Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models *

Wasim Sadiq and Maria E. Orlowska

Distributed Systems Technology Centre
Department of Computer Science & Electrical Engineering
The University of Queensland
Qld 4072, Australia
email: {wasim,maria}@dstc.edu.au

Abstract. The foundation of a process model lies in its control flow specifications. Using a generic process modeling language for workflows, we show how a control flow specification may contain certain structural conflicts that could compromise its correct execution. In general, identification of such conflicts is a computationally complex problem and requires development of effective algorithms specific for target system language. We present a visual verification approach and algorithm that employs a set of graph reduction rules to identify structural conflicts in process models for a generic workflow modeling language. We also provide insights into the correctness and complexity of the reduction process. The main contribution of the paper is a new technique for satisfying well-defined correctness criteria in process models.

1 Introduction

The workflow technology provides a flexible and appropriate environment to develop and maintain next generation of component-oriented enterprise-wide information systems. The production workflows, a subclass of workflows, support well-defined procedures for repetitive processes and provide means for automated coordination of activities that may span over several heterogeneous and mission-critical information systems of the organization. The production workflow applications are built upon business processes that are generally quite complex and involve a large number of activities and associated coordination constraints.

The objective of process modeling is to provide high-level specification of processes that are independent of target workflow management system. It is essential that a process model is properly defined, analyzed, verified, and refined before being deployed in a workflows management system.

A workflow is a set of tasks and associated execution constraints. A workflow management system coordinates the execution of these tasks to achieve some business

* The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

M. Jarke, A. Oberweis (Eds.): CAiSE'99, LNCS 1626, pp. 195-209, 1999.

objectives. Generally, tasks in a workflow are inter-related in such a way that initiation of one task is dependent on successful completion of a set of other tasks. Therefore, the order in which tasks are executed is very important. The control flow aspect of a process model captures the flow of execution from one task to another. This information is used by a workflow management system to properly order and schedule workflow tasks. Arguably, the control flow is the primary and the most important aspect of a process model. It builds a foundation to capture other workflow requirements.

The research presented in this paper addresses modeling and verification of production workflows. The Workflow Management Coalition [15] is developing a standard process definition language and an interface specification that could be used to transfer process models between products. The verification issues discussed in this paper are presented using a graphical process modeling language that is based on this standard.

The use of Petri Nets for workflow modeling has been explored in [1] and [7]. However, Petri Nets are not used in any of the major products. A few conceptual modeling languages and methodologies have also been proposed specifically for workflow technology [4] [5] [6] [10] [12] [13].

It is possible to introduce error situations while building large workflow specifications. Such modeling inconsistencies and errors may lead to undesirable execution of some or all possible instances of a workflow. It is essential to rectify such problems at design phase rather than after deploying the workflow application. We have found limited work in literature on workflows verification. In [9] a few verification issues of workflow structures have been examined and complexity of certain verification problems has been shown. In [12] some correctness issues in workflows modeling have been identified. In [1] the application of analysis techniques in Petri Nets domain has been explored for workflow verification.

The work presented in this paper differs from other approaches. It provides an effective approach and algorithm to gradually reduce a workflow graph through a set of reduction rules and allows visual identification of structural conflicts.

2 Process Modeling

To be able to present our reduction technique, we introduce a basic process modeling language that is based on generic modeling concepts as described in [15]. In this language, the process models are modeled using two types of objects: node and control flow.

Node is classified into two subclasses: task and condition. A *task*, graphically represented by a rectangle, represents the work to be done to achieve some objectives. It is also used to build sequential, and-split, and and-join structures. A *condition*, graphically represented by a circle, is used to construct or-split and or-join structures. A *control flow* links two nodes in the graph and is graphically represented by a directed edge.

By connecting nodes with control flows through five modeling structures, as shown in Figure 1, we build directed acyclic graphs (DAG) called workflow graphs where nodes represent vertices and control flows represent directed edges. From now on, we will refer to vertices as nodes and edges as flows.

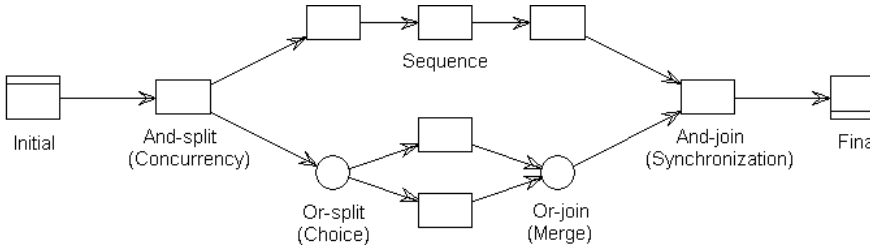

Fig. 1. Process modeling constructs

Sequence is the most basic modeling structure and defines the ordering of task execution. It is constructed by connecting at the most one incoming and one outgoing flow to a task. An *and-split* structure is used to represent concurrent paths within a workflow graph and is modeled by connecting two or more outgoing flows to a task. At certain points in workflows, it is essential to wait for the completion of more than one execution path to proceed further. An *and-join*, represented by more than one incoming flow to a task, is applied to synchronize such concurrent paths. An and-join task waits until all the incoming flows have been triggered.

An *or-split* structure is used to model mutually exclusive alternative paths and is constructed by attaching two or more outgoing flows to a condition object. At runtime, the workflow selects one of the alternative execution paths for a given instance of the business process by activating one of the outgoing flows originating from the or-split condition object. The or-split is exclusive and complete. The exclusive characteristic ensures that only one of the alternative paths is selected. The completeness characteristic guarantees that, if a condition object is activated, one of its outgoing flows will always be triggered. An *or-join* structure is "opposite" to the or-split structure. It is applied to join mutually exclusive alternative paths into one path by attaching two or more incoming flows to a condition object.

Since a workflow model is represented by a directed acyclic graph (DAG), it has at least one node that has no incoming flows (source) and at least one node that has no outgoing flows (sink). We call these *initial* and *final* nodes respectively. To uniquely identify a final node for a workflow graph, we join all split structures. Therefore, a workflow graph contains only one initial and one final node. A workflow instance completes its execution after its final node has completed its execution.

The generic process modeling language [15] also contains additional modeling structures to support nesting, blocks, and iteration. However, these are not discussed in this paper since the modeling approach used for these structures does not impact on the verification algorithm presented here. We recognize the importance of modeling and verifying other aspects like data flow, temporal constraints, execution, roles, and task classifications. However, this paper concentrates only on the verification of structural conflicts.

3 Structural Conflicts in Process Models

We identify two structural conflicts in process models: deadlock and lack of synchronization. As mentioned earlier, all split structures introduced after the initial node are closed through join structures before reaching the final node. That means an and-join is applied for joining and-split paths and an or-join for or-split paths. Joining exclusive or-split paths with an and-join results into a deadlock conflict. A deadlock at an and-join blocks the continuation of a workflow path since one or more of the preceding flows of the and-join are not triggered. Similarly, joining and-split concurrent paths with an or-join introduces lack of synchronization conflict. A lack of synchronization at an or-join node results into unintentional multiple activation of nodes that follow the or-join node.

Figure 2 shows a correct workflow graph and two graphs with deadlock and lack of synchronization structural conflicts. The conflict nodes are highlighted in the incorrect graphs.

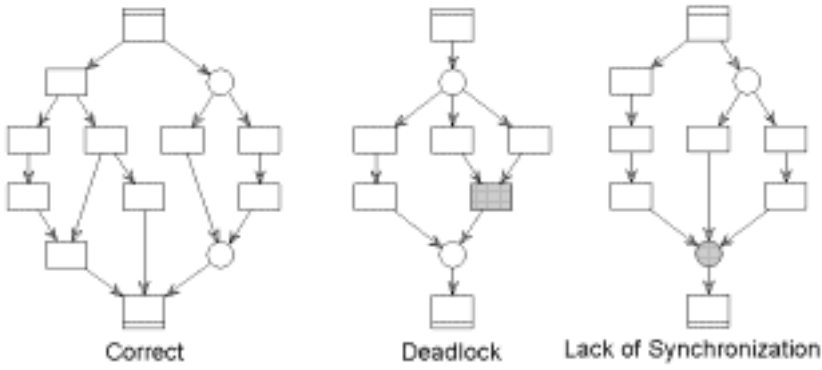

Fig. 2. Structural conflicts in workflow graphs

It is important to point out here that structural conflicts are not the only types of errors possible in process models. However, they do represent the primary source of errors in control flow specifications. Other modeling aspects may also affect the correct execution of workflows. For example, data flow modeling captures the data dependencies between activities. A dependent activity may not get the required data from the source activity at run-time because of incorrect modeling. The incorrect data mapping between activities and underlying application components may also cause incorrect execution of workflows. Similarly, control flow specification of a workflow model may not satisfy certain specified temporal constraints.

The existence of other errors in a workflow model does not introduce or remove structural conflicts. Therefore, the identification of structural conflicts in a workflow model can be performed independently from other types of verification analysis. The correctness of a complete workflow specification cannot be guaranteed just by removing structural conflicts from the workflow graph. However, the non-existence of structural conflicts in a workflow model guarantees that control flow specifications conform to a certain correctness criteria.

We partition the verification process for workflow structures into two phases. In first phase, basic syntax checking is performed to ensure that the model conforms to the modeling language syntax and that all necessary properties of its components have been defined. The verification of basic syntax is easy to facilitate and requires local analysis of workflow modeling objects and structures. An example of such verification is checking whether the workflow model is a DAG and is properly connected.

The second phase of verification requires a rigorous analysis of the workflow model. It attempts to identify inconsistencies in the model that could arise due to conflicting use of modeling structures. It is possible that such conflicting structures are placed at distant locations in the workflow graph. In such cases, it is difficult to identify the inconsistencies manually.

We need to define the concept of instance subgraphs before presenting the correctness criteria for workflow graphs. An instance subgraph represents a subset of workflow tasks that may be executed for a particular instance of a workflow. It can be generated by visiting its nodes on the semantic basis of underlying modeling structures. The subgraph representing the visited nodes and flows forms an instance subgraph. Figure 3 shows a workflow graph and its instance subgraphs.

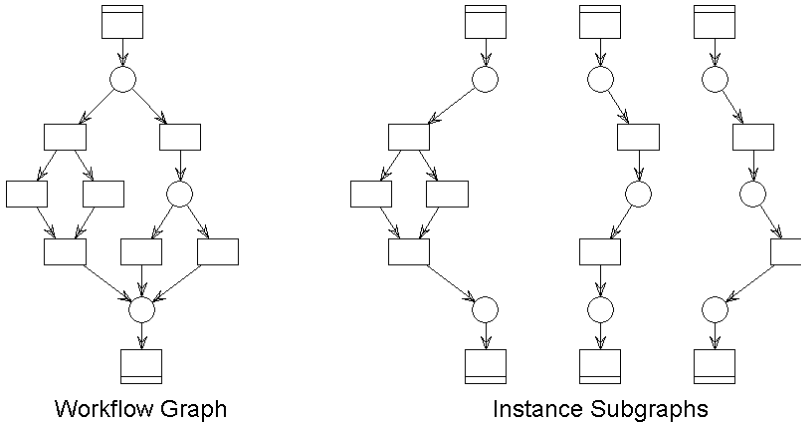

Fig. 3. A Workflow graph and its instance subgraphs

Correctness criteria 1 – deadlock free workflow graphs: A workflow graph is free of deadlock structural conflicts if it does not generate an instance subgraph that contains only a proper subset of the incoming nodes of an and-join node.

Correctness criteria 2 – lack of synchronization free workflow graphs: A workflow graph is free of lack of synchronization structural conflicts if it does not generate an instance subgraph that contains more than one incoming nodes of an or-join node.

The common point in both cases of verification is that in principle, we need to examine all possible instance subgraphs of a workflow. The or-split is the only structure in a workflow graph that introduces more than one possible instance subgraphs. A workflow graph without or-split structures would produce exactly the same instance subgraph as of the workflow graph. A workflow graph with a single or-split structure produces as many possible instance subgraphs as the number of outgoing flows from the or-split structure. However, the number of possible instance subgraphs could grow

exponentially as the number of or-split and or-join structures increases in a workflow specification. Therefore, a brute force method to generate all possible instance subgraphs of a workflow graph to ensure correctness is not computationally effective.

We present a formal notation of the workflow graphs that will be used in the verification algorithm as follows.

The workflow graph $G = (N, F)$ is a simple directed acyclic graph (DAG) where

- N is a finite set of nodes
- F is a finite set of control flows representing directed edges between two nodes

For each flow $f \in F$:

- $head[f] = n$ where $n \in N$ represents head node of f
- $tail[f] = n$ where $n \in N$ represents tail node of f

For each node $n \in N$:

- $type[n] \in \{$ TASK, CONDITION $\}$ represents type of n
- $dout[n]$ = out degree of n, i.e., number of outgoing flows from n
- $din[n]$ = in degree of n, i.e., number of incoming flows to n
- $Outflow[n] = \{ f : f \in F$ and $head[f] = n \}$, i.e., a set of outgoing flows from n
- $Inflow[n] = \{ f : f \in F$ and $tail[f] = n \}$, i.e., a set of incoming flows to n
- $Outnode[n] = \{ m : m \in N$ and $\exists f \in F$ where $head[f] = n$ and $tail[f] = m \}$, i.e., a set of succeeding nodes that are adjacent to n
- $Innode[n] = \{ m : m \in N$ and $\exists f \in F$ where $tail[f] = n$ and $head[f] = m \}$, i.e., a set of preceding nodes that are adjacent to n

The graph G meets following syntactical correctness properties:

- Condition nodes are not used in sequential structures, i.e., $\neg\exists\ n \in N$ where $type[n]$ = CONDITION and $din[n] \leq 1$ and $dout[n] \leq 1$
- G does not contain more than one initial node, i.e., $\exists\ n \in N$ where $din[n] = 0$ and
- ($\neg\exists\ m \in N$ where $din[m] = 0$ and $n \neq m$)
- G does not contain more than one final node: i.e., $\exists\ n \in N$ where $dout[n] = 0$ and
- ($\neg\exists\ m \in N$ where $dout[m] = 0$ and $n \neq m$)

3.1 Reduction Rules

In principle, the concept behind the verification approach and algorithm presented in this paper is simple. We remove all such structures from the workflow graph that are definitely correct. This is achieved by iteratively applying a conflict-preserving reduction process. The reduction process eventually reduces a structurally correct workflow graph to an empty graph. However, a workflow graph with structural conflicts is not completely reduced.

The reduction process makes use of three reduction rules – adjacent, closed, and overlapped – as long as they are able to reduce the graph.

Adjacent Reduction Rule

The adjacent reduction rule targets four types of components. We visit all nodes of the graph and check if applying adjacent reduction rule can reduce any of them.

Let us call the node being visited as the current node. We remove the current node from the graph if the number of flows attached to it is less than or equal to one. We also assume that when a node is removed from the graph, all flows attached to it are automatically removed. If the current node is forming a sequential structure, i.e., it has exactly one incoming and one outgoing flow, we change the tail of its incoming flow to the tail of its outgoing flow and remove it from the graph.

If the current node is not removed by first two criteria, it means that it is forming either a split or join structure since it would either have out degree or in degree or both that is more than one. We check if the current node has a single incoming flow and is introducing a split structure by having more than one outgoing flow. If the type of the current node is same as its preceding node, we move outgoing flows of the current node to the preceding node and remove the current node. Finally, if the last criterion is not met, we check if the current node has a single outgoing flow and is introducing a join structure through more than one incoming flow. If the type of the current node is same as its succeeding node, we move incoming flows of the current node to the succeeding node and delete the current node. This step is similar to the previous one except for the fact that it merges join structures whereas the previous step merges split structures. Figure 4 (a) shows examples of applying the adjacent reduction rule.

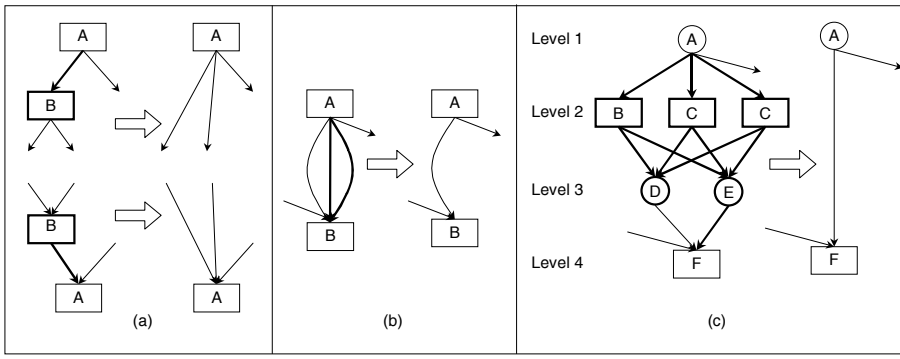

Fig. 4. Examples of reduction rules

Closed Reduction Rule

The application of adjacent reduction rule generally introduces closed components in workflow graphs. A closed component comprises two nodes of the same type that have more than one flow between them. The closed reduction rule deletes all but one flow between such nodes. Figure 4 (b) shows an example of the closed reduction rule. A graph may contain closed components only if some adjacent components are reduced.

Overlapped Reduction Rule

The overlapped reduction rule targets a specific class of components in workflow graphs that has an infrequent occurrence. Therefore, we invoke it only if the adjacent and closed reduction rules are unable to reduce the graph. An overlapped component of a workflow graph meets several properties that ensure non-existence of structural conflicts in it. Such a component has four levels as shown in Figure 4 (c). The source of the component at level 1 is always a condition and sink at level 4 is always a task. It has only task objects at level 2 and only condition objects at level 3. Each of the tasks at level 2 has outgoing flows to each of the conditions at level 3 and has exactly one incoming flow from the source at level 1. Each of the conditions at level 3 has incoming flows only from each of the tasks level 2 and has exactly one outgoing flow to the sink at level 4. The nodes at level 3 and 4 do not have any other control flows attached to them than the ones mentioned above. The overlapped reduction rule identifies components that meet all these properties and reduces them to a single control flow between source and sink of the component.

The workflow graph before reduction is assumed to meet syntactical correctness properties as described in previous section. However, during the reduction process, a reduced graph may not satisfy some of these properties. The adjacent reduction may introduce multiple flows between two nodes, hence transforming the simple graph into a multi-graph. In reduced graphs, multiple flows between two nodes represent existence of more than one reduced path between them. Similarly, the closed and overlapped reduction may introduce sequential condition nodes. Such a sequential condition node represent reduction of more than one or-split or or-join paths of the condition node into one. We allow the existence of these two syntactical errors in reduced workflow graphs.

3.2 Verification Algorithm

The three reduction rules have been combined in the following REDUCE(G) algorithm that takes a workflow graph G as input.

```
procedure REDUCE(G)
   lastsize ← size[G] + 1
   while lastsize > size[G] do
      lastsize ← size[G]
      for each node n ∈ N[G] do                    /* Adjacent components */
         if din[n] + dout[n] ≤ 1 then
            delete n
         else if din[n] = 1 and dout[n] = 1 then
               tail[top[Inflow[n]]] ← top[Outnode[n]]]
               delete n
            else if din[n] = 1 and dout[n] > 1 and type[n] = type[top[Innode[n]]] then
                  for each flow outflow ∈ Outflow[n] do
                     head[outflow] ← top[Innode[n]]
                  delete n
```

```
        else if dout[n] = 1 and din[n] > 1 and
                    type[n] = type[head[Outnode[n]]] then
            for each flow inflow ∈ Inflow[n] do
                tail[inflow] ← top[Outnode[n]]
            delete n
if lastsize < size[G] then                    /* Closed components */
    for each node n ∈ N[G] do
        if dout[n] > 1 then
            Nodeset ← {}
            for each flow outflow ∈ Outflow[n] do
                if type[n] = type[tail[outflow]] then
                    if tail[outflow] ∉ Nodeset then
                        Nodeset ← Nodeset ∪ { tail[outflow] }
                    else
                        delete outflow
if lastsize = size[G] then                    /* Overlapped components */
    for each node n ∈ N[G] do
        if type[n] = CONDITION and dout[n] = 1 and din[n] > 1 then
            level4 ← top[Outnode[n]]
            fn ← head[Innode[n]]
            if type[level4] = TASK and din[level4] > 1 and
                    type[fn] = TASK and dout[fn] > 1 and din[fn] = 1 then
                level1 ← head[Innode[fn]]
                if type[top] = CONDITION and dout[top] > 1 then
                    Level2 ← Innode[n]
                    Level3 ← Outnode[fn]
                    if ∀ node ∈ Level2 ( type[node] = TASK and
                            Innode[node] = { level1 } and
                            Outnode[node] = Level3 ) then
                        if ∀ node ∈ Level3 ( type[node] = CONDITION and
                                Outnode[node] = { level4 } and
                                Innode[node] = Level2 ) then
                            head[top[Outflow[n]]] ← level1
                            delete all node ∈ Level2
                            delete all node ∈ Level3
```

A workflow graph is shown in Figure 5 (a). The first application of adjacent rule reduces the workflow graph from (a) to (b). The closed rule removes multiple flows from closed components and transforms the graph to (c). The adjacent and closed rules are applied two more times to get (d). At this point, the adjacent and closed rules cannot reduce the graph any further. Therefore, the overlapped rule is applied to get (e). We get (f) by applying adjacent and closed rules two more times on (e). Finally, a single application of adjacent rule reduces the graph from (f) to an empty graph and hence showing that the workflow graph in (a) does not contain any structural conflicts.

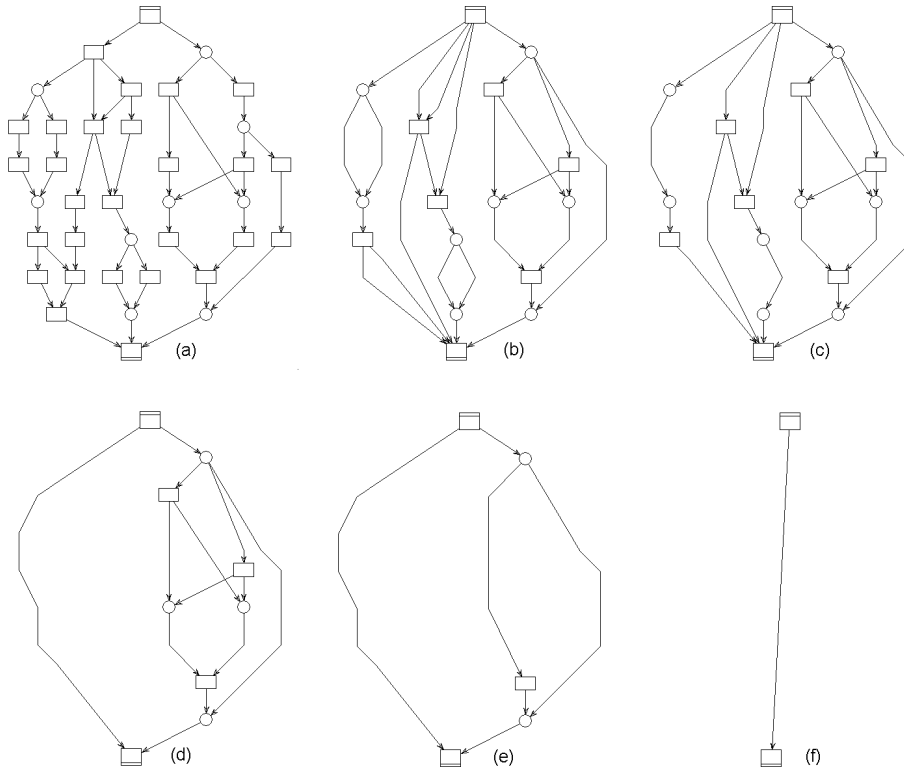

Fig. 5. Reducing a workflow graph containing structural conflicts

Figure 6 (a) shows a workflow graph where we have added an additional control flow to the workflow graph in 5 (a) that introduces a deadlock structural conflict. The reduction procedure is applied on (a) to get (b) that is not reducible any further. The non-reducible graph in (b) shows that a structural conflict exists in (a).

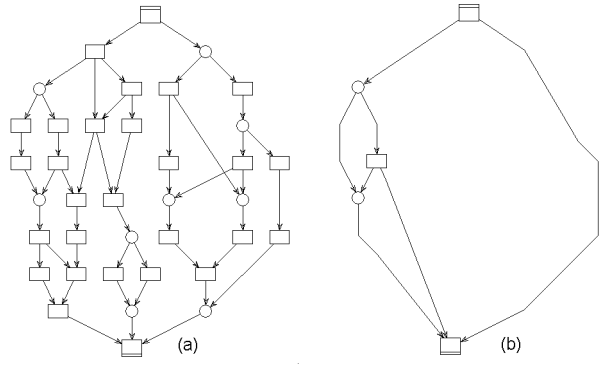

Fig. 6. Reducing a workflow graph with structural conflicts

3.3 Correctness and Complexity

In this section, we look at the correctness and complexity of the REDUCE(G) algorithm. Let the workflow graph G_i is the reduced graph after i iterations of the algorithm on G. We shall prove that:

- the adjacent, closed, and overlapped reduction rules do not generate structural conflicts, i.e., if G_i is correct then G_{i+1} is correct;
- the adjacent, closed, and overlapped reduction rules do not remove structural conflicts, i.e., if G_i is incorrect then G_{i+1} is incorrect; and,
- if G is correct, then reduction algorithm will always reduce G to an empty graph, equivalently, if reduction algorithm can not reduce G to an empty graph then G must contain at least one structural conflict.

First, we will look at individual reduction rules to prove that they meet the first two properties 1 and 2. The adjacent reduction rule targets four types of components as described in section 3.1. The nodes removed by the first two types represent sequential structures that do not introduce any split or join structures in workflow graph. Therefore, they cannot possibly contribute towards a structural conflict. At the same time, they cannot remove any existing structural conflict. The third type of adjacent reduction does not generate new structural conflicts since it only merges the split structure from the current node to the preceding node. The split structures are merged only if the type of the current node and the preceding node is same. If the current node's split structure contributes to a structural conflict that occurs in its succeeding paths, it would not be removed since the split structure is simply moved to the preceding node without any structural differences. At the same time, the current node has only a single preceding node and hence does not take part in a join structure. Therefore, the current node cannot be a part of a structural conflict with a split structure that is in its preceding paths. The fourth type of adjacent reduction is similar to the third one except that it merges join structures rather than split structures. Therefore, it also meets the first two correctness properties on similar basis.

We do not allow multiple flows between two nodes in a workflow graph. However, the adjacent reduction generally introduces components in reduced workflow graph that contain two nodes and more than one flow between them. Such multiple flows imply the existence of more than one path from the head node of the flows to their tail node. If the type of such two nodes is different, then the component represents a case of structural conflict and is not reduced by the algorithm. However, if the type is same, we remove all but one of the multiple flows since they cannot generate structural conflicts or remove existing ones. Multiple flows between two tasks represent concurrent paths and between two conditions represent alternative paths.

The basic property of an overlapped component is that none of the nodes between source and sink of the component are connected to any other nodes of the graph. At the same time, no structural conflict exists between source and sink. Therefore, reducing the whole overlapped component between sink and source of an overlapped component to a single control flow does not generate structural conflicts or remove existing ones.

So far, we have shown that all reduction rules of the algorithm meet first two correctness properties. To prove that algorithm meets the third property, we will show that if the algorithm does not reduce a workflow graph to an empty graph, then it contains at least one structural conflict.

Let G_k be a graph that is not reducible any further after applying k iterations of the reduction algorithm. Non-reducibility of graph G_k implies that it does not contain any sequential structures, i.e., either in degree or out degree or both of each graph node is more than one. It also implies that if a node has an in degree or out degree that is equal to one then the adjacent node attached to the single incoming or outgoing flow is of different type.

It is also possible that graph G_k contains a node that has more than one flow to another node. Non-reducibility of graph implies that such multiple flows could exist only between nodes of different types. Multiple flows from a condition to a task represent deadlock conflict and from a task to a condition represent lack of synchronization conflict. Therefore, if a non-reducible graph contains a node with multiple flows to another node then it always contains a structural conflict.

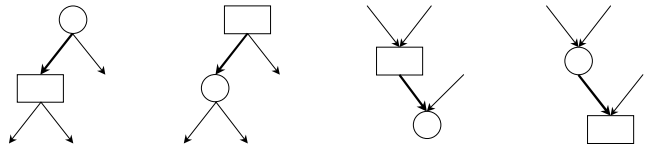

Fig. 7. Structures with single in / out degrees in non-reducible graphs

To proceed, we assume that G_k does not contain a node with more than one flow to another node, i.e., G_k is a simple DAG. We also know that G_k has a single source and a single sink. We know that a simple DAG with a single source and a single sink contains at least one node with a single in degree and another node with a single out degree. This property holds even for complete graphs. Since G_k is non-reducible, then the node on the other side of such a single incoming or outgoing flow is always of a different type, otherwise it would be reducible by adjacent reduction. There are only four possible cases of such structures as shown in Figure 7. It is easy to show that it is not possible to build a non-reducible correct graph with a single source and a single sink that contains any of these four structures. Therefore, a non-reducible graph always contains at least one structural conflict.

Now we look at the time complexity of the algorithm. In each iteration of the algorithm, we visit all nodes to identify and reduce graph objects that meet certain properties of the reduction rules. The checking of whether a graph object meets any of these properties is done in constant time since it takes only those objects into account that are adjacent to the current node. After each iteration, the algorithm continues only if the graph has reduced.

The worst case complexity of the algorithm is $O(n^2)$ where n represents the number of nodes and flows in the workflow graph. The worst case is for a workflow graph that is completely reducible and each iteration of the algorithm is able to reduce at the most one object. However, the average case complexity is much lower than $O(n^2)$, since the first few iterations dramatically reduce the size of a workflow graph and remaining iterations need to work on a much smaller workflow graph than the original graph.

4 Implementation

The algorithm presented in this paper has been implemented in a workflow modeling and verification tool. The tool, called FlowMake, provides workflow analysts and designers a well-defined framework to model and reason about various aspects of workflows. It has been designed to augment production workflow products with enhanced modeling capabilities and to provide a basis for expanding the scope of the verification. Figure 8 shows a screen snapshot of FlowMake where the reduction process is being applied to a process model.

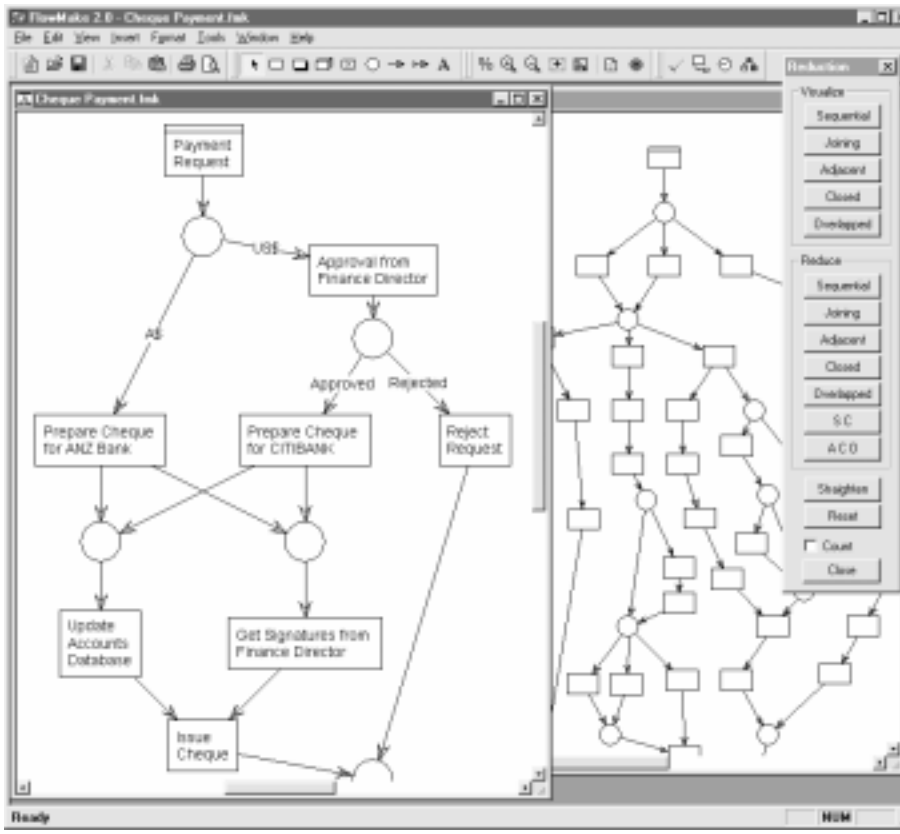

Fig. 8. FlowMake: workflow modeling and verification tool

FlowMake is composed of four major components: the repository, the workflow editor, the verification engine, and the interface. *Repository* maintains workflow models and has been implemented using relational technology. *Workflow Editor* provides a user-friendly graphical environment to maintain large workflow graphs. It is also used to visualize inconsistencies in design. *Verification Engine* implements the algorithms to check the consistency of workflow models. *Interface* component provides linkage to workflow products through import and export of workflow models. Presently, the tool allows importing of process models from IBM workflow product MQ

Workflow (previously known as FlowMark), analyzing them for structural conflicts, and exporting them back to the product. More information about FlowMake is available at http://www.dstc.edu.au/DDU/projects/FlowMake/.

5 Conclusions

We report on successful implementation of graph reduction techniques for detecting structural conflicts in process models. The implementation provides interface to a selected workflow product, IBM MQ Workflow, to demonstrate its applicability to a leading workflow management system.

To present the verification approach and algorithm, we introduced a basic process modeling language based on a process definition standard by Workflow Management Coalition. The language makes use of five modeling structures – sequence, and-split, and-join, or-split, and or-join – to build control flow specifications. We have identified two types of structural conflicts, deadlock and lack of synchronization, which could compromise the correctness of process models. The identification of these structural conflicts in workflow models is a complex problem. We have presented an effective graph reduction algorithm that can detect the existence of structural conflicts in workflow graphs. The basic idea behind verification approach is to remove all such structures from the workflow graph that are definitely correct. The algorithm reduces a workflow graph without structural conflicts to an empty graph. However, a workflow graph with structural conflicts is not completely reduced and structural conflicts are easily identifiable. The incremental reduction of workflow model also allows analysis of workflow graph components.

The main contribution of the paper is a new technique for identifying structural conflicts in process models. We believe that the ideas presented in this paper provide a basis for expanding the scope of verification in workflow products. The visual approach for identifying structural conflicts is useful, intuitive, and natural.

References

1. Aalst. WMP van der (1997). Verification of Workflow Nets. In P. Azema and G. Balbo, editors, Application and Theory of Petri Nets 1997, volume 1248 of Lecture Notes in Computer Science, pages 407-426. Springer-Verlag, Berlin, 1997.
2. Aalst. WMP van der (1998). The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers, 8(1):21--66, 1998.
3. Butler Report. Workflow: Integrating the Enterprise. The Butler Group, 1996.
4. Reichert M and Dadam P (1997). ADEPTflex - Supporting Dynamic Changes of Workflow without loosing control. Journal of Intelligent Information Systems (JIIS), Special Issue on Workflow and Process Management.
5. Carlsen S (1997). Conceptual Modeling and Composition of Flexible Workflow Models. PhD Thesis. Department of Computer Science and Information Science, Norwegian University of Science and Technology, Norway, 1997.
6. Casati F, Ceri S, Pernici B and Pozzi G (1995). Conceptual Modeling of Workflows. In M.P. Papazoglou, editor, Proceedings of the 14th In-ternational Object-Oriented and Entity-Relationship Modeling Conference, vol-ume 1021 of Lecture Notes in Computer Science, pages 341-354. Springer-Verlag.

7. Ellis CA and Nutt GJ (1993). Modelling and Enactment of Workflow Systems. In M. Ajmone Marasan, editor, Application and Theory of Petri Nets, Lecture Notes in Computer Science 691, pages 1-16, Springer-Verleg, Berlin, 1993.
8. Georgakopoulos D, Hornick M and Sheth A (1995) An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Journal on Distributed and Parallel Databases, 3(2):119-153.
9. Hofstede, AHM ter, Orlowska ME and Rajapakse J (1998). Verification Problems in Conceptual Workflow Specifications. Data & Knowledge Engineering, 24(3):239-256, January 1998.
10. Kuo D, Lawley M, Liu C and Orlowska ME (1996). A General Model for Nested Transactional Workflow. In Proceedings of the International Workshop on Advanced Transaction Models and Architecture (ATMA'96), Bombay India, pp.18-35, 1996.
11. Rajapakse J (1996). On Conceptual Workflow Specification and Verification. MSc Thesis. Department of Computer Science, The University of Queensland, Australia, 1996.
12. Sadiq W and Orlowska ME (1997). On Correctness Issues in Conceptual Modeling of Workflows. In Proceedings of the 5th European Conference on Information Systems (ECIS '97), Cork, Ireland, June 19-21, 1997.
13. Sadiq W and Orlowska ME (1999). On Capturing Process Requirements of Workflow Based Information Systems. In Proceedings of the 3rd International Conference on Business Information Systems (BIS '99), Poznan, Poland, April 14-16, 1999.
14. Workflow Management Coalition (1996) The Workflow Management Coalition Specifications - Terminology and Glossary. Issue 2.0, Document Number WFMC-TC-1011.
15. Workflow Management Coalition (1998). Interface 1: Process Definition Interchange, Process Model, Document Number WfMC TC-1016-P.