# ANALYZING PROCESS MODELS USING GRAPH REDUCTION TECHNIQUES

WASIM SADIQ AND MARIA E. ORLOWSKA

Distributed Systems Technology Centre
Department of Computer Science & Electrical Engineering
The University of Queensland, Australia

**Abstract** — The foundation of a process model lies in its structural specifications. Using a generic process modeling language for workflows, we show how a structural specification may contain deadlock and lack of synchronization conflicts that could compromise the correct execution of workflows. In general, identification of such conflicts is a computationally complex problem and requires development of effective algorithms specific for the target modeling 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 the given workflow modeling language. We also provide insights into the correctness and complexity of the reduction process. Finally, we show how the reduction algorithm may be used to count possible instance subgraphs of a correct process model. The main contribution of the paper is a new technique for satisfying well-defined correctness criteria in process models.

*Key words:* Business process modeling, Verification of process models, Workflow management systems.

## 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 a means for automated coordination of activities that may span over several heterogeneous and mission-critical information systems of an organization. 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 the 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. There are several aspects of a process model including structure, data flow, roles, application interface, temporal constraints, and others. The structural aspect of a workflow defines the way a workflow management system would order and schedule workflow tasks. Arguably, process structure is the primary and the most important aspect of a process model. It builds the foundation for capturing other aspects of the workflow requirements.

A workflow management system coordinates the execution of workflow tasks to achieve some business 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 structural aspect of a process model captures the flow of execution from one task to another.

The research presented in this paper addresses modeling and verification of production workflows. Various approaches to workflow modeling can be found in literature [4, 5, 6, 10, 12, 13]. The use of Petri Nets for workflow modeling has been explored in [1] and [7]. However, Petri Nets are not directly used in any of the major products. 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.

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 reduction and synthesis techniques have also been explored in Petri Nets theory for analysis of their well-formadness [17].

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. We will also show how the reduction rules may be used to count instance subgraphs of a process model. The concept of instance subgraph will be introduced later.

## 2. PROCESS MODELING STRUCTURES

To be able to present our graph reduction techniques for process model analysis, 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 transition. Node is classified into two subclasses: task and choice/merge coordinator. Figure 1 (a) shows graphical representation of modeling objects.

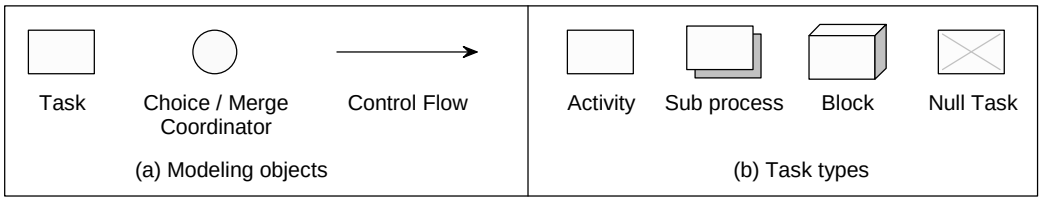


Fig. 1. Process modeling objects and task types

A *task*, graphically represented by a rectangle, represents the work to be done to achieve some objectives. It is also used to implicitly build sequence, fork, and synchronizer structures. It is the primary object in workflow specifications and could represent both automated and manual activities. Tasks are performed by assigned performers. The workflow specifications place less emphasis on the internal workings of tasks. Their aim is to capture the coordination requirements for performing a set of tasks for a given business process. They do, however, capture some information about the execution of tasks that are needed for coordination. All other modeling objects except the task are internal to the workflow management system and are used to specify the rules and constraints for the coordination of workflow execution. Tasks are further classified into four types: activity, sub process, block, and null task. An *activity* is an atomic piece of work coordinated by the workflow management system. The modeling and need for other three types of tasks will be discussed later in this section. Figure 1 (b) shows graphical representation of task types

A *choice/merge coordinator*, graphically represented by a circle, is used to explicitly build choice and merge structures. A *transition* links two nodes in the graph and is graphically represented by a directed edge. It shows the execution order and flow between its tail and head nodes.

By connecting nodes with transitions through modeling structures, as shown in Figure 2, we build directed acyclic graphs (DAG) called workflow graphs where nodes are represented by vertices and transitions are represented by directed edges. From now on, we will refer to vertices as nodes and edges as transitions.

*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 transition to a task.

A *fork* (and-split) structure is used to represent concurrent paths within a workflow graph and is modeled by connecting two or more outgoing transitions to a task. A fork does not enforce concurrent execution of the nodes that follow its outgoing transitions. It merely models the execution independence between concurrent paths. At run-time, the forked nodes are triggered at the same time but they may be performed later any time independent from each other. At certain points in workflows, it is essential to wait for the completion of more than one concurrent execution path to proceed further. A *synchronizer*

(and-join) structure, represented by attaching more than one incoming transition to a task, is applied to synchronize such concurrent paths. A task waits until all the incoming transitions have been triggered.

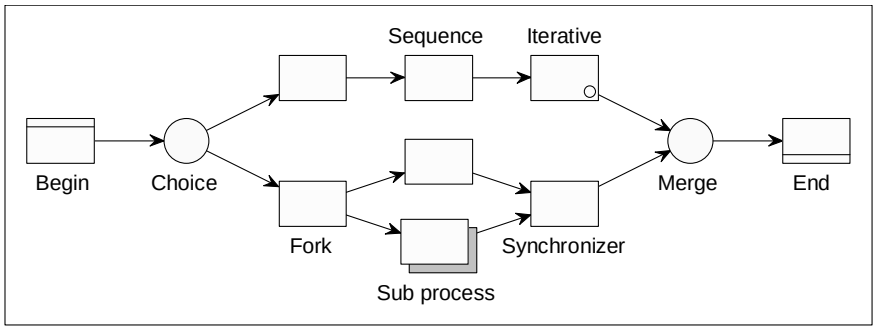


Fig. 2. Process modeling structures

A *choice* (or-split) structure is used to model mutually exclusive alternative paths and is constructed by attaching two or more transitions to a choice/merge coordinator object. At run-time, the workflow selects one of the alternative execution paths for a given instance of the business process by activating one of the transitions originating from the choice coordinator object. The choice structure is exclusive and complete. The exclusive characteristic ensures that only one of the alternative paths is selected. The completeness characteristic guarantees that, if a choice coordinator object is activated, one of its outgoing flows will always be triggered. There are several ways to ensure at the workflow engine level that a choice structure always triggers exactly one of its outgoing transitions. Generally, the choice structure uses workflow control data to choose one of its outgoing transitions. One of the ways to model choice structure is to assign a boolean transition condition to each of the outgoing transitions of the choice structure. At run-time, each of these expressions is evaluated and workflow engine selects one of the alternative workflow execution paths by triggering the appropriate outgoing transition from the choice coordinator.

A *merge* (or-join) structure is "opposite" to the choice structure. It is applied to join mutually exclusive alternative paths into one path by attaching two or more incoming transitions to a merge coordinator object.

It is possible to use a choice/merge coordinator that represents a merge structure to represent a choice structure as well. Similarly, the same task object may be used to represent a fork structure, a synchronizer structure or both.

We have choice / merge coordinator object to explicitly model choice and merge structures. However, fork and synchronizer structures are represented implicitly by directly connecting transitions to task objects. More than one outgoing transition from a task represents fork structure and more than one incoming transition to a task represents synchronizer structure. This approach keeps the resulting workflow model compact as well as graphically explicit. Nevertheless, in certain cases, it requires the use of *null* or router tasks to model proper coordination of flow and to conform to the syntactical correctness criteria of workflow structures. A null task is graphically represented by two shaded lines drawn between opposite corners of the rectangle and intersecting in the middle.

Since a workflow model is represented by a directed acyclic graph (DAG), it has at least one node that has no incoming transitions (source) and at least one node that has no outgoing transitions (sink). We call these *begin* and *end* nodes respectively. To uniquely identify a single end node for a workflow graph, we need to join all split structures. In such a case, a workflow graph contains only one begin and one end node. A workflow instance completes its execution after its end node has completed its execution. A bar at the top of a task or choice/merge coordinator represents a begin node. Similarly, a bar at the bottom represents an end node.

The *nesting* structure simplifies the workflow specifications through abstraction. Using this construct, we can encapsulate a workflow specification into a task and then use that nested task in other workflow specifications. For each execution of a nested task, the underlying workflow is executed. In this modeling language we differentiate between two types of nesting structures: sub process and block. A *sub process* task links to an externally defined process definition. At run time, for each execution of the sub process task, a process instance is created for the associated process definition. The externally defined process definition can instantiate independently as well as from the processes where it has been linked. A sub

process task is graphically represented through a shaded rectangle under the task rectangle. A *block task* represents an "in-line" sub process definition. The sub process structure allows us to reuse process definition as part of other process definitions through sub process task nodes. A block is a restrictive form of a sub process since it is not a separate process definition that can be reused. Block is a useful structure in simplifying complex process definitions and in modeling blocked iteration. A block task is graphically represented through a block shape task rectangle

The *iteration* structure is needed to model the repetition of a group of tasks within a workflow. One way to support iteration is through exit conditions. As long as a certain condition is not met, a particular task is repeatedly executed. The nesting structure could be used if there is a need to repeat a sub graph of the workflow model. A task that has an exit condition defined for iterative purposes is graphically represented through a small circle at the bottom right corner of the task rectangle.

A variety of process modeling languages exist both in research papers and commercial products [3]. Most of the languages support the generic modeling structures introduced here. We have defined this simple language to provide a basis for introducing the concept of structural conflicts and associated reduction rules for identifying them in a process model. However, the reduction principles for process model verification are applicable to other forms of process modeling languages after appropriately modifying the reduction rules.

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 paths introduced after the begin node through split structures are properly joined through join structures before reaching the end node. The synchronizer structures are applied to join fork paths and merge structures to join choice paths.

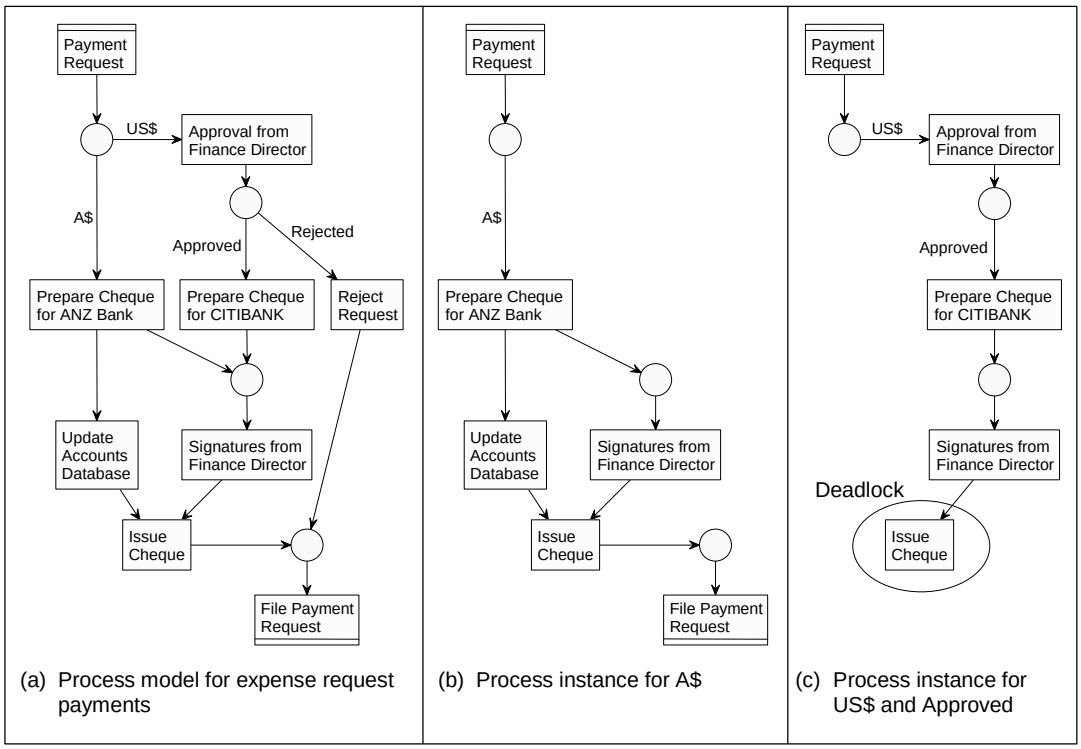


Fig. 3. Example process model with deadlock structural conflict

- *Deadlock* – Joining exclusive choice paths with a synchronizer results into a deadlock conflict. A deadlock at a synchronizer structure blocks the continuation of a workflow path since one or more of the preceding transitions of the synchronizer are not triggered.
- *Lack of Synchronization* – Joining fork concurrent paths with a merge structure introduces lack of synchronization conflict in the process model. A lack of synchronization at a merge structure results into unintentional multiple activation of nodes that follow the merge node.

Figure 3 (a) shows a workflow graph with deadlock structural conflict. This process model executes a subset of defined activities depending on the paths chosen at runtime through choice structures. In Figure 3 (b), one such possible instance is shown where expense payment request is for A$. This instance would execute successfully. However, as shown in Figure 3 (c), if the request is for US$ and is also approved by the Finance Director, the process instance would deadlock at *Issue Cheque* activity. According to the process model definition, *Issue Cheque* activity can only execute after synchronizing the completion of *Update Accounts Database* and *Signatures from Finance Director* activities. In this instance, because of incorrect modeling, *Update Accounts Database* will never be executed. This particular instance would not terminate properly and would result in incomplete execution of the business process.

For this particular example, there are three possible types of instances that can be executed at runtime. When we analyze these instance types individually, we realize that one of the instance types (US$ and Approved) would result into a deadlock and the other two (A$; US$ and Rejected) would execute correctly. For this simple example, it is not difficult to manually look at all possible instance types. However, for larger workflows, it is not feasible to generate all possible instance types of a process model to identify if any of them contain structural conflicts.

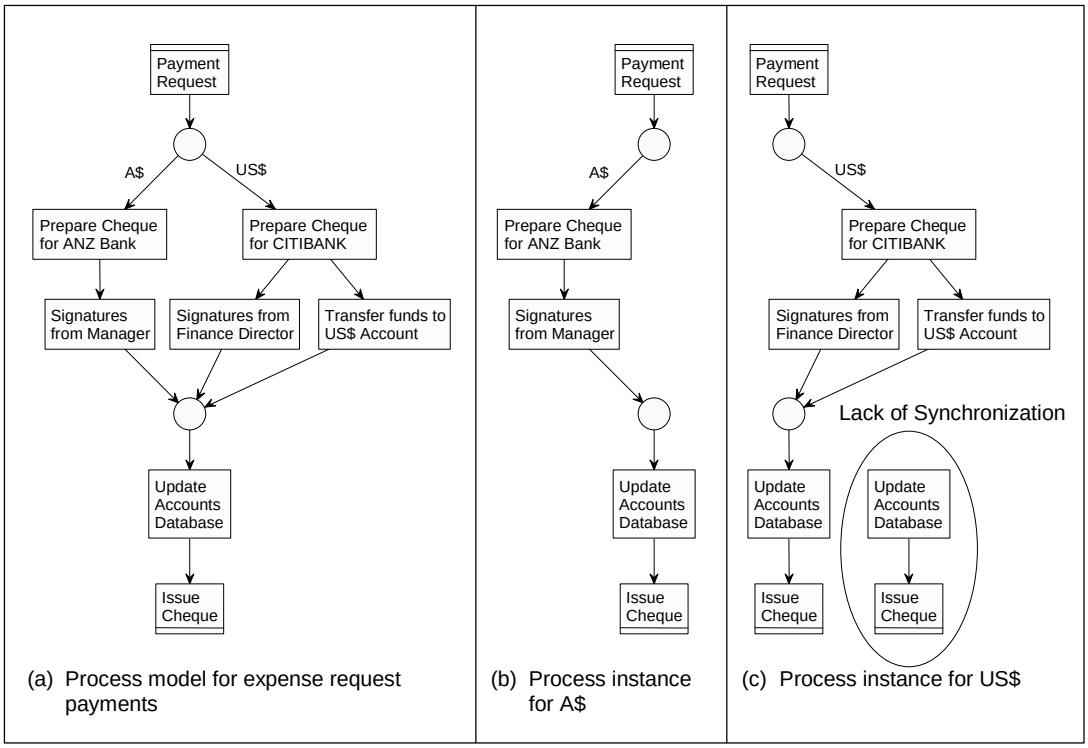


Fig. 4. Example process model with lack of synchronization structural conflict

Figure 4 (a) shows a workflow graph with the lack of synchronization structural conflict. In Figure 4 (b), a possible process instance is shown where expense payment request is for A$. This instance would execute correctly. However, as shown in Figure 4 (c), if the request is for US$, the process instance would create multiple instances of *Update Accounts Database* and *Issue Cheque* activities since two parallel activities *Signatures from Finance Director* and *Transfer funds to US$ Account* are not synchronized before the merge structure.

Figure 5 shows the correct process models for examples shown in Figure 3 and Figure 4. These two models do not contain any deadlock or lack of synchronization structural conflicts.

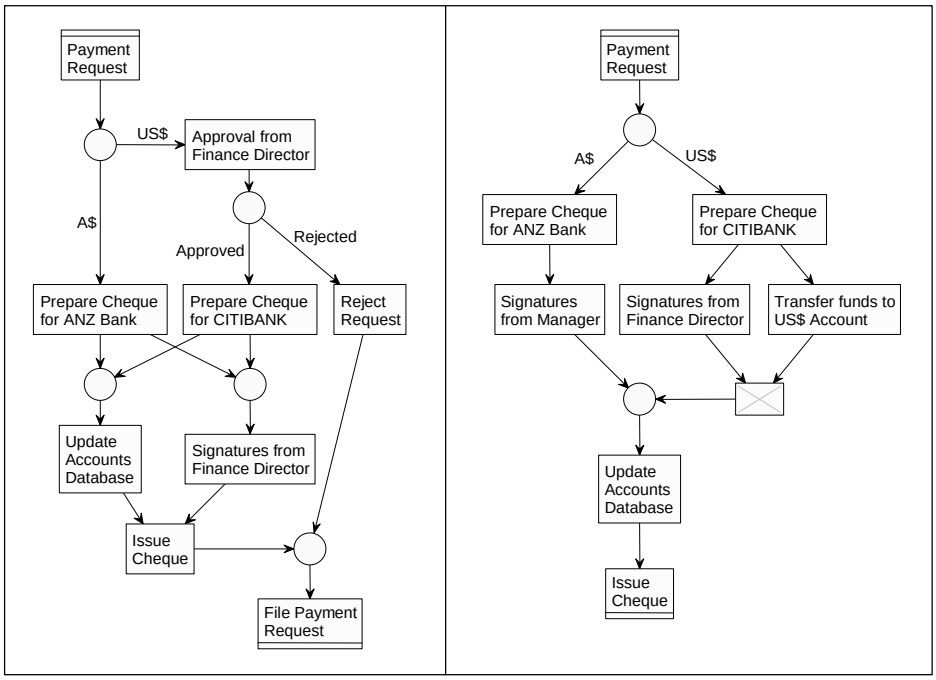


Fig. 5. Correct process models without structural conflicts

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 structural 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, structural 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 structural 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 6 shows a workflow graph and its instance subgraphs.

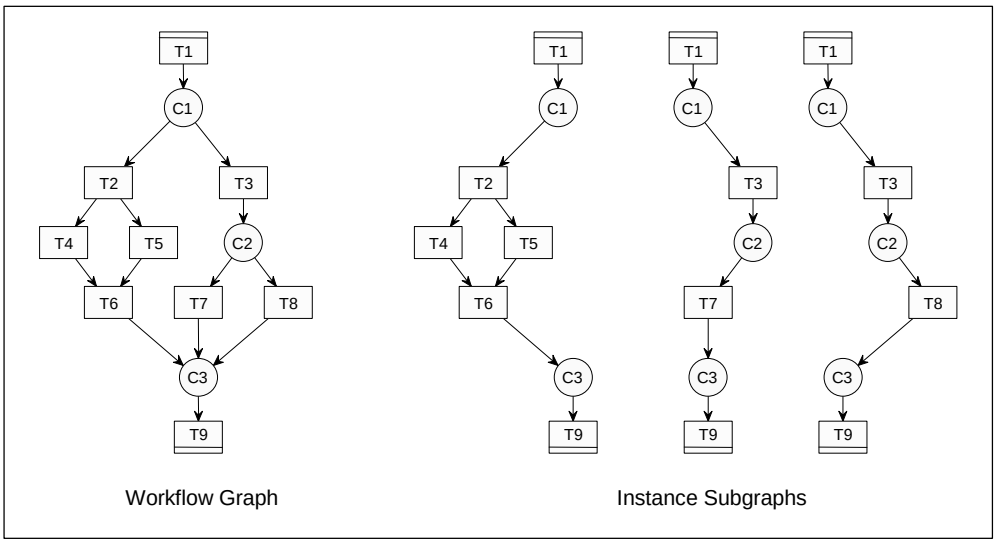


Fig. 6. A Workflow graph and its instance subgraphs

We now give the correctness criteria for workflow graphs:

- *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 a synchronizer 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 a merge node.

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

## 4. VERIFICATION OF STRUCTURAL CONFLICTS

In this section, we will use graph notations to represent workflow models and call them workflow graphs. We present a formal notation of the workflow graphs that will be used in the verification algorithm that follows. The notion of workflow graphs to be discussed here is based on the generic class of workflow modeling languages that we have introduced in section 2.

A process model based on this modeling language conforms to the following basic properties:

- it uses only core modeling structures, namely, sequence, choice, merge, fork, synchronization, begin/end, nesting, and iteration;
- it does not have any cycles and the iteration structure is supported only through blocked iteration; and,
- it has exactly a single begin and a single end task.

Each object (node and transition) of the workflow graph has some associated attributes. These attributes could be singular values or sets of other values or objects. We will use these attributes to define modeling structures and associated correctness criteria in graph notation. Later on these attributes will also be used in the specification of verification algorithms.

Attributes of objects are accessed using the attribute name followed by the name of the object in square brackets, i.e., an attribute definition is of the form $a[o] = v$, where $a$ represents the name of a particular attribute of object $o$ and $v$ represents the value of the attribute. If the value of an attribute is a singular value, its name starts with a lower case alphabetical letter. If value is a set, the name starts with an upper case letter.

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

- $N$ is a finite set of nodes
- $T$ is a finite set of transitions representing directed edges between two nodes
- $size[G] = size[N] + size[T]$ represents the total number of nodes and transitions in $G$

For each transition $t \in T$:

- $fromNode[t] = n$ where $n \in N$ represents from node of $t$
- $toNode[t] = n$ where $n \in N$ represents to node of $t$

For each node $n \in N$:

- $nodeType[n] \in \{$ TASK, CM $\}$ represents type of $n$, here CM stands for Choice / Merge coordinator
- $dout[n]$ = out degree of $n$, i.e., number of outgoing transitions from $n$
- $din[n]$ = in degree of $n$, i.e., number of incoming transitions to $n$
- $OutTrans[n] = \{ t : t \in T$ and $fromNode[t] = n \}$, i.e., a set of outgoing transitions from $n$
- $InTrans[n] = \{ t : t \in T$ and $toNode[t] = n \}$, i.e., a set of incoming transitions to $n$
- $OutNodes[n] = \{ m : m \in N$ and $\exists\ t \in T$ where $fromNode[t] = n$ and $toNode[t] = m \}$, i.e., a set of succeeding nodes that are adjacent to $n$
- $InNodes[n] = \{ m : m \in N$ and $\exists\ t \in T$ where $toNode[t] = n$ and $fromNode[t] = m \}$, i.e., a set of preceding nodes that are adjacent to $n$

The graph $G$ meets following syntactical correctness properties:

- Choice / merge coordinator nodes are not used in sequential structures, i.e., $\neg\exists\ n \in N$ where $nodeType[n] =$ CMC and $din[n] \leq 1$ and $dout[n] \leq 1$
- $G$ does not contain cycles, i.e., $\forall\ n_i, n_j \in N$, a path from $n_i$ to $n_j$ implies that $n_i \neq n_j$ (no self-loops) and no path exists from $n_j$ to $n_i$ (no cycles)
- $G$ does not contain more than one begin 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 end node: i.e., $\exists\ n \in N$ where $dout[n] = 0$ and ( $\neg\exists\ m \in N$ where $dout[m] = 0$ and $n \neq m$ )

### *4.1. 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 five reduction rules – terminal, sequential, adjacent, closed, and overlapped – as long as they are able to reduce the graph. We apply these reduction rules by visiting all nodes of the graph and checking if a reduction rule can be applied. We will call the node being visited as the current node while describing the reduction rules. We also assume that when a node is removed from the graph, all transitions attached to it are automatically removed.

#### *4.1.1. Terminal Reduction Rule*

The terminal reduction rule removes begin and end nodes from workflow graph only if they are not part of fork, choice, synchronizer, or merge structures. We remove the current node from the graph if the number of transitions attached to it is less than or equal to one.

Figure 7 shows an example of applying terminal reduction where we remove task *T1* to get a reduced graph from (a) to (b).

### *4.1.2. Sequential Reduction Rule*

If the current node being visited is forming a sequential structure, i.e., it has exactly one incoming and one outgoing flow, we change the *toNode* of its incoming transition to the *toNode* of its outgoing transition and remove the current node from the graph.

Figure 7 shows two examples of applying sequential reduction. First, we remove tasks *T3*, *T4*, *T5*, *T6*, and *T7* to get a new reduced graph from (b) to (c). It is applied again to remove choice / merge sequential nodes in (d) to get a new graph in (e).

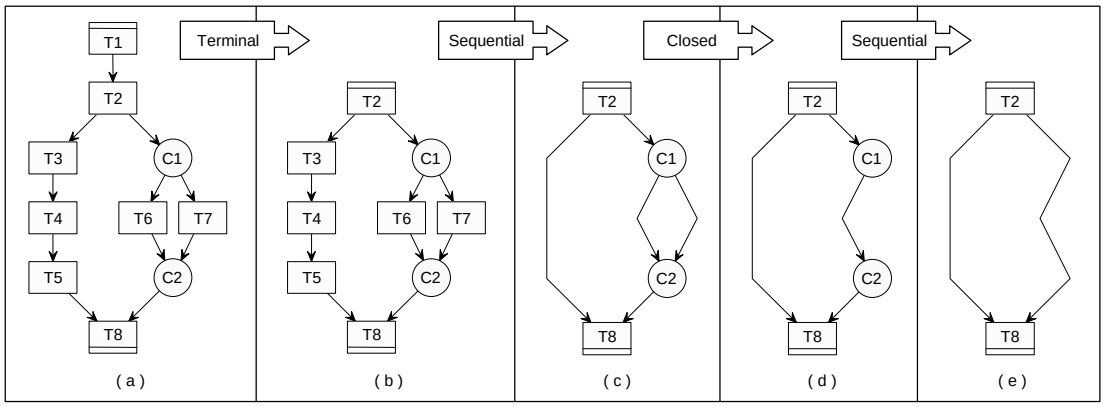


Fig. 7. Applying terminal, sequential and closed reduction rules

### *4.1.3. Adjacent Reduction Rule*

The adjacent reduction rule targets two types of components. If the current node is not removed by terminal and sequential reduction rule, 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. In this case, we check if the current node has a single incoming transition and is introducing a split structure by having more than one outgoing transition. If the type of the current node is same as its preceding node, we move outgoing transitions of the current node to the preceding node and remove the current node.

If the last criterion is not met, we check if the current node has a single outgoing transition and is introducing a join structure through more than one incoming transition. If the type of the current node is same as its succeeding node, we move incoming transitions 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.

In Figure 8 shows an example where we remove *T2*, *T8*, *C2*, and *C3* to get a reduced graph from (a) to (b) after applying the adjacent reduction rule. Application of sequential and closed reduction rules on (b) will further reduce the graph.

### *4.1.4. Closed Reduction Rule*

The application of sequential and adjacent reduction rules generally introduces closed components in workflow graphs. A closed component comprises two nodes of the same type that have more than one transition between them. The closed reduction rule deletes all but one transition between such nodes.

Figure 7 shows an example of the closed reduction rule. We remove one of the two transitions between *C1* and *C2* to get a reduced graph from (c) to (d). Similarly, one of the two transitions between *T2* and *T8* in (e) may also be removed using this rule. It is important to note that a graph may contain closed components only if some sequential and adjacent components are reduced.

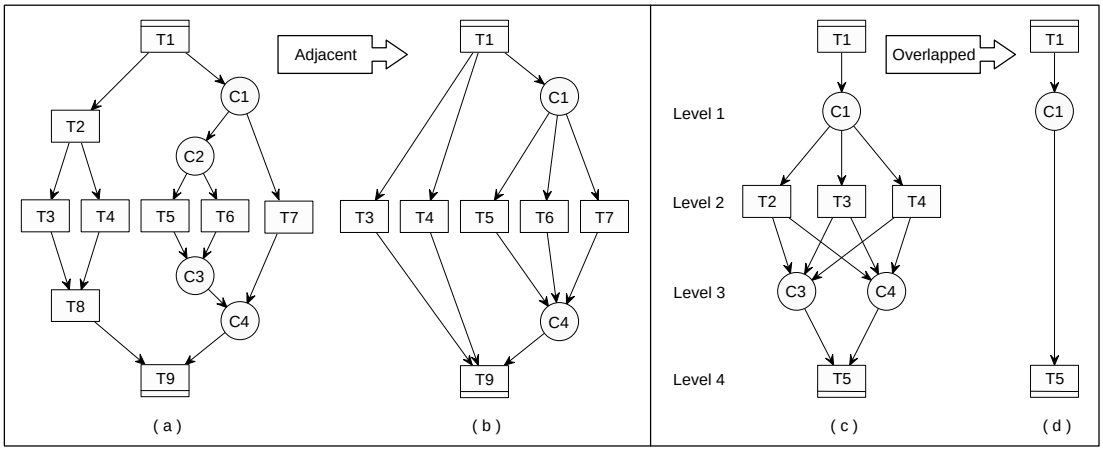


Fig. 8. Applying adjacent and overlapped reduction rules

*4.1.5. 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 8 (c).

The source of the component at level 1 is always a choice / merge and sink at level 4 is always a task. It has only task objects at level 2 and only choice / merge objects at level 3. Each of the task nodes at level 2 has outgoing transitions to each of the choice / merge nodes at level 3 and has exactly one incoming transition from the source at level 1. Each of the choice / merge node at level 3 has incoming transitions only from each of the task nodes at level 2 and has exactly one outgoing transition to the sink at level 4. The nodes at level 3 and 4 do not have any other transitions 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 transition 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 sequential and adjacent reduction rules may introduce multiple transitions between two nodes, hence transforming the simple graph into a multi-graph. In reduced graphs, multiple transitions 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 choice or merge paths of the choice / merge node into one. We allow the existence of these two syntactical errors in reduced workflow graphs.

*4.2. Verification Algorithm*

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

**procedure** REDUCE($G$)
  $lastsize \leftarrow size[G] + 1$
  **while** $lastsize > size[G]$ **do**
    $lastsize \leftarrow size[G]$

    /* Terminal, sequential and adjacent reduction */

    **for** each node $n \in N[G]$ **do**
      **if** $din[n] + dout[n] \leq 1$ **then**
        delete $n$
      **else if** $din[n] = 1$ and $dout[n] = 1$ **then**

```
            toNode[top[InTrans[n]]] ← top[OutNodes[n]]]
            delete n
        else if din[n] = 1 and dout[n] > 1 and nodeType[n] = nodeType[top[InNodes[n]]] then
            for each transition t ∈ OutTrans[n] do
                fromNode[t] ← top[InNodes[n]]
            delete n
        else if dout[n] = 1 and din[n] > 1 and nodeType[n] = nodeType[top[OutNodes[n]]] then
            for each transition t ∈ InTrans[n] do
                toNode[t] ← top[OutNodes[n]]
            delete n

/* Closed reduction */

if lastsize < size[G] then
    for each node n ∈ N[G] do
        if dout[n] > 1 then
            NodeSet ← {}
            for each transition t ∈ OutTrans[n] do
                if nodeType[n] = nodeType[toNode[t]] then
                    if toNode[t] ∉ NodeSet then
                        NodeSet ← NodeSet ∪ { toNode[t] }
                    else
                        delete t

/* Overlapped reduction */

if lastsize = size[G] then
    for each node n ∈ N[G] do
        if nodeType[n] = CM and dout[n] = 1 and din[n] > 1 then
            level4 ← top[OutNodes[n]]
            fn ← top[InNode[n]]
            if nodeType[level4] = TASK and din[level4] > 1 and
                    nodeType [fn] = TASK and dout[fn] > 1 and din[fn] = 1 then
                level1 ← top[InNodes[fn]]
                if type[level1] = CM and dout[level1] > 1 then
                    Level2 ← InNodes[n]
                    Level3 ← OutNodes[fn]
                    if ∀ node ∈ Level2 ( nodeType[node] = TASK and
                            InNodes[node] = { level1 } and
                            OutNodes[node] = Level3 ) then
                        if ∀ node ∈ Level3 ( nodeType[node] = CM and
                                OutNodes[node] = { level4 } and
                                InNodes[node] = Level2 ) then
                            fromNode[top[OutTrans[n]]] ← level1
                            delete all node ∈ Level2
                            delete all node ∈ Level3
```

The verification algorithm uses attributes of workflow graphs, nodes, and transitions that have been presented earlier. It also uses an attribute named *top*[*S*] that returns the first element of the set S. In the algorithm, this property is used for sets that contain only a single element.

A workflow graph is shown in Figure 9 (a). The application of sequential and adjacent rules reduces the workflow graph from (a) to (b). The closed rule removes multiple transitions from closed components and transforms the graph to (c). The sequential, adjacent and closed rules are applied two more times to get (d). At this point, the sequential, adjacent and closed rules cannot reduce the graph any further. Therefore, the overlapped rule is applied to get (e). We get (f) by applying sequential, adjacent and closed

rules two more times on (e). Finally, a single application of terminal 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.

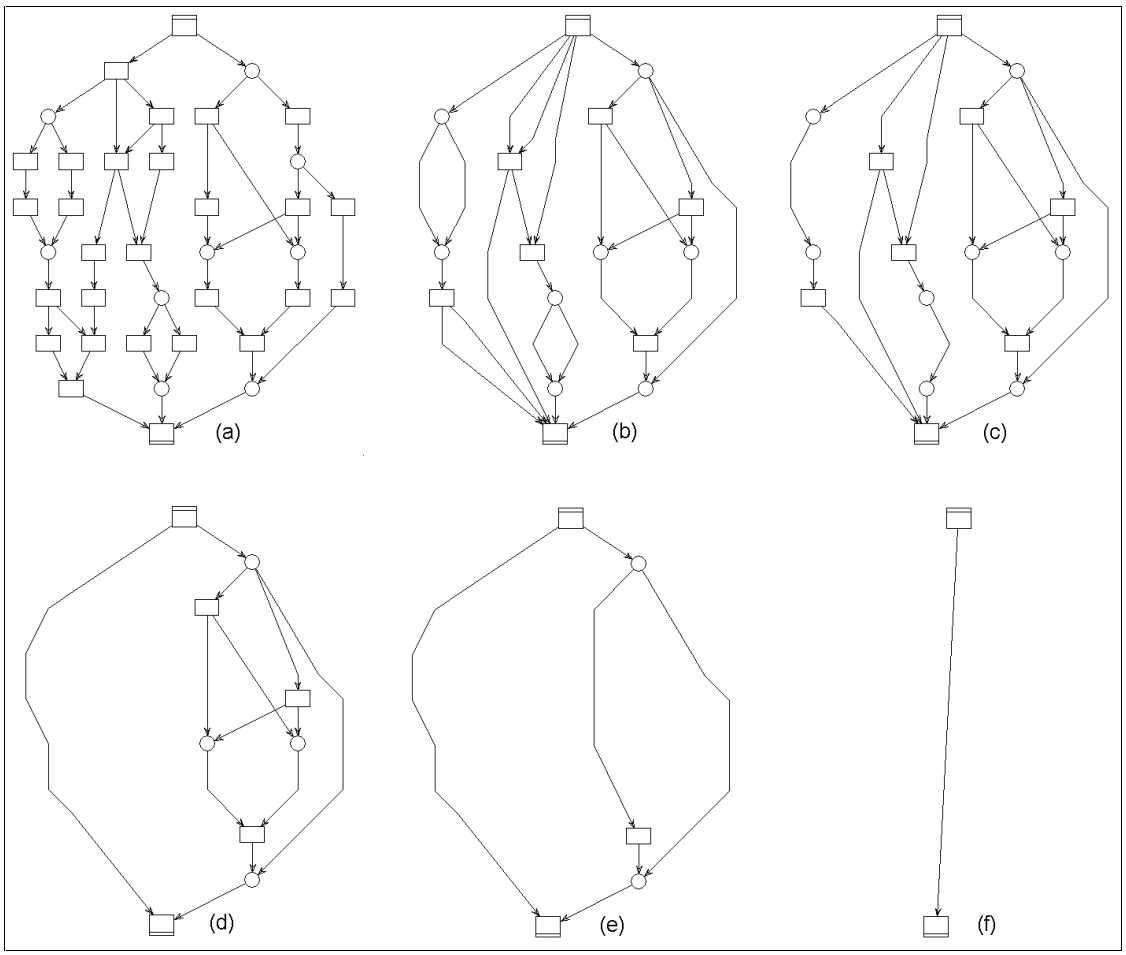


Fig. 9. Reducing a workflow graph without structural conflicts

Figure 10 (a) shows a workflow graph where we have added an additional transition to the workflow graph in 9 (a) that introduces a deadlock structural conflict. The reduction procedure is iteratively applied on (a) to get (b) that is not reducible any further and hence shows that a structural conflict exists in (a).

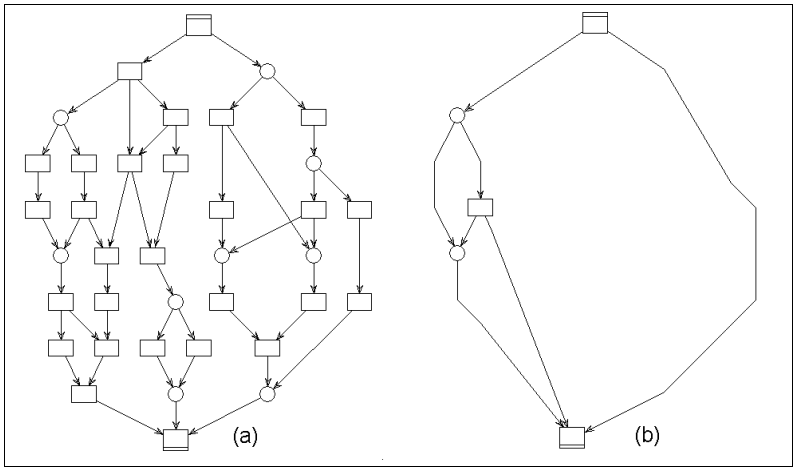


Fig. 10. Reducing a workflow graph with structural conflicts

### *4.3 Correctness and Complexity*

In each iteration of the algorithm, we check if the previous iteration has removed some nodes and/or transitions from the workflow graph *G*. We keep on iterating the reduction process as long it can reduce the size of process graph. If the outcome of REDUCE(*G*) algorithm is an empty graph then it means that the original process graph *G* before reduction does not contain any structural conflicts. Otherwise, it contains deadlock or lack of synchronization structural conflicts that would be clearly visible form the reduced graph.

Let the workflow graph $G_i$ is the reduced graph after *i* iterations of the algorithm on *G*. We shall prove that:

- the reduction rules do not generate structural conflicts, i.e., if $G_i$ is correct then $G_{i+1}$ is correct;
- the 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 nodes removed by the terminal and sequential structures do not introduce any split or join structures in a workflow graph. Therefore, they cannot possibly contribute towards a structural conflict. At the same time, they cannot remove any existing structural conflict.

The adjacent reduction rule targets two types of components. The first 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 second 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 application of sequential and adjacent reduction rules generally introduces components in a reduced workflow graph that contain two nodes and more than one transition between them. Such multiple transitions between two nodes imply the existence of more than one path between them. If the type of such two nodes is different, then the component represents a case of structural conflict and is not reduced by the closed reduction rule. However, if the type is same, we remove all but one of the multiple transitions since they cannot generate structural conflicts or remove existing ones. Multiple transitions between two tasks represent concurrent paths and between two choice / merge nodes 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 transition 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 terminal or 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 transitions to another node. Non-reducibility of graph implies that such multiple flows could exist only between nodes of different types. Multiple flows from a choice / merge node to a task represent deadlock conflict and from a task to

a choice / merge node 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.

To proceed, we can assume that $G_k$ does not contain a node with more than one transition 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.

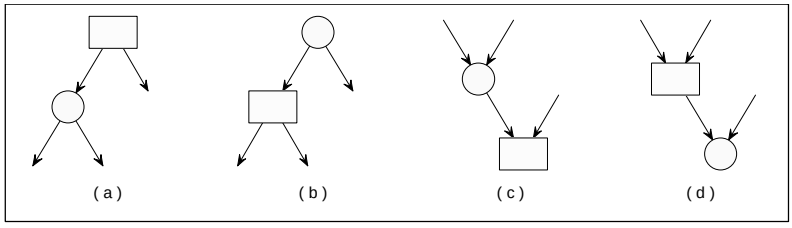


Fig. 11. Structures with single in / out degrees in non-reducible graphs

Since $G_k$ is non-reducible, then the node on the other side of such a single incoming or outgoing transition is always of a different type, otherwise it would be reducible by adjacent reduction rule. There are only four possible cases of such structures as shown in Figure 11. 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. There are only a few possibilities to build process graphs containing any of the structures in Figure 11 such that these structures are not reducible. As an example, let us consider at Figure 11 (a). The only way the choice structure is not reducible is if its outgoing transitions lead to a task object. However, that would result into a deadlock conflict. Similarly, for other structures we have to use objects of opposite types to build process graphs with a single source and a single sink and to ensure that these structures are not reducible. But introducing objects of opposite types would result into either deadlock or lack of synchronization structural conflicts.

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 transitions 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.

## 5. COUNTING INSTANCE SUBGRAPHS

A process model integrates different possible scenarios of process execution through modeling structures. An instance subgraph of a process model represents one of the possible execution scenarios of the underlying model. For example, Figure 6 shows a process model and its three possible instance subgraphs. As mentioned before, the number of possible instance subgraphs could grow exponentially depending on the number and placement of choice, merge, fork, and synchronizer structures.

Computing the number of possible instance subgraphs of a process model is a complex problem. The instance subgraph count is useful in assessing the complexity of a process model. It shows how many possible combinations of workflow tasks and their execution order may be created at run-time. The reduction algorithm presented earlier could also be used to count the number of possible instance subgraphs of process model.

The basic idea behind development of counting algorithm is to apply the reduction algorithm to reduce the workflow graph to only begin and end nodes and a single transition between them while counting the number of instance subgraphs between begin and end nodes.

Each transition $t \in T$ in the process graph is assigned an attribute named *count*[*t*] representing the number of instance subgraphs between the starting and ending nodes of the transition. Initially, the value of this attribute is set to 1 for all transitions. When the reduction process finishes, the attribute *count*[*t*] of the single remaining transition represents the number of instance subgraphs of the process graph.

For counting purposes, we do not apply the *terminal reduction* rule since it would reduce the process model to an empty graph. The other four reduction rules – sequential, adjacent, closed, and overlapped – are extended to handle the counting of instance subgraphs. The computation of instance subgraph count is based on combination principles that are applied to each of the four reduction rules. Rather than presenting the reduction algorithm again with extended statements for counting instance subgraphs, we will present the counting principles for each of the four reduction rules. The changes to the algorithm to incorporate these counting principles are straightforward.

The *sequential reduction* reduces all such nodes in the graph that have exactly one incoming and one outgoing transition. Figure 12 (S1, S2) shows the counting principle behind sequential reduction. It multiplies the *count* attribute of the incoming transition of the sequential node with the *count* attribute of the outgoing transition. After removing the sequential node, by applying the sequential reduction, the count attribute of the new transition between the parent and child nodes of the removed sequential node is set to the multiplication result. The basis for this counting principle is simple. If there are *x* instance subgraphs from node A to B, and *y* instance subgraphs from B to C, then there will be *x* * *y* instance subgraphs from A to C. This holds since for each instance subgraph from A to B, we can include one of the instance subgraphs from B to C in the target instance subgraph. The type of the sequential node being removed has no affect on the counting principle as shown in (S1) and (S2).

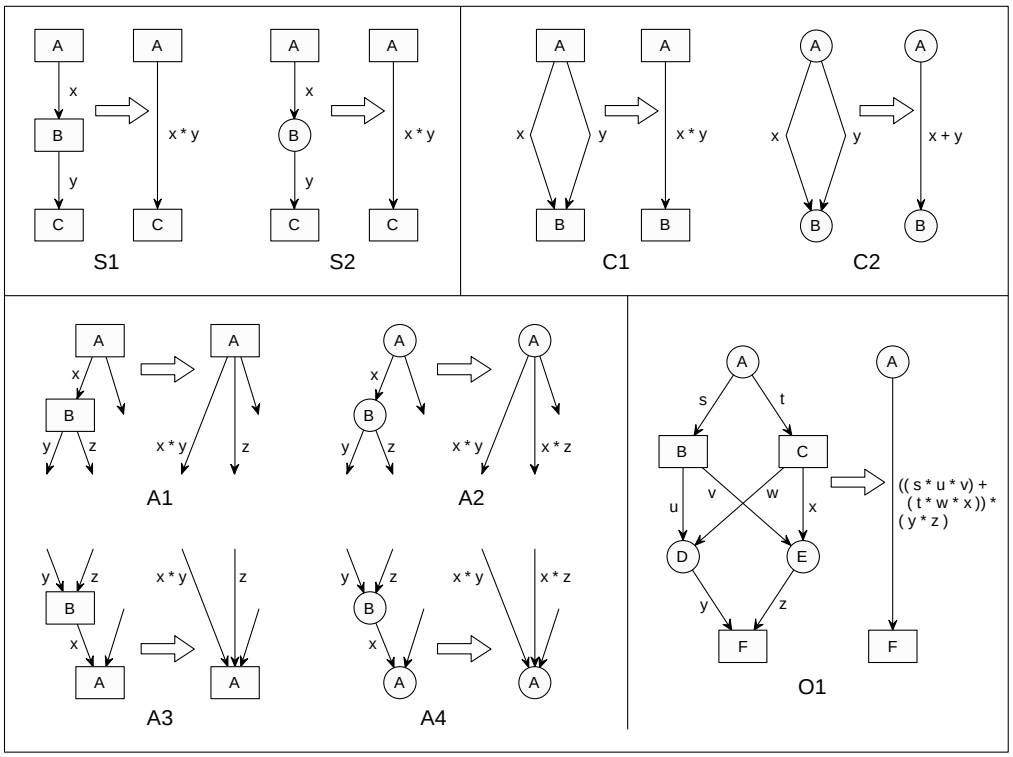


Fig. 12. Counting principles for sequential, adjacent, closed, and overlapped reduction rules

The *closed reduction* rule reduces multiple transitions between two nodes of the same type. In case of fork and synchronizer closed reduction, as shown in Figure 12 (C1), *count* attribute of all transitions between A and B are multiplied. All except one of the transitions are removed by applying closed reduction rule and the *count* attribute of the remaining transition is set to the multiplication result performed earlier. For each instance subgraph from A to B underneath transition associated with *x*, any

one of the instance subgraphs underneath the transition associated with *y* may be selected. Therefore, for fork and merge synchronize reduction, the total number of instance subgraphs from A to B will be *x* * *y*. In case of choice and merge closed reduction, as shown in (C2), addition is performed instead of multiplication. Whenever an instance graph is selected from one of the instance subgraphs underneath transition associated with *x*, none of the instance subgraphs underneath the transition associated with *y* will be selected because of the choice structure. Therefore, for choice and merge closed reduction, the total number of instance subgraphs from A to B will be *x* + *y*.

Figure 12 (A1, A2, A3, A4) shows the counting principles for *adjacent reduction*. This reduction rule merges the adjacent split and join structures of the same type. There are two types of split structures, fork and choice. The counting principles for these two cases are shown as A1 and A2 respectively. In case of fork, as shown in A1, the *count* attribute of one of the outgoing transitions of task B is multiplied with the *count* attribute of its single incoming transition. After that, by applying the adjacent reduction rule, all outgoing transitions from task B are moved to task A. Task B and its incoming transition are removed from the process graph. If there are *x* instance subgraphs from A to B then for each instance subgraph from A to B all outgoing subgraphs from B will be performed. As shown earlier, the closed reduction rule for fork and synchronizer structure multiplies *count* attributes of all fork transitions. Therefore, this principle multiplies just one of the outgoing transitions. Multiplying the *count* attribute of the incoming transition to any of the outgoing transitions would eventually produce the same result. In intermediate calculation, the multiplication would be different. But since the outgoing transitions will eventually be multiplied when reducing the closed structures, the final result would be unaffected. In case of choice, as shown in (A2), the *count* attributes of the incoming transition of task B is multiplied with the count attributes of each of its outgoing transitions. The computation of adjacent synchronizer and merge structures is based on similar principles.

An *overlapped* graph component combines choice, merge, fork, and synchronizer structures in such an interleaved way that the overlapped graph component is correct but not reducible through sequential, adjacent, and closed reduction rules. However, for counting purposes, we can apply sequential and closed reduction by breaking up the overlapped component. We show the counting principles for overlapped components using the example in Figure 12 (O1). There are two choices at A. If we choose to take the choice path associated with count *s*, then the total number of instance subgraphs from A to F would be ( s * u * v * y * z). Similarly, if we take the other path, the total number of instance subgraphs from A to F would be ( t * w * x * y * z). We can combine these two counts as a single number on the remaining one transition from A to F, after applying the overlapped reduction, as (( s * u * v) + ( t * w * x)) * ( y * z).

## 7. CONCLUSIONS

We report on successful application of graph reduction techniques for detecting structural conflicts in process models. To present the verification approach and algorithms for process models analysis, we introduced a generic process modeling language based on process definition standards by Workflow Management Coalition. The language makes use of five modeling structures – sequence, fork, synchronizer, choice, and merge – to build structural 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 process 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. An alternative use of reduction algorithm to count the number of instance subgraphs of a process model is also shown.

The research results presented in this paper have 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 13 shows a screen snapshot of FlowMake where the reduction process is being applied to a process model.

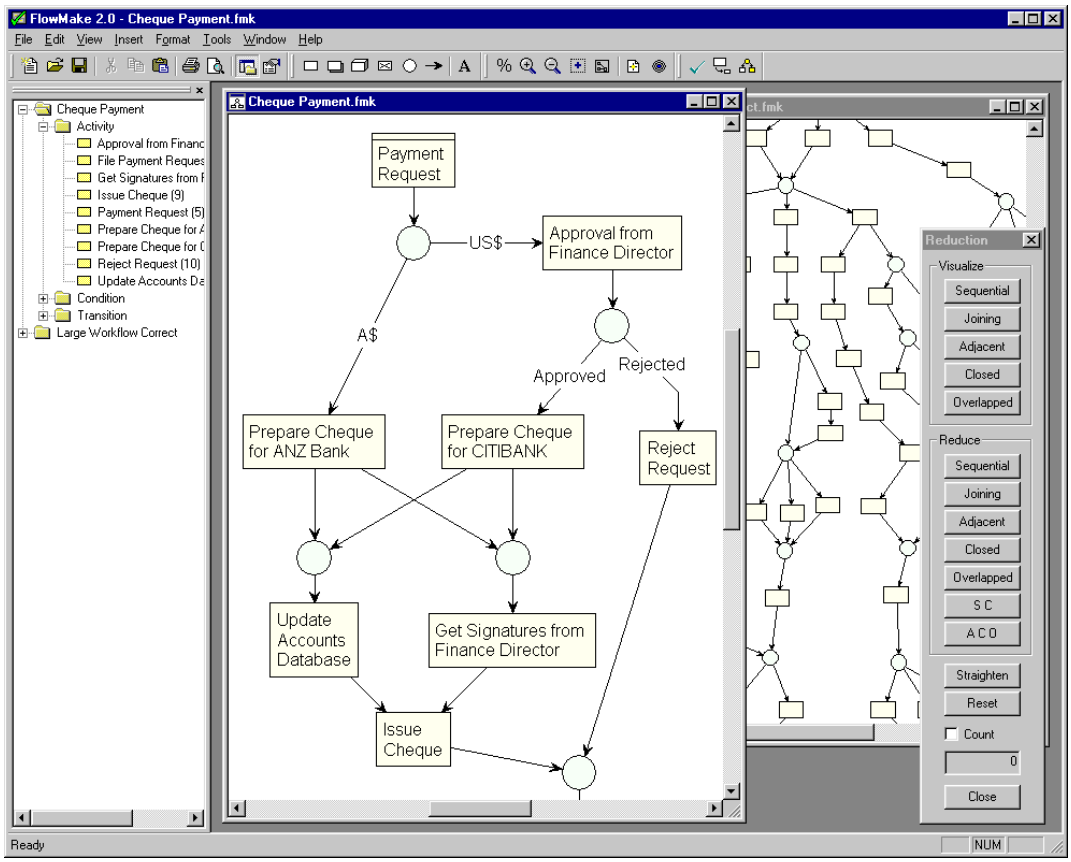


Fig.13. FlowMake: workflow modeling and verification tool

FlowMake is composed of three major components: the workflow editor, the verification engine, and the interface. *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/Research/projects/FlowMake/.

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.

*Acknowledgements* — 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.

## REFERENCES

[1] W.M.P. van der Aalst. Verification of workflow nets . In *P. Azema and G. Balbo , editors, Application and Theory of Petri Nets*, volume 1248 of Lecture Notes in Computer Science, pp. 407-426. Springer-Verlag, Berlin (1997)

[2] W.M.P. van der Aalst. 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] M. Reichert and P. Dadam. ADEPTflex - Supporting dynamic changes of workflow without loosing control. *Journal of Intelligent Information Systems (JIIS), Special Issue on Workflow and Process Management*, volume 10, number 2, pp. 93-129 (1998).

[5] S. Carlsen. 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] F. Casati F, S. Ceri, B. Pernici and G. Pozzi. Conceptual modeling of workflows. In *M.P. Papazoglou, editor, Proceedings of the 14th International Object-Oriented and Entity-Relationship Modeling Conference*, Gold Coast, Australia, volume 1021 of Lecture Notes in Computer Science, pp. 341-354. Springer-Verlag (1995).

[7] C.A. Ellis and G.J. Nutt. Modelling and enactment of workflow systems. In *M. Ajmone Marasan, editor, Application and Theory of Petri Nets*, Lecture Notes in Computer Science 691, pp. 1-16, Springer-Verleg, Berlin (1993).

[8] D. Georgakopoulos, M. Hornick and A. Sheth. An overview of workflow management: from process modeling to workflow automation infrastructure. *Journal on Distributed and Parallel Databases*, 3(2):119-153 (1995).

[9] A.H.M ter Hofstede, M.E. Orlowska and J. Rajapakse. Verification problems in conceptual workflow specifications. *Data & Knowledge Engineering*, 24(3):239-256 (1998).

[10] D. Kuo, M. Lawley, C. Liu and M.E. Orlowska. 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] J. Rajapakse. On Conceptual Workflow Specification and Verification. *MSc Thesis, Department of Computer Science, The University of Queensland*, Australia (1996).

[12] W. Sadiq and M.E. Orlowska. 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] W. Sadiq and M.E. Orlowska. On capturing process requirements of workflow based information systems. In *Proceedings of the 3rd International Conference on Business Information Systems (BIS '99)*, Poznan, Poland, pp. 281-294. Springer-Verlag 1999.

[14] Workflow Management Coalition. The workflow management coalition specifications - terminology and glossary. issue 2.0. *Document Number WFMC-TC-1011* (1996).

[15] Workflow Management Coalition. Interface 1: process definition interchange, process model. *Document Number WfMC TC-1016-P* (1998).

[16] W. Sadiq and M.E. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE '99)*, Heidelberg, Germany, Lecture Notes in Computer Science 1626. pp.195-209. Springer-Verlag (1999).

[17] J. Desel and J. Esparza. Free choice Petri nets. *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Great Britain (1995).

[18] M.N.S Swamy and K. Thulasiraman. Graphs, networks, and algorithms. *John Wiley and Sons*, Canada (1981).