Introducing Concurrency to Workflows: Theory and A Real-World Case Study

Laird Burns, Wai Yin Mok, and Wes N. Colley
College of Business Administration
University of Alabama in Huntsville
Huntsville, Alabama, USA, 35899

***Abstract*—Making use of the concepts of the activity diagrams of the Unified Modeling Language, this paper defines an important class of workflows, called flow independent workflows, which are deterministic in the sense that if a flow independent workflow is given the same multi-set of resources as input over and over again, it will produce the same output every time. After which, this paper provides a methodology, and its accompanying algorithms, that introduces concurrency to flow independent workflows by rearranging the action nodes in every flow of control. It then applies the methodology to a real-world case to demonstrate its usefulness. It concludes with future research possibilities that might extend the methodology to general workflows, i.e., not necessarily flow independent workflows.**

***Keywords*—Business process re-engineering; Unified Modeling Language; activity diagrams; flow independent workflows; concurrency**

I. Introduction

Business process re-engineering (BPR) has been a subject of intensive study since Michael Hammer published his seminal paper "Reengineering Work: Don't Automate, Obliterate" [6]. As a business management strategy, it focuses on the analysis and design of workflows and processes within an organization. Improving customer service and reducing operational costs are among its many goals. Many times BPR involves large scale modifications and redesigns of existing workflows. This paper, however, introduces a methodology of smaller scale that can speed up a specific class of workflows that has certain desirable properties. We then supplement the methodology with a real-world case study that demonstrates the usefulness of the methodology.

To formally prove the methodology really does what it claims, we need a mathematical foundation upon which the methodology is based. For this purpose, we adopt some of the ideas of The Unified Modeling Language (UML), which has been an industry standard for modeling software-intensive systems since 2000 [12]. Developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software in the 1990s, UML provides a set of graphic notation techniques to create visual models of object-oriented software-intensive systems [3]. Particularly relevant to this research is the activity diagrams, from which our methodology is derived.

Another important component of our methodology is the database concurrency theory [2], which has been an area of study in the last thirty years. Many recent studies applied this theory to workflows [1,4,5,7,8,9,10,11]. Database concurrency theory is mainly concerned with maintaining an orderly access of data items in the presence of multiple long-running concurrent database transactions. Without proper control, concurrent database transactions might read from and write to data items in an arbitrary order that will take a database from a correct state to an incorrect state. Placing locks on data items at the correct moment is a main mechanism the theory uses to control the execution of the transactions to ensure that the database will go from one correct state to another correct state.

This paper is organized as follows. In Section 2, we present a mathematical foundation for workflows and define a particular class of workflows, called flow independent workflows, that has the property that being deterministic. A workflow being deterministic means that it will always produce the same output for a given input. Section 3 presents the proposed workflow re-engineering methodology and the correctness proof. Section 4 gives the case study. We conclude and point out possible future researches in Section 5.

II. Fundamentals

A. *Workflows and UML Activity Diagrams*

UML has two types of diagrams: structural diagrams and behavioral diagrams [3]. Among all the behavioral diagrams of UML, activity diagrams are particularly relevant to this research because they specify the operational step-by-step processes of a system. Many concepts of activity diagrams are especially useful. For example, our definition for workflows adopts branches of control and concurrency from activity diagrams. Further, the concept of resources is also necessary in the definition because many workflows use, modify, consume and produce resources as they are executed. In addition, we also define the rules that govern the execution of workflows, i.e., the execution semantics of workflows.

Definition 1: A *workflow* is an 8-tuple (*Actions*, *Branches*, *Merges*, *Forks*, *Joins*, *Arrows*, *Resources*, *Constraints*). *Actions*, *Branches*, *Merges*, *Forks*, *Joins* are all finite sets of nodes (vertices) and *Arrows* is a finite set of directed edges (ordered pairs) of nodes in *Actions* ∪ *Branches* ∪ *Merges* ∪ *Forks* ∪ *Joins*. *Resources* is a finite multi-set of resources and *Constraints* is a finite set of constraints over the workflow. The following rules further refine these concepts.

1: *Resources* is a finite multi-set of $n \geq 1$ resources r_1, r_2, …, r_n. A resource r_i of *Resources* can only be in one of an enumerable number of states at any moment of the execution of the workflow. (A workflow might need more than one unit

of a resource type in its execution. Therefore, a multi-set, rather than a set, of resources is used in the definition because elements of a multi-set are not necessarily distinct. However, the subscripts can make the elements distinct. For example, two identical but distinct elements r_i and r_j in *Resources* can be distinguished by their subscripts if $i \neq j$. In a sense, the subscripts tag the resources and serve as unique identifiers.)

2: An action node in *Actions* denotes an action, which is executed completely or not at all. An action takes a finite amount of time to complete, although it may vary from time to time. Any action node v_i in *Actions* yields at most one directed edge of the form (v_i, v_j) in *Arrows*. (In other words, each action node has at most one out arrow.)

3: Each action node has four associated multi-sets of resources: *use*, *modify*, *consume*, and *produce*. *Use* is the finite multi-set of resources that the action uses in its execution. *Modify* is the finite multi-set of resources whose states will be changed by the action during its execution. *Consume* specifies the finite multi-set of resources that the action consumes during its execution. After the action consumes a resource r_i, r_i ceases to exist. *Produce* is the finite multi-set of resources that the action produces during its execution. The action can assign a resource r_i in one of its many allowable states when the action produces r_i. Naturally, *use* $\supseteq$ *modify* $\cup$ *consume* and *use* $\cap$ *produce* $= \emptyset$. We stipulate that if an action node is given the same multi-set *use* multiple times, the action node will always yield the same multi-sets *modify*, *consume*, and *produce* every time.

4: The nodes in *Branches* $\cup$ *Merges* $\cup$ *Forks* $\cup$ *Joins* are *control nodes*. Control nodes only direct the flows of control during the execution of the workflow and thus they do not modify, consume or produce the resources in *Resources*.

5: A *source node n* is a node that there is not a directed edge (v_i, n) in *Arrows*. A *sink node n* is a node that there is not a directed edge (n, v_i) in *Arrows*. Further, none of the directed edges in *Arrows* has the form (n, n) where $n \in$ *Actions* $\cup$ *Branches* $\cup$ *Merges* $\cup$ *Forks* $\cup$ *Joins*. (In other words, self-loops are not allowed.) Source nodes and sink nodes must be action nodes, although multiple source nodes and/or sink nodes are allowed.

6: Initially, there is not a flow of control in the workflow. When the execution of the workflow begins, simultaneously a separate flow of control will start at each source node. The execution of the workflow terminates when all of its flows of control terminate.

7: When a flow of control reaches v_i of a directed edge (v_i, v_j), whether v_i will start executing depends on if v_i uses any resources. If v_i does not use any resources, v_i immediately starts executing. If v_i uses some resources, then we further check whether the elements in v_i's *use* are also in *Resources*. If *use* $\subseteq$ *Resources*, then v_i starts executing. In both cases, after v_i stops executing, *Resources* = *Resources* $\cup$ v_i's *produce* $-$ v_i's *consume*, the elements in v_i's *modify*, which are also in *Resources*, have been modified, and v_j immediately starts executing. If *use* $\not\subseteq$ *Resources*, then the flow of control terminates at v_i with no change to *Resources*. (In the case that the resources of v_i are not immediately available to v_i when the flow of control reaches v_i, v_i may enter a waiting period, which can be modeled by adding a branch node of Rule 8 and/or a constraint of Rule 12. Example 3 illustrates how this can be done. Hence, there is no loss of generality to say that if *use* $\not\subseteq$ *Resources*, then the flow of control terminates at v_i. Note that any action of a reasonable workflow will not wait indefinitely. Therefore, we stipulate that the length of such a waiting period must be specified beforehand.)

The following rules are concerned with branch nodes, merge nodes, fork nodes and join nodes. All of these nodes are not source nodes and sink nodes. As a result, for every node $v_j \in$ *Branches* $\cup$ *Merges* $\cup$ *Forks* $\cup$ *Joins*, there are nodes v_i and v_k such that (v_i, v_j) and (v_j, v_k) are both in *Arrows*.

8: For any branch node v_j in *Branches*, there is a unique directed edge (v_i, v_j) in *Arrows* and there are $n \geq 1$ directed edges (v_j, v_{k1}), (v_j, v_{k2}), …, (v_j, v_{kn}) in *Arrows*. Further, there is a condition associated with (v_j, v_{kl}), $1 \leq l \leq n$. When a flow of control reaches v_j, at most one condition out of those associated with these n directed edges can be true at that moment. The flow of control then follows the unique directed edge whose condition is true. If the conditions associated with these n directed edges are all false, then the flow of control terminates at v_j. Since the conditions associated with the directed edges (v_j, v_{k1}), (v_j, v_{k2}), …, (v_j, v_{kn}) might involve the elements in *Resources*, v_j also has a multi-set *use*, which specifies the elements in *Resources* that are used in these conditions.

9: For any merge node v_j in *Merges*, there is a unique directed edge (v_j, v_k) in *Arrows* and there are $n > 1$ directed edges (v_{i1}, v_j), (v_{i2}, v_j), …, (v_{in}, v_j) in *Arrows*. For any flow of control that reaches any of $v_{i1}, v_{i2}, \ldots, v_{in}$ and after it stops executing, v_k immediately starts executing. Note that v_j does not need any resources in its execution.

10: For any fork node v_j in *Forks*, there is a unique directed edge (v_i, v_j) in *Arrows* and there are $n > 1$ directed edges (v_j, v_{k1}), (v_j, v_{k2}), …, (v_j, v_{kn}) in *Arrows*. When a flow of control reaches v_j via v_i, the flow of control terminates at v_j and n new flows of control $f_1, f_2, \ldots, f_n$ will be created, and each f_l starts at v_j and then immediately reaches v_{kl}, $1 \leq l \leq n$. Note that v_j does not need any resources in its execution.

11: For any join node v_j in *Joins*, there is a unique directed edge (v_j, v_k) in *Arrows* and there are $n > 1$ directed edges (v_{i1}, v_j), (v_{i2}, v_j), …, (v_{in}, v_j) in *Arrows*. A new flow of control starts at v_j implies that there are n flows of control $f_1, f_2, \ldots, f_n$ and each f_l, $1 \leq l \leq n$, reaches v_j via v_{il} and then terminates at v_j. We stipulate that every flow of control that reaches v_j via v_{il} for some l can only give raise to a single new flow of control starting at v_j. Note that v_j does not need any resources in its execution.

12: *Constraints* is a finite set of constraints defined on the workflow. □

Example 1: A moving company workflow might require a multi-set $\{truck_1, truck_2, worker_1, worker_2, worker_3, worker_4\}$ of two trucks and four workers to move a family. Although $truck_1$ and $truck_2$ might be identical, the subscripts distinguish them from each other. As for the status of a resource, for example a document might be in one of the states prepared,

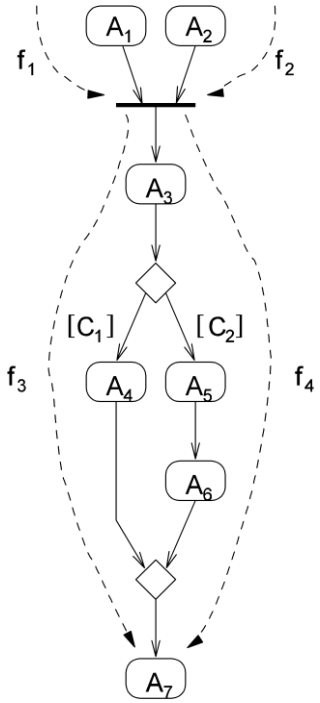

Fig.1. A sample workflow in UML notation

unsigned, and signed; and a temperature variable might assume one of its many permissible values. □

In most workflows, many actions cannot have a fixed duration to complete. For example, a mailman might be able to finish his job in 5 hours on a sunny day but not on a rainy day. Hence, unless otherwise specified, we leave the duration of an action open. When such constraints are necessary, they are added to the set *Constraints*.

We use UML notation to represent the components of a workflow. Action nodes are represented by rounded boxes, branch nodes and merge nodes by diamonds and fork nodes and join nodes by thick lines. Arrows naturally represent directed edges.

Example 2: Fig 1shows a workflow in UML notation. When the workflow begins executing, two flows of control f_1 and f_2, one starting at A_1 and the other at A_2, start simultaneously. These two flows of control are executed concurrently and synchronized at the join node, where each waits until the other reaches the join node. Then, they both terminate at the join node and a new flow of control starts at the join node. At the branch node, either C_1 or C_2 but not both is true. The flow of control follows the directed edge whose condition is true. Therefore, there are two possible flows of control f_3 and f_4 starting at the join node; but unlike f_1 and f_2, f_3 and f_4 cannot co-exist at the same time. They are merely two different possibilities. Finally, the flow of control reaches the merge node, where it simply continues on to A_7, the final action node. Although not shown in the Fig., the workflow in has a multi-set *Resources* and each action node has four associated finite multi-sets *use*, *modify*, *consume* and *produce*. □

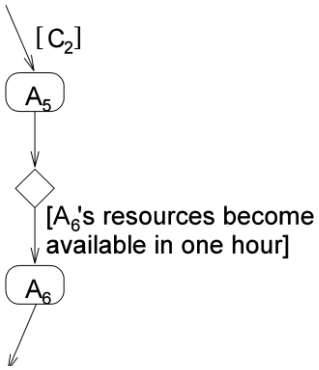

Fig.2. Adding a waiting period to A6 in Fig.1.

Example 3: Suppose the resources of action node A6 in are not available immediately when a flow of control reaches A6. A6 then has to wait for a while. To capture such a waiting period, a branch node can be added before A6 in the workflow. Since workflows cannot wait forever, there must be a specific time limit for the waiting period. Hence, the condition in specifies that the waiting period must be less than one hour, although some other time limit is also possible. If A6's resources become available within one hour, the follow of control continues to A6; otherwise, the condition fails and the flow of control terminates at the branch node. □

Example 4: If initially *Resources* = {a, b, c_1, c_2} and A_1's *use* = {a, d_1, d_2}, then the flow of control started at A_1 immediately terminates at A_1 because A_1 uses two d's in its execution, but there is none available in *Resources*. When that happens, even if A_2 successfully completes its action, no new flow of control will start at the join node because the join node will indefinitely wait for a flow of control coming from A_1, which will never come. □

Example 5: We might additionally add a constraint that specifies A_1 must complete its action within an hour or that the total time the workflow will take must be less than 5 hours. These constraints will then be added to the set *Constraints*. □

B. *Configuartions of Workflows*

Although self-loops are not allowed in a workflow, cycles are still possible. Hence, a workflow might not stop executing once it starts. Since most useful workflows terminate once they are given enough time and resources, non-terminating workflows are not considered any further.

Definition 2: A *terminating workflow* is one that will terminate on any given multi-set *Resources*. □

Example 6: Because it does not have any cycles, the workflow in Fig.1 always terminates not matter what the multi-set *Resources* is. □

Definition 3: A *configuration* of a workflow is the multi-set *Resources* together with the states of the elements in *Resources*. □

Every workflow has an initial configuration, which is the multi-set *Resources* together with the states of each of its elements before the workflow starts executing. After the

workflow stops executing, it then has a final configuration, which is the multi-set *Resources* together with the states of each of its elements after the workflow stops executing.

Definition 4: A *deterministic workflow* is a terminating workflow and for any given multi-set *Resources*, it always ends up in the same final configuration even if the workflow has to be executed multiple times. On the other hand, a *non-deterministic workflow* is also a terminating workflow and for some given multi-set *Resources*, it might not end up in the same final configuration if the workflow is executed multiple times. □

Although a deterministic workflow is a terminating workflow, the converse is not true. That is, it is possible that a terminating workflow is non-deterministic. Since most real-world workflows are supposed to give the same results even if they have to be executed more than once, deterministic workflows are desirable.

Example 7: Suppose *Resources* = {*a*, b_1, b_2, *c*}, and A_1's *use* = A_1's *modify* = A_2's *use* = A_2's *modify* = {*a*}, and in addition resource *a* is not in the multi-sets *modify*, *consume* and *produce* of the other action nodes in Fig.1. Since A_1 might finish before A_2 in one run, but in another run A_2 might finish before A_1, and since A_1 and A_2 both modify resource *a*, the workflow in Fig.1 would be non-deterministic under these conditions. □

Here, we give a sufficient condition for deterministic workflows. That is, if a workflow satisfies this condition, the workflow is deterministic.

Definition 5: Two (not necessarily distinct) actions A_i and A_j are *independent* if A_i's *use* ∩ (A_j's *modify* ∪ A_j's *consume* ∪ A_j's *produce*) = ∅ and A_j's *use* ∩ (A_i's *modify* ∪ A_i's *consume* ∪ A_i's *produce*) = ∅. □

Definition 6: Two distinct concurrent flows of control f_i and f_j of a workflow are *independent* if any pair of actions A_i of f_i and A_j of f_j are independent. A workflow is *flow independent* if any two of its distinct concurrent flows of control f_i and f_j are *independent*. □

Example 8: f_1 and f_2 are distinct concurrent flows of control in Fig.1. f_3 and f_4 are not concurrent flows of control because they cannot co-exist at the same time. □

Theorem 1: If a workflow *w* is flow independent, *w* is deterministic.

Proof sketch: We proceed by induction on the number *n* of concurrent flows of control of *w* at any moment. If $n = 1$, then at any moment *w* has at most one flow of control. Suppose this flow of control f_i has $n \geq 1$ nodes $v_1, v_2, \ldots, v_{n-1}, v_n$. By Rule 3 of Definition 1, when an action node is given the same multi-set *use* multiple times, the action node will always yield the same multi-sets *modify*, *consume*, and *produce* every time. Hence, every action node v_i of f_i is deterministic. It is clear that a sequential execution of $n \geq 1$ deterministic action nodes is also deterministic. Hence, the basis is established. Assume the induction hypothesis is true for $n = k$ for some $k \geq 1$. That is, *w* is deterministic if *w* has *k* or less concurrent flows of control at any moment. Suppose *w* has one more flow of control f_{k+1} at a moment. By the assumption that *w* is flow independent, any pair of actions A_i of f_{k+1} and A_j of f_l ($1 \leq l \leq k$) are independent. Hence, every action A_i of f_{k+1} does not depend on the output of the action nodes of f_l ($1 \leq l \leq k$). At this point the argument for the basis also applies to f_{k+1}. Hence, the workflow is deterministic. □

III. Workflows Re-Engineering

The main purpose of workflows re-engineering is to make workflows more efficient, or to reduce certain resources required by the workflows. For most cases, the goal is to reduce the completion time of a workflow. Because flow independent workflows have the desirable property that they are deterministic, in this section we focus on flow independent workflows and present a methodology that will reduce their completion time.

A. *Introducting Concurrency*

Introducing concurrency to a flow independent workflow obviously can reduce the time it takes to complete. However, the interactions of the actions and the resources of the workflow imply a certain order the actions must observe. Example 9: Consider the workflow in and suppose A_5's *use* = {*a*} and A_6's *modify* = {*a*}. Under this assumption, A_5 uses the resource *a* before it is modified by A_6. Hence, A_5 must precede A_6. Now suppose A_5's *modify* = {*a*} and A_6's *use* = {*a*}. In this case, A_6 uses the resource *a* modified by A_5. Hence, A_6 must follow A_5. Similarly, suppose A_5's *use* = {*a*} and A_6's *consume* = {*a*}. Then, A_5 must precede A_6 because after A_6, resource *a* will cease to exist. If A_5's *consume* = {*a*} and A_6's *use* = {*a*}, then A_6 cannot be executed because resource *a* no longer exists after A_5. Hence, the flow of control terminates at A_5. (Note that one may argue that it is incorrect to have A_5's *consume* = {*a*} and A_6's *use* = {*a*}. However, we have to be compliant to the semantics of the workflow and thus the order of execution of the actions in the workflow must be preserved.) For the case that A_5's *produce* = {*a*} and A_6's *use* = {*a*}, A_5 must precede A_6. On the other hand, if A_5's *use* = {*a*} and A_6's *produce* = {*a*}, then the flow of control terminates at A_5, which is similar to the case that A_5's *consume* = {*a*} and A_6's *use* = {*a*}. However, if A_5 and A_6 are independent, they can be executed concurrently. This can be done by adding a fork node as shown in Fig 3 As a result, two concurrent flows of control can start at A_5 and A_6 simultaneously. □

The main idea of our methodology is to introduce concurrency to a flow independent workflow; but at the same time we have to ensure that the resulting workflow is equivalent to the original.

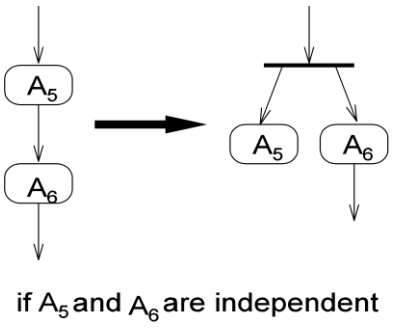

Fig.3. A_5 and A_6 can be executed concurrently if they are independent.

Definition 7: Two deterministic workflows are *equivalent* if they are given any multi-set *Resources* as their initial configurations, they will both end up in the same final configuration after they stop executing. □

B. *Partitioning Flows of Control into Sequences*

Lemma 1 points out some useful characteristics that will help introducing concurrency to flow independent workflows.

Lemma 1: Consider a flow of control f_i of $n \geq 1$ nodes v_1, v_2, …, v_{n-1}, v_n of a flow independent workflow. The following are all true for f_i.

a) v_1 cannot be a branch node.

b) Both v_1 and v_n cannot be a merge node.

c) Only v_1 or v_n can be a fork node.

d) Only v_1 or v_n can be a join node.

e) If v_i ($1 < i < n$) is a branch node, then each of v_1, …, v_{i-1} cannot be executed concurrently with any of v_{i+1}, …, v_n.

Proof sketch:

a) Rule 8 of Definition 1 states that when a flow of control reaches a branch node, the conditions associated with all of its outgoing directed edges will be evaluated and the flow of control will follow the unique directed edge whose condition is true. Hence, branch nodes do not start any flow of control and thus v_1 cannot be a branch node.

b) According to Rule 9 of Definition 1, merge nodes neither start nor terminate any flow of control. Given a merge node v_j, there are two directed edges (v_i, v_j) and (v_j, v_k) in Arrows and v_j simply passes the flow of control from v_i to v_k. Hence, v_1 and v_n cannot be a merge node.

c) Rule 10 of Definition 1 states that when a flow of control reaches a fork node, the flow of control will terminate at the fork node and then the fork node will yield more than one flow of control. Hence, if a v_i where $1 < i < n$ is a fork node, then the flow of control starting at v_1 will terminate at v_i. This contradicts the lemma statement that the flow of control terminates at v_n.

d) Rule 11 of Definition 1 states that when a flow of control reaches a join node, the flow of control will wait for and synchronize with the other flows of control destined for the join node. Then, they will all terminate at the join node and a new flow of control will start at the join node. Hence, if a v_i where $1 < i < n$ is a join node, the flow of control starting at v_1 will terminate at v_i. This contradicts the lemma statement that the flow of control terminates at v_n.

e) If v_i where $1 < i < n$ is a branch node, then by Rule 8 of Definition 1 there is a condition associated with the directed edge (v_i, v_{i+1}). The execution of any of v_{i+1}, …, v_n depends on the truth value of the condition. If the condition is true, v_{i+1}, …, v_n will be executed in this order; otherwise, none of them will be executed. On the other hand, the execution of v_1, …, v_{i-1} do not depend on the condition of the directed edge (v_i, v_{i+1}). As a result, each of v_1, …, v_{i-1} cannot be executed concurrently with any of v_{i+1}, …, v_n. □

While the other parts of Lemma 1 restrict the nodes in f_i, Part e relates directly to concurrency. With the additional restriction that none of the nodes of f_i can be replicated, Lemma 2 provides another result about merge nodes that also relates directly to concurrency.

Lemma 2: Consider a flow of control f_i of $n \geq 1$ nodes v_1, v_2, …, v_{n-1}, v_n of a flow independent workflow. If v_j ($1 < j < n$) is a merge node and none of the nodes of f_i can be replicated, then each of v_1, …, v_{j-1} cannot be executed concurrently with any of v_{j+1}, …, v_n.

Proof sketch: By Rule 9 of Definition 1, there are at least two directed edges (v_{j-1}, v_j) and (v_i, v_j) and a unique directed edge (v_j, v_{j+1}) for v_j. v_{j-1} is the node precedes v_j in f_i but v_i is not part of f_i. When a flow of control reaches v_j via v_{j-1}, v_j will pass the flow of control to v_{j+1}. Likewise, when a flow of control f_k ($i \neq k$) reaches v_j via v_i, v_j will pass the flow of control to v_{j+1}. If any of v_{j+1}, …, v_n is executed concurrently with any of v_1, …, v_{j-1}, the same node must be removed from v_{j+1}, …, v_n and must also be executed concurrently with some node before v_j on f_k as well. However, since nodes on f_i cannot be replicated, this is impossible. □

Given a flow of control f_i of a flow independent workflow, Lemmas 1 and 2 specifies certain restrictions on f_i. Particularly, the branch nodes and merge nodes partition f_i into sequences of action nodes. By Lemma 1e and Lemma 2, every action node of each of these sequences cannot be executed concurrently with any action node of the other sequences of f_i. However, we can rearrange the action nodes within a sequence so that some of them can be executed concurrently.

Example 10: Consider a flow of control f_i of a flow independent workflow that has two diamonds, where each diamond either represents a branch node or a merge node. As shown in Fig 4 these two diamonds partition f_i into three sequences of action nodes, where each sequence of action nodes is represented by a dashed line. □

To rearrange the action nodes within a sequence, we first need to define a relation on the action nodes. This relation will yield an order of execution on the action nodes.

Definition 8: Given a sequence of $n \geq 1$ action nodes v_1, v_2, …, v_{n-1}, v_n, we define a relation, denoted by $\lhd$, on the action nodes as follows:

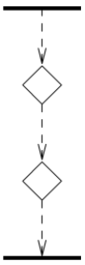

Fig.4. Three sequences of action nodes as a result of two branch/merge nodes.

a) $v_i \lhd v_j$ *if* $i < j$ *and* v_i *and* v_j *are not independent, or*

b) $v_i \lhd v_j$ *if* $v_i \lhd v_k$ *and* $v_k \lhd v_j$ *for some* v_k. □

The meaning of $v_i \lhd v_j$ is that v_i must be executed before v_j. Condition a is obvious because if v_i precedes v_j in the sequence and they are not independent, then v_i must be executed before v_j. (See Example 9.) Condition b says that if v_i must be executed before v_k and v_k must be executed before v_j, then v_i must (indirectly) be executed before v_j. Based on the relation $\lhd$, we can now define an execution order hierarchy for the action nodes as follows:

Definition 9: Given a sequence of $n \geq 1$ action nodes v_1, v_2, …, v_{n-1}, v_n, we define an execution order hierarchy h for the action nodes as follows:

a) If $v_i \not\lhd v_k$ *for each* v_i *such that* $v_i \neq v_k$, *then* v_k *is a root of h. (Note that it is possible that h may have more than one root.)*

b) If v_i *is a node in h and* $v_i \lhd v_j$ *and there does not exist an* v_k *such that* $v_i \lhd v_k$ *and* $v_k \lhd v_j$, *then* v_i *is a parent of* v_j *in h. (Note that an action node may have more than one parent in h. Also note that if* $v_i \lhd v_k$ *and* $v_k \lhd v_j$ *for some* v_k, *then* v_i *is not a parent of* v_j *in h. However,* v_i *is an ancestor of* v_j *in h.)*

Algorithms can be easily defined to generate the execution order hierarchy of Definition 9. One possibility is as follows.

Algorithm 1:

Input: a sequence of $n \geq 1$ action nodes v_1, v_2, …, v_{n-1}, v_n.

Output: an initially empty execution order hierarchy h of Definition 9 for the input action nodes.

1) Set $s = \emptyset$.

2) For $i = 1$ to n do

For $j = i+1$ to n do /* $i < j$ */

If v_i and v_j are not independent, insert (v_i, v_j) to s.

3) For $i = 1$ to n do

If v_i does not appear in h, make v_i a root of h.

For each ordered pair of the form (v_i, v_j) in s,

add a directed edge $v_i \longrightarrow v_j$ to h.

4) For $i = 1$ to n do /* Remove redundant edges from h. */

If $v_i \longrightarrow v_k$ is a directed edge in h but $v_i \longrightarrow v_k$ can be derived from the other directed edges in h, remove $v_i \longrightarrow v_k$ from h.

Example 11: Consider the sequence of action nodes in Fig 5 and further suppose the following:

1) A_1's *use* = {a}, A_1's *modify* = {a}, A_1's *produce* = {b, c},

2) A_2's *use* = {d}, A_2's *produce* = {e, f},

3) A_3's *use* = {a, b, e}, A_3's *modify* = {a}, A_3's *produce* = {g},

4) A_4's *use* = {c, f}, A_4's *modify* = {f}, A_4's *consume* = {c},

5) A_5's *use* = {a}, A_5's *produce* = {h},

6) A_6's *use* = {e}, A_6's *produce* = {i}.

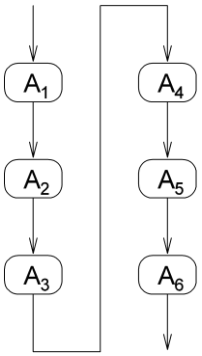

Fig.5. A sequence of action nodes

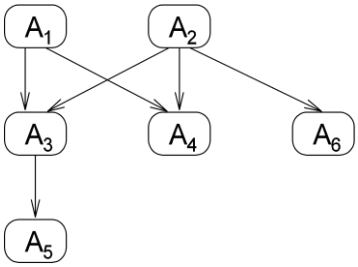

Fig.6. The execution order hierarchy of Example 11

After Step 2 of Algorithm 1, $s = \{A_1 \longrightarrow A_3, A_1 \longrightarrow A_4, A_1 \longrightarrow A_5, A_2 \longrightarrow A_3, A_2 \longrightarrow A_4, A_2 \longrightarrow A_6, A_3 \longrightarrow A_5\}$. In the first iteration of the for-loop of Step 3, A_1 becomes the first root of h. Then, the directed edges $A_1 \longrightarrow A_3$, $A_1 \longrightarrow A_4$, $A_1 \longrightarrow A_5$ are added to h. In the second iteration of the for-loop of Step 3, A_2 becomes the second root of h. Then, the directed edges $A_2 \longrightarrow A_3$, $A_2 \longrightarrow A_4$, $A_2 \longrightarrow A_6$ are added to h. In the third iteration of the for-loop of Step 3, A_3 does not become another root of h because A_3 already appears in h. Then, the directed edge $A_3 \longrightarrow A_5$ is added to h. Now, the directed edge $A_1 \longrightarrow A_5$ becomes redundant in h because $A_1 \longrightarrow A_5$ can be derived from $A_1 \longrightarrow A_3$ and $A_3 \longrightarrow A_5$, which are also in h. Thus, $A_1 \longrightarrow A_5$ is removed at Step 4. The result is the execution order hierarchy in Fig 6. □

Theorem 2: Given a sequence of $n \geq 1$ action nodes v_1, v_2, …, v_{n-1}, v_n, Algorithm 1 generates the execution order hierarchy of Definition 9 from the action nodes.

Proof sketch:

Step 2 of Algorithm 1 generates all ordered pairs $v_i \lhd v_j$ where there is no v_k such that $v_i \lhd v_k$ and $v_k \lhd v_j$. (That is, Step 2 of Algorithm 1 generates all ordered pairs $v_i \lhd v_j$ such that v_i must be executed directly, not indirectly, before v_j.) In fact, Step 2 of Algorithm 1 may as well generate some ordered pairs that can be derived from the other ordered pairs. Step 3 of Algorithm 1 simply adds all the directed edges to *h* based on the ordered pairs in *s*. Potentially it may add redundant directed edges to *h* that can be inferred from the other directed edges, e.g., $A_1 \longrightarrow A_5$ of Example 11. Step 4 of Algorithm 1 is designed to remove them. □

Transforming the execution order hierarchy of Definition 9 into a workflow with concurrency is straightforward. The idea is to use fork nodes to fork multiple flows of control and join nodes to synchronize these flows of control back into one.

Algorithm 2:

Input: an execution order hierarchy *h* generated by Algorithm 1.

Output: a workflow that is equivalent to the input sequence of action nodes to Algorithm 1.

1) For each action node v_i in h do,
If v_i has $n > 1$ outgoing directed edges, then

Add a new fork node *f* and a directed edge from v_i to *f*.

Make *f* to have *n* outgoing directed edges.

2) For each action node v_i in h do,

If v_i has $n > 1$ incoming directed edges, then

Add a new join node *j* and a directed edge from *j* to v_i.

Make *j* to have *n* incoming directed edges.

3) According to the directed edges in h, connect the added fork nodes and the join nodes and the rest of the action nodes in h.

4) If h has more than one root, add a fork node at the beginning of the resulting workflow such that it has out-going directed edges to all of the roots.

5) If h has more than one sink action node, add a join node at the end of the resulting workflow such that it has in-coming directed edges from all of the sink action nodes.

Example 12: In Fig 6, A_1 has two out-going directed edges. Thus, Algorithm 2 adds a fork node that has two out-going directed edges and a directed edge from A_1 to that fork node. Algorithm 2 does the same to A_2. A_3 has two in-coming directed edges. Thus, Algorithm 2 adds a join node that has two in-coming directed edges and a directed edge from that join node to A_3. Similarly, Algorithm 2 does the same to A_4. According to the directed edges in *h*, Algorithm 2 then connects the added fork nodes to the added join nodes and also the rest of the action nodes by directed edges. Because the execution order hierarchy has two roots, Algorithm 2 adds a fork node at the beginning of the workflow with out-going directed edges to the two roots. Since there are three sink action nodes, Algorithm 2 adds a join node at the end of the workflow to merge all of the flows of control back into one. The resulting workflow is shown in Fig 7. □

Theorem 3: Given a sequence of $n \geq 1$ action nodes v_1, v_2, …, v_{n-1}, v_n, our methodology generates an equivalent workflow from the sequence of action nodes.

Proof sketch:

The resulting workflow in UML notation is a simple translation from the execution order hierarchy of Definition 9. By Theorem 2, Algorithm 1 generates the execution order hierarchy of Definition 9 from the sequence of action nodes. It is therefore sufficient to consider the execution order hierarchy generated by Algorithm 1. We proceed by induction on the number *n* of action nodes in the sequence. When $n = 1$, it is clear that the sequence of action nodes is equivalent to the

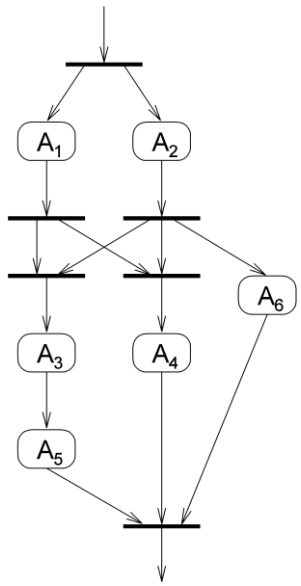

Fig.7. The resulting workflow generated from the execution order hierarchy in Fig 6.

execution order hierarchy because both of them have only one action node. We assume the induction hypothesis is true for $n = k$ for some $k \geq 1$. When $n = k+1$, the sequence has $k+1$ action nodes where v_{k+1} is the last action node added to the sequence. Step 2 of Algorithm 1 adds the ordered pair (v_i, v_{k+1}) to *s* for each $i \leq k$ such that v_i and v_{k+1} are not independent. Step 3 of Algorithm 1 adds the directed edge $v_i \longrightarrow v_{k+1}$ to the execution order hierarchy. As such, it is clear that if there is not a direct path from one action node v_i to another action node v_j in the execution order hierarchy, they are independent and thus v_i can be executed before v_j, or v_j can be executed before v_i, or v_i and v_j can be executed concurrently. Hence, the generated execution order hierarchy is equivalent to the original sequence of action nodes. □

IV. Case Study

The case study is drawn from a representative workflow ("phase" in research client terminology) from a much larger set of workflows at a large organization which serves as a broker between major international customers and multiple US

suppliers ("Broker"). Broker provides a wide range of products and services to their customers, from individual items to complex integrated systems. Their overall workflow breaks into four sequential major "phases." The first phase is initial interest from a customer in a broad product category, where the customer works with Broker to develop a structured expression of interest for a particular product set with a rough-order-of-magnitude (ROM) cost estimate. The second phase involves contract development (CD), where Broker develops and structures the contract to be executed in phase three, contract execution (CE). The fourth phase is contract closure. Elements of phase two and three are illustrated in the case study. Other phases are not shown for simplicity.

We provide two examples of workflow redesign in a case study. The first example introduces intra-phase concurrency in phase two, illustrated notionally in Fig 8, with a goal to reduce processing time within phase two. The second example introduces concurrency in inter-phase processes to convert certain non-deterministic processes to deterministic to reduce later contractual rework that causes multi-month delays. The goal in inter-phase workflow redesign is to reduce total system time.

A. *Intra-phase concurrency*

Phase two workflow translates the ROM developed in phase one into a formal itemized breakdown of costs and other contractual terms. Some phase 2 actions involve communications with US suppliers to get accurate cost and delivery information, interactions with US regulatory agencies to ensure appropriate legal authority to export is maintained, and interactions with customers. All new orders, whether for component parts or complex systems, flow through phase two.

In addition, modifications or amendments to existing orders also require the modified or amended orders to be reviewed through all phase two steps again to ensure the updated order is capable of being fulfilled during CE in phase three. Modifications to orders may be initiated in phase three for many reasons and may include, for example, transportation changes, corrections of misspecification of items or contractual terms, inadvertent errors, or changes in delivery time for one or more items in a particular order. The interactions with customers during CE quite often yield an amendment of the order that introduces a new workflow so the amended order can be properly reviewed and re-signed before once again proceeding through CE.

The duration of orders proceeding through phase two and three may span anywhere from several months to two years or more, depending on whether a particular order involves spare components parts, one or more major systems that are built to order, or a complex system comprised of complex machinery, on-site training upon delivery, and subscriptions for ongoing upgrades to technical manuals. With this complexity and multi-year order duration, modifications and amendments to orders comprise more than half of all orders. Thus one order may proceed through process workflows in phases two and three multiple times over the contract duration.

Our task was to map and model workflows to estimate Broker's employee workload levels. A major difficulty for Broker, particularly for complex integrated systems, is long lead-times associated with receiving 1) detailed cost and contract data from US suppliers, and 2) appropriate documentation for legal export from regulatory agencies. While the workflows are largely deterministic, the stochastic nature of time to receive responses from suppliers and customers means that the employees may be nearly idle while in the midst of one of these lead times, or extremely busy in transitioning from one customer action to processing an action for another customer. The latter may induce delays due to bottlenecks in flow dependent actions, which extends lead times and induces additional costs.

Broker's performance is evaluated by many customer-oriented metrics, including total processing time for each phase. Broker recently instituted targets for reducing total time for processing workflow in phase two for very complex

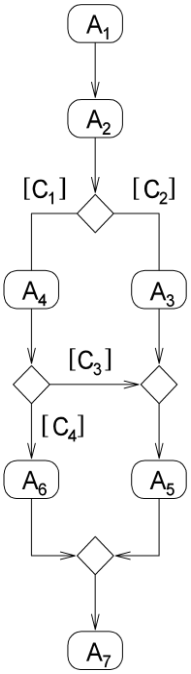

Fig.8. A real-world case study workflow

systems from l_1 to l_2, a reduction of 25%, which is measured in months. If the flows of control of phase 2 can be reengineered from serial to concurrent, reductions in processing time will help reduce maximum lead times to l_2. As such, reengineered workflows would reduce total lead time by days or weeks, and thus 1) greatly improve responsiveness of Broker and Broker's suppliers to customers, 2) reduce task load imbalances on Broker's employees, and 3) reduce cost to Broker.

Fig 8 shows the workflow based on the case study. Due to the confidentiality agreement, we cannot disclose the actual steps and the inputs and outputs of the workflow. Instead, we replace them by generic names. The workflow in Fig 8 cannot be hastened because, except for the sequence A_1, A_2, there is only a single action node in between any pair of branch node or merge node. However, a more in-depth analysis reveals that A_3 is not atomic. In fact, A_3 can be decomposed into three pairwise

independent action nodes $A_{3.1}$, $A_{3.2}$, and $A_{3.3}$. As a result, they can be executed concurrently, as shown in Fig 9. Likewise, A_5 can be decomposed into three atomic action nodes $A_{5.1}$, $A_{5.2}$ and

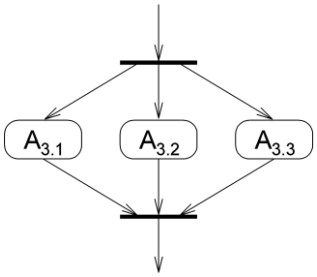

Fig.9. Rearranging three independent action nodes of A_3

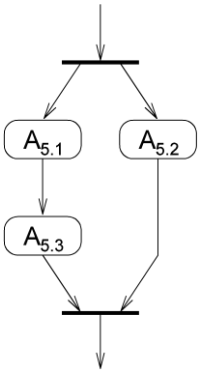

Fig.10. Rearranging three action nodes of A_5

$A_{5.3}$ of which $A_{5.1}$ and $A_{5.2}$ are not independent while the other pairs are independent. Thus, they can be rearranged into a workflow like the one in Fig 10.

The result of these reengineered workflows is reduced processing time to complete phase two, consistent with organizational goals, without an increase in resources or violations of workflow requirements or constraints. This example serves to illustrate how analysis and decomposition of processes provides opportunities to introduce workflow concurrency to serial tasks.

Currently, the principal barrier to introducing these concurrencies is lack of authority on the part of mid-level employees to move forward without prior approval by higher-level employees. For example, by policy $A_{3.2}$ currently has to wait on high-level approval on $A_{3.1}$ before it may proceed, although there is no strict reason, from the point of view of available resources $r_1 \ldots r_n$ other than the approval itself (r_j). We have therefore recommend Broker review some of the policies that introduce serial approval steps to evaluate their necessity.

Our analysis also identified opportunities where inter-phase workflows across phases two and three are designed to be independent, but analysis revealed that a significant percentage of contract modifications and amendments are the result of internal changes in how contracts are executed, as well as misspecifications and errors identified earlier require modifications and amendments above and beyond those initiated by the customer. The resultant workflows increase resource requirements, increase total processing time, and consume capacity that is constrained during fluctuations in daily and weekly workflow. We analyze this case in the next example.

B. *Inter-phase concurrency*

In addition to introducing concurrency to improve processing time in serial processes, such as in the phase two example above, our analysis identified non-serial processes in separate phases that unintentionally introduce non-standard practices in both phases. As noted, this causes inconsistencies and errors that make specific workflow processes non-deterministic. Due to the complex nature and long duration of many of these contracts, it is not surprising that these problems occur. But when they occur and contract modifications or amendments are necessary, the updated version of the contract must be re-processed through all of the phase two processes before the updated contract can be executed. This causes another full pass through phase two and through many of the processes in phase three. It is estimated the added workload is in the range of thirty percent of total workflows. In addition, often months can pass while the contract is re-specified, reviewed and re-signed. This affects resource costs, system capacity, total processing time, and customer satisfaction.

To address these problems, the goal in this workflow redesign example is to make more of the phase three processes deterministic by using workflow redesign to leverage the benefits of concurrency in phase two, specifically in contract review, verification, and structuring, to eliminate many of the latent contract problems identified only during CE in phase three. The original workflows are show in Fig.11 where the relevant processes for phase two are denoted as $A_7 - A_{12}$, and the processes for phase three are shown as $B_1 - B_8$. The workflows for phase two and three are f_5 and f_6, respectively. The final phase, contract closure, is not addresses in the process redesign and is denoted C_1 for simplicity.

The workflows of redesign interest in Fig.11 are discussed next. A_8 denotes a review process of customer requirements for a particular order by a CD manager ("CDM"). Due to the decreased phase two processing time targets for process time imposed by senior management, CDMs perform these review with input from suppliers that involve particular supplier requirements prior to forwarding the finalized customer requirements to the next phase. In A_{10}, the CDM translates these requirements from A_8 in to detailed contract requirements. In A_{11}, contract requirements are forwarded to the next process where the contract requirements are embedded in a finalized contract, ready for customer signature.

These three processes in the latter part of phase two are targeted for workflow redesign because many of the latent errors and misspecifications that arise months later in CE during phase three could be eliminated or mitigated during phase two.

It is noted that when working with organizational processes rather than information processing technology, it can be difficult to convert processes to deterministic. This challenge increases when a team that manages one phase for a key customer in phase two are located physically apart from a related customer team that manages in phase three. This example in our case study investigates how information uncertainty and inconsistencies between related contract specification teams and corresponding contract execution teams cause delays in contract execution,

We employ workflow concurrency as follows to accomplish this objective. For the workflow redesign of processes A_8, A_{10} and A_{11}, we first decompose workflow f_5 in Fig 11 into separate workflows and label them f_{5A}, f_{5B}, f_{5C}, f_{5D} and f_{5E} and f_{5F} (Fig 12). These workflows correspond to the six phase two processes identified in Fig 11 namely A_7, A_8, A_9, A_{10}, A_{11} and A_{12}, respectively. The three workflows, f_{5B}, f_{5D} and f_{5E}, will be redesigned in this example.

For the first workflow redesign in process A_8 (Fig 12), we introduce concurrent workflows using a fork node following A_7, where the new concurrent workflows denoted $f_{5B.1}$ and $f_{5B.2}$ correspond to processes A_8 and $B_{R1\text{-}8}$. This preserves the phase two review in A_8 by the CDM and introduces a concurrent review, denote $B_{R1\text{-}8}$, by the CE manager ("CEM"). The CEM uses a standardized review process that scans and identifies common errors, misspecifications, and other contractual problems that later drive increased workload.

The CEM review in $B_{R1\text{-}8}$ in Fig 12 includes issues typically identified in phase three during any of the processes B_1 through B_8 that could have been avoided through better initial specification of customer requirements and how they match Broker's organizational capabilities to meet the requirements. These review are processed through a join node into a final customer requirements document (not shown for simplicity).

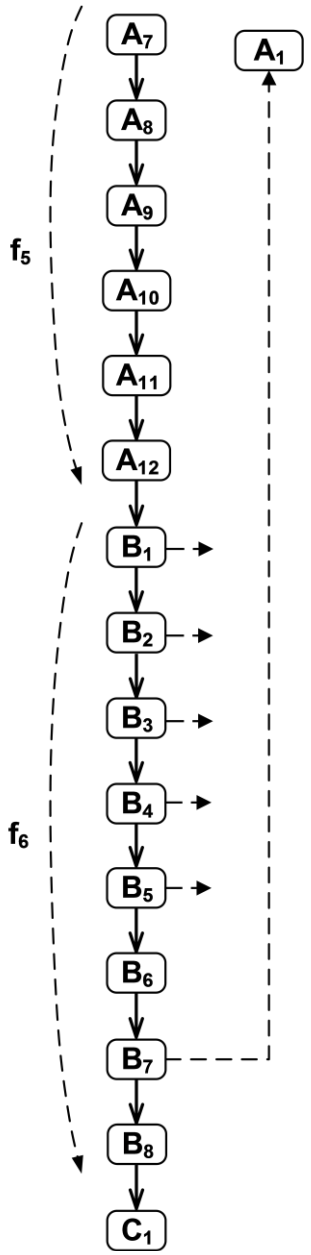

Fig.11. Original workflows for phase two (f_5) and phase three (f_6).

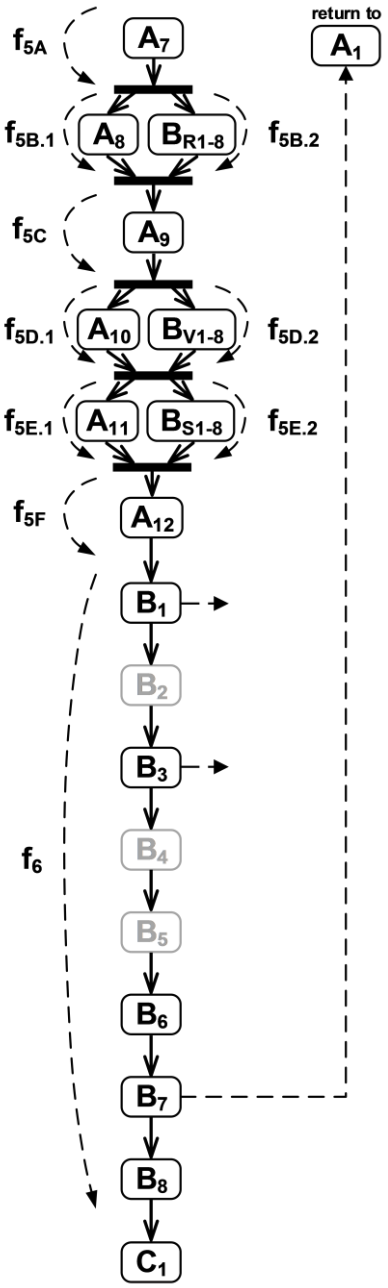

Fig.12. Redesigned intra-phase workflows using concurrency.

The second workflow redesign after the fork node following A_9, denoted $f_{5D.1}$ and $f_{5D.2}$ in Fig 12, introduces a concurrent review to verify detailed contract requirements in A_{10} and B_{V1-8}, which is driven by customer requirements derived in process A_8. Thus process A_{10} is also preserved, and the concurrent CEM review, B_{V1-8}, involves a standardized review to identify problematic contractual clauses and misspecifications in the draft of detailed contract clauses to ensure the contract clauses conform to customer requirements, Broker and supplier capabilities, relevant regulatory requirements, and international trade documentation and transportation requirements.

The detailed contract requirements resulting from A_{10} and B_{V1-8} in Fig 12 join after these processes, and immediately fork again to incorporate a concurrent review of the contract structure prior to final signatures. The CDM continues to perform a review of the contract structure in A_{11} while the CEM concurrently performs a review of known issues in contract structures that can affect phase three processing time, resource requirements, and customer satisfaction. Process flow then proceeds to A_{12}.

These concurrent reviews introduced in phase two are considered to be largely resource neutral, in that for most major contracts they already occur as part of a phase three process to identify problems that will occur later in time. Not all of these are caught in the phase three reviews during CE until they are ready for execution. In the longer term, concurrent reviews may be resource reducing when problems caught early during contract development eliminate the later need to expend resources and time to correct the contract or related supplier issues. Examples of these are shown in Fig 12, where processes B_2, B_4 and B_5 are represented in grey, signifying that these later processes do not need to be invoked for a particular contract if latent contract problems were identified and corrected in the concurrent reviews in phase 2.

C. *Benefits of concurrent reviews*

Introducing concurrent reviews for specifying customer requirements, detailed contract clauses, and contract structure much earlier in the process accomplishes several objectives. First, it identifies problems earlier, allowing more time to effectively react at lower cost while trying to maintain high levels of customer satisfaction. Second, by identifying and rectifying problems prior to formal contract signing, it avoids introducing contract modifications and amendments that must be processed through the full processes of phase two. This considerably reduces total process time to perform on the contract and initiate contract closure, reduces cost and resource consumption, reduces over-capacity utilization through reduced work flows, and improves customer satisfaction and Broker morale through improved customer performance and reduced execution of wasteful practices. It also avoids the requirement to run all contract modifications and amendments through all phase two processes, regardless of the extent of the contract modification or amendment.

These examples serve to introduce two type of concurrency that can be employed in organizational processes, with an intra-phase example of concurrency that employs decomposing serial processes into concurrent processing, and an example of non-serial inter-phase concurrency being employed earlier in organizational processes to convert certain non-deterministic work flows into deterministic workflows.

It is interesting to note that these workflow redesign opportunities existed prior to Broker introducing the phase two tightening of process performance times with the intent to better serve the customer. Thus the new shorter lead times to contract signature and execution did not create these problems, but they did exacerbate them. If concurrent workflows are introduced at processes A_8, A_{10} and A_{11} in phase two and maintained to effectively resolve the issues identified in this paper, the additional time to process these concurrent reviews

is on the order of days or perhaps a week or two, while the delays that the latent problems induce can be measured on the order of six to twelve months in additional delays. However, because of the current physical, organizational and social separation between the CDMs and CEMs, Broker is facing resistance in increasing the role of the CEMs in phase two.

This illustrates that the most effective policy to improve customer satisfaction is not to optimize time in a single phase at the expense of subsequent phases, but to consider the whole system and work to reduce total time from contract development to contract closure. Using concurrency in workflows can be effective in real organizations, and non-deterministic work flows can, in some cases, be converted to deterministic processes to reduce processing time and resource consumption while simultaneously improving customer satisfaction.

V. Conclusions & Future Research

Adopting some of the concepts of activity diagrams, this paper defined a mathematical foundation for workflows. It then defined deterministic workflows, a class of workflows that have the predictable property that they will produce the same results if they have to be executed over and over again. After which, it defined flow independent workflows, a class of workflows that are deterministic. A methodology was then presented, which can be applied to each flow of control of a flow independent workflow. The methodology rearranges the action nodes in a flow of control so that some action nodes can be executed concurrently. We also supplement the paper by a real-world workflow that demonstrates the usefulness of our methodology.

Although introducing concurrency to flow independent workflows can be done in polynomial time, ultimately the workflows must be executed by humans or machines or any combination of the two. Hence, another relevant research problem is to assign the action nodes of a flow independent workflow to processors, which will eventually be responsible for executing the action nodes.

However, the problem of assigning action nodes of a flow independent workflow to a fixed number of processors so that the completion time of the workflow is under a certain time constraint does not seem to have a polynomial time solution. In fact, it seems like the problem is NP-complete. One can easily see the problem that given a number m of action nodes of a flow independent workflow that are pairwise independent, assigning them to $n < m$ processors to be executed so that the completion time of the workflow is under a specific time constraint subsumes the bin-packing problem, a well-known NP-hard problem. (In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers each of volume v in a way that minimizes the number of bins used.) We will report the investigation of this problem in a future journal paper.

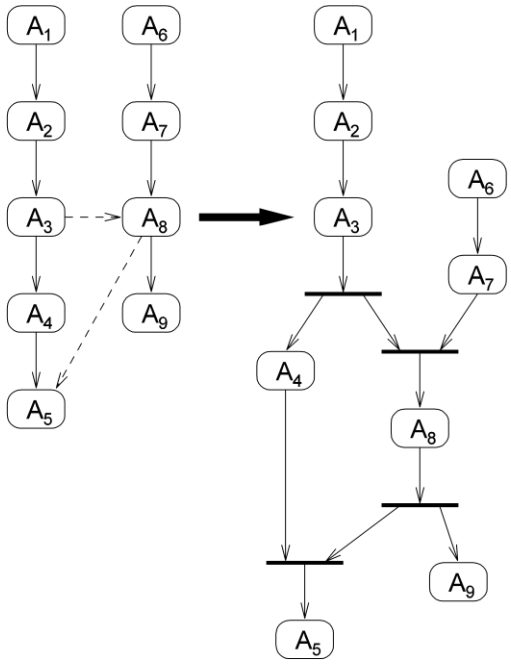

Fig.13. Adding fork nodes and join nodes to a not necessarily flow independent workflow.

Although flow independent workflows have the desirable property that being deterministic, as illustrated in this paper not all real-world workflows are flow independent. Another type of workflows have collaborating concurrent flows of control that pass data back and forth with one another. Although they may not be deterministic, nevertheless collaborating flows of control are quite common. Therefore, it is still useful to apply our methodology to a not necessarily flow independent workflow to speed it up.

The first step, however, is to identify the parts of the workflow to which our methodology can be applied. Although much more work is needed, we do have some preliminary ideas on this third approach. Fig 13 shows a workflow with two flows of control that are not independent. The dashed lines in Fig 13 denote data are passed from a sender action node to a recipient action node, and the recipient action node must wait for the data become available from the sender. In Fig 13, A_8 requires some data items from A_3 and A_8 send some data items to A_5.

Fig 13 also shows that after a careful analysis of the situation, in fact we can add fork nodes and join nodes to make the explicit modeling of passing data unnecessary. In addition, the added fork nodes and join nodes also delineate the sequences of action nodes to which our methodology can be applied. As shown in Fig 13, our methodology can now be applied to the sequences A_1, A_2, A_3 and A_6, A_7. We will also report these efforts in a future journal publication.

REFERENCES

[1] I. Arpinar, U. Halici, S. Arpinar, and A. Dogac, "Formalization of workflows and correctness issues in the presence of concurrency," vol. 7, pp. 199-248, April, 1999.

[2] P. A. Bernstein, and N. Goodman, "Concurrency Control in Distributed Database Systems," vol. 13, pp. 185-221, June 1981.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, 2nd edition, Addison-Wesley, 2005.

[4] F. Cédric, F. Dirk, and V. Hagen, "The relationship between workflow graphs and free-choice workflow nets," Information Systems, in press.

[5] Y. Gao, H. Ma, H. Zhang, X. Kong, and W. Wei, "Concurrency Optimized Task Scheduling for Workflows in Cloud," 2013 IEEE Sixth International Conference on Cloud Computing, pp. 709-716.

[6] M. Hammer, "Reengineering Work: Don't Automate, Obliterate," Harvard Business Review, vol. 68, pp. 104-112, Jul/Aug1990.

[7] J. Puustjarvi, "Workflow concurrency control," Computer Journal, vol. 44, pp. 42-53, 2001.

[8] X. S. Sherry, and J. L. Zhao, "Formal Workflow Design Analytics Using Data Flow Modeling," Decision Support Systems, vol. 55, pp. 270-283, Feb 2014.

[9] Y. Wang, and P. Lu, "Dataflow detection and applications to workflow scheduling," Concurrency And Computation-Practice & Experience, vol. 23, pp. 1261-1283, Aug 2010.

[10] Y. Wang, and P. Lu, "DDS: A deadlock detection-based scheduling algorithm for workflow computations in HPC systems with storage constraints," Parallel Computing, vol. 39, pp. 291-305, 2013.

[11] M. Wang, X. Zhang, L. Zhu, and L. Liao, "Trust-based workflow refactoring for concurrent scheduling in service-oriented environment," Concurrency And Computation-Practice & Experience, vol. 25, pp. 1879-1893, 2013.

[12] Object Management Group, "The Unified Modeling Language™ - UML," http://www.uml.org/