicates the number of times the node was visited in an execution of Verify_Instance_Subgraph procedure. MS algorithm was tested using various test graphs in the literature.

Performance of MS algorithm and Graph Reduction Algorithm were checked for various types of random workflow graphs. These random workflow graphs were generated using a random workflow graph generator which was implemented in C language on Linux platform. Random workflow graph generator takes two inputs: number of nodes, and also whether to generate a correct workflow graph or a wrong workflow graph, and creates a workflow graph accordingly. It uses several constructs such as Correct AND construct, Correct OR construct, Wrong AND construct, Wrong OR construct, AND cluster construct, OR cluster construct and First Level Overlapped construct similar to those given in (Perumal and Mahanti 2006).

Performance of these algorithms were measured using clock() function that measures the time taken by the CPU in seconds. Hardware configuration of the system was 64-bit Itanium server, with RAM of 8 GB, and hard disk capacity of 73 GB x 3 arranged as RAID-5.

Figure 6: Performance comparison across wrong workflow graphs of various sizes

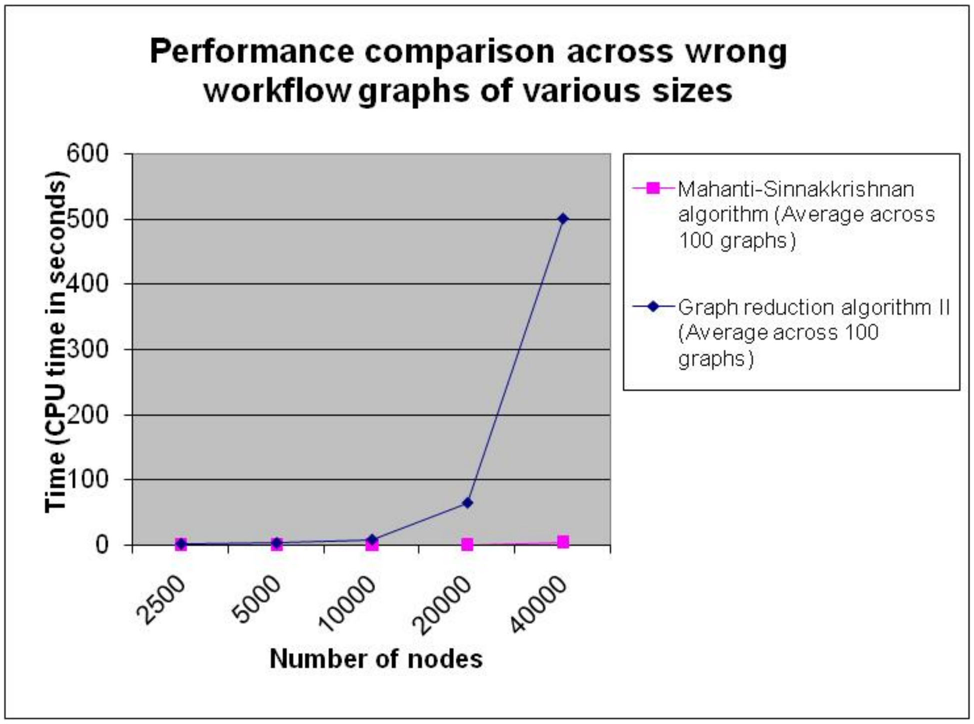


Figure 7: Performance comparison across correct workflow graphs of various sizes

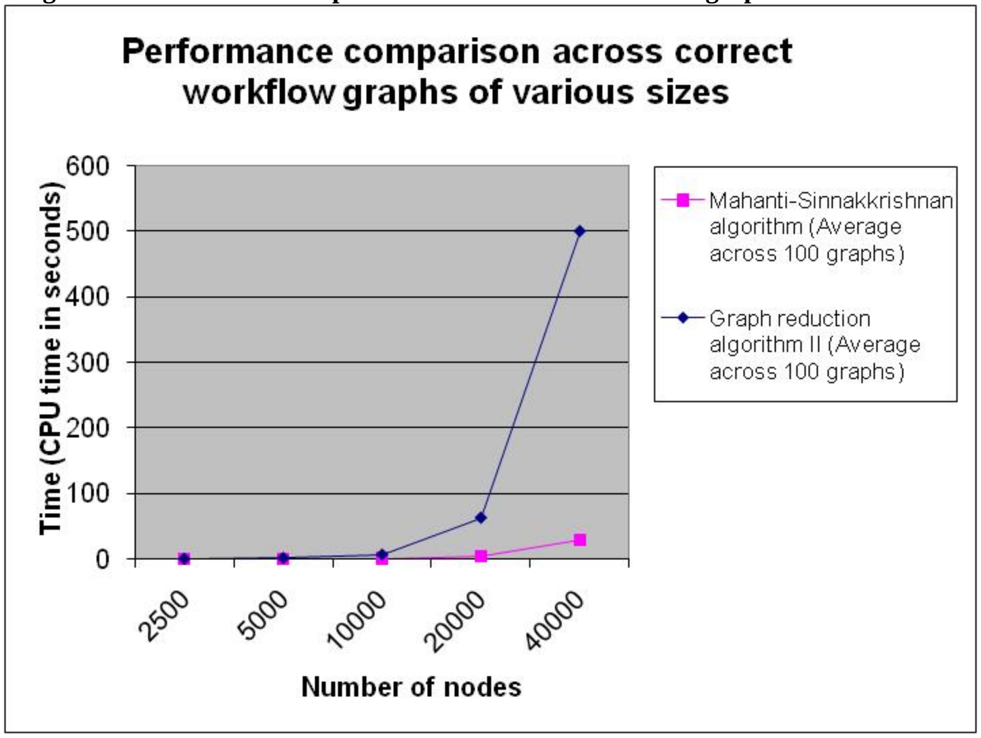


Figure 8: Performance comparison across wrong workflow graphs with varied proportion of OR-split nodes

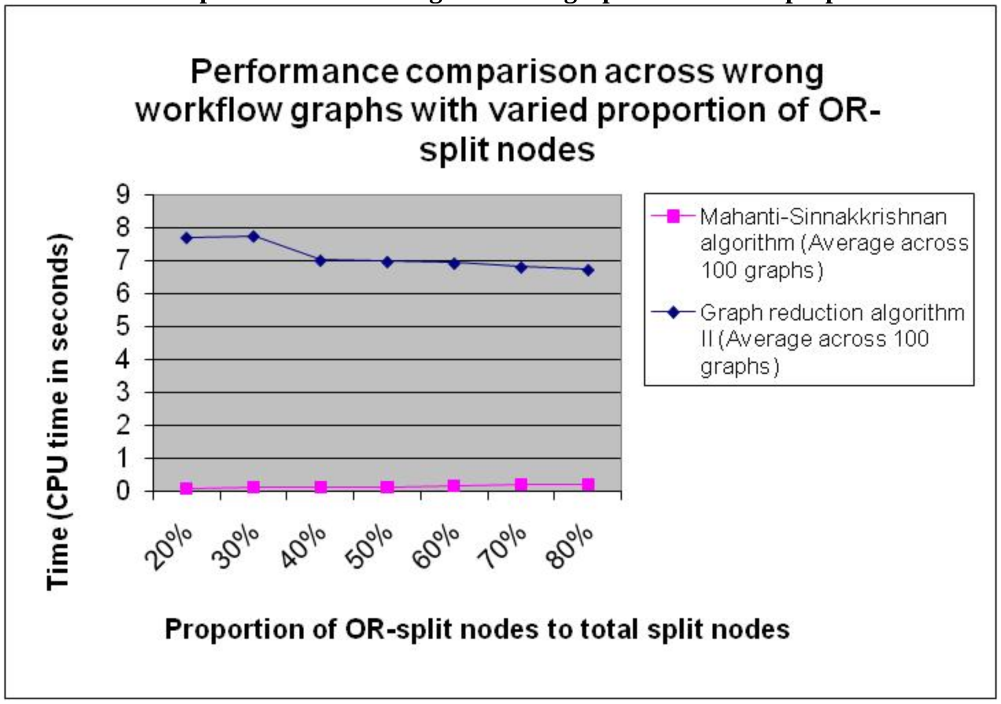


Figure 9: Performance comparison across correct workflow graphs with varied proportion of OR-split nodes

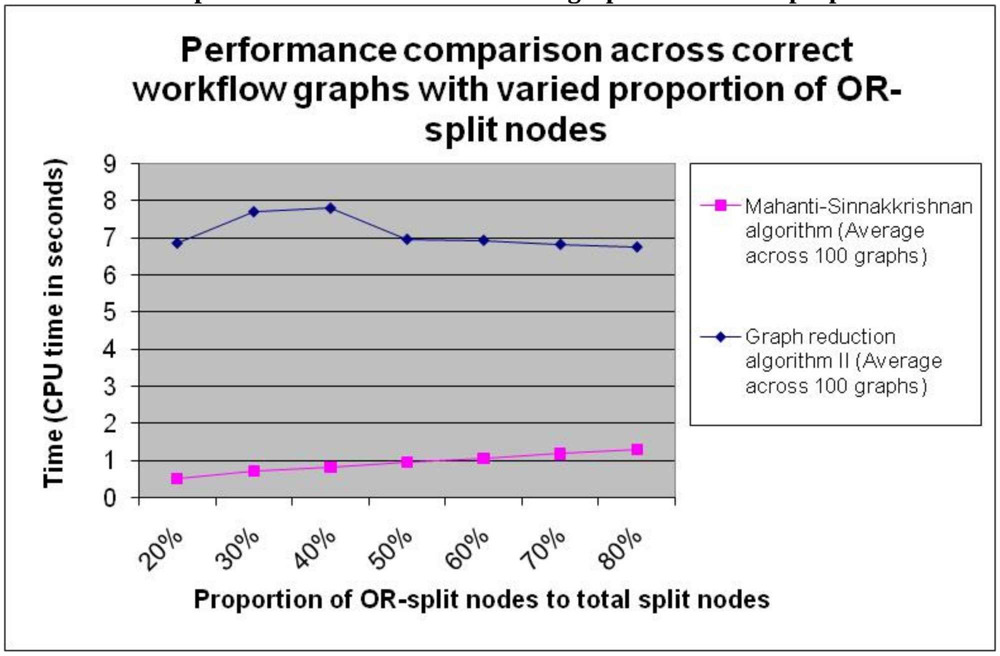


Figure 7 presents performance comparison between these two algorithms for correct workflow graphs with varied sizes, where size of the graph is measured in terms of number of nodes. Similarly, Figure 6 presents performance comparison for wrong workflow graphs with varied sizes. Figure 9 presents the performance comparison across correct workflow graphs with varied proportion of number of OR-split nodes to the total number of split nodes. This was tested with workflow graphs of size 10000. Similarly, Figure 8 presents the performance comparison across wrong workflow graphs with varied proportion of number of OR-split nodes to the total number of split nodes.

It could be seen from Figure 6 and Figure 7 that the time taken by Graph Reduction Algorithm II is several times the time taken by MS algorithm for verifying any random workflow graph.

When the proportion of number of OR-split nodes to the total number of split nodes increases, then the time taken by Graph Reduction Algorithm II initially increases and then decreases. This is shown in Figure 8 and Figure 9. This is because initially as the proportion increases, variety of various types of nodes in the workflow graph increases. Graph Reduction Algorithm II works better when the workflow graph has similar type of nodes. The same graph shows that the time taken by MS algorithm increases as the number of OR-split nodes increases. This is because MS algorithm verifies various instance subgraphs of the workflow graphs to verify the complete workflow graph, and the number of instance subgraphs increases if the proportion of number of OR-split nodes to the total number of split nodes increases.

6 THEORETICAL ANALYSIS

Definition 1:

- S_i, $1 \le i \le k$: i^{th} first level OR-split node in G. A first level OR-split node is defined as an OR-split node which does not have any other OR-split node as its predecessor in G.

- S_{i,p_u} : u^{th} hyperpath from the first level OR-split node S_i. Hyperpaths are numbered in depth first, left-to-right order. A hyperpath begins at a node and ends at the terminal node. For any hyperpath, one child node is included for every included OR-split node and all child nodes are included for all other included nodes. Note that an instance subgraph is a hyperpath from start node to end node. S_{i,p_1} and $S_{i,p_{last}}$ denote the first and last hyperpaths from the first level OR-split node S_i respectively.
- $p_n^{S_{i,p_u}}$: Set of parent nodes of *n* in u^{th} hyperpath from S_i.
- Any instance subgraph of *G* will contain all first level OR-split nodes and exactly one hyperpath from each of it. Now, let $(S_{1,p_{z_1}}, S_{2,p_{z_2}}, \dots, S_{k,p_{z_k}})$ represent $z_1^{th}, z_2^{th}, \dots, z_k^{th}$ hyperpaths from first level OR-split nodes, S_1, S_2, … , S_k respectively present in an instance subgraph. We also call this as a *"first-level-hyperpath-combination"* (or simply "combination") in an instance subgraph.
- p_n^G : Set of parent nodes of *n* in *G*.
- $p_n^{\mathcal{I}}$: Set of parent nodes of *n* in an instance subgraph $\mathcal{I}$.

Theorems:

Theorem D: Let the first level OR-split nodes $(S_1, S_2, \dots, S_k)$ in workflow graph *G* have $m_1, m_2, \dots m_k$ child nodes respectively. Let S_{i,l_i} denote that l_i^{th} child was chosen for S_i in an instance subgraph. Now, if an instance subgraph had the choice of child nodes for first-level OR-split nodes as $(S_{1,l_1}, S_{2,l_2}, \dots, S_{k,l_k})$, this was verified by MS algorithm, and *j* is the smallest index, $1 \le j \le k$, such that $lj < mj$, then this implies that,

$l_i = m_i, \forall\ i < j$, and

$l_i = 1, \forall\ i > j$.

Proof: This is due to the selection of PED-OR node in Prepare_for_next_instance and due to the following steps in Create_Instance_Subgraph of MS algorithm:
- "Push q to OR_split_stack", and
- "Push these child nodes to the top of Z in, say, left-to-right order such that the right-most child node is on top of Z". □

Lemmas:

Let *G* be a workflow graph and let *n* be an AND-merge node in *G* such that:

(i) there exists an instance subgraph $\mathcal{I}$ such that *n* belongs to $\mathcal{I}$ and there is a deadlock at *n* in $\mathcal{I}$, and

(ii) there does not exist any predecessor *q* of *n* in *G* where a deadlock or lack of synchronization error occurs in $\mathcal{I}$ or any other instance subgraph of *G*.

We use this definition of *n, G* and $\mathcal{I}$ for the following lemmas.

Lemma 1: Let S_{i,p_u} and S_{j,p_v} be two hyperpaths from two first level OR-split nodes S_i and S_j, $i \neq j$, which have paths passing through *n* in an instance subgraph of *G*.

Then, if there is any common OR-merge node *c* between these two hyperpaths such that *c* is a predecessor of *n*, then it should satisfy the condition, $p_c^{S_{i,p_u}} = p_c^{S_{j,p_v}}$.

Proof: By contradiction.

$p_c^{S_{i,p_u}} \neq p_c^{S_{j,p_v}}$ $\Rightarrow$ Multiple incoming edges to *c* $\Rightarrow$ Lack of synchronization error at *c*. By assumption, *n* does not have any predecessor in *G* which is causing error. □

Lemma 2: Let S_{i,p_u} and S_{j,p_v} be two hyperpaths from two first level OR-split nodes S_i and S_j, $i \neq j$, which have paths passing through *n* in an instance subgraph of *G*. Let there be a common AND-merge node *c* between these two hyperpaths such that *c* is a predecessor of *n*, and $p_c^{S_{i,p_u}} \neq p_c^{S_{j,p_v}}$.

Then,

(i) all other hyperpaths from S_i (or S_j) should also pass through *c*, and

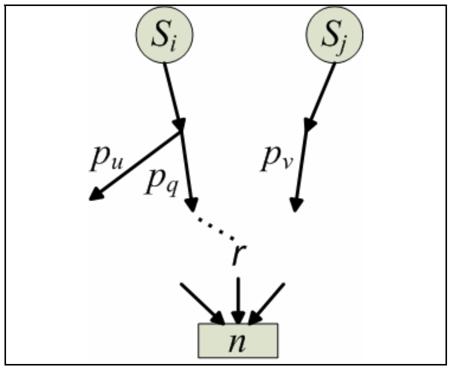
(ii) all other hyperpaths from S_i (or S_j) should have the same parent set for *c* as that of S_{i,p_u} (or S_{j,p_v}).

Proof: By contradiction.

This will lead to a deadlock at *c* for any other combination of hyperpaths from S_i and S_j. □

Lemma 3: Let S_{i,p_q} be a hyperpath from a first level OR-split node S_i having a link *(r,n)*, i.e., $\{r\} \subseteq p_n^{S_{i,p_q}}$. Let S_{i,p_u} be another hyperpath from S_i that does not pass through the link *(r,n)*, i.e., $\{r\} \not\subset p_n^{S_{i,p_u}}$. Then, any instance subgraph containing S_{i,p_u} cannot have any hyperpath S_{j,p_v} from any other first level OR-split node S_j, *j* ≠ *i*, such that S_{j,p_v} has the link *(r,n)*.

Figure 10. Workflow graph showing constructs for Lemma 3 statement



(Figure 10) shows in support of Lemma 3 that,

S_{i,p_u} and S_{j,p_v} exist together in an instance subgraph

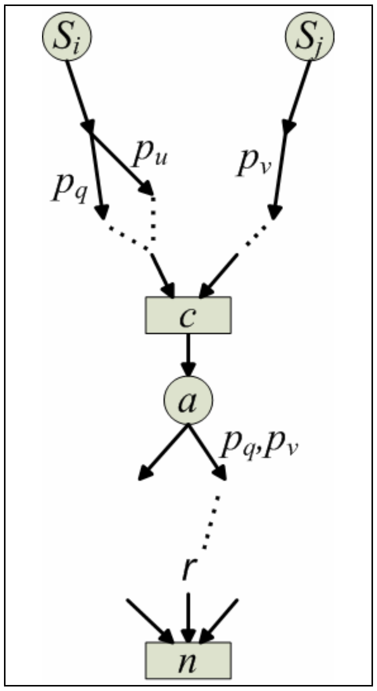
$\Rightarrow S_{j,p_v}$ does not have the link *(r,n)*.

Proof: If *r* does not have any other first level OR-split node as its predecessor in *G*, then it is trivially proved.

Now, the proof by contradiction.

Suppose *r* has a first level OR-split node S_j, $j \neq i$, as its predecessor. Let the hyperpath S_{j,p_v} have the link *(r,n)* and let both S_{i,p_u} and S_{j,p_v} be present in an instance subgraph.

Figure 11. Workflow graph showing the constructs for Lemma 3 Case 2



Let c be a node where S_{i,p_q} and S_{j,p_v} merge for the first time such that either there is a path from c to r in these hyperpaths, or $c = r$.

Case 1: c is an OR-merge node

This is not possible due to Lemma 1.

Case 2: c is an AND-merge node

Then, by lemma 2, all hyperpaths from S_i, thus S_{i,p_u}, should also pass through c. (Figure 11) shows the constructs for this case.

Now, since: (i). S_{i,p_u} and S_{j,p_v} both are present in an instance subgraph, and (ii). S_{j,p_v} has the path from c to n passing through r, it is a must that S_{i,p_u} will also pass through r to n – hence, the contradiction. □

Lemma 4: Let S_i be a first level OR-split node such that all parent nodes of n are its successors in G. Then, there exists a hyperpath S_{i,p_w} from S_i such that $p_n^{S_{i,p_w}} \subset p_n^G$ and $p_n^{S_{i,p_w}} \neq \{\}$.

Proof: Any instance subgraph of G will contain exactly one hyperpath from S_i.

If a hyperpath S_{i,p_u} from S_i has all the incoming edges of n, then an instance subgraph containing S_{i,p_u} will not have deadlock at n because all incoming edges of n will be present in this instance subgraph.

If a hyperpath S_{i,p_v} from S_i does not have any incoming edge of n, then using lemma 3, in an instance subgraph containing S_{i,p_v} there cannot be any hyperpath S_{j,p_y} from any first level OR-split node S_j such that S_{j,p_y} has a path passing through n. Thus, any instance subgraph containing S_{i,p_v} will not have node n in it.

Thus, an instance subgraph containing S_{i,p_u} will not have deadlock at n, and an instance subgraph containing S_{i,p_v} will not have node n itself. Hence, for an instance subgraph of G to have a deadlock at n, it should contain a hyperpath S_{i,p_w} from S_i such that $p_n^{S_{i,p_w}} \subset p_n^G$ and $p_n^{S_{i,p_w}} \neq \{\}$. □

Proof of Theorem-A stated in section 4.2:

Consider the AND-merge node *n* as given in the statement of the theorem. Since *n* is causing deadlock, it must be the case that the parent nodes of *n* in $\mathcal{I}$, i.e., $p_n^{\mathcal{I}}$ is a subset of p_n^G. To put it simply, we can say that some paths (i.e., sequence of arcs) from the root node to *n* have not been activated (or included) in $\mathcal{I}$. This inadequate number of paths from root node to *n* can only happen due to the selection of a single child node at every OR-split node included in $\mathcal{I}$, because all children of other nodes such as AND-split, OR-merge, AND-merge and sequence nodes are included in any instance subgraph of *G*. Thus, only OR-split nodes can cause variation in activation of paths between instance subgraphs. In this proof, we mainly focus on the first level OR-split nodes in *G*. Hence, a deadlock can only occur in a node which is a successor of at least one first level OR-split node.

A hyperpath is defined as, "various multi-pronged process paths that emanate from a node of a workflow process". Thus, detection of deadlock can be illustrated in terms of instance subgraphs that vary in terms of activation or non-activation of incoming edges to an erroneous AND-merge node through hyperpaths from various first level OR-split nodes.

Let $\mathcal{I}$ have the first-level-hyperpath-combination $(S_{1,p_{z_1}}, S_{2,p_{z_2}}, \ldots, S_{k,p_{z_k}})$ causing deadlock at AND-merge node *n*. We will prove that MS algorithm detects deadlock at node *n*, unless it terminates prior to that by finding some other error in *G*.

<u>Case 1:</u> $\mathcal{I}$ was checked by MS algorithm.

Then, the error would have been determined in procedure *VIS*.

<u>Case 2:</u> $\mathcal{I}$ was not checked by MS algorithm.

<u>Case 2.1:</u> Node *n* is successor of more than one first level OR-split node in *G*.

Consider a first level OR-split node S_i with the lowest index (i.e., *i* such that $1 \le i \le k$) that has multiple hyperpaths, including two hyperpaths S_{i,p_u} and S_{i,p_v} in *G* such that: (i) $(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}) \neq \{\}$ and (ii) the hyperpaths S_{i,p_u} and S_{i,p_v} were checked by MS algorithm. Consider an instance subgraph having the combination $(S_{1,p_{last}}, S_{2,p_{last}}, \ldots, S_{i-1,p_{last}}, S_{i,p_v}, S_{i+1,p_1}, \ldots, S_{k,p_1})$, which would have been verified by MS algorithm as per theorem-D.

<u>Case 2.1.1:</u> The combination $(S_{1,p_{last}}, S_{2,p_{last}}, \ldots, S_{i-1,p_{last}}, S_{i,p_v}, S_{i+1,p_1}, \ldots, S_{k,p_1})$ includes *n*.

Since $(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}) \neq \{\}$, at least one incoming edge of *n* will not be active. This is because, as per Lemma 3, parent nodes of *n* in the set $(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}})$ that are not reached through S_{i,p_v} will not be reached from any other first level OR-split node for this combination. Hence, this combination will lead to deadlock error.

Case 2.1.2: The combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_v}, S_{i+1,p_1}, ... , S_{k,p_1}) does not include *n*.

Case 2.1.2.1: $p_n^{S_{i,p_u}} \subset p_n^G$.

The combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_v}, S_{i+1,p_1}, ... , S_{k,p_1}) did not include *n*. Since $p_n^{S_{i,p_u}} \subset p_n^G$ and the combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_u}, S_{i+1,p_1}, ... , S_{k,p_1}) only replaces S_{i,p_v} by S_{i,p_u}, this combination is bound to cause deadlock error at n.

Case 2.1.2.2: $p_n^{S_{i,p_u}} = p_n^G$.

This implies that all parent nodes of *n* are successors of S_i in *G*. Since $\mathcal{J}$ is erroneous, using lemma 4, there exists a hyperpath S_{i,p_w} from S_i such that $p_n^{S_{i,p_w}} \subset p_n^G$ and $p_n^{S_{i,p_w}} \neq \{\}$. The combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_v}, S_{i+1,p_1}, ... , S_{k,p_1}) did not include *n*. Hence, hyperpaths from other first level OR-split nodes were not passing through *n* for this combination. Since the combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_w}, S_{i+1,p_1}, ... , S_{k,p_1}) only replaces S_{i,p_v} by S_{i,p_w}, this combination will lead to deadlock at n.

Case 2.2: Node *n* is a successor of exactly one first level OR-split node in *G*.

Case 2.2.1: All the paths from the start node to *n* in *G* pass through a first level OR-split node.

Let S_i be the only first level OR-split node having *n* as successor in *G*. Then, S_i will have two hyperpaths S_{i,p_u} and S_{i,p_v} in *G* such that $\left(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}\right) \neq \{\}$, $p_n^{S_{i,p_v}} \neq \{\}$, and that the hyperpath S_{i,p_v} was checked by MS algorithm. Consider the combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_v}, S_{i+1,p_1}, ... , S_{k,p_1}). This combination will lead to deadlock error as at least one of the incoming edges of *n* will not be active for this combination. This is because, $\left(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}\right) \neq \{\}$, and $p_n^{S_{i,p_v}} \neq \{\}$. Hence, proved.

Case 2.2.2: There is at least one path from the start node to *n* in *G* which is not passing through any of the first level OR-split nodes.

Let S_i be the only first level OR-split node having *n* as successor in *G*. Then, S_i will have two hyperpaths S_{i,p_u} and S_{i,p_v} in *G* such that $\left(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}\right) \neq \{\}$, and that the hyperpath S_{i,p_v} was checked by MS algorithm. Consider the combination ($S_{1,p_{last}}$, $S_{2,p_{last}}$, … , $S_{i-1,p_{last}}$, S_{i,p_v}, S_{i+1,p_1}, … , S_{k,p_1}). This combination will lead to deadlock error as at least one of the incoming edges of *n* will not be active for this combination. This is because, $\left(p_n^{S_{i,p_u}} - p_n^{S_{i,p_v}}\right) \neq \{\}$ and there is at least one path from the start node to *n* which is not passing through any of the first level OR-split nodes. Hence, proved. □

Definition 2:

- *Maximal Path* : A maximal path from a node *n* is a sequence of directed arcs starting at node *n* and ending at the terminal node.

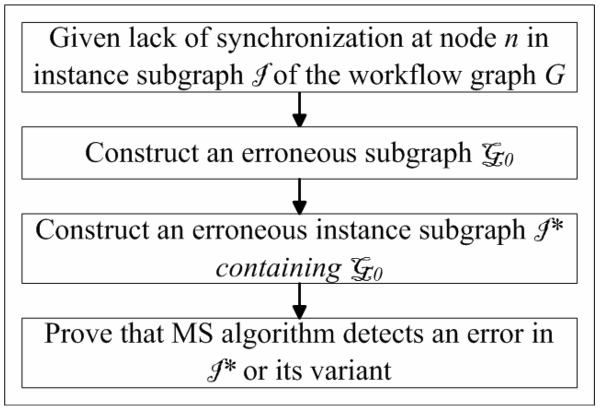
Additional Lemmas:

Lemma 5: All the nodes in *G* will be expanded by MS algorithm unless it terminates earlier by finding an error.

Proof: MS algorithm identifies various instance subgraphs in *G* in left-to-right, depth first manner and expands all the nodes in each instance subgraph unless it terminates earlier by finding an error. Hence, proved. □

Proof of Theorem-B stated in section 4.2:

Figure 12. Schematic diagram for the proof for Theorem-B



Given lack of synchronization at node *n* in instance subgraph $\mathcal{J}$ of the workflow graph *G*
Construct an erroneous subgraph $\mathcal{G}_0$
Construct an erroneous instance subgraph $\mathcal{J}$* *containing* $\mathcal{G}_0$
Prove that MS algorithm detects an error in $\mathcal{J}$* or its variant

Consider the OR-merge node *n* given in the statement of the theorem. Since $\mathcal{J}$ had lack of synchronization at *n*, it can be said that more than one path was merging at *n* in $\mathcal{J}$, i.e., $\left|p_n^{\mathcal{J}}\right| > 1$.

In this proof, first, we construct an erroneous subgraph $\mathcal{G}_0$ that has lack of synchronization at *n*, with a motive to construct one erroneous instance subgraph $\mathcal{J}$* containing this subgraph $\mathcal{G}_0$.

Then, we want to show that MS algorithm will either verify $\mathcal{I}^*$ or one of its variants to spot lack of synchronization error at *n*, unless it terminates prior to that by finding some other error in *G*.

(Figure 12) presents the schematic diagram for the proof.

Construction of erroneous subgraph $\mathcal{G}_0$:

Let *J* be the left-most, deepest AND-split node in *G* such that two different maximal paths from it merge at the OR-merge node *n*.

Let c_i and c_k be any two child nodes of *J* that have maximal paths passing through *n*.

Let P_1 and P_2 be two different maximal paths in *G* such that: (i) both P_1 and P_2 originate from node *J*, merge at node *n*, and end at the terminal node *t*, (ii) *n* has different parent nodes on P_1 and P_2, (iii) both P_1 and P_2 have identical sub-path from *n* to *t*, (iv) P_1 passes through c_i taking the left-most path from it with a condition to pass through *n*, and (v) P_2 passes through c_k taking the left-most path from it with a condition to pass through *n*.

Construction of erroneous instance subgraph $\mathcal{I}^*$:

Let $\mathcal{I}^*$ be an instance subgraph containing both P_1 and P_2, and hence having lack of synchronization at *n*. Without loss of generality, we assume $\mathcal{I}^*$ does not have any other error.

Proof that MS algorithm will detect error:

(Figure 13) presents traces in a workflow graph for various steps of this proof.

Case 1: MS algorithm verified an instance subgraph containing both the sub-paths of P_1 and P_2 from *J* to *n*.

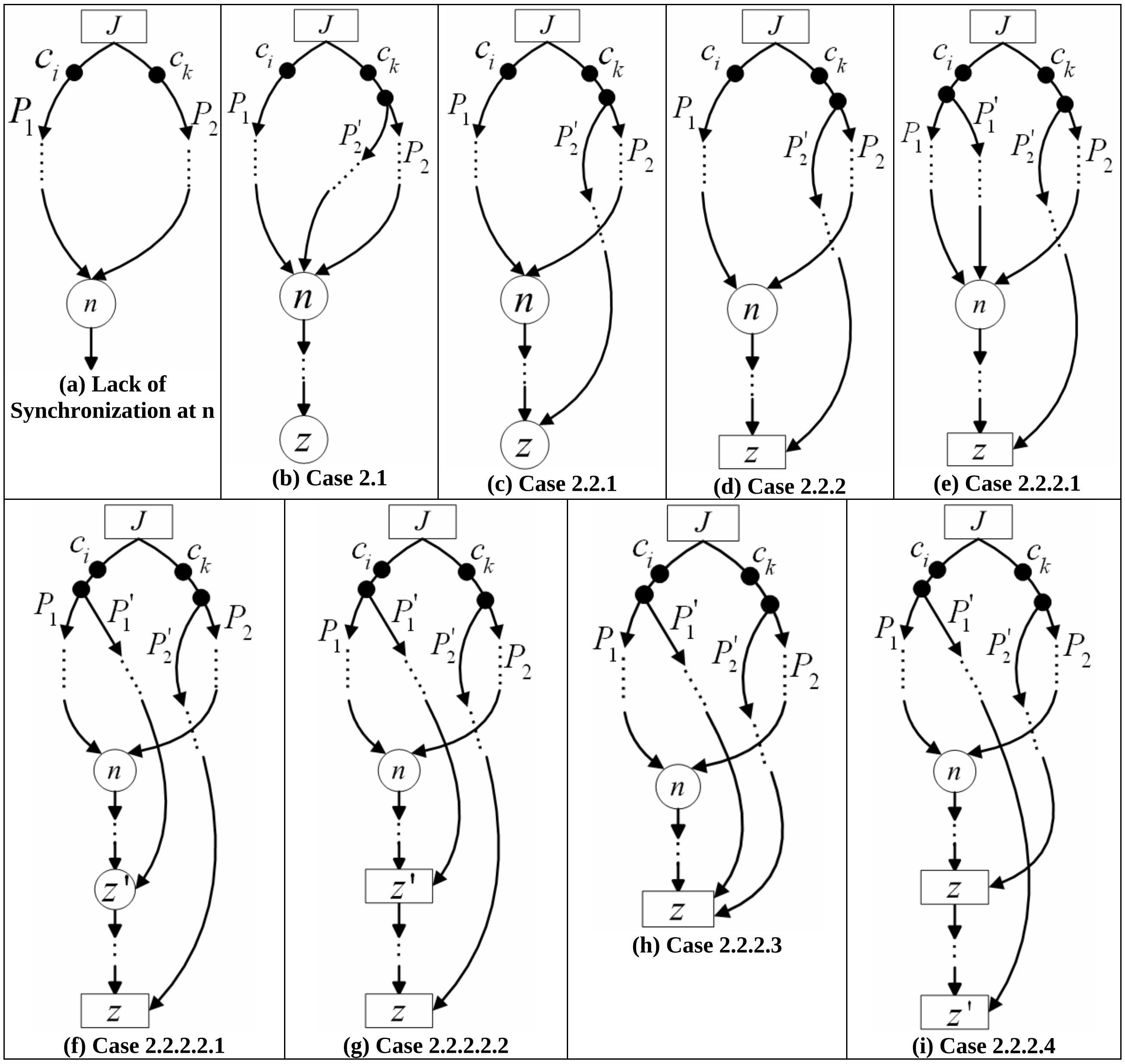
Then, MS algorithm would have reported lack of synchronization at *n* for this instance subgraph and hence proved.

Case 2: MS algorithm did not verify any instance subgraph containing both the sub-paths of P_1 and P_2 from *J* to *n*.

Now, consider the child node c_i of *J* on P_1 and c_k of *J* on P_2. Let the right-most maximal path from *J* in *G* passing through c_i be called P_1' and left-most maximal path from *J* in *G* passing through c_k be called P_2'.

Note that c_i can be same as *n*, and similarly, c_k can be same as *n*. However, both c_i and c_k cannot be same as *n* at the same time. This is because, *J* is an AND-split node and the workflow graph does not have more than one edge between any two nodes.

Figure 13. Various Steps of OR-Merge Proof



(a) Lack of Synchronization at n

(b) Case 2.1

(c) Case 2.2.1

(d) Case 2.2.2

(e) Case 2.2.2.1

(f) Case 2.2.2.2.1

(g) Case 2.2.2.2.2

(h) Case 2.2.2.3

(i) Case 2.2.2.4

Case 2.1: P_2' passes through *n*.

An instance subgraph containing the sub-paths of P_1 and P_2' from *J* to *n* will have lack of synchronization at *n*. At least, one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2: P_2' does not pass through *n*.

Let P_1 and P_2' first merge at a node *z*, which is a successor of *n*.

Case 2.2.1: *z* is an OR-merge node.

An instance subgraph containing the sub-paths of P_1 and $P_2^{'}$ from *J* to *z* will cause lack of synchronization at *z*. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2.2: *z* is an AND-merge node.

In that case, P_1 and $P_2^{'}$ will merge at AND-merge node *z* and will not have any error. Now, consider $P_1^{'}$.

Case 2.2.2.1: $P_1^{'}$ passes through *n*.

An instance subgraph containing the sub-paths of $P_1^{'}$ and P_2 from *J* to *n* will cause lack of synchronization at *n*. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2.2.2: $P_1^{'}$ and P_2 first merge at a node z' which is a successor of *n* and predecessor of *z*.

Case 2.2.2.2.1: z' is an OR-merge node.

An instance subgraph containing the sub-paths of $P_1^{'}$ and P_2 from *J* to z' will cause lack of synchronization at z'. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2.2.2.2: z' is an AND-merge node.

An instance subgraph containing $P_1^{'}$ and $P_2^{'}$ will deadlock at both z' and *z*, because the path from n to z' and the path from z' to z will be missing. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2.2.3: $P_1^{'}$ and P_2 first merge at node *z*.

An instance subgraph containing $P_1^{'}$ and $P_2^{'}$ will deadlock at *z* because the path from n to z will be missing. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D.

Case 2.2.2.4: $P_1^{'}$ and P_2 first merge at a node z' which is a successor of *z*.

An instance subgraph containing $P_1^{'}$ and $P_2^{'}$ will deadlock at *z* and z' because the path from n to z and the path from z to z' will be missing. At least one such instance subgraph would have been verified by MS algorithm as per Theorem-D. □

7 CONCLUSION AND FUTURE WORK

Workflow verification is crucial to the deployment of workflow in the business scenario. This is because if a workflow has structural conflicts, then it could lead to the disruption of the business services. Hence, the workflow has to be verified for structural conflicts before deployment. Workflow verification problem has been solved in a simple manner by MS algorithm. Since this algorithm imitates the basic depth first search technique, it is simple. Also, due to the design of the algorithm and the power of graph search principles, this algorithm has better complexity results over other existing algorithms.

As future work, MS algorithm can be extended for verifying workflow graphs with cycles. Initial results of a method extending MS algorithm for verifying cyclic workflow graphs is given in (Perumal and Mahanti 2006) and (Perumal and Mahanti 2007). Currently, we are working on theoretical analysis of this method.

We also intend to develop a GUI tool for tracing the progress of the algorithm visually. This will be useful for better understanding the algorithm. Also, any workflow process can be verified step by step through this tool.

REFERENCES

Basu, A., R.W. Blanning. 2000. A Formal Approach to Workflow Analysis. *Information Systems Research* **11**(1) 17 - 36.

Bi, H.H., J.L. Zhao. 2004. Applying Propositional Logic to Workflow Verification. *Information Technology and Management* **5**(3-4) 293-318.

Casati, F., S. Ceri, B. Pernici, G. Pozzi. 1998. Workflow Evolution. *Data and Knowledge Engineering* **24**(3) 211-238.

Choi, Y., J.L. Zhao. 2002. Matrix-based abstraction and verification of e-business processes *The First Workshop on e-Business*, Barcelona, Spain, 154-165.

Choi, Y., J.L. Zhao. 2005. Decomposition-Based Verification of Cyclic Workflows. D.A. Peled, Y.K. Tsay, eds. *Automated Technology for Verification and Analysis (ATVA 2005)*. Springer, Taipei, Taiwan, 84-98.

Dehnert, J., W.M.P.v.d. Aalst. 2004. Bridging The Gap Between Business Models And Workflow Specifications. *International Journal of Cooperative Information Systems* **13**(3) 289-332

Dumas, M., A.H.M. ter Hofstede. 2001. UML Activity Diagrams as a Workflow Specification Language. M. Gogolla, C. Kobryn, eds. *International Conference on The Unified Modeling Language. Modeling Languages, Concepts, and Tools* Springer-Verlag, Toronto, Ontario, Canada 76–90.

Eshuis, R., R. Wieringa. 2002. Verification support for workflow design with UML activity graphs *24th International Conference on Software Engineering*. ACM Press, Orlando, Florida, 166 - 176.

Georgakopolous, D., M. Hornick, A. Sheth. 1995. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* **3**(2) 119-153.

Lin, H., Z. Zhao, H. Li, Z. Chen. 2002. A Novel Graph Reduction Algorithm to Identify Structural Conflicts *Thirty-Fifth Annual Hawaii International Conference on System Science*. IEEE Computer Society Press, Maui, HW, 289.

Mahanti, A., A. Bagchi. 1985. AND/OR Graph Heuristic Search Methods. *J. ACM* **32**(1) 28-51.

Nilsson, N.J. 1982. *Principles of Artificial Intelligence*. Springer, Berlin.

Perumal, S., A. Mahanti. 2006. Cyclic Workflow Verification Algorithm for Workflow Graphs. Working paper, Indian Institute of Management Calcutta, Kolkata, India.

Perumal, S., A. Mahanti. 2007. MSCWV: Cyclic Workflow Verification Algorithm For Workflow Graphs. L. Fischer, ed. *2007 BPM & Workflow Handbook*. Future Strategies Inc., Lighthouse Point, FL.

Sadiq, W., M.E. Orlowska. 1999. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. M. Jarke, A. Oberweis, eds. *11th International Conference on Advanced Information Systems Engineering*. Springer, Heidelberg, Germany, 195-209.

Sadiq, W., M.E. Orlowska. 2000. Analyzing Process Models Using Graph Reduction Techniques. *Inf. Syst.* **25**(2) 117-134.

van der Aalst, W.M.P. 1998. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers* **8**(1) 21-66.

van der Aalst, W.M.P. 1999. Formalization and verification of event-driven process chains. *Information and Software Technology* **41**(10) 639-650.

van der Aalst, W.M.P., A. Hirnschall, H.M.W. (Eric) Verbeek. 2002. An Alternative Way to Analyze Workflow Graphs. A. Banks-Pidduck, J. Mylopoulos, C.C. Woo, M.T. Ozsu, eds. *14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*. Springer, Toronto, Ontario, Canada, 535-552.

van der Aalst, W.M.P., S. Jablonski. 2000. Dealing with Workflow Change: Identification of Issues and Solutions. *International Journal of Computer Systems, Science, and Engineering* **15**(5) 267-276.

van der Aalst, W.M.P., A.H.M. ter Hofstede. 2005. YAWL: yet another workflow language. *Information Systems* **30**(4) 245–275.

van der Aalst, W.M.P., A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros. 2003a. Workflow Patterns. *Distributed and Parallel Databases* **14**(1) 5-51.

van der Aalst, W.M.P., M. Weske, G. Wirtz. 2003b. Advanced topics in workflow management: Issues, Requirements and Solutions. *Journal of Integrated Design and Process Science* **7**(3) 49 - 77.

van Dongen, B.F., W.M.P. van der Aalst, H.M.W. Verbeek. 2005. Verification of EPCs: Using Reduction Rules and Petri Nets. O. Pastor, J. Falcão e Cunha, eds. *17th International Conference on Advanced Information Systems Engineering*. Springer, Porto, Portugal, 372-386.

Verbeek, H.M.W., W.M.P.v.d. Aalst, A.H.M.t. Hofstede. 2007. Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. *The Computer Journal* **50**(3) 294-314.

Verbeek, H.M.W., T. Basten, W.M.P. van der Aalst. 2001. Diagnosing Workflow Processes using Woflan. *Computer Journal* **44**(4) 246-279.

Verbeek, H.M.W., W.M.P. van der Aalst. 2000. Woflan 2.0: A Petri-Net-Based Workflow Diagnosis Tool. M. Nielsen, D. Simpson, eds. *21st International Conference on Application and Theory of Petri Nets*. Springer, Aarhus, Denmark, 475-484.

Wirtz, G., M. Weske, H. Giese. 2001. The OCoN Approach to Workflow Modeling in Object-Oriented Systems. *Information Systems Frontiers* **3**(3) 357–376.

Workflow Management Coalition. 1996. The workflow management coalition specifications - terminology and glossary. issue 2.0. Workflow Management Coalition, Winchester, UK.

Workflow Management Coalition. 2002. Workflow Process Definition Interface - XML Process Definition Language. Version 1.0. Workflow Management Coalition, Hingham, MA.

Workflow Management Coalition. 2005. Workflow Process Definition Interface - XML Process Definition Language. Version 2.00. Workflow Management Coalition, Lighthouse Point, FL.