行政院國家科學委員會專題研究計畫 成果報告

一個在編輯流程軟體規格時的遞增性資源分配分析之研究
研究成果報告(精簡版)

計 畫 類 別 ：個別型
計 畫 編 號 ：NSC 94-2213-E-009-135-
執 行 期 間 ：94 年 08 月 01 日至 95 年 07 月 31 日
執 行 單 位 ：國立交通大學資訊工程學系(所)

計 畫 主 持 人：王豐堅

計畫參與人員：博士班研究生-兼任助理：吳建德
碩士班研究生-兼任助理：林友涵、簡璞

處 理 方 式 ：本計畫可公開查詢

中 華 民 國 96 年 01 月 30 日

行政院國家科學委員會補助專題研究計畫 ■ 成 果 報 告
□期中進度報告

一個在編輯流程軟體規格時的遞增性資源分配分析之研究

計畫類別：■ 個別型計畫　□ 整合型計畫
計畫編號：NSC　94-2213-E-009-135
執行期間：　94 年 8 月 1 日至　95 年　7　月　31　日

計畫主持人： 王豐堅
計畫參與人員： 吳建德、林友涵、簡璞

成果報告類型(依經費核定清單規定繳交)：■精簡報告　□完整報告

本成果報告包括以下應繳交之附件：
□赴國外出差或研習心得報告一份
□赴大陸地區出差或研習心得報告一份
□出席國際學術會議心得報告及發表之論文各一份
□國際合作研究計畫國外研究報告書一份

處理方式：除產學合作研究計畫、提升產業技術及人才培育研究計畫、列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

執行單位：國立交通大月資訊工程學系網路軟體工程實驗室

中　華　民　國　96　年　1　月　30　日

摘要

工作流程管理技術提供對於企業內複雜商業流程(*business process*)的模組化管理，一般說來一個工作流程管理系統(*WFMS, Workflow Management System*)是由兩個主要的元件–設計環境與執行環境–所組成，為了要確保所建構目標工作流程系統的正確性，工作流程規格的結構、時序以及資源正確性都必須被檢驗。在本報告中，我們以遞增性的方法，針對設計過程中每次設計動作之後的資源統一性以及時序限制進行分析，並且提供適當的資訊給設計者以及系統維護者，以求幫助正確的建構有效率的工作流程應用程式。

關鍵字：工作流程管理系統，遞增性方法，資源統一性，時序限制

Abstract

Workflow management technology helps modulizing and controlling complex business processes within an enterprise. Generally speaking, a workflow management system (*WfMS*) is composed of two primary components, a design environment and a run-time system. Structural, timing and resource verifications of a workflow specification are required to help assure the correctness of the specified system. In this paper, we address an incremental methodology to analyze resource consistency and timing constraints after each editing activity of a workflow specification and to provide proper feedbacks to designer or maintainer of the workflow specification.

Keywords: Workflow Management System, WfMS, Incremental Methodlogy, Resource Consistency, Timing Constraints

The following paper published in **Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)** - Pages: 122 - 129

An Incremental Analysis to Workflow Specifications

Hwai-Jong Hsu, Da-Li Yang, and Feng-JianWang

Department of Computer Science and Information Engineering, National Chiao-Tung University

{hjhsu, dlyang, fjwang}@csie.nctu.edu.tw

Abstract

Workflow management technology helps modulizing and controlling complex business processes within an enterprise. Generally speaking, a workflow management system (WfMS) is composed of two primary components, a design environment and a run-time system. Structural, timing and resource verifications of a workflow specification are required to help assure the correctness of the specified system. In this paper, we address an incremental methodology to analyze resource consistency and timing constraints after each editing activity of a workflow specification and to provide proper feedbacks to designer or maintainer of the workflow specification.

1. Introduction

Electronic workflow integrates business rules and staffs inside an enterprise into an automatic information system. Inside a flow, the process (activities) and information flow between them are specified according to the business rules to accomplish specific tasks [1][12]. Furthermore, workflow specifications schedule tasks, and coordinates human resources and information system [13]. Modern workflow management systems (WfMSs) support environments for both workflow design and workflow enactment.

To assure the correctness of executing a workflow specification, analyses on structural integrity, temporal correctness, and resource conflicts are required. Structural analysis is given precedence over analysis of the other two, since the rest give a better result on the specification whose structure has been analyzed. Various methodologies for structural and temporal analysis of workflow system specifications have been developed and proved effective [3][5][6][7][9][10]. However, the methodologies proposed work only for static and total verifications of resource consistency. These methodologies might be ineffective in analyzing influence of workflow specifications with large amount of processes after each modification, and provide insufficient information to designers and maintainers [2].

In this paper, we present an incremental methodology for analysis of resource constraints in structuralized workflow specifications. The analysis works after each editing operation of workflow specification, and provides more precise and effective information to the designer/maintainer directly. Our approach focuses on influence of each editing operation, and therefore is more efficient than traditional methodology.

The paper is structured as follow: Section 2 describes the definitions and notations us. An incremental algorithm for analysis of resource conflicts in workflow specifications is constructed in section 3, and the temporal factor is considered in section 4. The time complexity and conclusion of our methodology are discussed in section 5 and 6.

2. Definitions and Notations

Directed acyclic graph (DAG) is a simplified model for workflow specifications [3] [8]. DAG can be applied for verifying consistency of control and data flows. We describe workflow specifications based on DAG with a five-tuple (N, F, R, S, E). S and E represent the start and end processes of the workflow. N represents the set of processes which can be distinguished as activity and control process. An activity process describes a task and control processes are and-split, and-join, xor-split, and xor-join process defined in [WfMC TG]. Each flow f in F, represented as (n_i, n_j) means a *transition* from process n_i to process n_j. Flow f is called an *out*-flow of process n_i, and an *in*-flow of process n_j; besides, n_i is the *source* process of flow f and n_j is the *sink* process of flow f. R is a set of sets of resources associated to each activity process in a workflow.

Definition 1 *(Workflow Specification)*

Workflow Specification ws = (N, F, R, S, E)

(1) N: a set of processes, where $\forall n \in N$, n.TYPE = {ACTIVITY, AND-SPLIT, XOR-SPLIT, AND-JOIN}

(2) F: a set of flow, where $\forall f \in F$, $f = (n_i, n_j)$, $n_i, n_j \in N \cup \{S, E\}$

(3) R: a set of sets of resources referenced by each process, where $\forall R_i \in R$, $R_i = \{r_n \mid r_n$ is a resource accessed by n_i, $n_i \in N$, n_i.Type = ACTIVITY$\}$

(4) S and E are the starting and the ending process correspondingly. S, E $\in$ N

[2] describes path, reachability, distance and ancestor. In order to construct our algorithm, we define path, reachability, distance, ancestor, distance to ancestor, nearest common ancestor and control block formally in following definitions.

Definition 2 *(Path)*

A *path* $p = (n_1, n_2, \ldots, n_t)$ where $\forall\ i, 1 \le i \le t-1$, $(n_i, n_{i+1}) \in F$

A *path* p is acyclic, if $\forall\ n_i, n_j \in p, i \ne j, n_i \ne n_j$

The *length* of an acyclic path p is denoted as |p|. |p| = t

Definition 3 *(Reachability)*

Process n_j is *reachabl*e from process n_i if there is an acyclic path $p = (n_i, \ldots, n_j)$ Reachable(n_i, n_j) is a boolean function to denote whether n_j is reachable from n_i

$$\text{Reachable}(n_i, n_j) = \begin{cases} \text{TRUE, if } \exists \text{ a path} \\ \qquad p = (n_i, \ldots, n_j) \\ \text{FALSE, otherwise.} \end{cases}$$

$$\text{Distance}(n_i, n_j) = \begin{cases} \text{MIN}(\ \{\ |p|\ \mid\ p = (n_i, \ldots, n_j) \text{ or} \\ \qquad p = (n_j, \ldots, n_i)\ \}), \text{ if} \\ \qquad \text{Reachable}(n_i, n_j) \text{ or} \\ \qquad \text{Reachable}(n_j, n_i) \\ +\infty, \text{otherwise.} \end{cases}$$

Definition 4 *(Distance)*

Distance(n_i, n_j) is a function returns the distance between two processes n_i and n_j

Definition 5 *(Ancestor)*

Process n_i is an *ancestor* of n_j if Reachable(n_i, n_j) = True

Process n_i is a *common ancestor* of n_j and n_k if n_i is an ancestor of both n_j and n_k

Definition 6 (*Distance to the Common Ancestor*)

Let n_i is the common ancestor of n_j and n_k. We define distance of n_j, n_k to its common ancestor n_i as DCA(n_i, n_j, n_k)

DCA(n_i, n_j, n_k) = MIN(Distance(n_i, n_j), Distance(n_i, n_k))

Definition 7 (*Nearest Common Ancestor*)

Process n_i is a *nearest common ancestor* of n_j and n_k where DCA(n_i, n_j, n_k) is the shortest among all the common ancestor of n_j and n_k.

In our algorithm, function NCA(n_i, n_j) is defined as a function returns the nearest common ancestor of process n_i and n_j

Definition 8 *(Control Block)*

B = (n_s, n_e, N_B) is a Control Block where n_s, $n_e \in N$, N_B is a subset of N. N_B contains each process n where Reachable(n_s, n) = Reachable(n, n_e) = true.

B.start = n_s, B.end = n_s and n_s.TYPE = AND_SPLIT if and only if n_e.TYPE = AND-JOIN or n_s.TYPE = XOR_SPLIT if and only if n_e.TYPE = XOR-JOIN.

Two control blocks B_1 and B_2 are said to be distinct if and only if $N_{B1} \cap N_{B2} = \phi$, B_1 is said to be totally contained by B_2 if and only if $N_{B1} \subseteq N_{B2}$.

In order to simplify the discussion of out algorithm, we assume that our algorithms are adopted only for *well-formed workflow specification*. The well-formed workflow specification is defined in definition 9.

Definition 9

(*Well-formed Workflow Specification*)

A well-formed workflow specification is a workflow in which all the processes are connected by flows and any two control blocks are either distinct or totally contained by one another.

3. An Incremental Algorithm for Analysis of Resource Conflicts in Workflow Specifications

In this section, first, we introduce the conditions leading to resource conflicts. Second, editing operations on activity processes potentially causing resource conflicts are described. Third, our algorithms to analyze the resource conflicts to the corresponding operations in a workflow specification are presented. The influence caused by editing operations on processes other than activity processes is not discussed.

3.1. The Conditions and Editing Operations Leading to Resource Conflicts

A resource conflict may occur when two or more activity processes refer to one common (sharable) resource concurrently. In a workflow specification, two activity processes are potentially resource conflict when the following two conditions simultaneously hold [2].

(1) Resource Dependency: Two activity processes are resource dependent if and only if they refer to the same sharable resource.

(2) Potential Concurrent Execution: Two activity processes are potentially executed in parallel if and only if they are not on the same path, and one of their nearest common ancestors is a control process of AND-SPLIT.

Definition 9 *(Resource Dependency)*

Let $n_i, n_j \in N$, $i \neq j$, and $n_i.Type = n_j.Type = ACTIVITY$. n_i and n_j are *resource dependent* if and only if $R_i, R_j \in R$, $R_i \cap R_j \neq \phi$

Definition 10 *(Potential Concurrent Execution)*

Let $n_i, n_j \in N$, $i \neq j$, and $n_i.Type = n_j.Type = ACTIVITY$. n_i and n_j are potentially executed concurrently if and only if $\forall$ path p, n_i in p, n_j not in p, n_i and n_j has a nearest common ancestor n_k of AND-SPLIT type where $DCA(n_k, n_i, n_j) > 0$

The following editing operations might produce or eliminate resource conflicts: (1) Adding or deleting a resource reference associated with an activity process, (2) Adding or deleting an activity process within a workflow specification. Both operations might affect resource dependencies within a workflow specification. Since the editing operations on processes other than activity processes are not discussed, there's no operation which directly changes potential concurrent executions.

Not all resource dependencies result in resource conflicts. There're no resource conflicts between two distinct processes which are on the same path, or whose nearest common ancestor is not AND-SPLIT when they are not in the same path.

3.2. The Algorithm for Detecting Resource Conflicts in a Well-Formed Workflow

To describe our algorithm clearly, we describe *split paths* and *resource lists* on a split path formally and in order to simplify our algorithm, the type of each resource reference is ignored.

Definition 11 *(Split Path)*

$\forall$ control block $B = (n_s, n_e, N_B)$, a path $p_s = (n_s, n_1, \ldots, n_k, n_e)$ where $n_1, \ldots, n_k \in N_B$ is a split path of B

Definition 12 *(Resource List on a SPLIT PATH)*

RL_{SPi} is a resource list on some split path SP_i, where

(1) $\forall$ n_j in SP_i, $RL_{SPi} = \bigcup R_j$

(2) $\forall$ r, r is a resource, $RL_{spi}.ProcessRef(r) = \{n_k | n_k \text{ in } SP_i, r \in R_k\}$

Resource lists are indexed by split paths among each control block, and we assume that whenever a split path is created during editing of workflow specification the corresponding resource list is also constructed. Each resource list records resource references of processes along

with the corresponding split path. The information used within an incremental algorithm is recorded in the resource list on each split path, and is updated when resource references and activity processes are added in or deleted from the workflow specification.
Besides we define the notation for resource conflicts and how to record resource conflict for each process for usage of incremental algorithm.

Definition 13 *(Resource Conflict)*

(r, n_i, n_j) is a resource conflict if and only if r is a resource, n_i and $n_j \in N$, $r \in R_i$, R_j, and n_i and n_j potentially concurrent in execution. W.L.O.G, (r, n_i, n_j) and (r, n_j, n_i) is considered as the same in our discussion.

Definition 14

(Resource Conflict set for a Process)

$n.Rd(r) = \{n_1, \ldots, n_k\}$ is and only if for any i = 1,.., k and $n \neq n_i$, (r, n, n_i) is a resource conflict

The notation in definition 14 would be used in section 4.

There are two algorithms CSP (Collect Split Paths) and CRD (Check Resource Dependency) constructed. CSP is used to collect the processes which might execute concurrently and CRD is used to detect the resource dependencies among them. Since not all resource dependencies result in resource conflicts, the CSP algorithm is executed first and the result set is passed to CRD for execution. CSP and CRD are applied when adding or deleting a resource *r* associated with a process *n*. Adding a process can be viewed as adding multiple resources to the process, and deleting a process can be viewed as deleting all the resources referenced by the process.

In CSP algorithm, first, a flow queue Q is initialized. Q is used to store the flows when CSP back tracks the processes starting from the target process *n*. At line 2 all the in-flows of *n* are put into Q. At line 5, the first flow $f = (n_i, n_j)$ in Q is dequeued and checked for its type. If the algorithm finds that the source process of the flow, n_i, is typed as AND-SPLIT, the resource list along with the split path which the process *n* belonging to is updated at line 8. The resource list is updated according to what editing operation that triggers the function. Adding resource reference to the target process can be viewed as adding resource reference to the split path where the target process *n* belongs to, and removing resource reference from the target process *n* can also operate in similar way. At line 9 to 10, the split paths from the AND-SPLIT process other than the path from the target process *n* are collected into the result SPLIT_PATH set CSP. At line 12, the in-flows of the n_i are enqueued into Q, and the loop start from line 4 to line 12 continues until Q is empty.

SPLIT_PATH SET CSP
(workflow specification ws,
process n, resource r**)**
1. CSP = ϕ
2. flow queue Q = ϕ ; // initialize flow queue
3. $\forall$ in-flow f of n, Q.enqueue(f); // put the inflow of process n into Q
4. while Q $\neq$ ϕ // back tracking ws from the process n
5. Let f = Q.dequeue() and assume that f = (n_i, n_j)
6. //collect information about parallel split path when //meet an AND-SPLIT
7. if (n_i.TYPE = AND-SPLIT) then
8. $\exists$ a split path SP_k where n_i, n $\in$ SP_k, Update Resource List of SP_k according to the editing operation on process n and r
9. $\forall$ out-flow of n_i f', where f' = (n_i, n_k), $n_k \neq n_j$
10. //collecting all the split paths other than the //source path into CSP
11. $\exists$ a split path SP_m where n_i, $n_k \in SP_m$, CSP = CSP + SP_m
12. $\forall$ in-flow f" of n_i, Q.enqueue(f")

We have shown that any process which is potentially concurrent to target process n must be concluded in some split path contained in SPLIT_PATH set CSP. With the SPLIT_PATH set collected in CSP algorithm, line 2 and 3 of CRD algorithm check the resource list along with each split path to see if the resource *r* is referenced by other processes in the split paths not containing process *n*. At line 4 we update the resource conflict list according to the resource conflict found in algorithm.

RESOURCE_CONFLICT SET CRD
(SPLIT_PATH set P, process n, resource *r* **)**
1. CRD = ϕ

```
2.  ∀ SPLIT_PATH SP_i ∈ P and n_j ∈ RL_SPi.ProcessRef( r )
3.    insert ( r , n, n_j) into CRD
4.    update n.Rd(r) and n_i,Rd(r)
```

4. An Incremental Algorithm to detect Resource Conflicts with Temporal Consideration

Two processes can have potential resource conflict as the last section describes, however, the conflict never happens if execution of both processes doesn't overlap. We define such resource conflict as resource conflict with temporal consideration.

In this section, first, the concept of Estimated Active Interval (EAI) is introduced. Second, Potential Overlapped Execution between activity processes is defined. The resource conflicts with temporal consideration are the resource conflicts which are potentially overlapped in execution. The incremental algorithm to detect resource conflict with temporal consideration is constructed.

4.1. Calculating EAI in a Workflow Specification

The earliest start time (EST) and the latest end time (LET) of process in a workflow can be calculated if the maximal and minimal durations of each process are described in a workflow specification. [4]. The time interval starts from EST to LET is named as Estimated Active Interval (EAI). The EAI for process *n* is denoted as [*EST(n), LET(n)*]. Reasonably, LET(n) must not be less to EST(n) to any process n. *EAI table* is used to record EAI values of all the processes in the workflow specification.

For the process n, *d(n)* and *D(n)* shows the minimal and maximal durations of *n*. For activity processes, the values are specified by the designer, and for control processes the values are initialized as zero. EST and LET of starting process are initialized as zero. Besides, EST and LET of the rest processes are calculated from their precedent process(s). In a well-formed workflow, there's only one precedent process for an activity, AND-SPLIT, or XOR-SPLIT process and there're multiple precedent processes to an AND-JOIN, XOR-JOIN, or End process. To lengthen D(n) postpones the LET of n and its following processes; on the contrary, to shorten the D(n) advances the LET of n and its following processes. To alter d(n) of the process n would not directly affect the EAI of n; however, to lengthen d(n) postpones the EST of n's following processes, and to shorten d(n) advances the EST of n's following processes.

After each temporal related editing operation, which means modification of D(n) or d(n) on some process n, the algorithm Calculate_EAI is adopted to calculate EAI for each effected process. After introducing the algorithm, we would show that the algorithm covers all the influenced processes. In this algorithm, we assume that there's no delay between end and start of each process

The workflow specification ws, the target process n, and a flag string *mode* are the input parameters of the algorithm Calculate_EAI. There're two values "*target*" and "*ripple*" for the flag string mode. When the Calculate_EAI is invoked when some temporal editing operation is committed, the flag string "*target*" is used, When the Calculate_EAI is invoked recursively by itself, the flag string "*ripple*" is used. At line 1 and 2, we store the original EST and LET value of target process *n* for later usage. At line 3 to 5, the EAI of process n which is typed as AND-JOIN is calculated. Process n can be fired only when all precedent processes of n are committed. Value of EST of n is the maximal earliest end time among all its precedent processes. The earliest end time of any process n_i is the summation of EST(n_i) and d(n_i). Since the D(n) is zero, LET of n is the maximal LET among all its precedent processes. At line 6 to 8, the EAI of process n which is typed as OR-JOIN or end process is calculated. Process n can be fired only when any precedent processes of n are committed. Value of EST of n is the minimal earliest end time among all its precedent processes. Since D(n) is zero, LET of n is the maximal LET among all its precedent processes. At line 9 to 11, EST value of the target process is equal to the earliest end time its precedent process, and LET value is the LET value of its precedent process plus the maximal working duration of itself. At line 13 to 16, the algorithm first stores the new EAI value of target process into storage. The algorithm will recursively continues when the algorithm is invoked after some editing operation or when EAI of the target process is changed and the ripple effect to its following processes must be calculated. The flag string "*ripple*" is used when Calculate_EAI recursively invoke itself.

```
VOID Calculate_EAI
  (Workflow Specification ws,
   Process n, String mode)
1.  EST_old = EST(n); // record the original values
2.  LET_old = LET_(n); // record the original values
3.  if (n.Type = AND-JOIN) then
```

```
4.      EST(n) = MAX( {EST(n_i) + d(n_i) |
                              ∀ flow f =    (n_i, n)} );
5.      LET(n) = MAX( {LET(n_i) |
                              ∀ flow f = (n_i, n)} ) ;
6.   else if (n.Type = XOR-JOIN or n = End Process) then
7.      EST(n) = MIN( {EST(n_i) + d(n_i) |
                              ∀ flow f = (n_i, n)} );
8.      LET(n) = MAX( {LET(n_i) |
                              ∀ flow f = (n_i, n)} ) ;
9.   else
10.     EST(n) = EST(n_i) + d(n_i);
11.     LET(n) = LET(n_i) + D(n);
12.  // store old data for analysis
13.  if ( mode = "target" or (EST(n) ≠ EST_old or
                                 LET(n) ≠ LET_old) )   then
14.     store EST_old to EST'(n);
15.     store LET_old to LET'(n)
16.     ∀ n' ∈ N, (n, n') ∈ F,
                Calculate_EAI(ws, n', "ripple"); // continue
```

4.2. The Definition of Resource Conflict with Temporal Consideration and the Temporal Related Editing Operations

With EAI for each activity process, two activity processes are potentially overlapped in execution if and only if their EAI are overlapped. Before we define the potential overlapped execution, two operators on EAI is defined as following.

Definition 14 *(Intersection between EAIs)*
EAI of two processes n_i, n_j are denoted as $[EST(n_i), LET(n_i)]$, $[EST(n_j), LET(n_j)]$ $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)]$ means the interaction of the two interval. $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)] = \phi$ if and only if $LET(n_i) < EST(n_j)$ or $LET(n_j) < EST(n_i)$; on the contrtry, $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)] \neq \phi$. If $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)] \neq \phi$, $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)]$ is defined as $[MAX(\{EST(n_i), EST(n_j)\}), MIN(\{LET(n_i), LET(n_j)\})]$

Definition 15 *(Total Containment between EAIs)*
$[EST(n_j), LET(n_j)] \subseteq [EST(n_i), LET(n_i)]$ means $[EST(n_j), LET(n_j)]$ is totally contained by $[EST(n_i), LET(n_i)]$. $[EST(n_j), LET(n_j)] \subseteq [EST(n_i), LET(n_i)]$ if and only if $EST(n_j) \leqq EST(n_i)$ and $LET(n_i) \geqq LET(n_j)$.

After the operators for EAIs are defined, the formal description of potential overlapped execution and the resource conflict with temporal consideration is defined in Definition 16 and 17.

Definition 16 *(Potential Overlapped Execution)*
Two activity processes n_i and n_j are potentially overlapped in execution if and only if $n_i, n_j \in N$ where $i \neq j$, $n_i.Type = n_j.Type = ACTIVITY$, and $[EST(n_i), LET(n_i)] \cap [EST(n_j), LET(n_j)] \neq \phi$

Definition 16
(Resource Conflict with Temporal Consideration)
(r, n_i, n_j) is a resource conflict with temporal consideration where r is a resource n_i and $n_j \in N$ if and only if $r \in R_i$, R_j, n_i and n_j are potentially concurrent potentially overlapped in execution. W.L.O.G, (r, n_i, n_j) and (r, n_j, n_i) is considered as the same in our discussion.

Adding or deleting a resource reference from some process might increase or eliminate one or more resource conflicts, and only the processes in resource conflict are required to be checked if there's any overlapping in their execution.

Adding a new process can be viewed as adding a new process with *d(n)* and *D(n)* zero, and causes no effect on EAI of existing processes. Deleting an activity process can be viewed as the values of *d(n)* and *D(n)* of the target process are modified to zero.

Directly modifying minimal and maximal execution duration of some process *n* (*d(n) or D(n)*) affects EAI of its following processes until the influence disappears. Such editing operation is called temporal related editing operations.

4.3. An Incremental Algorithm for Detecting Potential Overlapped Execution

The set RCT is used to store the resource conflicts with temporal consideration. Algorithm CTO finds the resource conflict with temporal consideration from the records of existing resource conflict. Algorithm CTO can be viewed as a patch of CSP and CRD algorithm to detect resource conflicts with temporal consideration after editing operations which are not temporal related.

The result set RD from CSP and CRD algorithm is used as the input parameter of algorithm CTO. At line 1 to 3, each resource conflict in set RD is checked, the resource conflicts in which the involved processes are potentially overlapped in execution are added in to set CTO. After these steps, CTO contains all the resource conflicts with temporal consideration after the editing operation. RCT contains the original resource conflicts with temporal consideration in the workflow specification. The elements in RCT but not in CTO are the resource conflicts with temporal consideration eliminated after the editing operation. On the other hand, the elements in CTO but not in RCT are the resource conflicts with temporal consideration created after the editing operation. At line 4 to 7, these elements are selected and the designer is informed about these eliminated or created resource conflicts with temporal consideration. At line 8, the set RCT is updated with CTO, all the resource conflicts with temporal consideration after the editing operation.

```
RESOURCE_CONFLICT_EXT SET CTO
        (RESOURCE_CONFLICT SET RD)
1.  ∀ (r, ni, nj) ∈ RD
2.    if ([EST(ni), LET(ni)] ∩ [EST(nj), LET(nj)]
                                        ≠ φ ) then
3.      add (r, ni, nj) to CTO;
4.  ∀ (r, ni, nj) ∈ (RCT – CTO)
5.      info( resource conflict with temporal
              consideration (r, ni, nj) is eliminated );
6.  ∀ (r, ni, nj) ∈ (CTO – RCT)
7.      info( resource conflict with temporal
              consideration (r, ni, nj) is created );
8.  RCT = CTO;
```

Modification of working duration which changes EST and LET values of the target process and its descendent processes affects execution overlapping between processes but doesn't affect resource conflict set found by CSP and CRD. The algorithm CDM (Calculating Duration Modification) is constructed to handle the change of resource conflicts with temporal consideration for such operations.

Assume that for the target process *n*, the temporal related editing operation changes the *EST(n)* to *EST'(n)* and *LET(n)* to *LET'(n)*. With the discussion in above section, we know that change of D(n) affects LET of n and its following processes, and change of d(n) affects EST of n's following processes. For each temporal editing operation and for the target process n and each effected processes only one of EST or LET is changed.

At line 1 to 5 of algorithm CDM, the situation $LET'(n) < LET(n)$ is handled. The EAI of n is shortened; therefore, for any existing resource conflict with temporal consideration (r, n_x, n), some existing overlapped process n_x might not overlap to n when their intersection interval is totally contained in the shortened interval. The resource conflict with temporal consideration (r, n_x, n) is eliminated when n_x and n are no longer potentially overlapped in execution. The elimination is updated to RCT and informed to the designer. At line 16 to 20, the situation $EST'(n_i) > EST(n_i)$ is handled in similar way.

At line 6 to 10, $LET'(n_i) > LET(n_i)$ is handled. In this situation, EAI of n is lengthened; therefore all the resource conflict related to n (r, n_x, n) are checked. For any such resource conflict, if EAI of n_x is intersected with the lengthened part and (r, n_x, n) is not a resource conflict with temporal consideration in advance. (r, n_x, n) becomes the new created resource conflict with temporal consideration after this temporal related editing operation. (r, n_x, n) is added into RCT, and the designer is informed as well. At line 11 to 15 the situation $EST'(n_i) < EST(n_i)$ is handled in similar way.

At line 21 and 22, the algorithm continues recursively when EAI of n is changed.

```
Void CDM
   (workflow specification ws, process n)
1.  if ( LET'(n) < LET(n) )   then
2.      ∀ (r, nx, n) ∈ RCT
3.        if ( [EST(nx), LET(nx)] ∩ [EST(n), LET(n)]  ⊆
```

```
                                        [LET'(n), LET(n)] )    then
4.          RCT = RCT –(r, n_x, n);
5.          info( resource conflict (r_i, n_x, n_i) is no longer
                    potentially overlapped );
6.  else if ( LET'(n) > LET(n) )    then
7.      ∀  r ∈ R(n), (r, n_x, n) ∈ n.rd(r)
8.        if ( [LET(n),LET'(n)] ∩ [EST(n_i),LET(n_i)] ≠ φ
                                  && (r, n_x, n)  ∉  RCT )    then
9.          RCT = RCT + (r, n_x, n);
10.         info( resource conflict with temporal
                    consideration(r_i, n_x, n_i) is created )
11. else if ( EST'(n) < EST(n) )    then
12.     ∀  r ∈ R(n), (r, n_x, n) ∈ n.rd(r)
13.       if ( [EST'(n),EST(n)] ∩ [EST(n_i),LET(n_i)] ≠ φ
                  && (r, n_x, n)  ∉  RCT )    then
14.         RCT = RCT + (r, n_x, n);
15.         info( resource conflict with temporal
                    consideration(r_i, n_x, n_i) is created )
16. else if ( EST'(n) > EST(n) )    then
17.     ∀  (r, n_x, n) ∈ RCT
18.       if ( [EST(n),LET(n)] ∩ [EST(n_i),LET(n_i)] ⊆
                                  [EST(n),EST'(n)] )    then
19.           RCT = RCT –(r, n_x, n);
20.         info( resource conflict (r_i, n_x, n_i) is no longer
                    potentially overlapped );
21.  if ( EST'(n) ≠ EST(n)) or (LET'(n) ≠ LET(n) )    then
22.      ∀  n'  ∈  N, (n, n')  ∈  F, CDM(ws, n');
```

5. Discussion of Time Complexity

Since the existing work on analysis of resource conflicts in workflow specification neglect the temporal factors, in comparison of time complexity of our approach and the traditional one, we also focus on the part without temporal consideration.

We simply discuss the time complexity of the algorithms through the number of processes required to be visited. With a workflow specification in which there are N processes. Since the total and static methodology for analysis of resource conflicts [2] visit all the processes in the workflow specification for each analysis, the worst case is O(N). In our approach, tracking of processes is required when detecting potential concurrent execution. The worst case is also O(N) when all the processes are sequentially ordered. However the calculation is only required when AND-SPLIT is met in algorithm CSP, and when all the processes are sequentially ordered, there's no AND-SPLIT in the workflow specification.

Although our approach has the same worst case with the total and static methodology, it has better time complexity in average cases.

The time complexity for total and static methodology in average case is still O(N), since it always tracks the whole schema. In our approach, the number of nodes required to be visited is in average equal to the length from the start process to the target process. Now we conclude the average case of our algorithm as $O(\log_k N)$, where k is the average number of branches of each process. Value of k is influenced by the structure of the workflow specification. With a fixed number of processes in a workflow specification, more control processes results more branches for each process, i.e. the more processes visited the less calculations required, and therefore, our approach is much better in average cases to the traditional approach.

6. Conclusions and Future Work

Workflow specification is a formal description of design and implementation of workflow applications. Proper environment for verification of structural, resource, and timing constraints helps designers produce workflow applications with high quality. There're various effective approaches for verification of structural and temporal correctness. However, there lacks a real time approach for designers to verify resource conflicts associated with the designer's editing activity. This paper describes an incremental algorithm verifying resource conflicts in workflow specification after every editing operation. We discuss the conditions causing resource conflicts and the algorithm to detect them. Our approach provides abundant information to the designer after each editing operation. The time complexity of efficiency about our algorithm is better then

the traditional approaches.
There're still some issues not discussed in this paper, for example, considering editing operations on control process, invocation of resource type and multiple resource instances, and analysis for resource conflicts when the workflow specification is altered during run-time. To implement and integrate the algorithm into existing WfMS is also necessary.

References

[1] WfMC, Workflow Management Coalition. http://www.wfmc.org/
[2] Hongchen Li, Yun Yang, T.Y. Chen, "Resource constraints analysis of workflow specifications", the Journal of System and Software 73 (2004).
[3] Sadiq, W., Orlowska, M.E., "Analysing process models using graph reduction techniques", Information System 25(2), p117-134. 2000.
[4] Marjanovic, O., "Dynamic verification of temporal constraints in production workflows.", Proceedings of the Australian Database Conference. IEEE Press, Canberra, Austalia, pp. 74-81.
[5] Aalst, W.M.P.v.d., Basten, T., Verbeek, H.M.W., Verkoulen, P.A.C., and Voorhoeve., M. "Adaptive Workflow: An Approach Based on Inheritance." In Proceedings of the IJCAI'99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business. 1999. Stockholm, Sweden.
[6] Fleurke, M and Ehrler, L, Purvis, M. A. (2003). "JBees - An Adaptive and Distributed Agent-based Workflow System", in Proceedings of the International Workshop on Collaboration Agents: Autonomous Agents for Collaborative Environments (COLA 2003), Halifax, Canada, October 2003.
[7] Adam, N., Atluri, V., Huang W., 1998. "Modeling and Analysis of Workflows using Petri-Nets", Journal of Intelligent Information System, 10(2), 1998.
[8] M. Reichert, P. dadam. "ADEPT-Supporting Dynamic Changes of Workflows without Losing Control", Journal of Intelligent Information System, 10(2), 1998.
[9] Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N.,1999. "Definition of deadlock patterns for business processes workflow models." Proceedings of the 32nd Hawaii International Conference on System Sciences, pp. 1-11.
[10] Singh, M.P., "Formal aspects of workflow management, Part 1: Semantics." Technical Report, Department of Computer Science, North Carolina State University, 1997.
[11] Shazia Sadiq, Maria Orlowska, Wasim Sadiq, Cameron Foulger, "Data Flow and Validation in Workflow Modeling", Conferences in Research and Practice in Information Technology, Vol. 27. 2003.
[12] David Hollingsworth, "The Workflow Reference Model", 1995
[13] David Hollingsworth, "The Workflow Reference Model: 10 Years On", 2004
[14] WfMC, "Workflow Management Coalition Terminology and Glossary, Document Number WFMC-TC-1011" Feb, 1999