國立交通大學

資訊工程學系

博 士 論 文

一個對時序工作流程管理系統進行分析的研究

A Study to Analyzing Temporal Workflow Management System

研 究 生：許懷中

指導教授：王豐堅　教授

中 華 民 國 一 百 年 七 月

一個對時序工作流程管理系統進行分析的研究

A Study to Analyzing Temporal Workflow Management System

研 究 生：許懷中　　Student：Hwai-Jung Hsu

指導教授：王豐堅　　Advisor：Feng-Jian Wang

國 立 交 通 大 學

資 訊 工 程 學 系

博 士 論 文

A Dissertation
Submitted to Institute of Computer Science and Engieering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Doctor
in
Computer Science

July 2011

Hsinchu, Taiwan, Republic of China

中華民國一百年七月

一個對時序工作流程管理系統進行分析的研究

學生：許懷中　　指導教授：王豐堅 博士

國立交通大學資訊工程學系（研究所）博士班

摘　　要

現代化企業利用工作流程管理系統統整文件、資訊系統以及人事組織以遂行其企業目的，而針對工作流程進行分析，有助於尋找企業程序中所隱藏的問題，避免工作流程執行時重複發生的錯誤，進而增進企業整體的效率；由於大部分具有良好行為之工作流程皆可轉換為結構化工作流程，因此結構化工作流程模型是在進行工作流程結構健全性分析時不可或缺的工具，此外，時序為進行工作流程正確性檢查、驗證以及工作流程效率分析上必須考量之因素，在此博士論文中，我們將時序因素與結構化工作流程模型結合統整而成一結構化時序工作流程模型，並且針對三種不同的領域，提出各自的分析方法；在組織分析領域，針對使用任務與角色為基存取控制模型的工作流程管理系統，我們建立了一個可以進行工作代理的執行框架，在此框架下，代理行為受到責任分擔以及企業政策的制約，使用者可以手動進行工作代理授權，而系統也可以自動將緊急的工作授權給適合的代理人；在資料分析領域，我們建立了一個從結構化時序工作流程中偵測異常文件使用的方法，藉由這個方法對工作流程定義進行靜態分析，可以有效避免由於異常資料操作所造成的系統意外行為；最後，針對資源領域，我們提出了一個可以在結構化時序工作流程建構的過程中，進行資源一致性與時序條件分析的遞增性分析方法，藉由我們的方法，工作流程設計者可以瞭解他所做的每一個設計決定對於整體工作流程定義的影響，並且修正由於錯誤的設計邏輯所造成的潛在資源衝突。

關鍵字：工作流程、工作流程管理系統、結構化時序工作流程、代理、任務與角色為基的存取控制模型、異常文件使用、資源衝突、遞增性分析方法

A Study to Analyzing Temporal Workflow Management System

Student: Hwai-Jung Hsu Advisor: Dr. Feng-Jian Wang

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

In modern enterprises, Workflow Management System (WfMS) coordinates data, resources, and organizations to enact workflows for various business objectives. Analysis of workflows facilitates locating problems in business processes and prevents repeated errors during workflow enactment. Structured workflow model is a useful tool for analysis of structural integrity because most well-behaved workflows can be reduced to structured workflows. Besides, temporal factors are also essential for workflow analysis, especially for validation, verification and performance analysis of workflows. In this dissertation, we integrate temporal factors into structured workflow model as Temporal Structured Workflow (TS workflow) model, and develop three distinct approaches for analysis of TS workflows in various perspectives. For the organization perspective, a framework for delegating works among users in a WfMS coordinated with Task-Role-based Access Control (TRBAC) model is established. With the framework, delegations can be enacted manually or automatically under restrictions like separation of duty (SOD) and management of enterprise policies. For the data perspective, a methodology detecting artifact anomalies in TS workflows is developed. By analyzing workflow schemas with our methodology, the unexpected run-time behavior generated from abnormal data manipulation can be prevented. Finally, for the resource perspective, an incremental approach is constructed to analyze resource consistencies and temporal constraints during construction of a loop-reduced TS workflow (LRTS workflow). With our approach, designers may realize the effect of each edit operation they made on the workflow schema under design, and correct the potential resource conflicts buried in business processes immediately.

Keywords: Workflow, Workflow Management System (WfMS), Temporal Structured Workflow, Delegation, Task-Role based Access Control (TRBAC), Artifact Anomalies, Resource Conflicts, and Incremental Methodology

誌　　謝

獻給在天上看顧著我的祖父

攻讀博士不僅僅是為了獲取一個學位，它是一場對於人生、自我以及真理的探索，如今這漫長、無止盡的旅程將要來到轉捩點，為了直到今日所完成的小小成就，有很多人值得感謝，首先要感謝我的指導教授王豐堅老師，感謝他在學術以及人生上對我的指導與幫助，沒有老師的耐心與包容，我無法在這條道路上堅持到底；感謝我的博士論文口試委員，朱治平教授、黃慶育教授、黃冠寰教授、梁德容教授、吳毅成教授以及黃世昆教授，感謝各位口委的指導，讓本博士論文得以順利完成；感謝所有曾經直接或間接參與過本論文相關研究的實驗室伙伴，許嘉麟、王建偉、王靜慧、張志絃、許熏任、楊大立、簡璞、陳建志等等，感謝他們在本研究中的貢獻；感謝我的祖父，雖然他沒能親眼看到我畢業，但是他一生努力掙來的這個家，撫育我成長茁壯；感謝我的祖母，她無怨無悔的全力支持，讓我得以心無旁騖地進行研究；感謝我的父親，沒有他的一席話，我不會踏上這條路，沒有他多年來的默默支持，我無以為繼；感謝我的母親，雖然她總是說自己不懂我在作些什麼，但是她的關愛是我人生的明燈；感謝我的姑姑，感謝她為這個家的奉獻，彌補了我因著忙碌而未足夠的孝道；感謝我的妻子，感謝她這些年在我身後的付出以及容忍我的任性，我們還將攜手創造未來；最後，感謝所有曾經在這論文以及我攻讀博士的過程中有所貢獻的人們，謝謝大家。

目錄 – Table of Contents

圖目錄 – Table of Figures

表目錄 – Table of Tables

定義目錄 – Table of Definitions

演算法目錄 – Table of Algorithms

輔助定理目錄 – Table of Lemmas

Chapter 1. Introduction

Enterprises define their business objectives in business processes, and workflows automate the business processes by completing tasks which realize parts of business goals in a particular order [1]. As a dominant factor in workflow management, developing appropriate analysis techniques for workflows is necessary [2]. Irani et al. state that workflow analysis facilitates locating problems in business processes and preventing repetition errors during workflow enactment. [3]. Vergidis et al. claim that workflow analysis adopts a range of different tactics such as simulation, diagnosis, validation, verification, and performance analysis to clarify the characteristics, potential conflicts, possible bottlenecks and any promising process alternatives [4].

To assure the correctness of workflow execution, analyses on structural integrity of workflows are widely studied. Adam's methodology detects inconsistent dependencies among tasks to assure the safety of a workflow [5]. van der Aalst et al. develop effective Petri-net based techniques to verify deadlocks, livelocks (infinite loops), and dead tasks from workflow schemas [2][6][7]. In [8], Kiepuszewski et al. define structured workflow model and claim that a structured workflow is well-behaved, i.e. free from deadlock and multiple active instances of the same activity. Kiepuszewski et al. also claim that although structured workflow model is less expressive, most arbitrary well-behaved workflows can be transformed into a structured workflow, and structured workflow model is a good tool for various kinds of workflow analysis.

Besides, combining timing constraints into analysis of workflow models is also familiar. Li et al. indicate that analysis of temporal factors is essential for validation of the interval

dependencies with temporal constraints in a workflow schema [9]. Adam et al. consider timing constraints as the external conditions for structural correctness of a Petri-net based workflow model [5]. Chen et al. develop an approach for dynamic verification of fixed-time constraints in grid workflow system [10]. From a graph based workflow model, Eder et al. develop a timed graph model to illustrate the working duration of activities among workflows with the corresponding earliest and latest finish time, and calculate the deadlines among internal activities to meet the overall temporal constraints on the basis of the model [11][12]. Marjanovic et al. build the timing model based on duration and instantiation space, and model the absolute and relative deadline constraints for dynamic verification [13]. Zhuge et al. consider durations of activities for temporal checking in both design-time and run-time and model the temporal factors in workflows as timed workflow model for further analysis. [14]

For the organization perspective, modern WfMS regulates activities of employees through varieties of access control methods. Among the methods, role-based access control (RBAC) model [15][16] grouping users with similar permissions into roles is a popular solution among enterprises. However, business processes are operated based on not only roles but also tasks. With both as core concepts, Oh et al. propose task-role-based access control (TRBAC) model to provide more modeling power for access control in WfMS [17]. Delegation which allows subjects like access rights or work items being authorized between users or roles during run-time is an interesting problem for workflow management [18] and is often studied on the basis of the corresponding access control model. For example, RBDM0 [19], RBDM1 [20], and the methods in [21], [22], and [23] describe various delegation models based on RBAC [15][16]. On the basis of RBAC, Crampton et al. describe an approach to manage delegation in WfMS, and raise several new issues about delegation of tasks for work-list-based WfMS [18]. Delegation for TRBAC is also studied in [24] and [25]. Jian et al. construct a framework and define the components for delegation in TRBAC [24], and Hsu et al. enhance the work by

considering temporal issues in [25].

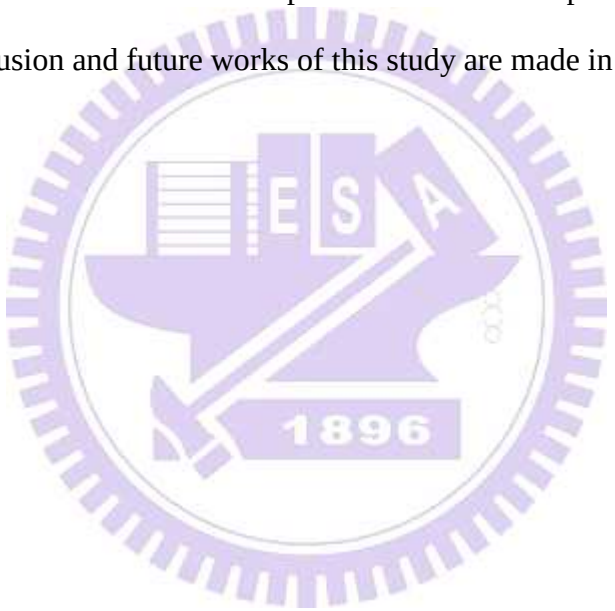
A well-structured workflow may still fail or produce unanticipated run-time behavior because of abnormal data manipulation, the artifact anomalies. Detect artifact anomalies in workflows checks possible data misuse buried in workflow specifications. Various methodologies have been developed for detection of artifact anomalies generated from structural relationships between activities in a workflow [26][27][28][29][30]. Sadiq et al. present seven basic data validation problems, redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data in structured workflow model [26][31]. Hsu et al. define preliminary improper artifact usages anomalies, and introduce the analysis of such anomalies in design phase of a structured workflow [27][28]. In [29], Wang et al. introduce a behavior model to describe the data behavior in a workflow and refine the work accomplished in [28] by improving its efficiency. In [30], Hsu et al. raise the issues about analyzing artifact anomalies in workflows adopting message passing data models, and describe a formal description for such anomalies. Nevertheless, how temporal factors may affect the analysis of artifact anomalies is still seldom addressed. The methodology detecting artifact anomalies generated from twisted temporal and structural relationships between activities in workflows should be further discussed on the basis of the previous studies.

As for the resource perspective, Reveliotis et al. construct a Petri-based model with consideration of resource allocation, and uses the model for structural and deadlock analysis of workflow applications [32][33]. Based on Zhuge's work [14], Li et al. estimate the active intervals of activities, and develops an algorithm to detect and remove resource conflicts with respect to both temporal and structural issues [34]. Zhong et al. adopt Li's methodology [34] onto a petri-net based workflow model, and develop an algorithm to detect resource conflicts when a new workflow being put into WfMS during run-time [35]. Based on [36], Hsu et al. develop an incremental methodology for analysis of resource constraints in structured

workflows with temporal consideration during design-time [37]. The generation or elimination of resource conflicts are tracked and alerted along with each edit operation made by designers of workflows [37]. However, the technique for structural analysis adopted in [37] is inefficient and can be revised with the methods proposed in [30].

In this dissertation, structured workflow modeled in [8] is extended as temporal structured workflow (*TS workflow*) model with the temporal issues considered in [34] and [37]. The techniques for structural and temporal analysis on TS workflows are first introduced, and the methodology to analyze TS workflows in organization, data, and resource perspectives are then discussed. For the organization perspective, the works accomplished in [25] are refined to adopt TS workflow model for temporal constraints. A delegation framework for the WfMS coordinated with TRBAC model is established, and a series of algorithms for delegation of task instances and exploration of delegatees are developed. With the framework, a user is able to delegate his work to another user through an approval process, and WfMS can automatically delegate an emergent work item to an appropriate delegatee. The constraints such as elimination of delegation loops and separation of duty (SOD) are validated for delegation requested by either users or WfMS during run-time. As for the data perspective, a formal model describing artifact anomalies in TS workflow is established on the basis of define-use-kill operations. The issues about the artifact anomalies produced from twisted structural and temporal relationships between activities in a TS workflow are discussed and modeled. The methodology for static analysis of artifact anomalies buried in a TS workflow is developed. Finally, for the resource perspective, the incremental methodology accomplished in [37] is refined to integrate TS workflow model and the analysis techniques proposed in [30]. The edit operations for constructing loop-reduced TS workflows (LRTS workflows) are first stated, and the methodology tracking down the generation and elimination of resource conflicts along with each edit operation made by designers is described.

The rest part of this dissertation is organized as following. In chapter 2, TS workflow model is sketched. The basic elements and the construction rules for TS workflow are described and the methods for analysis of temporal and structural properties in TS workflows are introduced. In chapter 3, a delegation framework for the WfMS coordinated with TRBAC model is introduced. In chapter 4, artifact operations and the corresponding artifact anomalies are first introduced, and the methodology detecting artifact anomalies in TS workflows is then described accordingly. In chapter 5, an incremental methodology tracking down the resource conflicts generated or eliminated in the steps of construction of an LRTS workflow is presented. The related works for each of the above topics are discussed separately at the end of the chapters, and the conclusion and future works of this study are made in chapter 6.



Chapter 2. Temporal Structured Workflow Model

2.1 Basic Elements

A workflow is composed of a *start process*, an *end process*, some *activity processes* and some *control processes*. The start (*ST*) process represents the entry point of a workflow, and the end (*END*) process indicates the termination point. An activity (*ACT*) process stands for a piece of work to be performed and describes one logical step within a workflow [1].

A control process is a routing construct used to control the divergence and convergence of sequence flows. The control processes can be classified as *AND-split* (*AS*), *AND-join* (*AJ*), *XOR-split* (*XS*), and *XOR-join* (*XJ*). An AND-split process within a workflow splits a single sequence of control into two or more sequences to allow simultaneous execution of activities; on the contrary, an AND-join process merges multiple parallel executing sequences into a single common sequence of control [13]. An XOR-split process within a workflow is the point where a single sequence of control decides a branch to take from multiple alternative branches, and an XOR-join process converges multiple alternative branches in a workflow [13].

Processes are connected by directed *flows*, the flow(s) leading to a process are called the *in-flow*(*s*) of the process, and the flow(s) departing from a process are called the *out-flow*(*s*) of the process. The process starting a flow is the *source process* of the flow, and the process ending a flow is the *sink process* of the flow. In a workflow, only AND-split and XOR-split processes have multiple out-flows, and only AND-join and XOR-join processes have multiple in-flows. Figure 1 illustrates the notation of the basic elements described above.

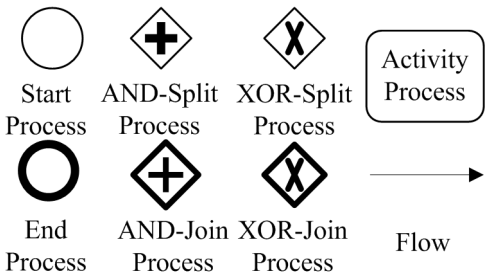


Figure 1 The Graphic Notations of the Basic Workflow Elements

With all the descriptions above, a workflow is modeled as following:

Definition 1 (Workflow Model)
A workflow w, $w = (P_w, F_w, s, e)$. and
P_w represents the set of the processes in w, and
$\forall p \in P_w$, $p.type \in$ {ACT, AS, AJ, XS, XJ, ST, END}
$F_w \subseteq P_w \times P_w$ represents the set of flows in w.
$\forall f \in F_w$, $f = (p, q)$ is the in-flow of process q and the out-flow of process p, and p is the source process of f, and q is the sink process of f.
$s \in P_w$ represents the start process of w, $s.type$ = ST, $\exists$ no in-flow to s.
$e \in P_w$ represents the end process of w, $e.type$ = END, $\exists$ no out-flow from e.

* In this dissertation, "=" denotes an assignment operator and "==" denotes a Boolean equality operator

A sequence of flow(s) forms a *path*, and is formally modeled as following:

Definition 2 (Path)
A path is notated as a series of processes quoted by a pair of angle brackets.
For a workflow w, a path, $<p_1, p_2, \ldots, p_k>$, from p_1 to p_k exists if and only if (p_1, p_2), (p_2, p_3), … $(p_{k-1}, p_k) \in F_w$.

2.2 Structured Workflow

A structured workflow is a workflow that is syntactically restricted in a number of ways. Control processes are organized in pair, an XOR-split process is paired with an XOR-join process, and an AND-split process is paired with an AND-join process. A control block is composed of a pair of control processes and the processes placed in between the pair of control processes. According to the type of the control processes, the control blocks can be classified as parallel structures, decision structures, and structured loops as Figure 2 illustrates. Each process in a structured workflow has at least one path from the start process to it, and at least one path

from it to the end process. Such restriction keeps a structured workflow well-behaved [4], i.e. a structured workflow is free from deadlocks and multiple active instances. Most arbitrary well-behaved workflows can be transformed to be structured without loss of their contexts [4]. Figure 2 shows the building blocks of a structured workflow according to the basic elements and constraints described above.

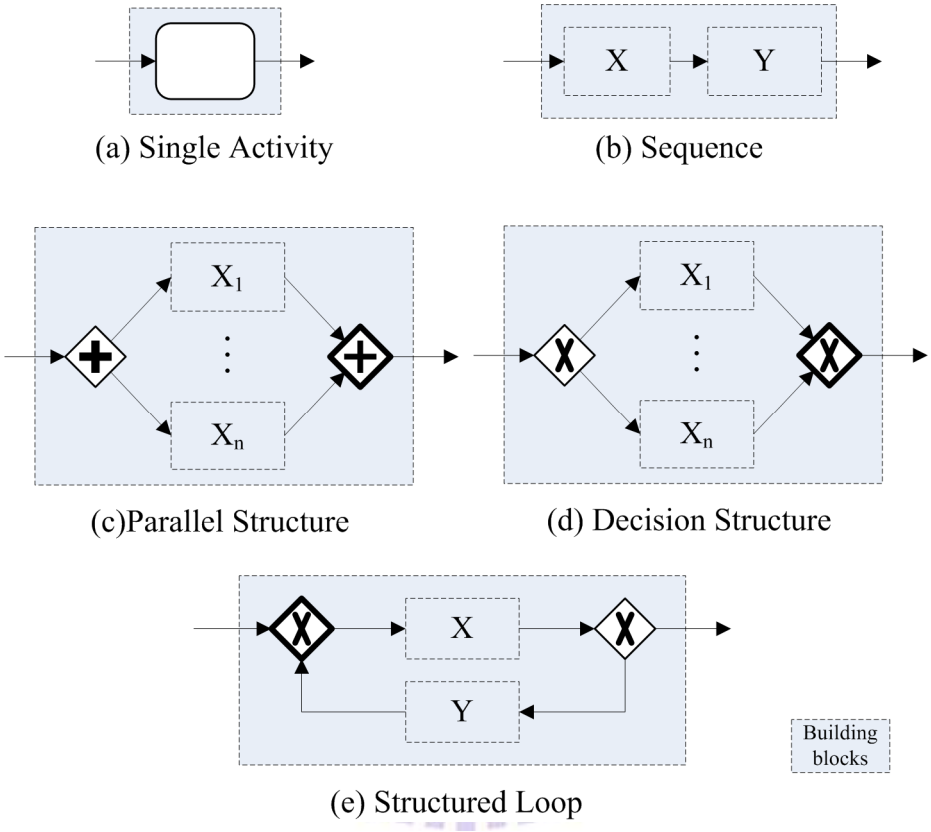


Figure 2 Building Blocks of a Structured Workflow

All the processes between the start and the end process in a structured workflow are organized with the building blocks shown in Figure 2. For Figure 2(c) and Figure 2(d), the blocks X_1, X_2, …, and X_n represent the branches split and converged in a parallel structure or a decision structure. Besides, in Figure 2(e), the structured loop acts like a do-while loop when block Y is null, and acts like a while loop when block X is null. Figure 3 illustrates the control graph of a sample structured workflow.

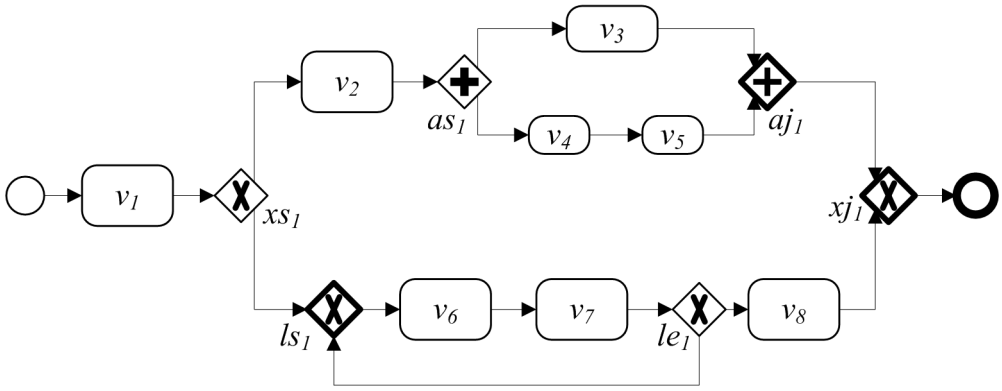


Figure 3 A Sample Structured Workflow

Two processes are *reachable* from one to the other if there exists a path between them, *parallel* if they reside on different branches of a parallel structure, and *exclusive* to each other if they reside on different branches of a decision structure. Take Figure 3 for example. The path $<v_1, xs_1, v_2, v_3>$ indicates that v_1 is reachable to v_3. v_3 and v_4 are parallel because they reside on different branches split from as_1. v_2 and v_8 are exclusive because they reside on different branches of the decision structure quoted by xs_1 and xj_1. In this dissertation, the above structural relationships between processes are notated as following Boolean functions:

Definition 3 (Structural Relationships in a Structured Workflow)
For a structured workflow w,
Reachable: $P_w \times P_w \Rightarrow$ {true, false}
Reachable(p, q) holds if and only if there exists a path from p to q.
Parallel: $P_w \times P_w \Rightarrow$ {true, false}
Parallel(p, q) holds if and only if p and q reside in different branches of a parallel structure.
Exclusive: $P_w \times P_w \Rightarrow$ {true, false}
Exclusive(p, q) holds if and only if p and q reside in different branches of a decision structure.

2.3 Temporal Structured Workflow

In [14], Zhuge models timed workflow by describing the maximal and minimum working durations for each activity. In this dissertation, a timed and structured workflow is named as a *Temporal Structured Workflow* (*TS workflow*) and is formally modeled as following:

Definition 4 (TS workflow)
A workflow *w* is temporal structured with following properties:
(1) *w* is structured, and
(2) $\forall p \in P_w$, d(*p*) and D(*p*) represents the minimum and maximum working duration of process *p*.

To facilitate discussion, we assume that if *p* is an activity process, $0 < d(p) \leq D(p)$; otherwise, d(*p*) = D(*p*) = 0. Figure 4 illustrates a sample TS workflow.

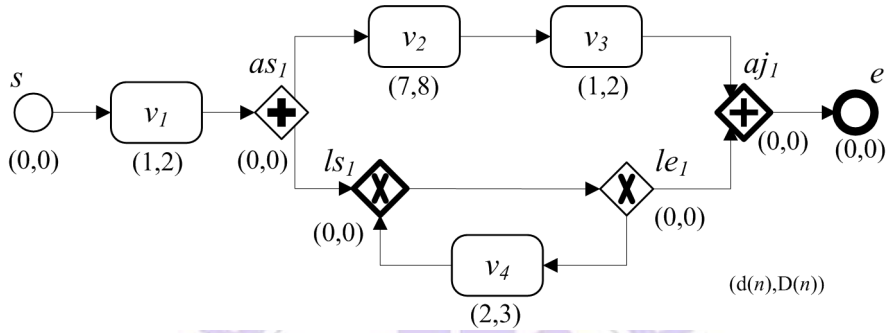


Figure 4 A Sample TS workflow

2.4 Analysis of Structural and Temporal Relationships between Processes in TS workflow

2.4.1 Loop Reduction

The structural and temporal relationships between processes are the bases of any further analysis of a TS workflow. In [9], [11], and [12], Hsu et al. give several methodology to reveal the structural and temporal relationships between processes in acyclic structured and timed workflows. In [28] and [29], Hsu and Wang et al. claim that in a structured workflow, all the possible state variations of the artifact operated in loops with more than two iterations are the same as those with exact two iterations. Therefore, they reduce a structured loop into a decision structure with three branches representing for no iteration, a single iteration, and two iterations for the analysis of artifact anomalies with better efficiency. In this dissertation, we adopt an approach similar to [7] and [8] to reduce the structured loops in a TS workflow as decision structures to retrieve structural and temporal information in a TS workflow as in [9], [11], and [12].

In a TS workflow, the number of iterations of a loop affects the active timing of processes succeeding to the loop. The loop reduction introduced in [7] and [8] may bring inaccuracy to the analysis of temporal factors, and is therefore not feasible for TS workflow. In [38], Leong considers the worst case scenarios for loops in a workflow and develops a methodology to detect whether the workflow possibly exceeds its deadline during run-time. Here, we combine Leong's concept and the methodology in [7] and [8] to describe a refined loop reduction method for the analysis of TS workflow.

First, it is assumed that the maximal number of iterations for a structured loop in a TS workflow is finite. In other words, the infinite loops are not discussed in this study. Based on the assumption, a structured loop is transformed into a decision structure with three branches: no iteration, a single iteration, and maximal iterations as Figure 5 illustrates.

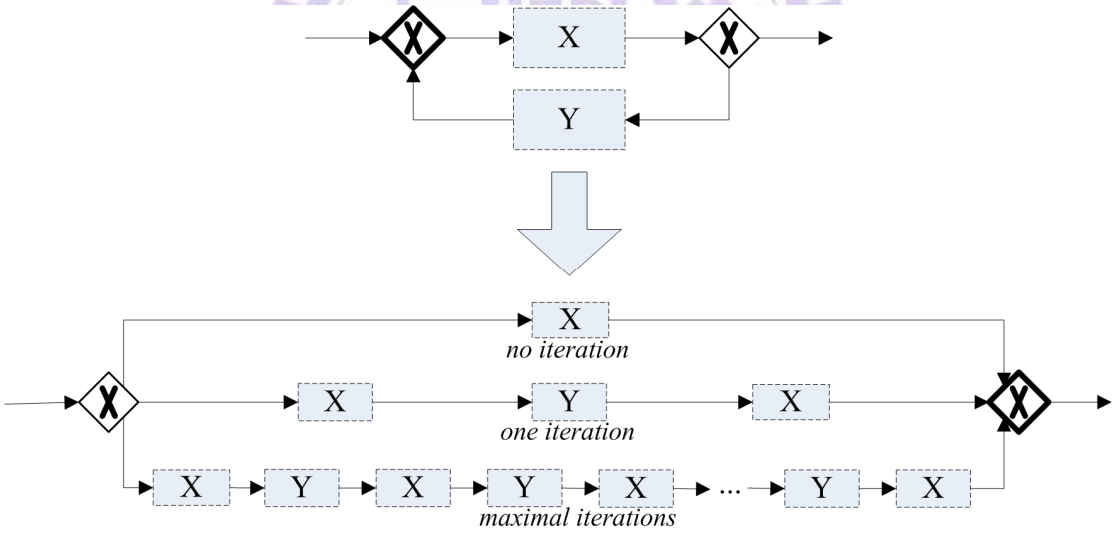


Figure 5 Refined Loop Reduction for TS Workflow Model

The refined loop reduction bring following advantage: (1) All the possible state variations of artifacts between iterations are still completely captured, (2) the active intervals of the processes succeeding to the structured loop can still be accurately estimated because the worst case scenario is considered, and (3) the methodology for acyclic structured workflow can be adopted in TS workflow because the structured loops are reduced. In this dissertation, loop-reduced TS workflows (*LRTS workflows*) are widely adopted in our methodology.

2.4.2 Analysis of Structural Relationships between Processes in LRTS workflow

The structural relationships between activity processes are the groundwork for analysis of TS workflow, and are described and proved in the following lemma.

> Lemma 1
> For an LRTS workflow w, p and $q \in P_w$, and $p.type == q.type ==$ ACT, one and exactly one of the following statements, Reachable(p, q), Reachable(q, p), Parallel(p, q), and Exclusive(p, q), holds.

Proof:

An LRTS workflow is still structured, and the lemma can be proved through the discussion of the construction rules of a structured workflow. Because a single activity process is a basic building block of a structured workflow, p and q can always be distributed into two different building blocks combined in a sequence, a parallel structure, or a decision structure illustrated in Figure 2.

Let b_p and b_q be the building blocks containing p and q separately. If b_p and b_q is combined in a sequence block, p and q are reachable from former to the later. Since w is loop-reduced, i.e. w is loop free, if Reachable(p, q) holds, Reachable(q, p) is false, and vice versa. Besides, according to the construction rules, there exist no paths between the building blocks split from an XOR/AND-split process. Therefore, Parallel(p, q) and Exclusive(p, q) can not hold in this case.

Otherwise, if b_p and b_q is combined in a decision block, b_p and b_q represents different branches split from the XOR-split process starting the decision structure. In other words, p and q resides in different branches of a decision structure, and therefore, Exclusive(p, q) holds. Since w is loop-reduced, there exist no paths between p and q, both Reachable(p, q) and Reachable(q, p) are false. On the other hand, according to the construction rules, since b_p and b_q reside on different branches of a decision structure, they can not reside in different branches of a parallel structure. Therefore, Parallel(p, q) does not hold. With similar reason, we can also show that when Parallel(p, q) holds, none of Reachable(p, q), Reachable(q, p), and Exclusive(p, q) holds, and hence, Lemma 1 is shown correct with all the statements above. □

In [9], Hsu et al. use a data structure, ABStack, to record the structural information of processes, and achieve an efficient analysis of the structural relationships between processes in an acyclic structured workflow. In this dissertation, the similar approach is adopted. All the flows in an LRTS workflow are tagged with a branch mark. The branch mark is a natural number ID for each out-flow split from an XOR/AND split process, and is -1 for any other flow

in the LRTS workflow. The branch mark in this dissertation is formally defined as following.

Definition 5 (Branch Mark)
For an LRTS workflow w,
BM_w: $F_w \Rightarrow$ INTEGER

$$\forall (p, p') \in F_w, \quad BM_w((p, p')) = \begin{cases} \text{a natural number if } p.type \in (XS, AS) \\ -1 \text{ otherwise} \end{cases}$$

For $p, q, q' \in P_w$, $p.type \in \{XS, AS\}$, and $(p, q), (p, q') \in F_w$,
$BM_w((p, q)) \neq BM_w((p, q'))$

A process in an LRTS workflow might reside in nested decision/parallel structures, and the structures are recorded in the ABStack corresponding to the process. Each of the structures is presented as a *structural item* composed of the split process starting the structure and the branch mark mapped to one of the out-flows of the split process. In the dissertation, an ABStack is notated as a series of structural items quoted by a pair of double angle brackets, "«" and "»". The items representing the inner structures are recorded higher in the ABStack, where the leftmost item is the top of the stack and the rightmost item is the bottom. The definition of an ABStack is formally described as following.

Definition 6 (ABStack)
$\forall p \in P_w$, *p.abstack* represents the ABStack corresponding to p.
A structural item, *stitem* = (*sp*, *bm*), is included in *p.abstack* if and only if
(1) $sp \in P_w$, $sp.type \in \{AS, XS\}$, and $\exists$ a path <*sp*, …, *p*, …, *jn*> in *w* where *jn* is the corresponding join process of *sp*.
(2) *bm* = BM((*sp*, *p'*)) where *p'* == *p* or Reachable(*p'*, *p*) == true.
p.abstack == « » if and only if *p* resides in no decision/parallel structure.
p.abstack == «(sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)» exists if and only if a path <sp_k, …, sp_2, …, sp_1, …, *p*, …, jn_1, …, jn_2, …, jn_k> exists.
To calculate ABStacks of the processes in an LRTS workflow, *push* and *pop* functions associated with ABStack are defined as following:
Let an ABStack *abs* == «(sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)»
Push(*abs*, (*sp*, *bm*)) returns a new ABStack *abs'*, where
abs' == «(*sp*, *bm*), (sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)»
Pop(*abs*) returns a new ABStack *abs'*, where
abs' == «(sp_2, bm_2), …, (sp_k, bm_k)»

Figure 6 illustrates how push and pop functions work for the calculation of the ABStacks corresponding to the processes in an LRTS workflow.

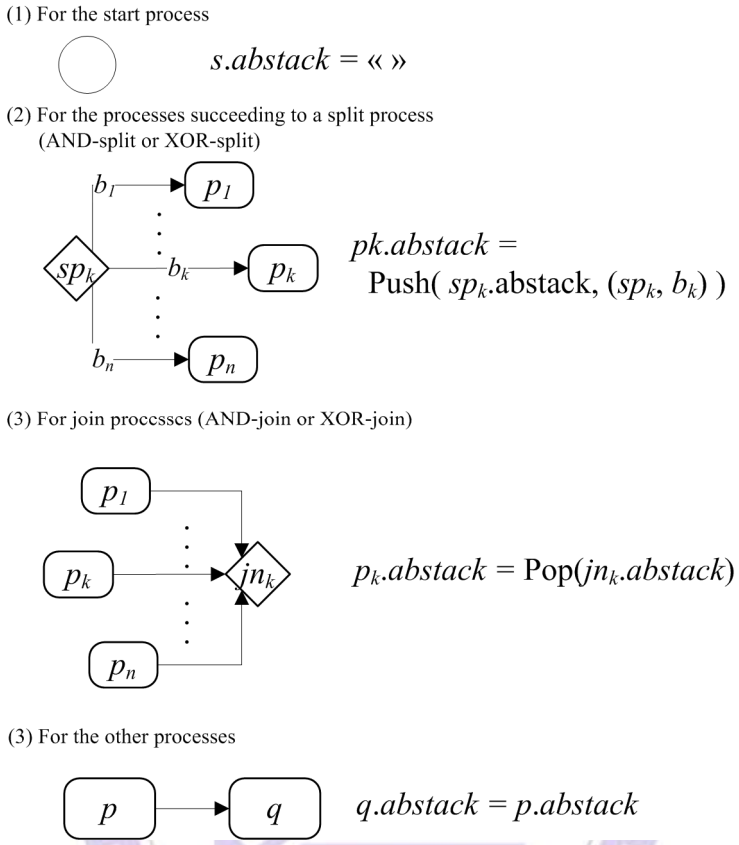


Figure 6 Calculation of ABStacks for Processes in an LRTS Workflow

Figure 7 illustrates a sample LRTS workflow decorated with ABStacks. Take process v_5 for example. The items (as_1, 2), and (xs_1, 1) in the ABStack of v_4 shows that v_4 resides on #2 branch split from the AND-split process as_1 and #1 branch split from the XOR-split process xs_1. The order of (as_1, 2) and (xs_1, 1) indicates that the parallel structure started from as_1 is nestedly contained by the decision structure started from xs_1.

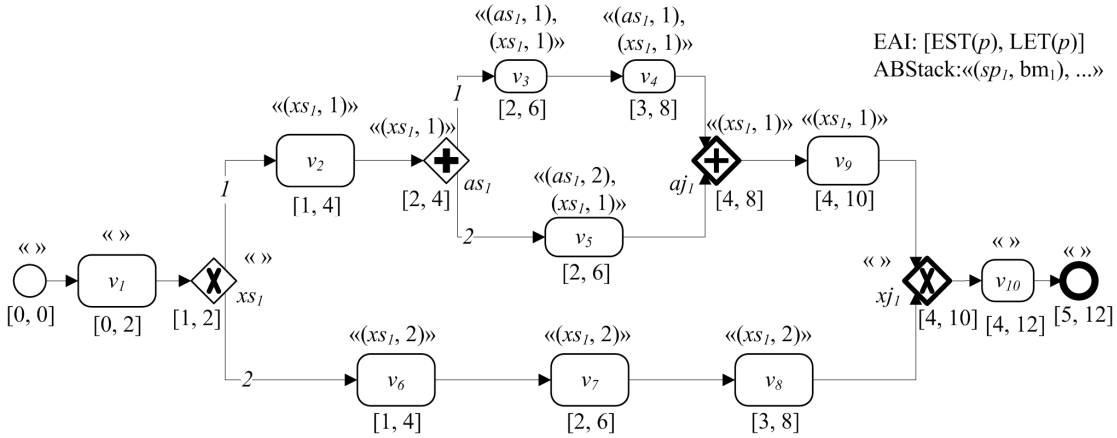


Figure 7 A Sample TS workflow with ABStacks and EAIs

Besides, the structural items (as_1, 1) and (as_1, 2) in the ABStacks of v_4 and v_5 correspondingly indicate that v_4 and v_5 reside on different branches split from AND-split process, as_1. In other words, v_4 and v_5 are parallel. The parallelism or exclusiveness between processes can be identified through comparing the ABStacks of the corresponding processes, and Lemma 2 shows how ABStacks work for identification of structural relationships between processes in an LRTS workflow.

Lemma 2
For an LRTS workflow *w*, and *p*, *q*$\in$ P_w
(1) Parallel(*p*, *q*) holds if and only if
$\exists$ (*sp*, *bm*)$\in$ *p.abstack* and (*sp*, *bm'*)$\in$ *q.abstack* where *sp*$\in$ P_w, *sp.type* == AS and *bm* $\neq$ *bm'*.
(2) Exclusive(*p*, *q*) holds if and only if
$\exists$ (*sp*, *bm*)$\in$ *p.abstack* and (*sp*, *bm'*)$\in$ *q.abstack* where *sp*$\in$ P_w, *sp.type* == XS and *bm* $\neq$ *bm'*.

Proof:

Consider the if-part of statement (1), according to Definition 6, if $\exists$ (*sp*, *bm*)$\in$ *p.abstack* and (*sp*, *bm'*)$\in$ *q.abstack* where *sp*$\in$ P_w, *sp.type* == AS and *bm* $\neq$ *bm'*, there exists a process *m* that *bm* == (*sp*, *m*), and *m* is either equivalent to *p* or Reachable(*m*, *p*) holds. Similarly, there exists another process *n* for *q*. *bm* $\neq$ *bm'* indicates that *m* $\neq$ *n*, and *p* and *q* reside on different branches split from the AND-split process, *sp*. Thus Parallel(*p*, *q*) holds and the if part is shown correct.

As for the only-if-part, if Parallel(*p*, *q*) == true, *p* and *q* reside on different branches of a parallel structure. Let *sp* be the AND-split process starting the parallel structure, and *jn* be the

AND-join process terminating it. The nodes in the path from *sp* to *p* are totally different from those in the path from *sp* to *q*. Besides *sp* and *jn*, two distinct paths, <*sp*, …, *p*, …, *jn*> and <*sp*, …, *q*, …, *jn*>, exist. Therefore, there exists a process *m* that (*sp*, *m*)$\in F_w$ and either *m* is equivalent to *p* or Reachable(*m*, *p*) == true. Similarly, there also exists such a process *n* for *q*. *m* and *n* can not be the same process because they reside on different branches split from *sp*, and thus, BM(*sp*, *m*) ≠ BM(*sp*, *n*). According to Definition 6, (*sp*, BM(*sp*, *m*)) is included in *p.abstack*, and (*sp*, BM(*sp*, *n*)) is included in *q.abstack*. The only-if part of statement (1) of the lemma is proved.

Part (2) can be proved similarly and the proof is omitted here. With the paragraphs above, Lemma 2 is shown correct.□

2.4.3 Analysis of Twisted Temporal and Structural Relationships between Processes in LRTS workflow

In a TS workflow, the temporal and structural relationships between processes are twisted. This section firstly shows how to identify the temporal property between processes.

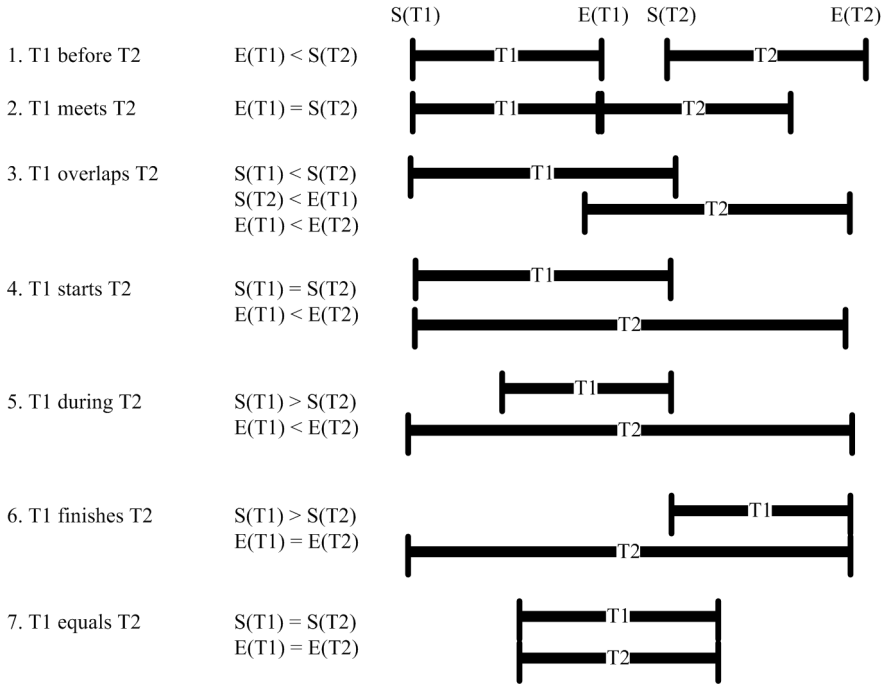


Figure 8 The Temporal Relationships between Time Intervals [39]

A time interval is duration of a segment of time. In [39], Allen defines seven reasoning relationships between time intervals. Figure 8 illustrates the temporal relationships adopted in this dissertation on the basis of Allen's definition, and Definition 7 describes the formal

definition of time intervals and the temporal relationships between time intervals adopted in this dissertation.

Definition 7 (Time Intervals)
A time interval ti = [S(ti), E(ti)] indicates a duration from the time point S(ti) to E(ti), E(ti) ≥ S(ti).
A time point tp can be represented as a time interval [tp, tp], and *ctime* is the time point indicating the current time.
For any two time intervals ti_1 and ti_2,
ti_1 is *before* ti_2, notated as $ti_1 \prec_{TI} ti_2$, if and only if E(ti_1) ≤ S(ti_2).
ti_1 is after ti_2, notated as $ti_1 \succ_{TI} ti_2$, if and only if ti_2 is before ti_1.
ti_1 overlaps ti_2, notated as $ti_1 \approx_{TI} ti_2$, if and only if
MIN({E(ti_1), E(ti_2)}) – MAX({S(ti_1), S(ti_2)}) > 0
ti_2 *contains* ti_1, notated as $ti_2 \supseteq_{TI} ti_1$, if and only if S(ti_2) ≤ S(ti_1) and E(ti_2) ≥ E(ti_1).

In Definition 7, two utility functions MAX and MIN are invoked. Function MAX returns the element with the maximum value among the parameter set, and function MIN returns the minimal one.

In [23] and [40], Joshi et al use not only individual time intervals but also the periodic temporal expressions to describe the temporal constraints in roles for temporal RBAC model. For example, the expression "*the night time duty is activated 6pm to 11pm every Wednesday and Friday*" indicates that the permissions for night time duty are activated during certain repeated time durations. The periodic temporal expressions can be viewed as a combination of multiple time intervals, and are grouped as a *time description* as following definition.

Definition 8 (Time Descriptions)
A time description td is a set of time intervals. For any two time intervals ti_z and ti_y in td, ti_x and ti_y are exclusive. On the other hand, for any two non-empty time description td_a and td_b, td_a *contains* td_b notated as $td_a \supseteq_{TD} td_b$ if and only if $\forall\, ti_b \in td_b$, $\exists\, ti_a \in td_a$ such that $ti_a \supseteq_{TI} ti_b$.

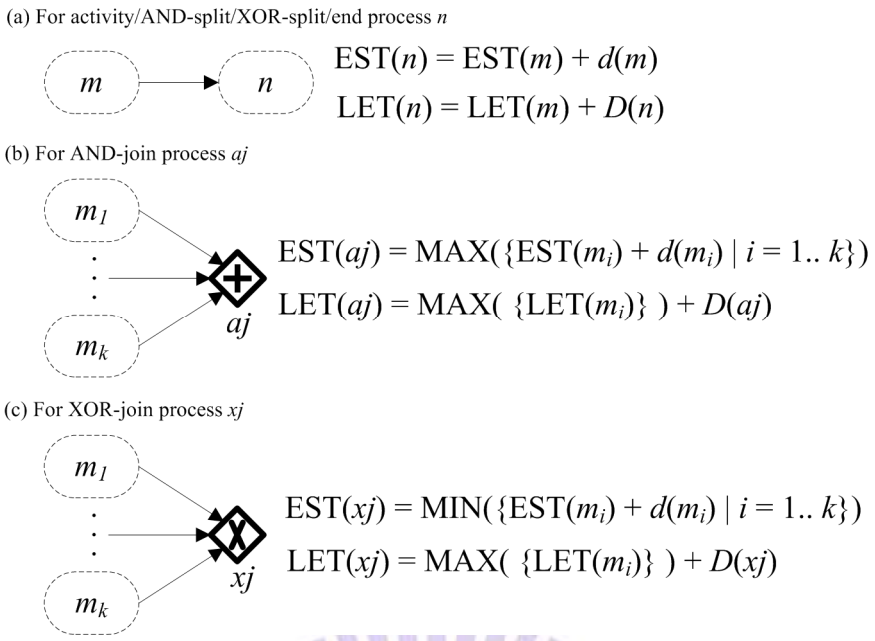


Figure 9 Calculation of EAIs in an LRTS Workflow [34][37]

In [34] and [37], the minimum and maximum working durations are used to estimate the active duration of a process corresponding to the start of workflow. The *Estimated Active Interval* (EAI) of a process is a time interval indicating when the process can be initialized and when it should be terminated. In this dissertation, the Estimated Active Interval of a process *p*, notated as EAI(*p*) is defined as following:

> Definition 9 (Estimated Active Interval)
> For a TS workflow *w* and a process $p \in P_w$,
> EAI(*p*) = [EST(*p*), LET(*p*)], and corresponding to when *w* starts:
> EST(*p*) indicates the earliest time that *p* can be initialized.
> LET(*p*) indicates the latest time that *p* must terminate.

With the assumption that the EST and LET of the start process of a TS workflow are zero, the methodology described in [34] and [37] is adopted to calculate the EAIs of processes in an LRTS workflow as Figure 9 illustrates.

With Lemma 1 and Lemma 2, whether two processes in an LRTS workflow are exclusive, parallel, or reachable from one to the other is identified with corresponding ABStacks. The path direction of two reachable processes can be further derived according to

the corresponding EAIs, and the following lemmas show how EAIs can be adopted in analysis of LRTS workflow.

> Lemma 3
> For an LRTS workflow w, p and $q \in P_w$, $q.type$ == ACT,
> if Reachable(p, q), LET(p) < LET(q)

Proof:

Reachable(p, q) represents that the path $<p, m_1, m_2, \ldots, m_n, q>$ exists. Now we prove the lemma with mathematical induction. For $n = 0$, $(p, q) \in F_w$, since $q.type$ = ACT, D(q) > 0 and LET(q) = LET(p) + D(q). LET(p) < LET(q) holds.

Hypothesis: The lemma holds when $n < k$.

For $n = k$, LET(q) = LET(m_k) + D(q) and LET(m_k) < LET(q). According to the construction rule of TS workflow, $m_k.type \neq$ S, E, and $m_k.type \in$ {AS, XS, AJ, XJ, ACT}. The following conditions should be discussed:

For any $1 \leq i \leq k$, if there exists an m_i where m_i.type = ACT, according to the hypothesis, LET(p) < LET(m_i) and LET(m_i) < LET(q). Therefore, LET(p) < LET(q). Otherwise, for any $1 \leq i \leq k$, $m_i.type \in$ {AS, XS, AJ, XJ}, according to the EAI calculation methods, for any $(u, m_i) \in F_w$, LET(u) ≤ LET(m_i). Since there exists a path from p to m_i, LET(p) ≤ LET(m_i). On the other hand, according to the hypothesis, LET(m_i) < LET(q). Therefore, LET(p) < LET(q). With statements above, we know the lemma holds for $n = k$, and on the basis of mathematical induction, Lemma 3 is proved. □

> Lemma 4
> For an LRTS workflow w, p and $q \in P_w$, $p.type$ == $q.type$ == ACT,
> if Parrallel(p, q) == Exclusive(p, q) == false, and LET(p) < LET(q),
> Reachable(p, q) == true.

Lemma 4 can be shown correct with Lemma 1 and the construction rule of LRTS workflow. Lemma 4 describes that if two activity processes in an LRTS workflow are not mutually parallel or exclusive, the process with larger LET is reachable from the process with smaller LET. From Lemma 1, we know that in an LRTS workflow, two processes are either parallel, exclusive, or reachable from one to the other. Therefore, Lemma 4 can be re-stated as Lemma 5 that if two activity processes in an LRTS workflow are reachable from one to the other, the activity processes with larger LET is reachable from the one with smaller LET.

Lemma 5
For an LRTS workflow w, $p, q \in P_w$, $p.type == q.type ==$ ACT,
If (Reachable(p, q) $\oplus$ Reachable(q, p)) == true, and LET(p) < LET(q),
Reachable(p, q) == true.

On the other hand, two processes are concurrent if and only if they are structurally parallel and overlapped in EAIs. On the basis of Lemma 5, a process is before another one if one of the following statements holds, (1) the latter is structurally reachable from the former, and (2) they are structurally parallel and the EAI of the former is before the EAI of the latter. The definition of the structural and temporal relationships in LRTS workflow is formally described as following.

Definition 10 (Structural and Temporal Relationships in LRTS workflow)
For an LRTS workflow w,
Concurrent: $P_w \times P_w \Rightarrow$ {true, false}
Concurrent(p, q) == true if and only if
(Parallel(p, q) $\wedge$ EAI(p) $\approx_{TI}$ EAI(q)) == true.
Before: $P_w \times P_w \Rightarrow$ {true, false}
Before(p, q) == true if and only if
(Reachable(p, q) $\vee$ (Parallel(p, q) $\wedge$ EAI(p) $\prec_{TI}$ EAI(q))) == true.
After: $P_w \times P_w \Rightarrow$ {true, false}
After(p, q) == true if and only if Before(q, p) == true.

Chapter 3. A Delegation Framework for WfMS based on Task-Role based Access Control and TS workflow

Tasks represent the basic logical steps of business processes, and roles combining users with similar responsibility together are the core components for access control management in modern WfMS. With both tasks and roles as core concept, we introduce a delegation mechanism for WfMS coordinated with TRBAC. In section 3.1, TRBAC and the issues related to delegation in WfMS are sketched. The task and role models adopted in this dissertation are described in section 3.2, and our delegation framework for WfMS and TRBAC is depicted in section 3.3. In section 3.4, a case study is made, and the related works are discussed and compared with our methodology in section 3.5.

3.1 Background

3.1.1 Task-Role based Access Control Model

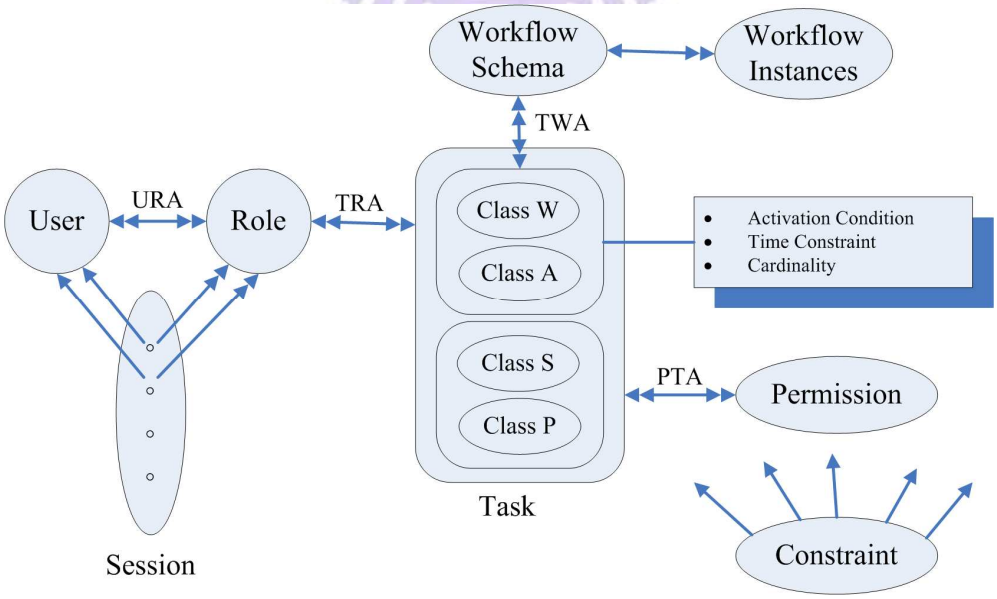


Figure 10 The TRBAC Model [17]

Based on RBAC96 [15][16], TRBAC model [17] illustrated in Figure 10 works for modern enterprise environments in which tasks are the fundamental units of business processes. TRBAC model binds permissions on tasks and groups users operating the same tasks into roles. Rather than accessing business objects directly [15][16], users accomplish their works through tasks in which permissions are properly defined and protected. Restricting the access rights of business objects on tasks facilitates permission management and reduces the risks of inappropriate permission authority made by users. In TRBAC [17], tasks are classified into four classes according to whether the task participates in a business process and whether the task is inherited by the ancestor job. The classes of tasks in TRBAC model are illustrated in Table 1.

Table 1 Classes of Tasks in TRBAC Model [17]

	Non-inheritable	**Inheritable**
Passive access	P (private)	S (supervision)
Active access	W (workflow)	A (approval)

3.1.2 Delegation Approaches in RBAC and TRBAC

Delegation is to authorize subjects like access rights or works between users or roles, and is often built based on access control models. The user (or role) authorizing the subject is the *delegator*, and the one who receives it is the *delegatee*. In RBAC [15][16], permissions to business objects, like documents or devices, etc. are bound with roles. RBDM0 [19] provides a flexible way for granting and revoking permissions between roles. RBDM1 [20], an extension of RBDM0 [19], is more realistic since it organizes roles with hierarchy. Both techniques are focused on delegation of roles among human users through identifying *can-delegation* relationships between roles.

In [21], the essence of this delegation model is that a user delegates a particular right to another user, and delegation of partial permissions is allowed. Osborn separates users in organization, role hierarchies, and relationships among privileges into different graph models in [22] and [41], and shows a simple way to delegate privileges to users by creating a delegatee

role. In [18], Crampton gives a further discussion about both granting and transferring access rights between roles. When access rights are granted from the delegator to the delegatee, the delegated access rights are available for both the delegator and the delegatee [18]. On the other hand, if the access rights are transferred, only the delegatee holds the access rights after the delegation [18]. Besides, Crampton considers both can-delegate and can-receive relationships, and introduces the concept of administrative scope to improve the efficiencies in delegation controlling [18].

Besides, tasks, the basic logic units of business processes, should also be considered in delegation. In [18], Crampton addresses the issues like upward delegation and authorization of appropriate permissions for delegation to adopt RBAC-based delegation mechanism in task-based WfMS. Bammigatti associates tasks into permission management and develops a new model for using RBAC in workflow system [42]. In TAC model [43], the permissions possessed by roles and required by tasks are described separately, and the assignment of tasks to roles is thus constrained. With such constraints, a protocol enabling delegation of task instances from users to roles is established [43].

TRBAC [17] binds the permissions with tasks, and the tasks with roles. With the roles assigned to users, users access business objects and accomplish their duties through tasks. Therefore, authorization of permissions is not necessary for delegation of tasks and task instances in TRBAC. In our previous work [24], a delegation framework for TRBAC has been initially established without considering the temporal issues. Zhang et al. develops a delegation model for time constraints-based TRBAC [44]. However, Zhang reduces TRBAC model as TRBACM model in which permissions and tasks are separately bound with roles. In [44], users delegate permissions together with tasks to accomplish their works, and the methodology violates the primary sprits of TRBAC [17].

3.1.3 Separation of Duty

Separation of Duty (*SOD*) is a security principle which requires multiple users to be responsible for the completion of a work [45]. Since delegation transfers permissions and tasks among users, the delegation approaches also follows the SOD policy of the corresponding access control system. In RBDM1 [16], Ferraiolo defines SOD as "*For a particular set of transactions, no single individual is allowed to execute all the transactions within the set.*" Botha discusses SOD in workflow environments both statically and dynamically [46]. In Botha's study, four possible conflicts, conflicting roles, conflicting permissions, conflicting users, and conflicting tasks, are described, and the corresponding methods for the conflicts are developed.

TRBAC [17] offers SOD policy at both task and instance level, and defines that some tasks are *mutually-exclusive* to each other. In task level SOD, for the roles played by a user, none of the tasks assigned to the roles are mutually-exclusive. In instance level SOD, the policy is effective for the tasks belonging to the same workflow instance. The task instances instantiated from the mutually-exclusive tasks in a workflow instance can not be executed by the same user [17]. In this dissertation, we follow the SOD policy established in TRBAC when delegation.

3.2 Task and Role Model

Tasks are the basic components describing pieces of works in logical steps within a workflow [1], and are modeled as activity processes in TS workflow model. *Permissions* are the rules describing the admission in accessing business objects such as documents or computation resources. In this dissertation, individual permissions are bound with tasks on the basis of TRBAC. Besides, it is assumed that only the tasks related to enactment of workflows can be delegated during run-time, and therefore, only "Workflow" and "Approval" are considered as the classes of tasks in this dissertation. Task is formally defined as following.

Definition 11 (Task)
For a TS workflow w, $T_w = \{t \mid t \in P_w, t.type == ACT\}$ is the set of tasks in w.
Let T be the set of all the tasks managed by WfMS,
$T_w \subseteq T$, $\forall t \in T$, the following properties are additionally modeled:
P_t is the set of permissions to business objects bound on t.
R_t is the set of roles assigned to t.
$t.class \in$ {Workflow, Approval} is the class of t.

During run-time, TS workflow instances are instantiated from a TS workflow, and the tasks in the TS workflow are also instantiated as *Task Instances*. Task instances are the basic units for daily duties. When a task instance is going to be executed, the system offers the instance to a role in accordance with the corresponding TS workflow. Then, the instance is allocated to the work list of one of the users playing the offered role. The user executes the instances in his work list, and submits the instance whenever it is complete. A task instance is suspended once the responsible user becomes unavailable for a certain time, and is resumed from suspension when the responsible user is again available. A task instance is failed if it is not completed in its active interval, and is discarded if it is not executed until the end of the TS workflow instance. The active interval of a task instance is obtained from the EAI of its source task and the starting time of the TS workflow instance, and indicates when it can be started and the corresponding deadline. TS workflow instance and task instances are modeled as following.

Definition 12 (TS workflow Instance and Task Instance)
A TS workflow instance $wi = (w, I_{wi}, st)$.
w is the TS workflow instantiating wi
I_{wi} is the set of the task instances instantiated from the tasks in T_w.
Let I be the set of all the task instances managed by WfMS.
$I_{wi} \subseteq I$, $\forall i \in I_{wi}$, $i = (wi, tk, ar, s, eu, ai)$.
$tk \in T_{wi.w}$ is the task instantiating i.
$ar \in R_{tk}$ is the role i offered.
$s \in$ {Initiated, Discarded, Offered, Allocated, Completed, Suspended, Failed} is the status of i.
eu is the user executing the task instance.
$ai = [wi.st + EST(tk), wi.st + LET(tk.eai)]$ is the active interval of i.
st is the time point wi being initialized.

Users are the participants of business processes. A user may play multiple roles for various businesses, and a role can be played by multiple users also. During run-time, users execute task instances in their work list to accomplish their daily duties. The status of a user is normally *available* and is transited to *unavailable* when he/she is not available for work. A user is formally modeled as following.

Definition 13 (User)
Let U be the set of all the users managed by WfMS.
$\forall u \in U$, $u = (R_u, WL, cs)$.
R_u is the set of roles played by u.
$WL = \{ i \mid i \in I, i.eu = u, i.s \in \{\text{Allocated, Completed, Suspended, Failed}\} \}$ is the work list of u.
$cs \in \{\text{Available, Unavailable}\}$ is the current status of u.

Roles represent the collections of users with common responsibilities [15][16]. In this dissertation, a role is modeled as a collection of the users responsible for the same tasks with certain timing constraints. The definition roles are formally described as follows.

Definition 14 (Role)
Let R be the set of all the roles managed by WfMS.
$\forall r \in R$, $r = (U_r, T_r, etd)$.
U_r is the set of users playing r.
T_r is the set of tasks assigned to r.
etd is a time description indicating when r is active.

Roles are organized with the *role hierarchy*. The role hierarchy indicates inheritance relationships and partial orders between roles to reflect the organization lines of authority or responsibility [15]. In this dissertation, the role hierarchy is modeled with directed acyclic graph (DAG) like in [15], [47], and [48]. Among the role hierarchy, the roles in higher positions possess larger authority, and the connected roles are more coherent than disconnected ones [15][47][48]. The number of edges between two connected roles in the role hierarchy is defined as their *distance*. The roles closer in distance are related more tightly than roles farther. The role hierarchy and the function calculating the distance between two roles in a hierarchy are defined

in the following definition.

Definition 15 (The Role Hierarchy)
The role hierarchy $RH \subseteq R \times R$.
$\forall (r_1, r_2) \in RH$, (r_1, r_2) shows a partial order that all inheritable tasks assigned to r_1 can also be assigned to r_2.
$\forall r, r' \in R$, $r' \succ_{rh} r$ holds if there exists $(r, r_1), .., (r_k, r') \in RH^*$.
RH is acyclic, and if $r' \succ_{rh} r$ holds, $r \succ_{rh} r'$ does not.
DisRH() shows the distance between two roles in the role hierarchy:
(1) DisRH(r, r) = 0,
(2) if $r' \succ_{rh} r$, DisRH(r, r') = -(k+1) and DisRH(r', r) = k+1,
(3) DisRH(r, r') is undefined while neither $r' \succ_{rh} r$ nor $r \succ_{rh} r'$ holds.

3.3 Delegation Framework for WfMS on TRBAC

3.3.1 The properties of Delegation

A delegation is primarily composed of a delegator, a delegatee and a delegating subject. In TRBAC, since the permissions are bound with tasks, the task instances are delegated between users during run-time. For each delegation, the delegator, the delegatee, the delegated task instance, and the delegation duration are recorded in a delegation record. In this dissertation, the duration is constrained not exceeding the active interval of the delegated task instance. Our framework allows multi-level delegation [19][21], and a task instance might be delegated several times. For each delegated task instance, a delegation record keeps tracking its status no matter how many times it is delegated. All the delegators who once delegated the task instance are put into the historical delegator list in the corresponding delegation record.

Besides, we assume that the maximal times that a task instance can be delegated are constrained by an enterprise policy named the *Maximal Levels of Delegation* (*MLD*). MLD is a non-negative integer. If MLD is equal to 1, multi-level-delegation is forbidden. With above features, the format of a delegation record is defined in Definition 16. When a delegation occurs, the corresponding record is attached to the task instance for reference as definition 10 shows.

Definition 16 (Delegation Record)

Let D be the set of all the delegation records managed by WfMS.

$\forall d \in$ D, d = (*di*, *dr*, *de*, *dur*, HDRL).

di is the delegated task instance, and $\forall d' \in$ D, if $d' \neq d$, $d'.di \neq d.di$.

dr$\in$ U is the original delegator.

de$\in$ U, is the current delegatee.

dur is a time interval indicating during when *d* is effective, and $di.ai \supseteq_{TI} dur$.

HDRL = $\{u_1, u_2, \ldots, u_k\}$ is the historical delegator list. u_1 == *dr*, and $\forall u_m \in$ HDRL, $m<k$, u_m delegated *di* to u_{m+1}, and u_k delegated *di* to *de*. |HDRL| $\leq$ MLD.

Definition 17 (Delegation Records in Task Instances)

For any task instance *i*, if *i* is delegated, *i.dr*$\in$ D, *i.dr.di* == *i*; otherwise, *i.dr* == Ø.

Algorithm 1 describes how a task instance is delegated in our framework.

Algorithm 1 Delegation Algorithm - *DA*

Input: the delegating task instance *dti*,
the delegatee *u*, and
the designated delegating duration *ddur*

Pre-Condition: $dti.ai \supseteq_{TI} ddur$

```
DA {
01: if (dti.dr ≠ Ø) {
02:     if( |dti.dr.HDRL|+1 > MLD )
03:         EXCEPTION( MAX_DELEGATION_LEVEL_REACHED );
04:     else {
05:         add dti.eu to dti.dr.HDRL;
06:         dti.dr.dur = ddur;
07:         dti.dr.de = u;
08:     }
09: } else {
10:     dti.dr = (dti, dti.eu, u, ddur, {dti.eu});
11:     add dti.dr to D;
12: }
13: remove dti from dti.eu.WL;
14: add dti to u.WL;
15: dti.eu = u;
}
```

The system invokes Algorithm 1 when delegating a task instance to the designated delegatee. At line 1, the algorithm checks whether the input task instance has been delegated. If so, the algorithms checks the size of historical delegator list of the task instance at line 2 to assure that the delegation does not violate the restriction held by MLD. According to the input parameter, the delegation record is updated from line 5 to 7. Otherwise, the task instance is delegated for the first time. A new delegation record is created and attached to *dti* at line 10 and 11. After the delegation record is well updated or created, the task instance is transferred from the delegator's work list to the delegatee's from line 13 to 15

3.3.2 Delegatee Decision

Algorithm 1 does not concern whether a delegatee is appropriate for delegation or not. In multi-level-delegation, if a task instance is delegated to one of the delegators who once delegated the task instance, a delegation loop occurs. Delegation loop causes redundancy in business and should be avoided [49]. Algorithm 2 is constructed to remove users inducing delegation loop from the candidate users.

Algorithm 2 Removing Users Causing Delegation Loop - *RUDL*
Input: the candidate user set CUS, and
the target task instance *ti*
Pre-Condition: CUS $\subseteq$ U
User Set RUDL {
01: if (*ti*.*dr* $\neq$ Ø)
02: CUS = CUS \ (*ti*.*dr*.HDRL);
03: return CUS;
}

Taking a set of users and a task instance as the input parameters, Algorithm 2 eliminates users causing delegation loop from the input user set. Each delegator user who once delegated the instance is recorded in the historical delegator list of the delegation record. After removing the historical delegators from the input user set at line 2, CUS is returned at line 3.

SOD is another issue in delegatee decision. Since delegation happens during run-time, we focus on maintaining instance level SOD policy for the task instances in a TS workflow instance. For each TS workflow, the mutually-exclusive tasks are grouped in records, and a task might belong to multiple records. For example, task "auditing" is mutually-exclusive to both task "ordering" and "purchasing", but "ordering" and "purchasing" are not mutually-exclusive. Therefore, two records are established, and "auditing" is contained in both records with "ordering" and "purchasing" separately. The record for mutually-exclusive tasks and the SOD constraints adopted in this dissertation is defined as follow.

Definition 18 (Mutually Exclusive Tasks)
MET is the set of all the records of mutually-exclusive tasks.
$\forall$ *met*$\in$ MET, *met* = (*w*, T_{met}).
w is a TS workflow.
$T_{met} \subseteq T_w$ is a set of mutually-exclusive tasks
$\forall$ t_i, $t_j \in T_{met}$, t_i and t_j are mutually-exclusive.

Definition 19 (Instance Level SOD constraints)
$\forall$ workflow instance *wi*, ti_1, $ti_2 \in I_{wi,}$ and (*wi.w*, T_{met})$\in$ MET,
If $ti_1.tk$, $ti_2.tk \in T_{met}$, $ti_1.eu \neq ti_2.eu$.

In a delegation, SOD also holds. For a task instance *ti* which is being delegated, a user executing a task instance mutually-exclusive to *ti* can not be the delegatee of *ti*. Taking a set of candidate delegatees and a task instance as the input parameters, Algorithm 3 eliminates the users violating instance SOD from the candidate delegatees.

Algorithm 3 Removing Users Involved in Mutually-Exclusive Tasks - *RUMET*
Input: the candidate user set CUS, and
the target task instance *ti*
Pre-Condition: CUS$\subseteq$U
User Set RUMET {
01: $\forall$ $i \in I_{ti.wi} \backslash \{ti\}$ {
02: if ($\exists$ *met*$\in$ MET, *met.w* == *ti.wi.w*, *i.tk*, *ti.tk*$\in T_{met}$)
03: remove *i.eu* from CUS;
04: }

05: return CUS;
}

To show the correctness of Algorithm 3, we prove that the following lemma holds.

Lemma 6
Algorithm 3 follows the SOD constraints defined in Definition 19.

Proof:

By way of contradiction (*B.W.O.C*), we assume that a user $u \in$ RUMET(CUS, *ti*) is now executing *ti'* which is mutually-exclusive to *ti*. Let a workflow instance wi_1 contains *ti'* and *ti*. With the assumption, *ti'.tk* and *ti.tk* are mutually-exclusive to each other, and *ti'.eu* = *u*. Since *ti.wi* = wi_1 and *ti'* $\in I_{ti.wi}$, *ti'* is selected at line 1. On the other hand, by definition 11, there exists *met* $\in$ MET that *met.w* = *ti.wi.rws* = *ti'.wi.rws* = wi_1*.rws* and *ti'.tk*, *ti.tk* $\in T_{met}$. Therefore, the expression at line 2 is true for *ti'* and *ti'.eu* is removed from the result set at line 3. Thus, *u* is not included in the result set and the assumption is contradicted. Algorithm 3 follows SOD constraints defined in Definition 19. □

Intuitively, an unavailable user can not be the delegatee of any delegation, and a task instance should not be delegated to the user currently executing it. Concluding these two issues and the algorithms described in this section, the algorithm removing inappropriate users from a set of candidate delegatees is constructed as follows.

Algorithm 4 Removing Inappropriate Users - *RIU*
Input: the candidate delegatee set CDS, and
the delegating task instance *dti*
Pre-Condition: CDS $\subseteq$ U
Candidate Delegatee Set RIU {
01: $\forall u \in$ CDS,
02: if (*u.cs* == Unavailable) remove *u* from CDS;
03: CDS = RUMET(RUDL(CDS, *tdi*), *tdi*) \ {*dti.eu*};
04: return CDS;
}

Algorithm 4 first removes the unavailable users from the input set at line 2. At line 3, the algorithm invokes Algorithm 2 and Algorithm 3 to remove the users causing delegation loop or violating SOD. After removing the current executing user of *dti*, Algorithm 4 returns the result set at line 4.

3.3.3 Delegation from System Request

When a suspended task instance is nearly timed out, the system might need to spontaneously request a delegation for the task instance. We assume that a suspended task instance is *emergent*, and need to be delegated automatically if the proportion of its remaining active interval is less than an enterprise policy named the *Emergent Execution Ratio* (*EER*), a real number ranged from 0 to 1. To automatically delegate the emergent task instance, WfMS needs first decide an appropriate delegatee.

The role hierarchy indicates the organization lines of authority and responsibility [15], and can be used for exploration of possible candidate delegatees. Along with the role hierarchy, a task instance can be delegated upward or downward. When a task instance of daily works is being delegated, the system gathers the users playing lower roles related to the offered role of the instance as candidate delegatees. On the other hand, for the instances of the tasks related to decision making, the system commits an upward discovery from the offered role in the role hierarchy for the candidate delegatees. Besides, the users playing roles closer to the offered role in the role hierarchies are considered as better candidates in delegatee decision. Based on Definition 15, the algorithm discovering the role hierarchy for the candidate delegatees is constructed as follows.

Algorithm 5 Discovering the Role Hierarchy - *DRH*
Input: the delegating task instance *dti*
Candidate Delegatee Set DRH {
01: if(*dti*.*tk*.*type* == Approval) $p = 1$;
02: else if(*dti*.*tk*.*type* == Workflow) $p = -1$;
03: m = 0;
04: US = Ø;
05: loop {
06: GR = Ø;
07: $\forall r \in R$, DisRH(*dti*.*ar*, *r*) == $p*m$
08: add *r* to GR;

```
09:    if ( GR == Ø ) return Ø;
10:    ∀ r∈ GR, r.etd ⊇_TD {dti.ai}
11:        US = US ∪ U_r;
12:    US = RIU ( US, dti );
13:    if ( US == Ø ) m = m + 1;
14:    else break;
15: }
16: return US;
}
```

At line 1 and 2, according to the class of the *dti*'s task, the algorithm decides the direction to explore the role hierarchy. Algorithm 5 commits an upward discovery for the tasks typed "Approval" or a downward discovery for the tasks typed "Workflow". From line 5 to 14, the algorithm does a breadth first search in the role hierarchy. At line 9, empty GR set represents that all roles connected to the offered role along with the designated direction in the role hierarchy are explored, and no proper delegatee is found. Therefore, Algorithm 5 returns Ø as the result. If GR is not empty, the users playing roles in GR is gathered into user set US and filtered with Algorithm 4. If US set is not empty after the removal of conflict users, the algorithm returns US as the result set. Otherwise, the discovery continues with further distances.

With Algorithm 5, the algorithm for delegation requested by the system is described as Algorithm 6. WfMS tracks the status of the executing task instances, and invokes algorithm 6 whenever an emergent task instance is found. Algorithm 6 acquires the candidate delegatees by exploring the role hierarchy with Algorithm 5 at line 2. Exception is raised if Algorithm 5 returns no candidates. Otherwise, Algorithm 6 randomly chooses a delegatee from the candidate delegatees and invokes Algorithm 1 to delegate the emergent task instance.

Algorithm 6 Delegation from System Request - *DSR*
Input: the delegating task instance *dti*
Pre-Condition: *dti.s* = Suspended, E(*dti.ai*) > *ctime*
(E(*dti.ai*) - *ctime*) / (E(*dti.ai*) - S(*dti.ai*)) < EER

```
DSR {
01:  CDS = DRH(dti);
02:  if( CDS == Ø ) EXCEPTION( NO_PROPER_DELEGATEE );
03:  else {
04:      randomly choose a user u from CDS;
05:      DA( dti, u, [ctime, E(dti.ai)] );
06:  }
}
```

3.3.4 Delegation from User Request

Many modern enterprises adopt user-authorized delegation as the primary delegation methodology. The RBAC-based studies like [20], [22], [18], and [23], also describes how roles and permissions are delegated under user authorization.

With our framework, a user can authorize two types of delegation. First, a user may delegate task instances currently allocated to him. Second, a user may delegate the task instances going to be allocated to him during a specific period.

To request a delegation, the delegator fills in an authorization form which designates the delegating subject, the delegatee and the activation duration of the delegation. For the first type of delegation, the delegator designates a task instance residing in his work list as the delegating subject. The duration to authorize the delegation must be contained by the active interval of the delegating task instance. For the second type of delegation, the delegator designates an executable task by any of the roles he playing as the delegating subject. The duration to authorize the delegation must be contained by the effective duration of the role.

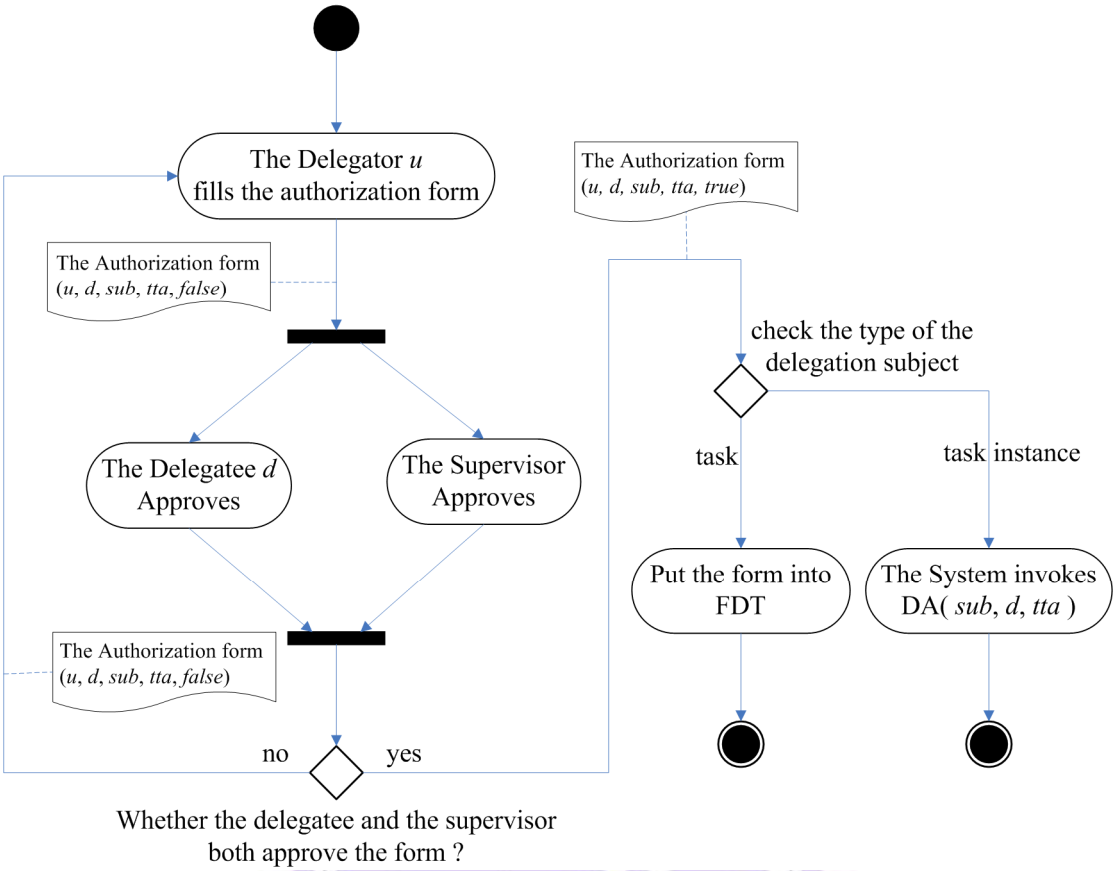


Figure 11 The Process for Delegation from User Request

After accomplishing the authorization form, the delegator user submits the form to request the approval from his supervisor and the designated delegatee. If the delegation is approved, for the first type of delegation, the designated task instance is delegated to the delegatee user with algorithm 1 immediately. For the second type of delegation, the approved form is put into *Forthcoming Delegation Table* (FDT). According to the form, the task instances of the designated task which is allocated to the delegator in the specified duration are delegated to the designated delegatee. Figure 11 represents the process of delegation from user request, the authorization form is defined in Definition 20, FDT is defined in Definition 21, and finally, Algorithm 7 shows how WfMS handles the second type of delegation.

Definition 20 (Authorization Form) Let AP be the set of all authorization forms. $\forall ap \in$ AP, $ap = (dr, de, sub, tta, is\_approved)$. dr is the delegator user, $dr \in$ U.

de is the designated delegatee user, $dr \in U$, $dr \neq de$.
sub is the subject of delegation, $sub \in T \cup I$, and
if $sub \in T$, $\exists r \in u.R_U \cap sub.R_T$ and $r.etd \supseteq_{TD} \{tta\}$,
otherwise, if $sub \in I$, $sub \in dr.WL$, $sub.s \neq$ Completed, and $sub.ai \supseteq_{TI} tta$.
tta, time to authorize, is the time interval that *dr* delegates the subject to *de*.
is_approved is a Boolean variable showing whether *ap* is approved.

Definition 21 (Approved Form)
$\forall ap \in AP$, if $ap \in FDT$, $ap.sub \in T$, $ap.is\_approved$ = true, and $E(ap.tta) \geq$ ctime.

Algorithm 7 Handle Forthcoming Delegation - *HFD*
Input: a task instance *i*,
a user *u*
Pre-Condition: *i* is allocating to *u*
HFD {
01: if ($\exists ap \in FDT$, $ap.dr == u$, and $ap.sub == i.tk$) {
02: if(RIU({*ap.de*}, *i*) $\neq \emptyset$)
03: DA(*i*, *ap.de*, [*ctime*, *min*(E(*ap.tta*), E(*i.ai*))]);
04: else EXCEPTION(INAPPROPRIATE_DELEGATEE);
05: }
}

The system invokes Algorithm 7 whenever a task instance is being allocated to its execution user. At line 1, the algorithm first checks if the user and the task of the task instance are recorded on an authorization form in FDT. If the task instance is authorized to be delegated, Algorithm 7 then invokes Algorithm 4 to check whether the designated delegatee user on the form violates any delegation constraints. If the check is not passed, an exception is raised and further handling is necessary. According to different policies, the hanging task instance might be handled manually or delegated by WfMS automatically. Otherwise, Algorithm 1 is invoked to perform the delegation.

3.3.5 Revocation

A successful delegation can be revoked by its delegator before it ends [18]. To revoke a delegated task, the authorization form is simply removed from the FDT. On the other hand, for

revocation of a delegated task instance, the delegatee's contribution on the task instance might be preserved or discarded according to the system settings and the enterprise policies. The task instance is transferred back to the work list of the user requesting the revocation, the *revoker*, and the revoker continues executing the task instance after the revocation.

Revoking a multi-level delegation is complex. For a multi-level delegation, all the users recorded in the historical delegator list might revoke the delegation. If the revoker is the original delegator, after the delegated task instance is transferred back, the delegation record is eliminated. Otherwise, if the revoker is the other delegator in the historical delegator list, the revoker becomes the delegatee of the delegation after the revocation. The revoker and the other delegators behind the revoker are removed from the historical delegator list.

When a delegation runs out of its effective duration, the system revokes it automatically. The delegated task instance is transferred back like the revocation is requested by the original delegator. Algorithm 8 is constructed as follows for revocation.

Algorithm 8 Revocation Algorithm - *RA*
Input: the subject to be revoked *rsub*,
the revoker *u*
Pre-Condition: *rsub*$\in$ T$\cup$I, *u*$\in$ U
RA {
01: if (*rsub*$\in$ T && $\exists$ *ap*$\in$ FDT that *ap.dr*=*u*, and *ap.sub*=*rsub*)
02: remove *ap* from FDT;
03: else if (*rsub*$\in$ I && *rsub.dr* == *d* that *u*$\in$ *d*.HDRL, and *rsub.s*$\neq$ Completed) {
04: alert *dti.eu* that *d* is going to be revoked;
05: remove *rsub* from *rsub.eu*.WL;
06: add *rsub* to *u*.WL;
07: *rsub.eu* = *u*;
08: if(*u* == *d.dr*) {
09: remove *d* from D
10: *rsub.dr* = Ø;
11: } else {
12: *d.de* = *u*;

```
13:       u and all the users behind u in the d.HDRL are removed from d.HDRL;
14:    }
15:    alert dti.eu dti is transferred back to his work list;
16: } else EXCEPTION(INVALID_REVOCATION);
}
```

Algorithm 8 takes the subject being revoked and the revoker as the input parameters. If the subject is a task, Algorithm 8 checks whether there is any corresponding authorization form, and removes the form from FDT at line 1 and 2. Otherwise, if the subject is a task instance, Algorithm 8 checks the corresponding delegation record to assure the revocation is valid at line 3. If valid, the current delegatee of the delegated instance is first alerted at line 4. The delegated instance is removed from the delegatee's work list, and transferred to the revoker from line 5 to 7. If the revoker is the original delegator of the delegation, the delegation record is eliminated from line 8 to 10. Otherwise, the record is updated. The delegatee is assigned to the revoker at line 9; the revoker and the delegators behind him are removed from the historical delegator list at line 10. The revoker is alerted at line 15. At line 16, an exception is raised if the revocation is invalid.

3.4 Case Study

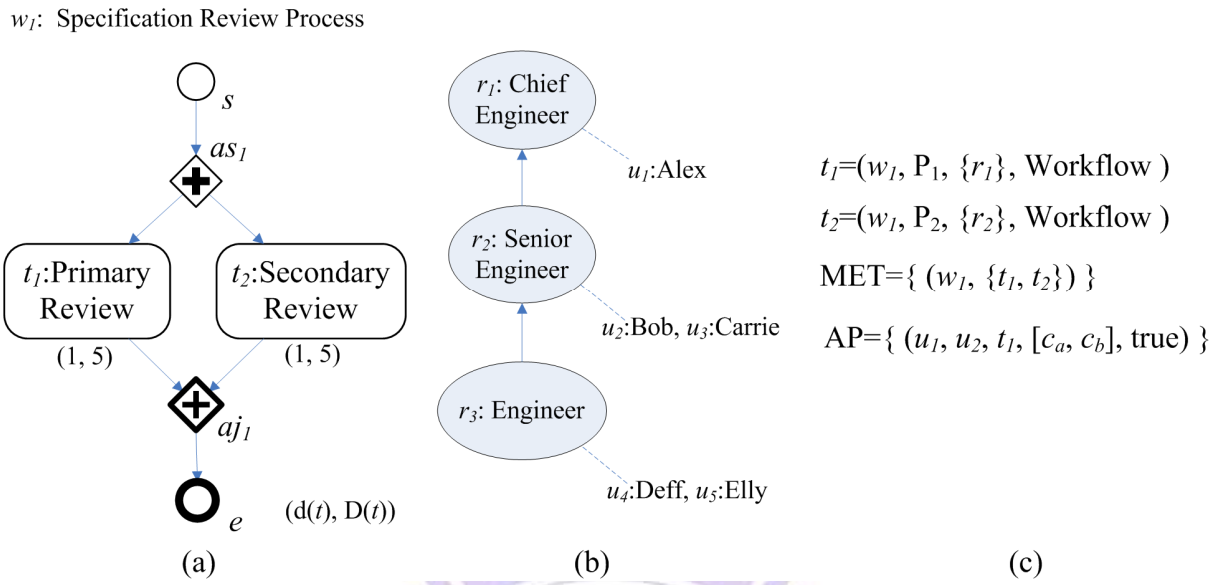


Figure 12 (a) The Sample TS Workflow Specification, (b) The Sample Role Hierarchy and User Assignment, and (c) The Information about Tasks, Mutually-exclusive Tasks, and Authorization Applications

In this section, we adopt a specification review process as an example to show the feasibility of our approach. The workflow specification of the review process, the partial role hierarchy, and the other related information are illustrated in Figure 12. In this case, the review process is composed of two tasks, primary review and secondary review. Chief Engineer is in charge of the primary review, and Senior Engineer is responsible for the secondary one. These two review tasks are mutually-exclusive. Their EAI are both [0, 5] after calculation. Since these two tasks reside on different branches split from the AND-split process, as_1, they are concurrent during execution.

Let Alex is busy in his duty, and apply for delegation of all the reviews allocated to him during the time interval [c_a, c_b]. The application is approved by Bob, the designated delegatee, and his supervisor. In other words, all the review jobs allocated to Alex during [c_a, c_b] would be delegated to Bob instead. At time c_1, $c_a < c_1$ and $c_1+5 < c_b$, a workflow instance of w_1, wi_1 = ({i_t_1, i_t_2}, w_1, c_1), is instantiated so that the task instances i_t_1 and i_t_2 are instantiated on the basis of t_1 and t_2. i_t_1 and i_t_2 are offered to Chief Engineer and Senior Engineer, and allocated

to Alex and Carrie correspondingly. Now, i_t_1 = (wi_1, t_1, r_1, Allocated, u_1, [c_1, c_1+5], Ø), and i_t_1 = (wi_1, t_2, r_2, Allocated, u_3, [c_1, c_1+5], Ø). Because Alex has been approved to delegate all the reviews during [c_a, c_b] to Bob, Algorithm 7 invokes Algorithm 1 to delegate i_t_1 to Bob. The delegation record d = (i_1, u_1, u_2, [c_1, c_1+5], {u_1}) is created, and i_t_1 becomes (wi_1, t_1, r_1, Allocated, u_2, [c_1, c_1+5], d) after the delegation.

At time c_2 which is in the middle of the active interval of i_t_1, $c_1 < c_2 < c_1$+5, Bob gets an emergent call and becomes unavailable right away. The task instances in his work list are all suspended. Let us assume that EER equals to 1. Thus, WfMS invokes Algorithm 6 to delegate i_t_1 to another appropriate delegatee immediately. In Algorithm 6, Algorithm 5 is first invoked to explore the role hierarchy for a proper delegatee. Because t_1 is typed "Workflow", the role hierarchy is explored downward from Chief Engineer, the role i_t_1 offered. Alex is the only user now playing Chief Engineer, and is eliminated from the candidate delegatee set by Algorithm 2 to avoid delegation loop. When considering Senior Engineer, Carrie is eliminated from the candidate set by Algorithm 3 because of the SOD policy, and Bob is eliminated from the candidate set by Algorithm 4 because he is unavailable. No users playing Senior Engineer are appropriate to take the task. Therefore, Engineer is then considered. After all, Deff and Elly are included in the candidate set, and Deff is randomly decided as the new delegatee of i_t_1. Algorithm 1 is invoked to delegate i_t_1 to Deff. d is updated as (i_1, u_1, u_4, [c_2, c_1+5], {u_1, u_2}), and i_t_1 is updated as (wi_1, t_1, r_1, Allocated, u_4, [c_1, c_1+5], d).

At c_3, $c_2 < c_3 < c_1$+5, Alex finishes his jobs ahead of time, and decides to finish i_t_1 himself. Alex invokes Algorithm 8 to revoke i_t_1. Deff is first alerted and i_t_1 is then revoked. The delegation record d is removed, and i_t_1 is updated as (wi_1, t_1, r_1, Allocated, u_1, [c_1, c_1+5], Ø). In summary, this case demonstrates the delegations requested from a user and the system, and indicates how the constraints like delegation loop and SOD work in automatic delegatee decision.

3.5 Discussion

In this section, we compare our framework with the latest popular approaches: [20], [18], [43], and [44]. Table 2 illustrates the characteristics of above approaches and ours correspondingly.

Table 2 Comparison of Characteristics of Various Delegation Models

Characteristics	**RBDM1** [20]	**Crampton**[18]	**Gaaloul** [43]	**VTTRDM** [44]	**Our Approach**
Access Control	RBAC [15][16]	RBAC [15][16]	TAC [43]	TRBACM [44]	TRBAC [17]
Delegation of Permissions	Grant	Grant & Transfer	No	Grant	No
Delegation of Tasks	No	No	Transfer	Yes	Transfer
Delegation of Task Instances	No	No	No	No	Transfer
Time Constraints	No	No	No	Yes	Yes
Automatic Delegation	No	No	No	No	Yes

RBDM1 [20] is a classic delegation model for RBAC [15][16], and can be adopted in managing delegation of permissions between users. Crampton et al. develop another RBAC-based delegation model for workflow systems [18]. Crampton's approach allows both grant and transfer operations for delegation of permissions while RBDM1 adopts only grant operation [18]. Crampton also raises the issues like upward delegation and permission authorization for delegation of tasks in work-list based workflow systems [18]. However, both RBDM1 and Crampton's approach describe no methods about delegation of tasks.

With various access control models based on tasks and roles, Gaaloul's methodology [43], VTTRDM [44], and our approach can be adopted in managing delegation of tasks for workflow systems. Gaaloul's methodology describes constraints for delegation of tasks based on Task-oriented Access Control (TAC) model [43]. TAC model describes the permissions which a role owns and a task needs. Gaaloul's methodology allows a user to delegate his tasks to a role

which has sufficient permissions to execute the tasks. Since Gaaloul's methodology allows no delegation of permissions, it is limited and inflexible when selecting the delegatee for a delegation.

In VTTRDM [44], both permissions and tasks can be delegated between users. RBDM1 [20] is adopted in VTTRDM to manage the delegation of permissions. When delegating a task, if the delegatee does not have sufficient permissions to execute the task, permission delegation from the delegator to the delegatee is necessary to enable the execution [44]. Since in VTTRDM, the delegated permission is not limited being used for the delegated tasks only, security risk exists.

In our approach, tasks are delegated through user's authorization. Our approach is based on TRBAC, and a task is executed with a set of associate permissions. Therefore, the delegatee can execute the delegated task without delegation of permissions, and the security risks brought by delegation of tasks are eliminated. Besides, in [50], delegation is defined as "*A user allocates a task instance previously allocated to him to another user.*" While delegation of task instances is ignored in [43], and [44], our approach clearly states how to delegate task instances between users. For delegation of task instances, our methods could gather candidate delegatees and remove inappropriate users from the candidates. With our approach, a workflow system can automatically delegate an emergent task instance to an appropriate user to prevent the task instance from failure.

Regarding temporal issues, in VTTRDM, delegation is effective during a single time interval, and the delegated tasks are revoked after the interval [44]. Our approach is based on the time constraints between the delegated task instances and the related roles. Because a role might be activated in multiple time intervals, multiple or periodical time intervals are considered in our approach to provide a more realistic temporal constraints.

Chapter 4. Detecting Artifact Anomalies in TS workflow

A well-structured workflow may more possibly fail or produce unanticipated run-time behavior because of abnormal data manipulation [26][27][28][29][51]. The anomalies might be yielded differently when the temporal issues are considered. Thus, it is worthwhile to study how to detect artifact anomalies in TS workflow. In this chapter, artifact anomalies in TS workflow is first stated and modeled in section 4.1. The methodology detecting artifact anomalies in TS workflow is described in section 4.2. A case study is then introduced in section 4.3 to illustrate the feasibility of our methodology. Finally, in section 4.4, the related works are discussed and compared with our methodology.

4.1 Artifact Anomalies in TS workflow

4.1.1 Artifact Operations

In this dissertation, we assume that an activity process in a TS workflow may operate an artifact as one of the following ways: define (*Def*), use (*Use*) and kill (*Kill*). Defining an artifact is to assign a value to the artifact, and when an artifact is first defined, it is initialized. An activity process references an artifact through using it, and an artifact can not be used without definition. Killing an artifact is to remove the definition of the artifact, and using a killed artifact before it is defined again leads to errors during execution. As for the control processes in a TS workflow, it is assumed that they all do no operation (*Nop*) on any artifacts.

An artifact in a TS workflow is initially stated *undefined* (*UD*), and turns to *defined&no-use* (*DN*) after it is defined. When a DN artifact is used, its state becomes *defined&referenced* (*DR*). A DR artifact remains DR after being used, and transits to DN after

being defined again. An artifact in any states becomes UD after being killed.

On the other hand, the artifact operations made by concurrent processes are executed with undetermined order and might generate ambiguity to artifacts. When several concurrent processes operate on the same artifact, they race against each other for accessing the artifact and anomalies might thus be generated. For example, let one process make a definition to an artifact, and another one kills the artifact concurrently. The existence of the definition of the artifact becomes ambiguous because the execution order between the kill and the definition is not determined during design-time. These operations, called *Racing Operations*, require additional consideration during analysis, and are categorized according to the operations involved as following:

(1) *Racing Definition*(*s*)*&Kill*(*s*), abbreviated as *RDK*, represents a racing operation composed of both definition(s) and kill(s) with none or any usage(s).

(2) *Racing Definitions*, abbreviated as *RDS*, represents a racing operation composed of multiple definitions and no kills with none or any usage(s).

(3) *Racing Kills*, abbreviated as *RKS*, represents a racing operation composed of no definitions and multiple kills with none or any usage(s).

(4) *Racing Definition&Usage*(*s*), abbreviated as *RDU*, represents a racing operation composed of a single definition, any usage(s) and no kills.

(5) *Racing Usage(s)&Kill*, abbreviated as *RUK*, represents a racing operation composed of no definitions, any usage(s), and a single kill.

(6) *Racing Usages*, abbreviated as *RUS*, represents a racing operation composed of multiple usages only.

As the example mentioned above, an RDK or an RDS introduces state ambiguous (*AB*) to the artifact. Besides, an artifact transits to state UD after an RKS or an RUK, and state DR after an RDU. Since the artifact state after a usage varies based on the input state of the artifact, the artifact state after an RUS requires additional consideration in merging the input states of the usages involved in the RUS. The artifact and its related operations are modeled in Definition 22, and Figure 13 illustrates how artifact transits its state with different artifact operations.

Definition 22 (Artifact Model in TS workflow)
For an LRTS workflow *w*,
The set of all the artifacts operated in *w* is notated as A_w.
$\forall a \in A_w$, a.state$\in$ {UD, DN, DR, AB}.
The artifact operation made by processes in *w* is described as a relationship AOP:
AOP: $\{p \mid p \in P_w, p.type == ACT\} \times A_w \Rightarrow$ {Nop, Def, Use, Kill}
$\{p \mid p \in P_w, p.type \neq ACT\} \times A_w \Rightarrow$ {Nop}

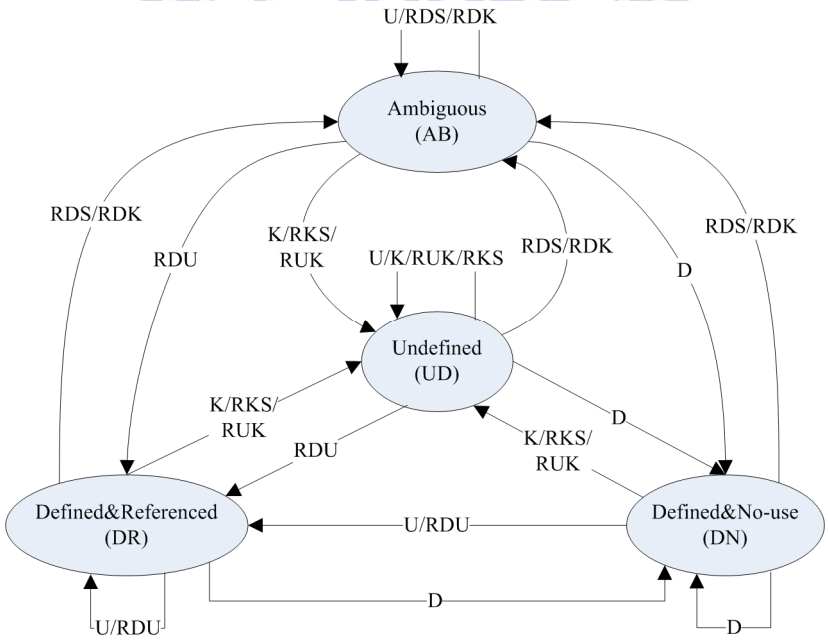


Figure 13 The Artifact State Transit Diagram

4.1.2 Artifact Anomalies

Artifact anomalies are generated from various structural and temporal relationships between artifact operations, and can be classified into four classes: *Useless Definition*, *Undefined Usage*, *Null Kill*, and *Ambiguous Usage*:

(1) Useless Definition:

Killing or defining a DN artifact makes the previous definition useless because the definition is destroyed (or redefined) without any usage. If an artifact remains DN at the end process, its definition is also useless because it is not used before the end of the workflow. A useless definition is a kind of redundancy indicating there might be logic error in the workflow schema and should be warned to designers.

(2) Undefined Usage:

An activity process might not be correctly executed if the essential artifact is not properly defined. Therefore, an undefined usage, i.e. using an UD artifact, is an error leading to faulty execution, and is necessary to be handled by the workflow designers.

(3) Null Kill:

A null kill represents a process try to remove an inexistent definition; e.g. to kill a UD artifact. A null kill is a kind of redundancy, and designers should be noticed about it.

(4) Ambiguous Usage:

An ambiguous usage means that an activity process uses an artifact which is ambiguous in definitions or in states. Therefore, the direct usage of an AB artifact is an ambiguous usage. The usage(s) involved in an RDS, an RDK, or an RDU are also ambiguous usages. Besides, if an artifact is stated DR/DN before an RKU, the usage(s) involved in the RKU is also ambiguous. Similarly, when an UD artifact meets an RDU, the definition in the RDU may not be made in time for the usages, and the usage(s) involved in the RDU is also ambiguous.

4.2 The Methodology Detecting Artifact Anomalies in LRTS workflow

In this section, the methodology detecting artifact anomalies in TS workflow is introduced. To simplify our discussion, the structured loops in all the TS workflows under analysis are first reduced with the methodology introduced in section 2.4.1, and the anomaly detection is made for LRTS workflows.

Our methodology is divided into three parts. In section 4.2.1, we first describe how to traverse an LRTS workflow to collect the structural and temporal relationships between the processes and the artifact operations. In section 4.2.2, according to the structural and temporal relationships gathered in the first part, the methodology analyzing relationships between the artifact operations are described. Finally, on the basis of the analysis made in the second part, the methodology detecting artifact anomalies in an LRTS workflow is concluded in section 4.2.3

4.2.1 Gathering Structural, Temporal, Artifact Information in LRTS workflow

In this section, we describe an algorithm to traverse an LRTS workflow to collect the ABStacks, EAIs, and the artifact operations made by activity processes in the LRTS workflow. The EAIs and ABStacks are calculated with the methods illustrated in Figure 6 and Figure 9 correspondingly. For each artifact *a*, an artifact operation list, notated as $AOPL_a$, is established. The definition of the list is formally described as following:

> Definition 23 (Artifact Operation List)
> For an LRTS workflow *w* and $\forall a \in A_w$,
> $AOPL_a$ is the list of artifact operations working on *a*,
> $\forall op \in AOPL_a$, $op = (p, a, est, let, type)$,
> $p \in P_w$, $p.type \in \{ACT, END\}$,
> $est = EST(p)$, and $let = LET(p)$, and
> $type = AOP(p, a)$.

With the definition, the algorithm gathering structural, temporal, artifact information in

LRTS workflow is described as following:

Algorithm 9 Information Gathering - *IG*
Input: an LRTS workflow *w*
Pre-Condition: *w.s.mark* == true, EAI(*w.s*) == [0, 0], *w.s.abstack* == « »
$\forall p \in P_w \backslash \{w.s\}$, *p.mark* == false

```
IG {
01: Queue tq;
02: ∀ (w.s, n)∈ F_w,
03:     tq.enqueue( n );
04: loop {
05:     Process p = tq.dequeue();
06:     if( (p.type∈ {AJ, XJ}) && (∃ (p', p)∈ F_w, p'.mark == false) ) continue;
07:     p.mark = true;
08:     calculate EAI(p);
09:     calculate p.abstack;
10:     if( p.type == ACT )
11:         ∀ a∈ A_w, AOP(p, a) ≠ Nop,
12:             add (p, a, EST(p), LET(p), AOP(p, a)) to AOPL_a;
13:     else if( p.type == END ) {
14:         ∀ a∈ A_w, add (p, a, EST(p), LET(p), AOP(p, a)) to AOPL_a;
15:         break;
16:     }
17:     ∀ (p, p')∈ F_w, tq.enqueue(p');
18: }
19: ∀ a∈ A_w, Sorting AOPL_a by LET
}
```

In Algorithm 9, a traverse queue is introduced to hold the order of traversal of processes in an LRTS workflow. Starting from the start process, the processes in a TS workflow is traversed along with flows. The EAIs, ABStacks, and artifact operations lists are calculated and collected correspondingly. To prevent unnecessary redundancy, a Boolean flag *mark* is given to each process. Besides the start process, the mark of each process in *w* is initialized as false, and when a process is calculated, its mark turned to true. Since a join process may have several in-flows, a Boolean expression is checked at line 6 to assure that the join process is calculated only when

each of its source process is calculated. Algorithm 9 records the artifact operation made by each activity process at line 12 and the "no operation" made by the end process in $AOPL_a$ at line 14 for further analysis of the definitions remaining useless at the end of *w*. At line 19, artifact operation list corresponding to each artifact is sorted by LET.

4.2.2 Collecting Structural and Temporal Relationships between Artifact Operations in LRTS workflow

Artifact operations are made by activity processes. Based on the structural and temporal relationships between the processes, the operations effective on the same artifact can be before, after, concurrent, or exclusive to each other. To identify these relationships between artifact operations is the foundation of analysis of artifact anomalies. Here, we first define the structural and temporal relationships between artifact operations as following:

Definition 24 (Relationships between Artifact Operations)
For an LRTS workflow *w* and $\forall a \in A_w$,
$\forall op_i, op_j \in AOPL_a$,
Before(op_i, op_j) == true if and only if Before($op_i.p$, $op_j.p$) == true.
After(op_i, op_j) == true if and only if After($op_i.p$, $op_j.p$) == true.
Concurrent(op_i, op_j) == true if and only if Concurrent($op_i.p$, $op_j.p$) == true.
Exclusive(op_i, op_j) == true if and only if Exclusive($op_i.p$, $op_j.p$) == true.

According to Definition 10, Definition 24, Lemma 1, and Lemma 3, the following lemma holds.

Lemma 7
For two operation *op* and $op' \in AOPL_a$,
(1) If Before(*op*, *op'*), $op.let < op'.let$
(2) If $op.let < op'.let$, After(*op*, *op'*) == false

Algorithm 10 is introduced to collect operations concurrent to each operation in an $AOPL_a$. To facilitate our discussion, it is assumed that each $AOPL_a$ is indexed, and $op_i \in AOPL_a$ indicates the *i*th operation in the list. Because $AOPL_a$ is sorted by LETs at the last part of

Algorithm 9, for $0 < i < j$, LET($op_i.p$) ≤ LET($op_j.p$). Besides, for any op_i in AOPL$_a$, ConcD_op_i, is the set collecting the definitions concurrent to op_i, and ConcK_op_i collects kills correspondingly. These sets are defined as following:

Definition 25 (Records of Relationships between Artifact Operations)
For an LRTS workflow w and $\forall a \in A_w$,
$\forall op_i \in$ AOPL$_a$,
ConcD_op_i =
{op | $op \in$ AOPL$_a$, $op.type$ == Def, Concurrent($op_i.p$, $op.p$) == true}
ConcK_op_i =
{op | $op \in$ AOPL$_a$, $op.type$ == Kill, Concurrent($op_i.p$, $op.p$) == true}

With the records, Algorithm 10 is constructed as following:

Algorithm 10 Identifying Concurrent Operations - *ICO*
Input: an artifact a
Pre-Condition: $a \in A_w$, and w is manipulated by Algorithm 9
ICO {
01: for(i = 1 to |AOPL$_a$|) {
02: if($op_i.p.abstack \neq$ « » {
03: $j = i + 1$;
04: while($j \leq$ |AOPL$_a$|) {
05: if (Concurrent($op_i.p$, $op_j.p$)){
06: if($op_i.type$ == Def) add op_i to ConcD_op_j;
07: else if($op_i.type$ == Kill) add op_i to ConcK_op_j;
08: if($op_j.type$ == Def) add op_j to ConcD_op_i;
09: else if($op_j.type$ == Kill) add op_j to ConcK_op_i;
10: }
11: j++;
12: }
13: }
14: }
}

Because AOPL$_a$ is sorted by LETs in Algorithm 9, Algorithm 10 checks each operation in AOPL$_a$ in order. For any $op_i \in$ AOPL$_a$, Algorithm 10 first checks if it resides in some parallel or decision structure(s) at line 2. If not, op_i can not be concurrent or exclusive to any other

operations. From line 3 to line 14, the algorithm checks the operations which are succeeding to op_i in $AOPL_a$ in order. If the operation under checking is concurrent to op_i, the records for both operations are updated.

For an artifact operation, the operations directly before it generate/carry its input artifact state, and might make it an artifact anomaly. For example, when a kill directly before a usage, i.e. no other operations between them, the usage is an undefined usage. We define the relationship *directly before* between artifact operations on the basis of Definition 24 as following:

Definition 26 (Directly Before)
For an LRTS workflow *w* and $\forall a \in A_w$,
$\forall op, op' \in AOPL_a$, *op* is *directly before* *op'* if and only if *op* is before *op'*, and $\exists$ no *op"* $\in AOPL_a$ that *op"* is after *op* and before *op'*.
$\forall op \in AOPL_a$, $DB4_{op} = \{ op' \mid op' \in AOPL_a$ and *op'* is directly before *op*$\}$

According to Definition 10, Definition 24, and Lemma 5, for any two artifact operations effective on artifact *a*, *op* and *op'*, if *op'* is before *op*, *op'.let* < *op.let*. Therefore, the operations directly before *op* can be identified by analyzing the sub-list of $AOPL_a$ where the operations in the sub-list are all with smaller index in $AOPL_a$ than *op*. The sub-list is defined as following:

Definition 27 (The List of Operations with Smaller LET than Operation *op*)
$\forall op \in AOPL_a$,
$OPL_{op} = \{op' \mid op' \in AOPL_a$, and *op'.let* < *op.let*$\}$
Similar to $AOPL_a$, OPL_{op} is sorted and indexed with LETs

The algorithm collecting the operations directly before one another operation is described as following.

Algorithm 11 Collecting Directly Before Operations – *CDBO*
Input: an artifact operation *op*,
Pre-Condition: $AOPL_a$ has been produced by Algorithm 9, $op \in AOPL_a$
Operation Set CDBO {
01: $DB4_{op} = \emptyset$;

```
02: for( i = |OPL_op| to 1 ) {
03:     if ( ( Concurrent(op, op_i) || Exclusive(op, op_i) ) == false ) {
04:         if ( ResultSet == Ø ) add op_i to ResultSet;
05:         else if(  ∃ no op'∈  DB4_op that Before(op_i, op') == true )
06:             add op_i to ResultSet;
07:     }
08: }
09: return DB4_op;
}
```

For the input artifact operation *op*, Algorithm 11 calculates $DB4_{op}$ from its corresponding artifact operation list. Algorithm 11 checks the operations in OPL_{op} with reverse order. According to Lemma 1 and Definition 10, the processes in an LRTS workflow are either before, after, concurrent or exclusive to each other, and so are the operations. The operations concurrent or exclusive to *op* are excluded at line 3. With Lemma 4, the first operation found passing the checking at line 3 is directly before *op*. According to Definition 26, if *op'* and *op"* are both directly before *op*, *op'* can not be before *op"* and vice versa. Therefore, the algorithm continues gathering the other directly before operations with the statement at line 6 after the first one is found.

To show Algorithm 11 is correct, the following lemma is depicted and proved.

> Lemma 8
> For any artifact operation *op* and *op'*, *op'* is directly before *op* if and only if *op'*∈ CDBO(*op*)

Proof:

We first show the if-part is correct. B.W.O.C, it is assumed that *op'*∈ CDBO(*op*), but is not directly before *op*. According to the algorithm, the result set of Algorithm 11 is a sub-set of OPL_{op}. Therefore, *op'*∈ OPL_{op}, *op'.let* < *op.let*, and *op'* can not be after *op* on the basis of Lemma 7. Besides, *op'* must pass the checking at line 3, *op'* is not concurrent or exclusive to *op*. Based on Lemma 4 and Definition 24, *op'* is before *op*. Since *op'* is not directly before *op*, according to Definition 26, there must exist another operation *op"* which is after *op'* and before *op*. Because *op'*∈ CDBO(*op*), *op'* must be collected in the result set at line 4 or line 6 in Algorithm 11. Since *op"* is after *op'*, *op'.let* < *op".let*. *op"* has a larger index than *op'* in OPL_{op}, and is touched by Algorithm 11 earlier than *op'* does. Therefore, either *op"* is directly

before *op* or not, *op'* can not be collected in the result set at line 4 or line 6. *op'*$\notin$ CDBO(*op*) which is a contradiction, and the if-part of Lemma 8 is shown correct.

As for the only-if-part, B.W.O.C, we assume that *op'* is directly before *op* and *op'*$\notin$ CDBO(*op*). The assumption indicates that *op'* is before *op*. According to Lemma 7, *op'.let* < *op.let*, and thus *op'* belongs to OPL_{op}. *op'* also passes the checking at line 3 based on Lemma 4 and Definition 24. If the result set is empty when Algorithm 11 touches *op'*, *op'* is inserted into the result set at line 4 because *op'* is before *op*. Otherwise, *op'* is added into the result set at line 6 because *op'* is directly before *op* and there exist no other operations after *op'* in OPL_{op}. Therefore, *op'*$\in$ CDBO(*op*) which is a contradiction, and the only-if-part of Lemma 8 is shown correct. With the proofs of the both direction, Lemma 8 is proved. □

For an artifact operation *op*, multiple operations directly before it might exist. According to Definition 26, the operations are not before or after each other. On the basis of Lemma 1 and Definition 10, the operations are mutually concurrent or exclusive, and are possibly organized as the following cases:

(1) All the operations are concurrent to each other.

(2) All the operations are exclusive to each other.

(3) The operations can be divided into several distinct groups where the operations in the same group are concurrent to each other, and the operations belonging to different groups are all mutually exclusive.

(4) The operations can be organized into several varied groups where the operations in the same group are concurrent to each other, and the operations belonging to different groups are either identical or mutually exclusive.

The operations in case (1) compose a racing operation. In case (2), each operation is considered separately during analysis because only one of the operations is executed during run-time. Case (3) and (4) happen when the operations are made by processes reside in nestedly organized decision and parallel structures. Since only one of the branches in a decision structure is taken during run-time, the operations reside in different branches of a decision structure are

separately analyzed with the operations concurrent to them. Figure 14 illustrates two partial LRTS workflow schemas as the examples of case (3) and (4).

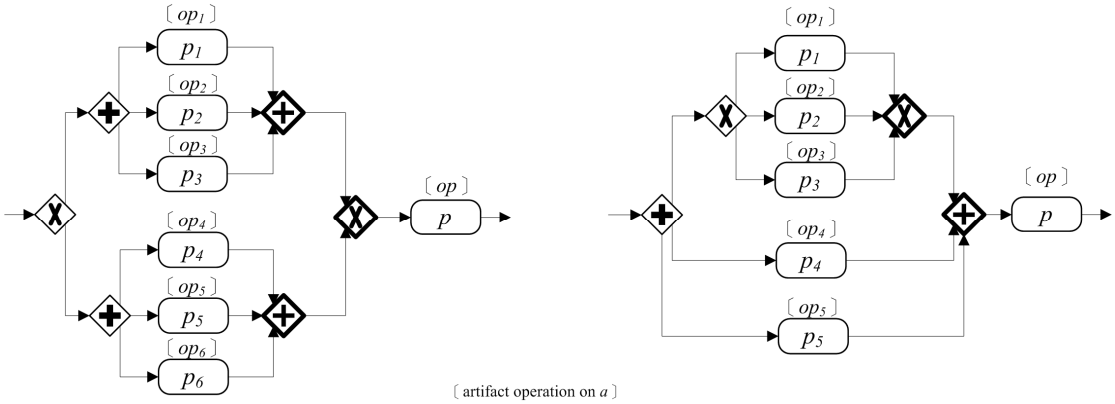


Figure 14 Examples for Nestedly Organized Decision and Parallel Structures

In Figure 14, we assume that the EAIs of the activity processes with footnotes are all overlapped, and all the operations made by them are thus directly before *op*. Figure 14(a) illustrates an example of case (3) mentioned above. In Figure 14(a), the operations directly before *op* can be divided into two distinct groups {op_1, op_2, op_3} and {op_4, op_5, op_6}. The operations are concurrent to the ones within the same group and are exclusive to the ones belonging to different groups. Figure 14(b) illustrates an example of the case (4). op_1, op_2, and op_3 are mutually exclusive and should be separately considered when analysis. However, each of them is concurrent to op_4 and op_5. Therefore, the operations are organized with three groups, {op_1, op_4, op_5}, {op_2, op_4, op_5}, and {op_3, op_4, op_5}. The operations in the same group are concurrent to each other, and the exclusive operations are distributed among different groups.

Definition 28 (Set of Operation Sets derived from $DB4_{op}$)
$\forall\, op \in AOPL_a$,
$DB4OPS_{op}$ = {OPS | $OPS \subseteq DB4_{op}$, $\forall\, op', op'' \in OPS$, Concurrent($op'$, op'') == true, and $\forall\, op^3 \in DB4_{op} \backslash OPS$, $\exists\, op^4 \in OPS$ that Exclusive(op^3, op^4) == true }

Each of the groups, the operation sets, in which all the operations are mutually concurrent represents an execution case during run-time. With Definition 26, the set of the operation sets

derived from $DB4_{op}$ is defined as above.

However, to retrieve all such operation sets from $DB4_{op}$ is equivalent to solve the well-known NP-hard problem "Maximal Clique Enumeration Problem [52]." Although many studies and efficient algorithms like [53] and [54] has been developed for this problem, to discuss the solution for maximal clique enumeration problem is beyond the scope of this dissertation. To illustrate our methodology, we describe a polynomial algorithm to manufacture $DB4OPS_{op}$ satisfying the cases (1), (2), (3) completely and case (4) partially from $DB4_{op}$. The algorithm is described as following.

Algorithm 12 Collecting Directly Before Operation Sets - *CDBOPS*
Input: an operation *op*,
Pre-Condition: $DB4_{op}$ has been calculated by Algorithm 11
Set of Operation Sets CDBOPS {
01: $DB4OPS_{op}$ = Ø;
02: duplicate $DB4_{op}$ to BaseSet;
03: while(BaseSet ≠ Ø) {
04: CurrentOPS = Ø;
05: choose and remove arbitrary operation *op'* from BaseSet;
06: duplicate $DB4_{op}$ \{*op'*} to CountSet;
07: add *op'* to CurrentOPS;
08: while(CountSet ≠ Ø) {
09: choose and remove arbitrary *op"* from BaseSet;
10: if($\forall op^3 \in$ CurrentOPS, Concurrent(*op"*, op^3) == true) {
11: add *op"* to CurrentOPS;
12: remove *op"* from BaseSet;
13: }
14: }
15: add CurrentOPS to ResultSet;
16: }
17: return $DB4OPS_{op}$;
}

First, the algorithm duplicates $DB4_{op}$ to BaseSet at line 2. The codes from line 3 to 16 form a loop. In the loop, an operation *op'* is arbitrarily chosen from BaseSet, and all the operations

concurrent to *op'* and each other are gathered and put into CurrentOPS. CurrentOPS is added to the result set as one of the operation sets found by the algorithm at the end of the loop. The operations in CurrentOPS are removed from BaseSet, and the next loop starts if there is still operation remaining in BaseSet. Because any operations in $DB4_{op}$ are mutually concurrent or exclusive, the operation chosen in the next loop is exclusive to at least one of the operations gathered in this loop. Besides, the algorithm starts collecting an operation sets from different operations every loop, and thus none of the operation sets collected in Algorithm 12 are identical. After each operation in BaseSet is distributed into some operation set, the algorithm returns the calculated $DB4OPS_{op}$ at line 17.

To depict the correctness and the effectiveness of Algorithm 12, we show that the following lemmas hold.

> Lemma 9
> The result set returned by Algorithm 12 follows Definition 28.

Proof:

Let OPS be one of the operation set collected in CDBOPS(*op*). According to the pre-condition of Algorithm 12, $DB4_{op}$ has been calculated by Algorithm 11, and according to Lemma 1, Definition 10, and Definition 26, the operations in $DB4_{op}$ are either mutually concurrent or exclusive. From line 7 and line 11 of Algorithm 12, we know that all the operations collected in OPS are mutually concurrent. The operations gathered in OPS are removed from BaseSet at line 12. Therefore, for any operation remaining in BaseSet, there exists at least one operation exclusive to it in OPS. Because BaseSet is duplicated from DB4 at line 2, OPS follows Definition 28, and Lemma 9 is thus shown correct. □

Before Algorithm 12 is introduced, four possible cases of the set of operation sets derived from $DB4_{op}$ are described, and we claim the capability of Algorithm 12 based on the cases. Here, we show the claim holds with the following lemma.

> Lemma 10
> Algorithm 12 is able to find the operation sets for case (1), (2), and (3) completely, and for case (4) partially.

Proof:

The cases are separately discussed as following:

(1) All the operations in $DB4_{op}$ are concurrent to each other.

In this case, the algorithm collects all the operations in the first loop of the algorithm. Only one operation set is included in the result set of Algorithm 12.

(2) All the operations in $DB4_{op}$ are exclusive to each other.

In this case, an operation is collected in an individual operation set in each loop. Let the size of $DB4_{op}$ be N. As the result, N particular operation sets are collected in $DB4OPS_{op}$, and the union of the sets is identical to $DB4_{op}$.

(3) The operations in $DB4_{op}$ can be divided into several distinct groups where the operations in the same group are concurrent to each other, and the operations belonging to different groups are all mutually exclusive.

In this case, $DB4_{op}$ can be divided into several distinct operation sets following Definition 28. However, the operations included in different sets are all mutually exclusive. According to Lemma 9, the operation sets collected by Algorithm 12 follow Definition 28. On the basis of the algorithm, each operation in $DB4_{op}$ is collected into some operation set in $DB4OPS_{op}$. Therefore, all the operation sets in this case can be found by Algorithm 12.

(4) The operations in $DB4_{op}$ can be organized into several varied groups where the operations in the same group are concurrent to each other, and the operations belonging to different groups are either identical or mutually exclusive.

In Algorithm 12, at least one operation is removed from BaseSet in the loop starting from line 3, and therefore the algorithm derives at most N operation sets from $DB4_{op}$. For case (4), the number of operation sets identified by Algorithm 12 is less than N, but the number of operation sets in this case might exceed N. The operation sets in case (4) follow Definition 28, and so is Algorithm 12. Since the number of operation sets in case (4) might exceed the maximal capability of Algorithm 12. Obviously, Algorithm 12 identifies the operation sets for case (4) only partially. □

4.2.3 Detecting Blank Branch

Besides the cases described above, analysis of blank branches, i.e. the branches in a decision structure where no process residing in the branch has operations effective on the same artifact, is still ignored. Figure 15 illustrates parts of an LRTS workflow that the definitions made by v_1 and v_2 are directly before the usage made by v_4. The definitions should be considered separately during analysis because they are exclusively executed during run-time.

However, if the third branch is taken during execution, the usage made by v_4 is undefined because the definition of *a* is killed by v_0, and no further definition is made by activity processes on the third branch. The third branch is a blank branch which generates a blind spot in our methodology.

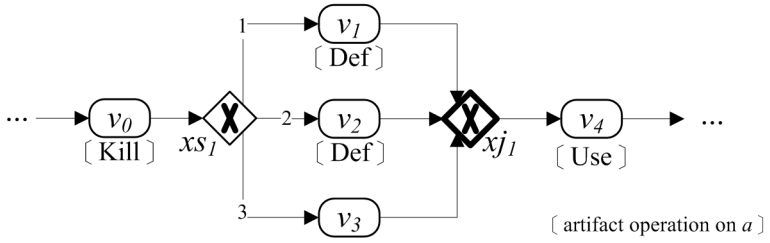


Figure 15 An Example of a Blank Branch

For any operation *op*, to eliminate the effect brought by blank branches when calculating its input states, all the operations reside in the decision structure with blank branches should be removed from OPL_{op}, and $DB4_{op}$ can then be recalculated for analysis. Algorithm 13 detects blank branches from the directly before operations of the input operation.

Algorithm 13 Detecting Blank Branch - *DBB*
Input: an operation *op*,
Pre-Condition: $DB4_{op}$ has been calculated by Algorithm 11
Branch Set DBB {
01: XSSet = Ø;
02: AllBranch = Ø;
03: OpBranch = Ø;
04: ∀ *op'*∈ $DB4_{op}$ {
05: ∀ *si*∈ (*op'.p.abstack**op.p.abstack*) where *si.p.type* == XS {
06: if(*si.p*∉ XSSet) {
07: ∀ out-flow of *si.p*, *f*, add (*si.p*, BM(*f*)) to AllBranch;
08: add *si.p* to XSSet;
09: }
10: add *si* to OpBranch;
11: }
12: }
13: BlankBranch = AllBranch\OpBranch
14: return BlankBranch;
}

The temporary sets used in the algorithm are initialized from line 1 to 3. At line 5, the algorithm checks if there exists a decision structure that (1) the structure is converged before *op* and (2) an operation in $DB4_{op}$ resides in the structure. At line 7, Algorithm 13 records the structural items representing all the branches of the decision structure in AllBranch. For any operation in $DB4_{op}$, if the operation resides in some decision structure, the algorithm collects the branch of the structure where the operation resides in OpBranch at line 10. At line 13, the blank branches are derived from the difference between AllBranch and OpBranch as BlankBranch. BlankBranch is then returned as the result set for further analysis.

In Algorithm 13, all the branches of the decision structures with the operations directly before *op* are collected in AllBranch, and the individual branches resided by the operations are recorded in OpBranch. If all the branches collected in AllBranch are resided by the operations directly before *op*, no blank branch exists. Otherwise, the differences between AllBranch and OpBranch are the branches without operations effective on *op.a*, i.e. the blank branches.

4.2.4 Identifying Artifact Anomalies in an LRTS workflow

In this section, the algorithm integrating all the information gathered above to identify the artifact anomalies in an LRTS workflow is introduced. An operation transits the state of artifacts as Figure 13 illustrates, and artifact anomalies are produced when operations effective on artifacts with inappropriate state. For an artifact operation *op*, the artifact state produced by *op*, i.e. *op*'s output state, is recorded in *op*.OutState, and its input state is calculated from the output states of the operation(s) directly before it. Since only one of the mutually exclusive input operations is executed during run-time, the input states from these operations are discussed separately, and an operation might thus produce multiple output states accordingly. States of artifacts are recorded as state items, and are modeled as following.

Definition 29 (Records of Artifact States)
∀ state item *stItem*, *stItem* = (*st*, SRC),
stItem.*st* represents the output artifact state of *op*.*a*, and
stItem.SRC indicates the source operations producing the state.

With the definition above, Algorithm 14 describes the methodology to calculate the input state for each operation.

Algorithm 14 Gathering Input States of an Operation - *GIS*
Input: an LRTS workflow *w*,
an operation *op*
Pre-Condition: $DB4OPS_{op}$ has been calculated by Algorithm 12
Set of State Items GIS {
01: InStates = Ø;
02: if($DB4OPS_{op}$ == Ø)
03: add (UD, {*w*.*s*}}) to InStates;
04: else ∀ OPS∈ $DB4OPS_{op}$ {
05: if(OPS is an RDS/RDK)
06: add (AB, {*op'* | *op'*∈ OPS, *op*.*type*∈ {Def, Kill}}) to InStates;
07: else if (OPS is an RDU)
08: add (DR, { *op'* | *op'*∈ OPS, *op*.*type* == Def}) to InStates;
09: else if (OPS is an RKS/RKU)
10: add (UD, { *op'* | *op'*∈ OPS, *op*.*type* == Kill}) to InStates;
11: else if (OPS is an RUS) {
12: if (∀ *op'*∈ OPS, ∃ *si*∈ *op'*.OutState that *si*.*st* == UD) {
13: UDSRC = Ø;
14: ∀ *op'*∈ OPS and *si*∈ *op'*.OutState,
15: if(*si*.*st* == UD) UDSRC = UDSRC∪ *si*.SRC;
16: add(UD, UDSRC) to InStates;
17: }
18: if(∃ *op'*∈ OPS and *si*∈ *op'*.OutState that *si*.*st*∈ (AB, DR)) {
19: ABSRC = Ø;
20: DRSRC = Ø;
21: ∀ *op'*∈ OPS and *si*∈ *op'*.OutState {
22: if(*si*.*st* == AB) ABSRC = ABSRC∪ *si*.SRC;
23: else if(*si*.*st* == DR) DRSRC = DRSRC∪ *si*.SRC;
24: }
25: if (ABSRC ≠ Ø) add(AB, ABSRC) to InStates;

```
26:            if ( DRSRC ≠ Ø) add( DR, DRSRC ) to InStates;
27:        }
28:    } else  ∀ op'∈ OPS, InStates = InStates ∪ op'.outStates;
29: return InStates;
}
```

The algorithm shows how to collect the input state of operation *op* from $DB4OPS_{op}$. An empty $DB4OPS_{op}$ indicates that no operation is operated before *op*. In this dissertation, we assume that all the artifacts are initialized with state UD, and the state item (UD, {*w*.*s*}) is inserted to the result set in this circumstance. If $DB4OPS_{op}$ is not an empty set, the algorithm calculates the input state of *op* from each operation set in $DB4OPS_{op}$. An operation set containing multiple operations composes a racing operation, and the algorithm gives the input state of *op* generated from an RDS, RDK, RDU, RKS, and RKU from line 5 to 10 based on the description in section 4.1.1. For an RUS, if all the usages involved in the RUS propagate state UD in their output states, the artifact might be undefined after the RUS, and state UD is included in *op*'s input states accordingly. On the other hand, if there exists a usage involving in the RUS propagating state AB for the target artifact, the target artifact might be ambiguous in definition before *op* is operated. Similarly, if the target artifact is defined in one of the usages involved in the RUS, DR is recorded as one of the input states of *op*. The method to calculate the artifact states generated from an RUS is described from line 12 to 26 in the algorithm. Finally, if the operation set contains only one single operation. The input state of *op* is simply equivalent to the output state of the operation, and is handled at line 28. The input states of *op* are identified for each operation set collected by Algorithm 12. The completeness of the input states gathered by Algorithm 14 is restricted by the capability of Algorithm 12.

According to the type of an operation and its corresponding input state, whether an artifact anomaly is generated from the operation can be detected. The artifact anomalies are recorded in Artifact Anomaly Table (AAT) modeled as following:

Definition 30 (Artifact Anomaly Table)
Let AAT_w be the artifact anomaly table for an LRTS workflow *w*
$\forall$ *aar*$\in$ AAT_w, *aar* = (*op*, *type*, SRC),
aar.*op* indicates the abnormal artifact operation,
aar.*type*$\in$ {Useless Definition, Null Kill, Undefined Usage, Ambiguous Usage}
indicates the anomaly type, and
aar.*SRC* represents the set of operations leading to the anomaly.

For each record in AAT_w, the source operations producing the anomaly are recorded. For example, a usage of an artifact is undefined because a kill removes the definition of the artifact before it. The kill is recorded in the artifact anomaly record to provide information for fixing of the anomaly. The following algorithm illustrates detection of artifact anomalies and calculation of the output states for operations with different types.

Algorithm 15 Identifying Artifact Anomalies for No Operations - *IAAN*
Input: an LRTS workflow *w*,
an artifact operation *op*, and
a set of state items InState
Pre-Condition: *op*.*type* == Nop
IAAN {
01: $\forall$ *stItem*$\in$ InState,
02: if(*stItem*.*state* == DN)
03: add(*op'*| *op'*$\in$ *stItem*.SRC, Useless Definition, {*op*}) to AAT_w;
04: *op*.OutState = InState;
}

For an artifact *a*, the no operation made by the end process is recorded in $AOPL_a$ to detect if any useless definition exists at the end of the LRTS workflow. Since only a definition transits an artifact to state DN, the algorithm records the operations generating DN state directly before the end of the LRTS workflow as useless definitions.

Algorithm 16 Identifying Artifact Anomalies for Definitions - *IAAD*
Input: an LRTS workflow *w*,
an artifact operation *op*, and
a set of state items InState
Pre-Condition: *op*.*type* == Def

```
IAAD {
01: ∀ stItem∈ InState,
02:     if( stItem.state == DN )
03:         add(op'| op'∈ stItem.SRC, Useless Definition, {op} ) to AAT_w;
04: op.OutState = { ( DN, {op} ) };
}
```

Algorithm 16 identifies the artifact anomalies generated from a definition, and calculate its output state. For an artifact *a*, a definition which is not referenced by any usages before being defined again is a useless definition. Finally, a definition transits *a* to state DN, and the output state generated by the definition is recorded accordingly.

```
Algorithm 17 Identifying Artifact Anomalies for Kills - IAAK
Input: an LRTS workflow w,
       an artifact operation op, and
       a set of state items InState
Pre-Condition: op.type == Kill
IAAK {
01: ∀ stItem∈ InState,
02:     if( stItem.state == DN )
03:         add( op'| op'∈ stItem.SRC, Useless Definition, {op} ) to AAT_w;
04:     else if( stItem.state == UD ) add( op, Null Kill, stItem.SRC ) to AAT_w;
05: op_i.OutState = { ( UD, {op_i} ) };
}
```

Algorithm 17 identifies the artifact anomalies generated from a kill, and calculates its output state. A definition which is killed before being referenced is also useless, and the anomaly is detected at line 2 and 3. Besides, if an artifact remains undefined before a kill, the kill is redundant, and a Null Kill is raised accordingly. A kill transits an artifact to state UD, and the output state generated from the kill is recorded at line 5.

```
Algorithm 18 Identifying Artifact Anomalies for Usages - IAAU
Input: an LRTS workflow w,
       an artifact operation op, and
       a set of state items InState
Pre-Condition: op.type = Use
```

```
IAAU {
01: ∀ stItem∈ InState {
02:     if( stItem.state == AB )
03:         add( op, Ambiguous Usage, stItem.SRC∪
04:                 ConcD_op∪ConcK_op ) to AAT_w;
05:     else if( stItem.state == UD ) {
06:         if(ConcD_op ≠ Ø )
07:             add (op, Ambiguous Usage, stItem.SRC∪ConcD_op) to AAT_w;
08:         else add(op, Undefined Usage, stItem.SRC) to AAT_w;
09:     }
10:     else if( stItem.state∈ {DR, DN} )
11:         if( ConcD_op∪ConcK_op ≠ Ø )
12:             add(op, Ambiguous Usage, stItem.SRC∪
13:                 ConcD_op∪ConcK_op) to AAT_w;
14:     if( stItem.state == DN ) add (DR, stItem.SRC) to op_i.OutState;
15:     else add stItem to op_i.OutState;
16: }
}
```

Algorithm 18 identifies whether a usage is abnormal, and calculates its output state. The input state AB indicates that the artifact is ambiguous in definition when the operation being operated, and makes the usage an ambiguous usage. If the input state of the usage is UD, the algorithm checks if there is any definition concurrent to the usage from at line 6. If no concurrent definition exists, the usage is undefined. Otherwise, the usage is ambiguous because it may reference an undefined artifact or the value defined by the concurrent definition(s). If the input state of the usage is DN or DR, the concurrent definitions or kills which cause ambiguity to the usage are checked at line 11, and an Ambiguous Usage is raised if any ambiguity exists. The usage transits a DN artifact to state DR or simply propagates the input states to the following operations otherwise.

The expressions adopted in Algorithm 15 to Algorithm 18 are stated based on the description in section 4.1. With all the definitions and algorithms described in this chapter, the methodology detecting artifact anomalies in a TS workflow is introduced as following.

```
Algorithm 19 Identifying Artifact Anomalies - IAA
Input: an LRTS workflow w
IAA {
01: IG( w );
02: ∀ a∈ A_w {
03:     ICO( a );
04:     for( i = 1 to |AOPL_a| ) {
05:         while( true ) {
06:             CDBO( op_i );
07:             CDBOPS( op_i );
08:             InState = GIS( w, op_i );
09:             if( op_i.type == Nop ) IAAN( op_i, InState, w );
10:             else if ( op_i.type == Def ) IAAD( op_i, InState, w );
11:             else if ( op_i.type == Kill ) IAAK( op_i, InState, w );
12:             else if ( op_i.type == Use ) IAAU( op_i, InState, w );
13:             BlankBranch = DBB( op_i );
14:             if( BlankBranch == Ø ) break
15:             else
16:                 ∀ op∈ OPL_opi,
17:                     if( ∃ si∈ BlankBranch, and si'∈ op.p.abstack, where si.sp == si'.sp )
18:                         remove op from OPL_opi;
19:         }
20:     }
21: }
}
```

At line 1, the algorithm first invokes Algorithm 9 to collect structural and temporal information like EAIs, ABStacks, and artifact operation lists for the input LRTS workflow. For each artifact *a*, Algorithm 19 then identifies the concurrency between artifact operations with Algorithm 10 at line 3, and starts analysis of the each operation in $AOPL_a$ in order from line 4. Algorithm 11 is invoked at line 6 to collect the operations directly before op_i, and the operation sets directly before op_i is manufactured by Algorithm 12 from the previous result at line 7. At line 8, Algorithm 14 gathers the input state of op_i, and invokes corresponding algorithms from line 9 to 12 to detect artifact anomalies and calculate the output state of op_i. At line 13, Algorithm 13 is invoked to detect if there is any blank branch before op_i. If not, the anomaly

detection work for op_i is accomplished. Otherwise, all the operations residing in the decision structure with blank branches are removed from OPL_{opi}, and Algorithm 19 repeats analysis of artifact anomalies for op_i until all the blank branches considered. The completeness of the artifact anomalies detected in our methodology is decided by the completeness of the operation sets identified by Algorithm 12. Developing an algorithm able to collecting more operation sets is helpful in enhancing our methodology, and is left as a future work of this study.

4.3 Case Study

In this section, a case study is made to illustrate the feasibility of our methodology.

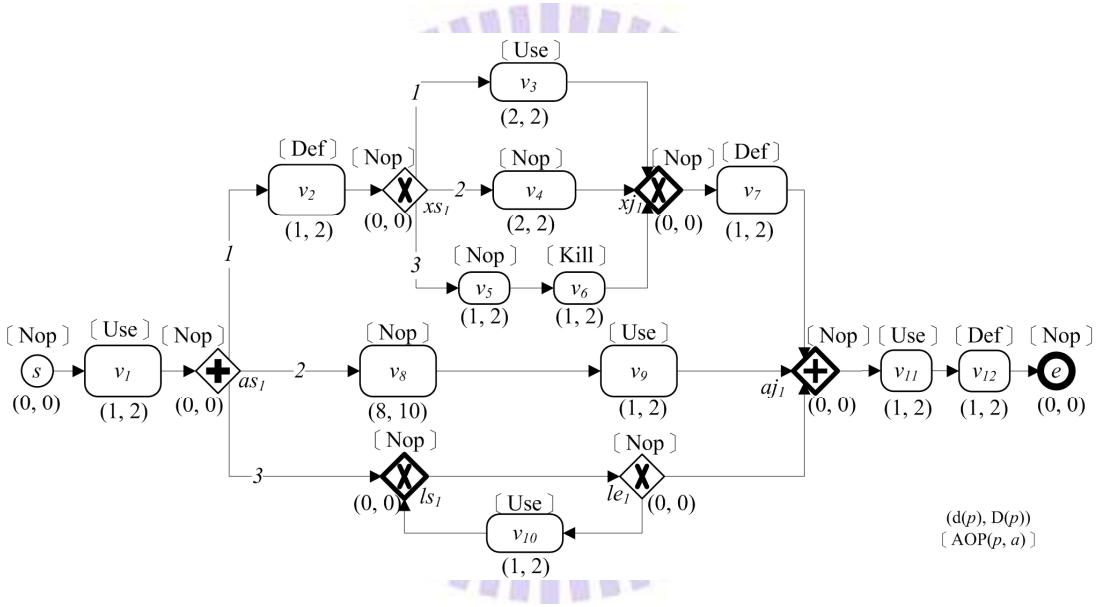


Figure 16 The Sample TS Workflow for the Case Study in Chapter 4

Figure 16 shows the sample TS workflow for our case study. The processes, flows, working durations, and the artifact operations made on artifact *a* are illustrated in the sample. To analyze the sample TS workflow with our methodology, the structured loops in the TS workflow should first be reduced. After loop reduction, the LRTS workflow generated from the sample TS workflow are illustrated as Figure 17. Then, Algorithm 9 is invoked to gather the temporal and structural information such as the EAI and the ABStack for each process, and the artifact operation list for each artifact.

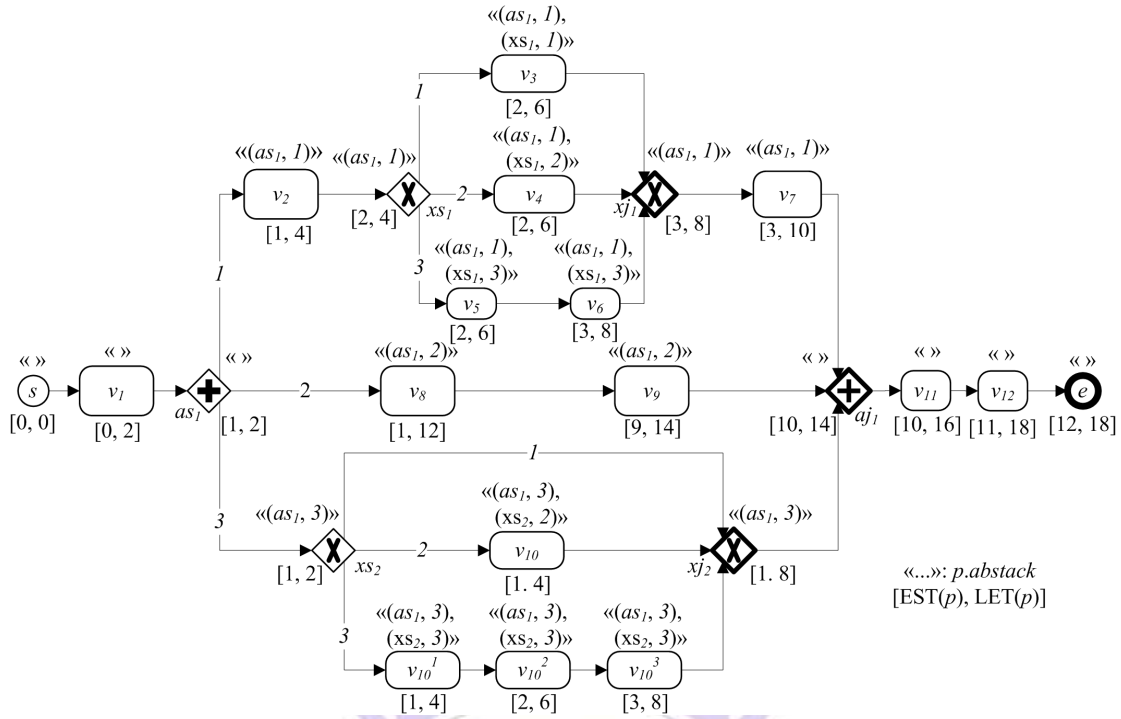


Figure 17 The Sample LRTS Workflow Derived from Figure 16 with Decoration of EAIs and ABStacks

Table 3 illustrates the artifact operation list and the concurrent operations for artifact *a* generated by Algorithm 9 and Algorithm 10.

Table 3 Artifact Operation List for *a*, and the Corresponding Concurrent Operations

op_i	$AOPL_a$	ConcD	ConcK
op_1	(v_1, *a*, 0, 2, Use)	Ø	Ø
op_2	(v_2, *a*, 1, 4, Def)	Ø	Ø
op_3	(v_{10}, *a*, 1, 4, Use)	{op_2}	{op_7}
op_4	(v_{10}^1, *a*, 1, 4, Use)	{op_2}	{op_7}
op_5	(v_3, *a*, 2, 6, Use)	Ø	Ø
op_6	(v_{10}^2, *a*, 2, 6, Use)	{op_2. op_9}	{op_7}
op_7	(v_6, *a*, 3, 8, Kill)	Ø	Ø
op_8	(v_{10}^3, *a*, 3, 8, Use)	{op_2. op_9}	{op_7}
op_9	(v_7, *a*, 3, 10, Def)	Ø	Ø
op_{10}	(v_9, *a*, 9, 14, Use)	{op_9}	Ø
op_{11}	(v_{11}, *a*, 10, 16, Use)	Ø	Ø
op_{12}	(v_{12}, *a*, 11, 18, Def)	Ø	Ø
op_{13}	(*e*, *a*, 12, 18, Nop)	Ø	Ø

To be brief, we do not show all the details of detecting artifact anomalies in this case study,

and focus on two representative examples, op_9 and op_{10}. Therefore, we assume that the operations before op_9 are calculated already, and Table 4 shows the output state of the operations with LETs smaller then op_9's.

Table 4 The Output State of the Operations before op_9 is Calculated

op_i	OutState
op_1	{ (UD, {s}) }
op_2	{ (DN, {op_2}) }
op_3	{ (UD, {s}) }
op_4	{ (UD, {s}) }
op_5	{ (DR, {op_2}) }
op_6	{ (UD, {s}) }
op_7	{ (UD, {op_7}) }
op_8	{ (AB, {s, op_2, op_9}) }

op_1 is an undefined usage because it is operated before any activity process gives definition to artifact a. op_3, op_4, op_6, and op_7 are ambiguous usages because there exist definition concurrent to them. Before op_9 is calculated, the artifact anomaly table, AAT_w, records the following anomalies:

AAT_w = { (op_1, Undefined Usage, {s}), (op_3, Ambiguous Usage, {s, op_2}), (op_4, Ambiguous Usage, {s, op_2}), (op_6, Ambiguous Usage, {s, op_2, op_7}), (op_7, Ambiguous Usage, {s, op_2, op_7}) }

For op_9, Algorithm 19 retrieve all the operations with smaller LET from $AOPL_a$ as OPL_{op9}, {op_1, op_2, op_3, op_4, op_5, op_6, op_7, op_8}, and invokes Algorithm 11 to calculate $DB4_{op9}$, {op_5, op_7}. Since all the operations directly before op_9 are mutually exclusive, i.e. the case (2) described in section 4.2, the $DB4OPS_{op9}$ is calculated from Algorithm 12 as { {op_5}, {op_7} }. With $DB4OPS_{op9}$, Algorithm 14 gathers the input states of op_9 as the union of the output states of op_5 and op_7 as { (DR, {op_2}), (UD, {op_7}) }. op_9 is a definition, and Algorithm 16 is invoked for detection of artifact anomalies and generation of its output state. As a result, no artifact anomaly is found and the output state of op_9 is generated as { (DN, {op_9}) }. However, during

the blank branch detection, (xs_1, 2) is found a blank branch, and the operation in the same decision structure should be removed to eliminate the effect of blank branch. op_5 and op_7 is removed from OPL_{op9}. $DB4_{op9}$, $DB4OPS_{op9}$, and the InState of op_9 are recalculated as {op_2}, {{op_2}}, and { (DN, {op_2}) }. After invoking Algorithm 16 once again, an artifact anomaly (op_2, Useless Definition, {op_9}) is raised because the definition made by op_2 is not used before redefinition when the blank branch is taken.

$DB4_{op10}$ is generated as {op_3, op_5, op_7, op_8}, and $DB4OPS_{op10}$ is generated as { {op_3, op_5}, {op_7, op_8} }. Since this case is relatively simple, we can easily identify that the operation sets {op_3, op_7} and {op_5, op_8} is neglected in our methodology. With $DB4OPS_{op10}$, the input states of op_{10} are generated. According to the definition of racing operations introduced in section 4.1.1, {op_3, op_5} is an RUS and {op_7, op_8} is an RKU, and { (DR, {op_2}) } and { (UD, {op_7}) } are generated as op_{10}'s input states correspondingly. Algorithm 18 is invoked to detect artifact anomalies and identify the output state of op_{10}. Two artifact anomalies, (op_{10}, Ambiguous Usage, {op_7, op_9}) and (op_{10}, Ambiguous Usage, {op_2, op_9}), are generated because op_9 makes a definition to *a* concurrently, and generates ambiguity to op_{10}. The output states of op_{10} is { (DR, {op_2}), (UD, {op_7}) }. Then the algorithm removes the blank branches for op_{10}, and finds no further anomalies.

Except for the artifact anomalies listed and described above, (op_{13}, Useless Definition, {e}) are detected and recorded to AAT_w when Algorithm 19 completes its work throughout *w*. The useless definition is detected at the end process of the LRTS workflow because the definition made by op_{13} is not used by any other activity process until the end of *w*.

4.4 Discussion

4.4.1 Related Works in Analysis of Artifact Anomalies

Sun et al. extend the Activity Diagram in UML for modeling data flow in a business process [51]. Three classes of data-flow anomalies, missing data, redundant data, and conflicting data, are defined. With the routing information defined in a workflow specification, a detecting algorithm for the data-flow anomalies is constructed [51]. However, Sun et al. do not build an explicit data model in characterizing the data behaviors, and consider only read and initial write in data operations.

In [26], Sadiq et al. reveal the importance about the validation of workflow data, and introduce seven basic data validation problems, Redundant Data, Lost Data, Missing Data, Mismatched Data, Inconsistent Data, Misdirected Data, and Insufficient Data in workflow models. Redundant Data occur when designers specify an activity to define a data item which is not required by any other succeeding activities. Lost Data occur when designers specify two activities that may be executed in parallel to define the same data item, and one of the definitions is lost when the data item is preempted by the process executed in advance. Missing Data occurs when designers specify an activity to consume a data item which is never defined by any preceding activities. Mismatched Data arise when the structure of data is incompatible between the definition and the usage of the data. Inconsistent data happen when the data required by a workflow are externally updated by other applications during the workflow execution, and the polluted data might cause errors of the workflow. Misdirected Data occur when the direction of the data flow is conflict with the direction of the control flow of the workflow. Insufficient Data happen when the data specified by designers is insufficient to successfully complete an activity.

Destruction of artifacts is not considered in both Sun and Sadiq's studies. In [27] and [28],

Hsu et al. consider the effect of destroying an artifact and re-model the inaccurate artifact manipulation by separating initialization and update as two different artifact operations. In [28], six inaccurate artifact usages, No Producer, No Consumer, Redundant Specification, Contradiction, Parallel Hazard, and Branch Hazard are defined. No Producer is a warning indicating that a data item is operated before it is specified. No Consumer indicates that an artifact is not requested after its definition (initialization). Redundant Specification indicates that an artifact is repeatedly specified in a workflow. Contradiction implies the defect that the state of an artifact is not matched to the pre-condition or post-condition of the activity accessing it. Parallel Hazard occurs due to conflict interleaving of concurrent artifact operations, and is recognized if multiple concurrent activities operate on the same artifact. Branch Hazard occurs when branches in a decision structure contain operations on artifacts have been selected, or when there is inconsistency between the condition testing in the XOR-split process or the branches in the decision structure.

In [29], Wang et al. develop a systematic notation to describe artifact anomalies and simplify the description of artifact anomalies from [28] into three categories, Missing Production, Redundant Write, and Conflict Write. Missing Production occurs when an artifact is consumed before it is produced or after it is destroyed. Redundant Write occurs when an artifact is written by an activity but the artifact is neither required by the succeeding activities nor a member of the process outputs. Conflict Write occurs when parallel processes race their access to the same artifact. According to different structural relationships between activities accessing some artifacts, thirteen abnormal usage patterns are described for the three categories to follow the previous models made by Sadiq et al. [26], Hsu et al. [29], and Sun et al. [51],

4.4.2 Comparison between Our Approach and the Related Works

Table 5 Comparison between Our Approach and the Related Works

<table>
<tr><th>Our Approach</th><th colspan="2">Sun et al. [51]</th><th>Sadiq et al. [26]</th><th>Hsu et. al [28]</th><th colspan="2">Wang et al. [29]</th></tr>
<tr><td rowspan="4">Undefined Usage</td><td rowspan="4">Missing Data</td><td>Absence of Initialization</td><td>Missing Data</td><td rowspan="2">No Producer</td><td rowspan="4">Missing Production</td><td>No Production</td></tr>
<tr><td>Delayed Initialization</td><td rowspan="2">Misdirected Data</td><td>Delayed Production</td></tr>
<tr><td rowspan="2">Improper Routing</td><td rowspan="2">Branch Hazard</td><td>Conditional Production</td></tr>
<tr><td>N/A</td><td>Exclusive Production</td></tr>
<tr><td>Uncertain Availability</td><td>Misdirected Data</td><td>Parallel Hazard</td><td>Uncertain Production</td></tr>
<tr><td rowspan="2">Useless Definition</td><td rowspan="2">Redundant Data</td><td>Contingent Redundancy</td><td rowspan="2">Redundant Data
Mismatched Data</td><td>Branch Hazard</td><td rowspan="2">Redundant Write</td><td>Conditional Consumption after Last Write</td></tr>
<tr><td>Inevitable Redundancy</td><td>No Consumer</td><td>No Consumption after Last Write</td></tr>
<tr><td>Ambiguous Usage</td><td>Conflict Data</td><td>Multiple Initialization</td><td>Lost Data</td><td>Contradiction</td><td>Conflict Write</td><td>Multiple Parallel Production</td></tr>
<tr><td>N/A</td><td>N/A</td><td>N/A</td><td>Insufficient Data
Mismatched Data</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>Null Kill</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>Temporal Consideration</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>Anomaly Source Tracking</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
</table>

Table 5 lists and compares the features between the related works and our approach. Artifact anomalies are appealed with different names in previous studies, but can still be mapped into the three basic categories made in [51]. By comparing the definition of the artifact anomalies defined in our approach and the related works, we conclude that Undefined Usage and Useless Definition are directly mapped into Missing Data and Redundant Data described in [51]. On the other hand, the Conflict Data defined in [51] are anomalies generated when multiple definitions are made in parallel. In our approach, the concurrent definitions are considered being executed with undetermined order, and generate ambiguity in artifacts. They

are not directly considered as an anomaly because (1) an anomaly actually occurs when a usage refers to the ambiguous definitions, and (2) similar anomaly may also occur when there exist kills or definitions concurrent to usages. Therefore, Ambiguous Usage is categorized in this dissertation, and covers Conflict Data discussed in the previous works. Besides, Sadiq et al. additionally define Insufficient Data and Mismatched Data in [26] for conflicts about contents or format between definitions and usages. Since the studies made in [28], [29], [51] and this dissertation do not discuss the contents of artifacts, Insufficient Data and Mismatched Data are ignored in these studies. Finally, although destruction of artifacts is considered in [28] and [29], the redundancy generated by unnecessary destruction is not discussed in these works. In our studies, Null Kill is categorized and detected by our approach to eliminate such redundancies.

Our approach also considers how temporal factors may affect the detection of artifact anomalies. The twisted temporal and structural relationships between activity processes are modeled and analyzed, and the artifact anomalies generated along with them are detected. Besides, when the previous works only focus on detection of artifact anomalies, our approach also helps designers locating the problems hidden in a workflow schema with providing the information about the sources leading to artifact anomalies.

Chapter 5. Incremental Detection of Resource Conflicts in LRTS Workflow

In this chapter, an incremental methodology detecting the resource conflicts generated/eliminated during construction of LRTS workflows along with each edit operation made by designers is described. With the methodology, designers obtain information after each step they made, and may respond to any conflicts immediately. In section 5.1, the resource conflicts in LRTS workflow are first defined, and the edit operations and additional elements necessary for building an LRTS workflow are modeled in section 5.2. The methods for incremental detection of resource conflicts are depicted in section 5.3. Several examples are described in section 5.4 to illustrate the feasibility of our methodology, and the related works are discussed in section 5.5.

5.1 Resource Conflicts in LRTS workflow

Each activity in a workflow needs certain resources to accomplish its business objective. In this dissertation, it is assumed that all the resources required by an LRTS workflow *w* are recorded in the set RES_w, and designers may assign resource in RES_w to activity processes in P_w to show that the resource is necessary to the process. The resource model used in this dissertation is defined as following.

Definition 31 (Resources)
For an LRTS workflow *w*,
$\forall r \in RES_w$ and $p \in P_w$, *p.type* == ACT
Ref: $RES_w \times P_w \Rightarrow$ Boolean
Ref(*r*, *p*) == true indicates that *r* is accessed by *p*

In [37], two processes are defined having *resource conflict* if the following conditions hold:

(1) The two processes have *resource dependency* on a resource, i.e. the two processes access the same resource.

(2) The two processes have *potentially concurrent execution*, i.e. the two processes reside on different branches split from an AND-split process with overlapped EAIs.

In this dissertation, we assume that all the resource conflicts buried in LRTS workflow w would be recorded in the set RCT_w. Besides, to tracking the generation or elimination of resource conflicts, pairs of processes which satisfy the following conditions are also recorded as *potential resource conflicts* in the set $PRCT_w$: (1) the processes are resource dependency, (2) the processes reside in different branches split from an AND-split process and their EAIs are not overlapped, i.e. they are parallel but not concurrent. On the basis of Definition 3, Definition 10, and [37], the resource conflict in an LRTS workflow is defined as following.

Definition 32 (Resource Conflict)
For an LRTS workflow w,
$RCT_w = \{(r, p, q) \mid (\text{Ref}(r, p) \wedge \text{Ref}(r, q)) == \text{true, and Concurrent}(p, q) == \text{true } \}$
$PRCT_w = \{(r, p, q) \mid (\text{Ref}(r, p) \wedge \text{Ref}(r, q)) == \text{true, Concurrent}(p, q) == \text{false, and Parallel}(p, q) == \text{true } \}$

Since Concurrent(p, q) is equivalent to Concurrent(q, p), the resource conflict (r, p, q) is equivalent to (r, q, p). To simplify our discussion, we assume that adding/removing (r, p, q) into/from RCT_w is equivalent to adding/removing (r, q, p) into/from RCT_w. In other words, (r, p, q)$\in RCT_w$ if and only if (r, q, p)$\in RCT_w$. The assumption also holds for $PRCT_w$.

5.2 Edit Operations for LRTS workflow

To trace resource conflicts generated in an LRTS workflow during design-time, the edit operations designers may adopt to develop the LRTS workflow are first addressed. Since an

LRTS workflow is structured [8], the construction of an LRTS workflow follows the constraints described in chapter 2. Therefore, the edit operations are restricted as following:

(1) Only an activity process can be directly inserted/removed into/from an LRTS workflow. Control processes must be inserted/removed into/from an LRTS workflow in pairs.

(2) Designers can only alter the working durations or resource references of activity processes.

(3) The design of an LRTS workflow is started from a basic LRTS workflow, and designers edit the LRTS workflow until all the design works are completed. The definition of basic LRTS workflow is described in Definition 33.

Definition 33 (Basic LRTS workflow)
A basic LRTS workflow $w = (\{s, e\}, \{(s, e)\}, s, e)$
$D(s) = d(s) = D(e) = d(e) = 0$
$EAI(s) = EAI(e) = [0, 0]$

In order to keep the control processes inserted/removed into/from an LRTS workflow in pairs, additional records for control blocks are introduced. A control block is composed of a split process starting a decision/parallel structure and a join process converging the structure. Besides, the record also marks a natural number counter to provide distinct ID for each branch in the corresponding decision/parallel structure. The record for a control block is modeled as following.

Definition 34 (Control Blocks)
For an LRTS workflow w, CB_w records all the control blocks in w.
$\forall cb \in CB_w$,
$cb = (st, end, br\_count)$
$st \in P_w$, $st.type \in \{AS, XS\}$
$end \in P_w$, $end.type == \begin{cases} AJ \text{ if } st.type == AS \\ XJ \text{ if } st.type == XS \end{cases}$
br_count is a natural number indicating the branch mark for the new branch splitting from *sp*.

$\forall sp \in P_w$, $sp.type \in$ {AS, XS}, $\exists cb \in CB_w$ that $sp == cb.st$
$\forall jn \in P_w$, $jn.type \in$ {AJ, XJ}, $\exists cb \in CB_w$ that $jn == cb.end$
$\forall cb \in CB_w$, Reachable($cb.st$, $cb.end$) == true
$\forall cb, cb' \in CB_w$,
(1) $cb.st \neq cb'.st$, and $cb.end \neq cb'.end$, and
(2) Reachable($cb.st$, $cb'.st$) == true if and only if
(Reachable($cb.end$, $cb'.st$) $\oplus$ Reachable($cb'.end$, $cb.end$)) == true.

Starting from a basic LRTS workflow *w*, the edit operations discussed in this dissertation are listed as following:

(1) Inserting activity process *p* into an existent flow *f*

Pre-Condition: $f = (p', p'') \in F_w$, BM(p', p'') == *bm*
Post-Condition: $(p', p'') \notin F_w$, (p', p), $(p, p'') \in F_w$,
$p.type$ = ACT, d(p) == D(p) == 0,
BM((p', p)) == *bm*, BM((p, p'')) == -1
Comments: Designers use this operation to insert an activity process *p* into an existent flow *f* in *w*. *f* is replaced by two new flows, the in-flow of *p* which is connected to the source process of *f* and the out-flow of *p* which is connected to the sink process of *f*. The minimum and maximum working durations of *p* are both assumed to be zero. The branch mark of the in-flow of *p* is given as the replaced one, and the branch mark of the out-flow of *p* is set as Ø because *p* is an activity process.

(2) Inserting a new decision structure quoted by *sp* and *jn* into an existent flow *f*

Pre-Condition: $f = (p', p'') \in F_w$, BM(p', p'') == *bm*
Post-Condition: $(p', p'') \notin F_w$, (p', sp), (sp, jn), $(jn, p'') \in F_w$,
$sp.type$ == XS, $jn.type$ == XJ, $(sp, jn, 1) \in CB_w$
BM((p', sp)) == *bm*, BM((sp, jn)) == 0, BM((jn, p'')) == -1
Comments: Designers use this operation to inset a decision structure quoted by *sp*, an XOR-split process, and *jn*, an XOR-join process, into an existent flow *f* in *w*. *f* is replaced similarly as in operation (1). A control block record is generated with this operation. Both *sp* and *jn* are recorded, and the corresponding courter for the branches in the decision structure is initialized as 1. Besides, the branch mark of (sp, jn), the flow of the first branch split from *sp*, is initialized as 0.

(3) Inserting a new parallel structure quoted by *sp* and *jn* into an existent flow *f*

Pre-Condition: $f = (p', p'') \in F_w$, $BM(p', p'') == bm$
Post-Condition: $(p', p'') \notin F_w$, (p', sp), (sp, jn), $(jn, p'') \in F_w$,
$sp.type == AS$, $jn.type == AJ$, $(sp, jn, 1) \in CB_w$
$BM((p', sp)) == bm$, $BM((sp, jn)) == 0$, $BM((jn, p'')) == -1$
Comments: To insert a new parallel structure into *w* is similar to insert a new decision structure. The only difference between them is that *sp* is an AND-split process, and *jn* is typed AND-join.

(4) Inserting a new branch to a decision/parallel structure

Pre-Condition: $(sp, jn, br\_count) \in CB_w$, $(sp, jn) \notin F_w$
Post-Condition: $(sp, jn) \in F_w$, $BM((sp, jn)) == br\_count++$
Comments: To simplify our discussion, a flow between a pair of split and join processes is allowed being inserted only when no such flow exists in *w*. The flow (*sp*, *jn*) is added to F_w. and its branch mark is set to the current value of the corresponding branch counter. The counter is added by 1 after the insertion.

(5) Adding a resource reference to an activity process

Pre-Condition: $r \in R_w$, $p \in P_w$, $p.type == ACT$, $Ref(r, p) == false$
Post-Condition: $Ref(r, p) == true$
Comments: Designers use this operation to indicate that access of the resource *r* is necessary for activity process *p*.

(6) Removing a resource reference from an activity process

Pre-Condition: $r \in R_w$, $p \in P_w$, $p.type == ACT$, $Ref(r, p) == true$
Post-Condition: $Ref(r, p) == false$
Comments: Designers use this operation to remove the resource reference of resource *r* from activity process *p*.

(7) Setting minimal working duration of an activity process

Pre-Action: $var = in\_value - d(p)$
Pre-Condition: $p \in P_w$, $0 \leq in\_value \leq D(p)$, $var \neq 0$
Post-Condition: $d(p) == in\_value$
Comments: Designers use this operation to designate the minimal working duration of an activity process in *w* to the specific input value, *in_value*. To simplify our discussion, *in_value* must be a non-negative integer is equal or smaller than

the maximal working duration of the target activity process. To facilitate our detection of resource conflicts, the variation of d(p) is recorded as *var* before the operation is invoked.

(8) Setting maximal working duration of an activity process

Pre-Action: $var = in\_value - D(p)$

Pre-Condition: $p \in P_w$, $d(p) \leq in\_value$, $var \neq 0$

Post-Condition: $D(p) == in\_value$

Comments: Designers use this operation to designate the maximal working duration of an activity process in w to the specific input value, *in_value*. *in_value* must be a non-negative integer which is equal to or larger than the maximal working duration of the target activity process. To facilitate our detection of resource conflicts, the variation of D(p) is recorded as *var* before the operation is invoked.

(9) Removing the activity process p from w

Pre-Condition: $(p', p), (p, p'') \in F_w$, $p.type == ACT$, $d(p) == 0$, $D(p) == 0$,
$\forall r \in R_w$, $Ref(r, p) == false$

Post-Condition: $p \notin P_w$, $(p, p'') \in F_w$

Comments: Designers use this operation to remove an activity process from w. To simplify our discussion, it is assumed that before the removal of the activity process, the resource references of the activity process are first removed, and the corresponding minimum and maximal working durations are set to be 0.

(10) Removing the empty branch from w

Pre-Condition: $(sp, jn, br\_count) \in CB_w$, $(sp, jn) \in F_w$, and a path $<sp, p_1, \ldots, p_k, jn>$ exists.

Post-Condition: $(sp, jn) \notin F_w$

Comments: Designers use this operation to remove the empty branch in a control block from w. In order to keep the integrity of w, the removal which disconnects w is forbidden.

(11) Removing the empty control block quoted by sp and jn from w

Pre-Condition: $(sp, jn, br\_count) \in CB_w$, $(p', sp), (sp, jn), (jn, p'') \in F_w$,
$\exists$ no $p \in P_w$ that $(Reachable(sp, p) \wedge Reachable(p, jn)) == true$

Post-Condition: $(p', p'') \in F_w$, $sp, jn \notin P_w$,
$\forall cb \in CB_w$, $cb.st \neq sp$, $cb.end \neq jn$

Comments: Designers use this operation to remove a control block from w. To simplify

our discussion, only the control block containing no processes inside is allowed being removed from *w*.

5.3 An Incremental Algorithm Detecting Resource Conflicts in TS workflow

The methods incrementally detecting resource conflicts along with the operations are introduced in this section. According to Definition 32, resource conflicts might be generated/eliminated after operation (5), (6), (7), and (8), and the methods would be invoked as the post actions of the operations. Besides, among the operations above, the ABStack(s) of the inserted process(es) should be established after operations (1), (2), and (3), and the EAIs among the LRTS workflow under editing are necessary being updated after operation (7) and (8). The calculation of ABStacks and EAIs are also described as the post actions of the edit operations. In this section, the methodology to calculate EAI changes after modification of working durations is first introduced in section 5.3.1. In section 5.3.2 and 5.3.3, the methods detecting generation or elimination of resource conflicts after operation (5), (6), (7), and (8) are separately discussed. In section 5.3.4, the post actions of each edit operations are described.

5.3.1 Updating Estimated Active Interval for Processes after Edit Operation

Changing EAI of a process ripples to its descendent processes. Algorithm 20 works after any working duration modification is made. The EAIs of all the affected processes are updated, and the set containing the processes are returned for further analysis of resource conflicts.

Algorithm 20 Calculate EAI - *CEAI*
Input: an LRTS workflow *w*,
an activity process *ip*
Pre-Condition: $ip \in P_w$, *ip.type* == ACT
Pre-Condition: $\forall p \in P_w$, *p.mark* == false;
Process set CEAI {
01: CP = Ø;
02: Queue *tq*;
03: *tq.enqueue*(*ip*);
04: $\forall (ip, p) \in F_w$, *tq.enqueue*(*p*);

```
05: while(tq is not empty) {
06:     p = tq.dequeue;
07:     if(p.type∈ {AJ, XJ}) && (∃ (p', p)∈ F_w, p'.mark == false) ) continue;
08:     p.mark = true;
09:     oest = EST(p);
10:     olet = LET(p);
11:     if(p.type == AJ) {
12:         EST(p) = MAX( {EST(p') + d(p') | (p', p)∈ F_w} );
13:         LET(p) = MAX( {LET(p') | (p', p)∈ F_w } );
14:     }
15:     else if(p.type == OJ) {
16:         EST(p) = min( {EST(p') + d(p') | (p', p)∈ F_w } );
17:         LET(p) = MAX( {LET(p') | (p', p)∈ F_w } ) ;
18:     else {
19:         EST(p) = EST(p') + d(p') | (p', p)∈ F_w;
20:         LET(q) = LET(p') + D(p) | (p', p)∈ F_w;
21:     }
22:     if( oest≠ EST(p) || olet≠ LET(p) ) {
23:         add p to CP;
24:         ∀ (p, p')∈ F_w, if(p'∉ tq) tq.enqueue(p');
25:     }
26: return CP;
}
```

Algorithm 20 is assumed being invoked after designers make a modification to the minimal/maximal working duration of an activity process *ip*. Similar to Algorithm 9, a traverse queue is adopted to traverse all the processes reachable from *ip*, and the algorithms checks the process in the queue one by one. Let the process currently being checked be *p*. At line 9 and 10, the original value of EAI(*p*) is recorded, and the algorithm updates EAI(*p*) according to its type from line 11 to 21. From line 22 to 25, the algorithm compares the current EAI(*p*) to the original one. If EAI(*p*) is changed after the operation, the algorithm inserts *p* into CP, puts the process(es) succeeding to *p* into the traverse queue, and continues the calculation. The algorithm halts when EAI stops changing at some join process or when the end process is met. CP, a set collecting all the processes with altered EAIs, is returned as the result set for further analysis of resource

conflicts. We show the correctness of the algorithm by proving the following lemma.

> Lemma 11
> After a duration modification has been made on an activity process *ip* in an LRTS workflow *w*, the following statements hold:
> (1) $\forall p \in P_w$, EAI(*p*) changes if and only if $p \in$ CEAI(*ip*, *w*)
> (2) $\forall p \in P_w$, only when EAI(*p*) should be altered, $p \in$ CEAI(*ip*, *w*)

Proof:

For the first statement, any EAI change in *w* is accomplished by the codes from line 11 to line 21 in Algorithm 20 only. Therefore, when any EAI change occurs, the process with altered EAI is found at line 22 and is put into the result set CP at line 23. The process with no EAI change is not put into CP in this algorithm.

For the second statement, on the basis of Definition 9 and the methods illustrated in Figure 9, EAI(*p*) is changed only when (1) D(*ip*) is modified and *p* == *ip*, (2) d(*ip*) is modified and $(ip, p) \in F_w$, or (3) $(p', p) \in F_w$, and EAI(*p'*) is changed. When d(*ip*) or *D*(*ip*) is modified, *ip* and its succeeding process(es) are enqueued into *tq* at line 3 and line 4. Therefore, the EAI changes originated from condition (1) or (2) are calculated by the codes from line 11 to line 21, and thus *p* is put into CP at line 24. For condition (3), if EAI(*p'*) is changed, *p* is enqueued into *tq* at line 24 because $(p', p) \in F_w$. EAI(*p*) would be calculated by the codes from line 11 to line 21, and *p* is put into CP at line 23 if EAI(*p*) is altered. The processes with EAI changes from the three above conditions are all included in CP.

With the proof above, Lemma 11 is shown correct. □

5.3.2 Identifying Generation or Elimination of Resource Conflicts after Adding/Removing a Resource Reference to/from an Activity Process

From section 5.2, the edit operation (5) and (6) change the resource references of an activity processes. The resource dependencies among processes might be generated or eliminated with the operations, and therefore, resource conflicts (or the potential ones) are generated or eliminated accordingly.

Operation (5) adds the reference of a resource r to an activity process p. A resource conflicts is generated if there exists another activity process which also references r and is concurrent to p. Similarly, if there exists another activity process which references r and is parallel but not concurrent to p, a potential resource conflict is produced. As following,

Algorithm 21 detects the generation of (potential) resource conflicts after a new resource reference is added to an activity process, and alerts designers for each new generated resource conflict.

Algorithm 21 Detecting Resource Conflict for New Resource Reference - *DRCNRR*
Input: a resource r, an activity process p, an LRTS workflow w
Pre-Condition: $r \in RES_w$, $p \in P_w$, and Ref(r, p) == true
DRCNRR {
01: $\forall p' \in P_w \backslash \{p\}$ {
02: if(Ref(r, p') == true && Parallel(p, p') == true) {
03: if(EAI(p) $\approx_{TI}$ EAI(p')) {
04: add (r, p, p') to RCT_w;
05: alert(Resource conflict (r, p, p') is generated);
06: }
07: else add (r, p, p') to $PRCT_w$
08: }
09: }
}

On the other hand, when a resource reference is removed from an activity process, all the resource conflicts (or the potential ones) related to the activity process and the resource are eliminated. Algorithm 22 updates RCT_w and $PRCT_w$ after removing a resource reference from an activity process, and raises alerts for any elimination of resource conflicts.

Algorithm 22 Updating Resource Conflict for Removal of Resource Reference - *URCRRR*
Input: a resource r, an activity process p, an LRTS workflow w
Pre-Condition: $r \in RES_w$, $p \in P_w$, and Ref(r, p) == false
URCRRR {
01: $\forall (r, p, p') \in RCT_w$, {
02: remove (r, p, p') from RCT_w;
03: alert(Resource conflict (r, p, p') is eliminated);
04: }
05: $\forall (r, p, p') \in PRCT_w$, remove ($r$, p, p') from $PRCT_w$;
}

5.3.3 Identifying Generation or Elimination of Resource Conflicts after Alteration of EAIs

Two concurrent processes might not be concurrent any more if their EAIs are no longer overlapped after an edit operation. According to section 5.2, concurrencies between processes might be generated or eliminated after operation (7) or (8) is invoked. In this section, first we show that the operations changing EAIs in an LRTS workflow bring the same effect to all the affected processes with the following lemma.

> Lemma 12
> For an LRTS workflow w, and $\forall p, q \in P_w$, if EAI(p) changes after a design operation, EAI(q) is either altered the same way or remains unchanged after the operation.

Proof:

Algorithm 20 collects all the processes whose EAI is changed after an edit operation. Therefore, if we can show that the EAIs of the processes collected by Algorithm 20 are all altered the same way, Lemma 12 is shown correct. On the basis of the discussion made for Algorithm 20, it is known that for any process $p \in P_w$, EAI(p) is changed because of the following situations:

(1) D(p) is altered.

(2) (p', p)$\in F_w$, and d(p') is altered.

(3) (p', p)$\in F_w$, and EAI(p') is altered.

According to the methods illustrated in Figure 9, for process p, only LET(p) is altered in situation (1), and only EST(p) is altered for situation (2). Since a designer is allowed to modify only the minimum or maximum working duration of a single activity process, only one of the EST(p) and LET(p) can be changed in situation (3).

For situation (1), let LET'(p) be the original value of LET(p), and D(p) is changed from v to v', and LET(p) = LET'(p) + (v' - v). Let (p, p'')$\in F_w$. LET(p'') = LET(p) + D(p) = LET'(p) + (v' - v) + D(p'') = LET'(p'') + (v'- v), and therefore LET(p'') is changed the same way as LET(p) did. According to the calculation in Algorithm 20, all the affected processes invokes the same formula for the changes of LETs, and therefore are altered the same way LET(p) did. Lemma 12 holds for situation (1).

For situation (2), let EST'(p) be the original value of EST(p), and d(p) is changed from v to v', and EST(p) = EST'(p) + (v' - v). Let (p, p'')$\in F_w$. EST(p'') = EST(p) + d(p) = EST'(p) + (v' - v) + d(p) = EST'(p'') + (v' - v). EST(p'') is changed the same way as EST(p) did. According to the calculation in Algorithm 20, all the affected processes invoke the same formula for the changes of EST's, and therefore are altered the same way as EST(p) did.

Lemma 12 holds for situation (2).

For situation (3), since the designer may not modify EAIs directly, there must exist some process n that D(n) or d(n) is changed by designers. The proofs made above can be adopted for n, and show that Lemma 12 holds for situation (3).

Since Lemma 12 holds in all the situations leading to change of EAI(p), Lemma 12 is shown correct. □

For any process q whose EAI is changed after this operation, if EST'(q) < EST(q) or LET'(q) > LET(q), EAI(q) expands after the operation; otherwise, EAI(q) shrinks. With Lemma 12, it is known that the EAIs change the same way after operation (7) or (8) is invoked. In other words, for any q, EAI(q) is either expanded or shrunk. According to the definition of operation (7) and (8) stated in section 5.2, let p be the process whose working duration is modified, v represents the original value of d(p) or D(p), v' represents the new value assigned, and $var = (v' - v)$ represents the variation of the modified working duration, and from the proof of Lemma 12, the variation of EST(q) or LET(q) is also var. If d(p) decreases (i.e. $var < 0$ in operation (7)), EAI(q) is expanded for $|var|$ time units; otherwise, EAI(q) is shrunk. On the contrary, if D(p) increases (i.e. $var > 0$ in operation (8)), EAI(q) is expanded for $|var|$ time units; otherwise, EAI(q) is shrunk.

In the following lemma, we show that for any processes in an LRTS workflow, shrink of its EAI creates no resource conflicts, and expansion of its EAI eliminates no ones.

> Lemma 13
> For an LRTS workflow w, and $\forall p \in P_w$, the shrink of EAI(p) does not generate any new resource conflict, and the expansion of EAI(p) eliminates no resource conflicts.

Proof:

The lemma is shown correct through following discussions:

(1) The shrink of EAI(p) can not generate new resource conflicts.

B.W.O.C, it is assumed that (r, p, q) is a new resource conflict generated from shrink of EAI(p). Let EAI'(p) be the original value of EAI(p). EAI(p) $\approx_{TI}$ EAI(q) and ~(EAI'(p) $\approx_{TI}$ EAI'(q)) both hold. Since EAI(p) is shrunk, one of the statement EST(p) > EST'(p) or LET(p) < LET'(p) holds. First, we discuss the case that EST(p) > EST'(p). According to Lemma 12, LET(p) == LET'(p), LET(q) == LET'(q), and EST(q) ≥

EST'(q). Since ~(EAI'(p) $\approx_{TI}$ EAI'(q)), MIN({LET'(p), LET'(q)}) – MAX({EST'(p), EST'(p')}) ≤ 0. Concluding the descriptions above, MIN({LET(p), LET(q)}) – MAX({EST(p), EST(p')}) ≤ 0, and therefore ~(EAI(p) $\approx_{TI}$ EAI(q)) which is a contradiction. The case that LET(p) < LET'(p) can be proved the similar way, and therefore the first statement of Lemma 13 is shown correct.

(2) The expansion of EAI(n) can not eliminate any resource conflict.

B.W.O.C, it is assumed that (r, p, q) is a resource conflict eliminated from expansion of EAI(p). Therefore, ~(EAI(p) $\approx_{TI}$ EAI(q)) and EAI'(p) $\approx_{TI}$ EAI'(q) both hold. Since EAI(p) is shrunk, one of the statement EST(p) < EST'(p) or LET(p) > LET'(p) holds. First, we discuss the case that EST(p) < EST'(p). According to Lemma 12, LET(p) == LET'(p), LET(q) == LET'(q), and EST(q) ≤ EST'(q). Since EAI'(p) $\approx_{TI}$ EAI'(q), MIN({LET'(p), LET'(q)}) – MAX({EST'(p), EST'(p')}) > 0. Concluding the descriptions above, MIN({LET(p), LET(q)}) – MAX({EST(p), EST(p')}) > 0, and therefore EAI(p) $\approx_{TI}$ EAI(q) which is a contradiction. The case that LET(p) > LET'(p) can be proved the similar way, and therefore the second statement of Lemma 13 is shown correct.

By (1) and (2), Lemma 13 is shown correct. □

Since the operations causing EAI expansion does not affect the structure of the LRTS workflow or the resource references among the processes, the resource conflicts generated from the operation must be a potential resource conflict before the operation is made. After operation (7) or (8) is invoked, if the EAI of the target process is expanded, Algorithm 23 checks each potential resource conflict in $PRCT_w$. If there exists any potential resource conflict that the processes involved in it become concurrent, Algorithm 23 transfers the resource conflict from $PRCT_w$ to RCT_w and raises an alert to designers about the generation of the resource conflict.

On the other hand, if the EAI of the target process shrinks after invocation of operation (7) or (8). Algorithm 24 checks RCT_w to assure that whether there exists any existent resource conflicts that the processes involved in it are no longer concurrent, i.e. the EAIs of the processes are no longer overlapped. If so, Algorithm 24 transfers the resource conflict from RCT_w to $PRCT_w$ and raises an alert to designers about the elimination of the resource conflict. The details of Algorithm 23 and Algorithm 24 are described as following.

Algorithm 23 Detecting Resource Conflict after EAI Expansion - *DRCEE*

Input: A set of processes PSet

Pre-Condition: PSet $\subseteq P_w$, $\forall p \in$ PSet, *p.type* == ACT

```
DRCEE {
01: ∀ p∈ PSet {
02:     ∀ (r, p, q)∈ PRCT_w {
03:       if( EAI(p) ≈_TI EAI(q) )
04:         remove (r, p, q) from PRCT_w;
05:         add (r, p, q) into RCT_w;
06:         alert( Resource conflict (r, p, p') is generated );
07:       }
08:    }
09: }
}
```

Algorithm 24 Updating Resource Conflict after EAI Shrink - *URCES*

Input: A set of processes PSet

Pre-Condition: PSet $\subseteq P_w$, $\forall p \in$ PSet, *p.type* == ACT

```
URCES {
01:   ∀ p∈ PSet {
02:     ∀ (r, p, q)∈ RCT_w {
03:       if( !(EAI(p) ≈_TI EAI(q)) )
04:         remove (r, p, q) from RCT_w;
05:         add (r, p, q) into PRCT_w;
06:         alert( Resource conflict (r, p, p') is eliminated );
07:       }
08:     }
09:   }
}
```

5.3.4 Combining the Algorithms with Edit Operations

With all the methods constructed above, the post actions of edit operations are described as following:

(1) Inserting activity process *p* into an existent flow

Post-Action: $p.abstack = \begin{cases} \text{Push}(p'.abstack, bm), \text{if } p'.type \in \{\text{AS, XS}\} \\ p'.abstack, \text{otherwise} \end{cases}$

$\text{EST}(p) = \text{EST}(p') + d(p')$, $\text{LET}(q) = \text{LET}(p') + D(p)$

Comments: The ABStack and EAI corresponding to *p* are calculated based on the methods described in section 2.4.2 and 2.4.3.

(2) Inserting a new decision structure quoted by *sp* and *jn* into an existent flow

Post-Action: $sp.abstack = \begin{cases} \text{Push}(p'.abstack, bm) \text{ if } p'.type \in \{\text{AS, XS}\} \\ p'.abstack \text{ otherwise} \end{cases}$

$jn.abstack = sp.abstack$

$\text{EST}(sp) = \text{EST}(p') + d(p')$, $\text{LET}(sp) = \text{LET}(p') + D(p)$

$\text{EST}(jn) = \text{EST}(sp)$, $\text{LET}(jn) = \text{LET}(sp)$

Comments: The ABStacks and EAIs corresponding to *sp* and *jn* are calculated based on the methods described in section 2.4.2 and 2.4.3.

(3) Inserting a new parallel structure quoted by *sp* and *jn* into an existent flow

Post-Action: $sp.abstack = \begin{cases} \text{Push}(p'.abstack, bm) \text{ if } p'.type \in \{\text{AS, XS}\} \\ p'.abstack \text{ otherwise} \end{cases}$

$jn.abstack = sp.abstack$

$\text{EST}(sp) = \text{EST}(p') + d(p')$, $\text{LET}(sp) = \text{LET}(p') + D(p)$

$\text{EST}(jn) = \text{EST}(sp)$, $\text{LET}(jn) = \text{LET}(sp)$

Comments: The ABStacks and EAIs corresponding to *sp* and *jn* are calculated based on the methods described in section 2.4.2 and 2.4.3.

(5) Adding a resource reference to an activity process

Post-Action: invoking DRCNRR(*r*, *p*)

Comments: Algorithm 21 is invoked to detect the resource conflicts generated because of the operation.

(6) Removing a resource reference from an activity process

Post-Action: invoking URCRRR(*r*, *p*)

Comments: Algorithm 22 is invoked to remove the resource conflicts eliminated because of the operation.

(7) Setting minimal working duration of an activity process

Post-Action: invoking $\begin{cases} \text{DRCEE}(\text{CEAI}(w, p)) \text{ if } var < 0 \\ \text{URCES}(\text{CEAI}(w, p)) \text{ otherwise} \end{cases}$

Comments: After the EAIs affected by this operation are updated, Algorithm 23 is invoked if the modified d(p) is smaller than the original one, and Algorithm 24 is invoked otherwise.

(8) Setting maximal working duration of an activity process

Post-Action: invoking $\begin{cases} \text{DRCEE}(\text{CEAI}(w, p)) \text{ if } var > 0 \\ \text{URCES}(\text{CEAI}(w, p)) \text{ otherwise} \end{cases}$

Comments: After the EAIs affected by this operation are updated, Algorithm 24 is invoked if the modified D(p) is smaller than the original one, and Algorithm 23 is invoked otherwise.

5.4 Case Study

To demonstrate our methodology, three cases are studied in this section. First, we show how to detect the resource conflict generated by a resource assignment, second, the effect brought by changing the working duration of an activity process is presented, and at last, we show the influences about removal of an activity process. We assume that designers has edited the sample LRTS workflow w from a basic LRTS workflow as illustrated in Figure 18, and our case study starts accordingly.

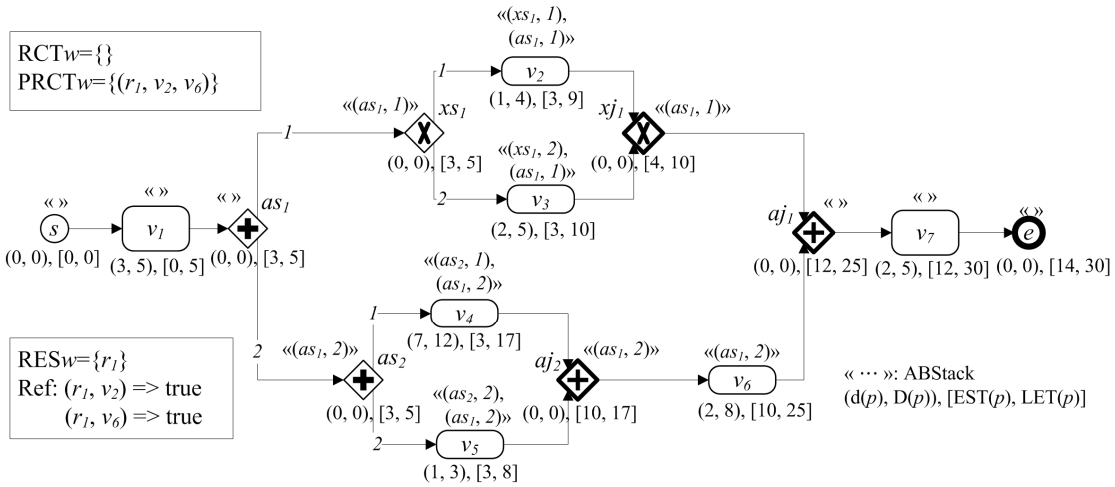


Figure 18 The Sample LRTS Workflow for the Case Study in Chapter 5

5.4.1 Case 1: Adding a Resource Reference

With the sample LRTS workflow in Figure 18, designers add resource reference r_1 to activity process v_4. Ref(r_1, v_4) becomes true after the operation. With the discussions made in section 5.3, Algorithm 21, DRCNRR(r_1, v_4), is invoked.

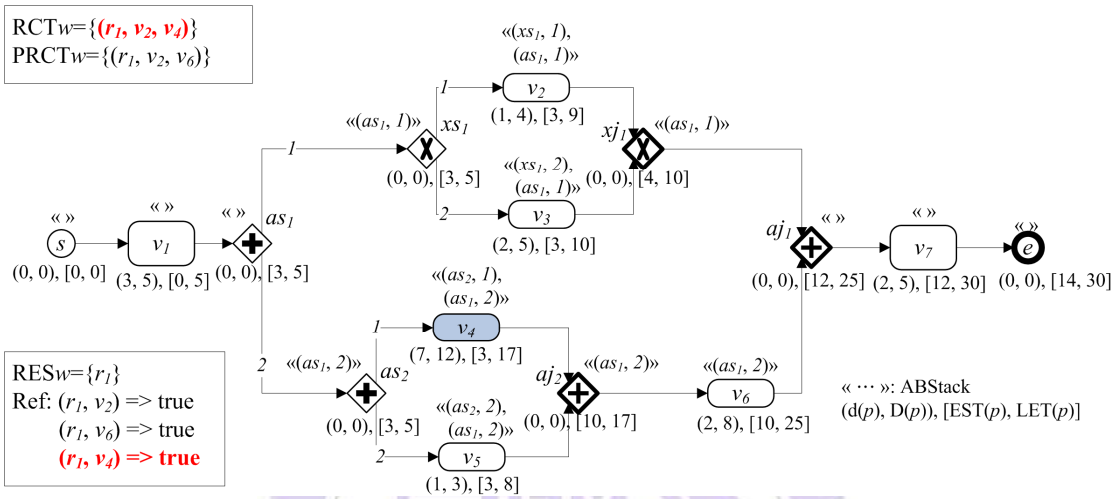


Figure 19 The Sample LRTS Workflow after Adding a New Resource Reference

Algorithm 21 checks the structural and temporal relationships between v_4 and the other processes referring to r_1, i.e. v_2 and v_6. According to the EAIs and the ABStacks of the processes, the concurrency between v_2 and v_4 is identified, and a new resource conflict (r_1, v_2, v_4) is generated. (r_1, v_2, v_4) is put into RCT_w, and a corresponding alert is raised for designers. After the operation, the LRTS workflow is updated as Figure 19, and the altered parts are marked with different colors.

5.4.2 Case 2: Modification of the Working Duration of an Activity Process

After adding a resource reference to v_4, designers modify the minimal working duration of v_4 from 7 to 4. Algorithm 20 is then invoked and updates the EAIs of the processes aj_2, v_6, aj_1, v_7, and e. Since d(v_4) is decreased from 7 to 4, the EAIs are expanded. Algorithm 23 is invoked. Since EAI(v_6) is expanded from [10, 25] to [7, 25] and is overlapped to EAI(v_2), the potential resource conflict (r_1, v_2, v_6) becomes an actual one after this operation. (r_1, v_2, v_6) is transferred

from $PRCT_w$ to RCT_w, and a corresponding alert is raised to give a warning to designers about the generation of the resource conflict. After the operation, the TS workflow is updated as Figure 20, and the altered parts are marked with different colors.

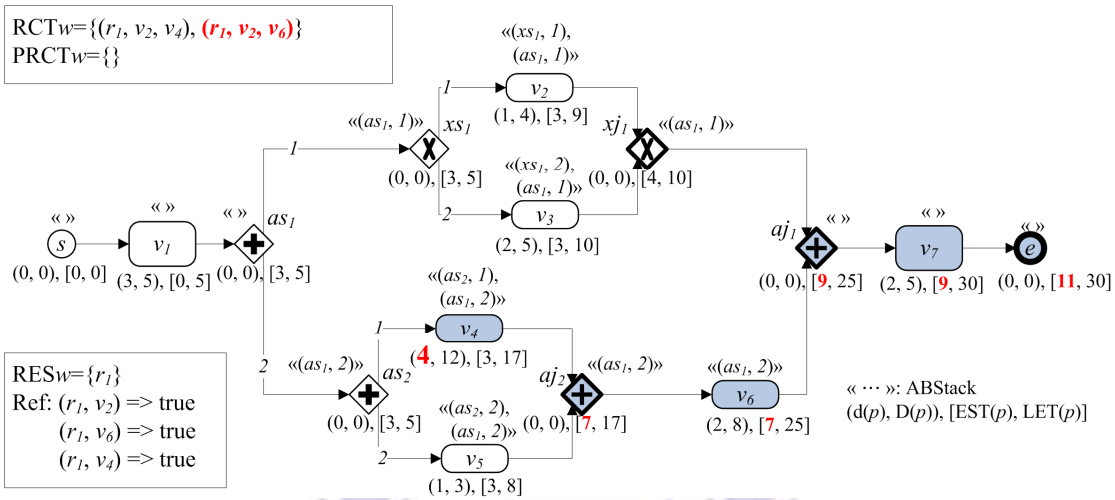


Figure 20 The Sample LRTS Workflow after Modification of a Working Duration

5.4.3 Case 3: Removing an Activity Process

After modifying the minimal working duration of v_4, designers decide to delete the activity process v_6. Before v_6 is actually removed, its resource references should be first removed, and its working durations are set to zero. Therefore, Ref(r_1, v_6) is set to be false, and all the resource conflict related to r_1 and v_6 are eliminated. The EAIs of the processes succeeding to v_6 are updated after d(v_6) and D(v_6) are set to zero. Since none of the affected process makes any references to resources, no resource conflict are generated or eliminated here. Finally, v_6 is removed from w, the flows (aj_2, v_7) and (v_7, aj_1) are removed, and (aj_2, aj_1) is added into F_w instead. After the operation, the sample LRTS workflow is updated as Figure 21, and the altered parts are marked with different colors.

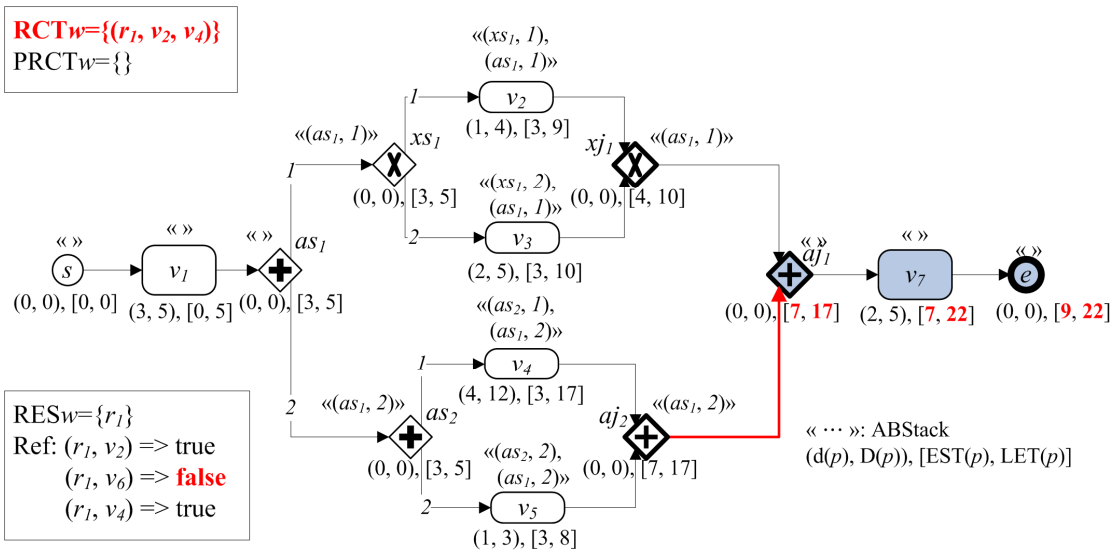


Figure 21 The Sample LRTS Workflow after Deleting an Activity Process

5.5 Related Works

Resource allocation is a popular topic in analysis of workflow models. In [55], Tang et al. extend a Petri-net based workflow model for composition of web-services and resources. In [32] and [33], Reveliotis et al. integrate resource allocation systems into a workflow model to analyze deadlocks and synchronization problems. Sun et al. extend the approach developed in [9] with additional resource constraints for analysis of performance among workflows [56]. Russel et al. conclude various representation and utilization of resources in workflows as 43 resource allocation patterns, and discuss coordination among workflow, human resources and external resources in detail [50].

In [57], Xiao et al. define the execution duration of a workflow and develop an approach to analyze the resource feasibility in the workflow during its execution. The approach tracks the resource occupation made by individual activities in a workflow, and keeps resources feasible when parallel access of resources happens. Wang et al. present a modeling and analysis approach for workflows with resources and non-determined time constraints on petri-nets [58]. The resources and the activities in workflows are modeled with different kinds of places in petri-nets. By analyzing the reachability graph of the R/NT_WF_Net, the implementation cases

which satisfy various timing constraints (from the best to the worst) are discussed and categorized.

In [34], Li et al. model resources and temporal constraints in workflow specification for analysis, and develop a methodology to detect generation and elimination of resource conflicts in timed workflow specifications. Because Li et al. establish a complete model for analysis of resource constraints in timed workflows, several studies like [35] and [36] follow Li's approach for further analysis of resource constraints among timed workflow. Zhong et al. apply Li's timing model to establish a Petri-net based workflow model for verification of resource constraints among concurrent workflows [35]. In [36], Hsu et al. focus on providing information to the workflow designers about resource conflicts in a workflow specification during design-time with an incremental algorithm. The works in [36] is revised in [37], and is further discussed in this dissertation to adopt the methods raised in [30] on TS workflow model for analysis.

Based on the former studies on static timing management of workflow specifications, Li continued his own research by analyzing the resource and temporal constraints between distinct workflow instances dynamically [59]. In [59], the concept of reference points is introduced to show the relative timing constraints between the activities in different workflow instances. According to a pre-specified reference point in each workflow, if any resource conflict exists, the EAIs of the processes involved in the conflict are adjusted. On the other hand, Wang et al. presents a modeling and analysis approach for workflows with resources and non-determined timing constraints on petri-nets [58]. On the other hand, Delias et al. propose an algorithm to minimize the resource conflicts subject to temporal constraints and simultaneously optimizes throughput or utilization of resources among workflow instances [60]. Rather than totally avoiding resource conflicts, Delias' approach optimizes the utilization of resources by maximizing overlapping between tasks which will eventually use different resources of the

same type [60]. The resource and temporal factors are formulated in a matrix to achieve an efficient optimal solution for run-time resource scheduling [60].

Chapter 6. Conclusion and Future Works

In this dissertation, the structural and temporal issues in workflow analysis are considered and modeled with TS workflow. Based on TS workflow model, three distinct analysis approaches for various perspectives are developed accordingly. For the organization perspective, the issues for delegation in WfMS coordinated with TRBAC are discussed. The constraints on delegation like delegation loops, separation of duty, and various enterprise policies etc. are detected and followed dynamically. With our methodology, users are able to request delegations for their works manually, and WfMS can delegate an emergent task to an appropriate delegatee automatically. For the data perspective, on the basis of define-use-kill operations, the artifact anomalies generated from the twisted temporal and structural relationships between processes in TS workflow are stated. The racing behavior from the concurrent activities are categorized and discussed, and a methodology detecting artifact anomalies in a TS workflow through static analysis is established. For the resource perspective, an incremental methodology verifying resource conflicts in a TS workflow along with every edit operation made by designers is described. The relationships between edit operations and generation/elimination of resource conflicts are discussed with both structural and temporal consideration. With the methodology, designers realize the effect of each edit operation they made, and acquire information to help correcting resource conflicts in their design.

In the future, several issues can be further studied on the basis of the methodologies in this dissertation. First, for the delegation framework, the feasibility and security issues in sharing a task instance among users can be studied to adopt grant operation in delegation of task instances. Users' capability and pleasure should also be considered in automatic delegation by applying

the techniques based on knowledge management. Second, for the detection of artifact anomalies, a solution to group the operation sets with completeness and better efficiency should be studied. An incremental algorithm to detect artifact anomalies generated or eliminated by edit operations made by designers can be constructed. Besides, the actions made on artifacts and processes need being studied with consideration of the dependencies caused by the actions among activity processes in more details. Third, for the verification of resource conflicts, our methodology can be extended to detect conflicts generated across various workflows. Besides, multiple instances of a resource type should be considered, and the delays caused by flows may also be included in temporal analysis.

Reference

[1] Workflow Management Coalition, “Workflow Management Coalition: Terminology & Glossary,” Document Number WFMC-TC-1011, 1999

[2] W. M. P van der Aalst, “The application of Petri Nets to Workflow Management,” in Journal of Circuits, Systems, and Computers, Vol. 8, Issue 1, pp. 21-46, 1998

[3] Z. Irani, V. Hlupic, and G. M. Giaglis, “Busincess Process Reengineering: An Analysis Perspective,” in Journal of Flexible Manufacturing Systems, Vol. 14, pp. 5-10, 2002

[4] K. Vergidis, Ashutosh Tiwari, and Basim Majeed, “Business Process Analysis and Optimization beyond Reengineering,” in IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews, Vol. 38, Issue 1, pp. 69-82, 2008

[5] N. R. Adam, V. Atluri, and W.-K. Huang, “Modeling and Analysis of Worfklows Using Petri Nets,” in Journal of Intelligent Information Systems, Vol. 10, Issue 2, pp. 131-158, 1998

[6] W. M. P. van der Aalst, and A. H. M. ter Hofstede, “Verification of Workflow Task Structures: A Petri-net Approach,” in Information System Vol. 25, Issue 1, pp. 43-69, 2000

[7] W. M. P. van der Aalst, K.M. van Hee, and R.A. van der Toorn, "Adaptive Workflow: An Approach Based on Inheritance," in the Proceedings of the Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business, pp. 36-45, 1999

[8] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler, “On Structured Workflow Modelling,” in Lecture Notes in Computer Science, Vol. 1789, pp. 431-445, 2000

[9] J. Li, Y. Fan, and M. Zhou, “Performance Modeling and Analysis of Workflow,” in IEEE Transaction on Systems, Man, and Cybernetics - Part A: Systems and Humans, Vol. 34, Issue 2, pp.229-242, 2004

[10] J. Chen, and Y. Yang, “Temporal Dependency based Checkpoint Selection for Dynamic

Verification of Fixed-time Constraints in Grid Workflow Systems," in the Proceedings of the 30th International Conference on Software Engineering, pp. 141-150, 2008

[11] J. Eder, E. Panagos, H. Pozewaunig, and M. Rabinovich, "Time Management in Workflow Systems," in the Proceedings of International Conference on Business Information Systems, pp. 266-280, 1999

[12] J. Eder, E. Panagos, and M. Rabinovich, "Time Constraints in Workflow Systems," in Lecture Notes in Computer Science, Vol. 1626, pp. 286-300, 1999

[13] O. Marjanovic, "Dynamic Verification of Temporal Constraints in Production Workflows," in the Proceedings of the 11th Australian Database Conference, pp. 74-81, 2000

[14] H. Zhuge, T.-Y. Cheung, and H.-K. Pung, "A Timed Workflow Process Model," in Journal of Systems and Software, Vol. 55, Issue 2, pp. 231-243, 2001

[15] R. S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-Based Access Control Models," IEEE Computer, Vol. 29, Issue 2, pp. 38-47, 1996

[16] D. Ferraiolo and R. Kuhn, "Role-based Access Controls," in the Proceedings of the 15th National Computer Security Conference, Vol. 2, pp. 554-563, 1992

[17] S. Oh, and S. Park, "Task-role-based access control model," in Information Systems, Vol. 28, Issue 6, pp. 533-562, 2003

[18] J. Crampton and H. Khambhammettu, "Delegation in Role-Based Access Control," in International Journal of Information Security, Vol. 7, Issue 2, pp. 123-136, 2008

[19] E. Barka and R. Sandhu, "A Role-Based Delegation Model and Some Extensions," in the Proceedings of the 23rd National Information Systems Security Conference, pp. 101-114, 2000.

[20] E. Barka, and R. Sandhu, "Role-Based Delegation Model/Hierarchical Roles (RBDM1)," in the Proceedings of 20th Computer Security Applications Conference, 2004, pp. 396-404.

[21] J. Wainer, and A. Kumar, "A Fine-grained, Controllable, User-to-User Delegation Method in RBAC," in the Proceedings of the 10th ACM symposium on Access control models and technologies, pp. 59-65, 2005

[22] H. Wang, and S. Osborn, "Delegation in the Role Graph Model," in the Proceedings of the 11th ACM symposium on Access control models and technologies, pp. 91-100, 2006

[23] J. B. D. Joshi, and E. Bertino, “Fine-grained Role-based Delegation in Presence of the Hybrid Role Hierarchy,” in the Proceedings of the 11th ACM symposium on Access control model and technologies, pp. 81-90, 2006

[24] P. Jian, H.-J. Hsu, and F.-J. Wang, "A Delegation Framework for Access Control in WfMS based on Tasks and Roles" in the Proceedings of 12th IEEE International Workshop on Future Trends of Distributed Computing Systems, pp.165-171, 2008

[25] H.-J. Hsu and F.-J. Wang, “A Delegation Framework for Task-Role based Access Control in WfMS,” in Journal of Information Science and Engineering, Vol. 27, Issue 3, pp. 1011-1028, 2011

[26] S. Sadiq, M. E. Orlowska, W. Sadiq, and C. Foulger, “Data flow and validation in workflow modeling,” in the Proceedings of the 15th Conference on Australasian Database, Vol. 27, pp. 207-214, 2004

[27] F.-J. Wang, C.-L. Hsu, and H.-J. Hsu, “Analyzing Inaccurate Artifact Usages in a Workflow Schema,” in the Proceedings of the 30th Annual International Computer Software and Application Conference, Vol. 2, pp. 109-114, 2006

[28] C.-L. Hsu, H.-J. Hsu, and F.-J. Wang, “Analysing Inaccurate Artifact Usages in Workflow Specifications,” in IET Software, Vol. 1, Issue 4, pp. 188-205, 2007

[29] C.-H. Wang, and F.-J. Wang, “Detecting Artifact Anomalies in Business Process Specification with a Formal Model,” in Journal of Systems and Software, Vol. 82, Issue 10, pp. 1064-1212, 2009

[30] H.-J. Hsu, and F.-J. Wang, "Using Artifact Flow Diagrams to Model Artifact Usage

Anomalies," in the Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference, Vol. 2, pp.275-280, 2009

[31] W. Sadiq, and M. E. Orlowska, “Analyzing Process Models Using Graph Reduction Ttechniques,” in Information System, Vol. 25, Issue 2, pp. 117-134, 2000

[32] S. Reveliotis, “Structural Analysis of Resource Allocation Systems with Synchronization Constraints,” in the Proceedings of the IEEE International Conference on Robotics & Automation, pp. 1045-1049, 2003

[33] J. Park, and S. Reveliotis, “Deadlock Avoidance in Sequential Resource Allocation Systems with Multiple Resource Acquisitions and Flexible Routings,” in IEEE Transactions on Automatic Control, Vol. 46, pp. 1572-1583, 2001

[34] H. Li, Y. Yang, and T. Y. Chen, “Resource Constraints Analysis of Workflow Specifications,” in Journal of Systems and Software, Vol. 73, Issue 2, pp. 271-285, 2004

[35] J. Zhong, and B. Song, “Verification of Resource Constraints for Concurrent Workflows,” in the Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 253-261, 2005

[36] H.-J. Hsu, D.-L. Yang, and F.-J. Wang, “An Incremental Analysis to Workflow Specifications,” in the Proceedings of the 12th Asia-Pacific Software Engineering Conference, pp. 122-129, 2005

[37] H.-J. Hsu and F.-J. Wang, “An Incremental Analysis for Resource Conflicts to Workflow Specifications,” in Journal of Systems and Software, Vol. 81, Issue 10, pp. 1770-1783, 2008

[38] I.-F. Leong, and Y.-W. Si, "Temporal Exception Prediction for Loops in Resource Constrained Concurrent Workflows," in the Proceedings of 6th IEEE International Conference on e-Business Engineering, pp. 310-315, 2009

[39] J. F. Allen, “Maintaining knowledge about temporal intervals,” in Communication of the ACM, Vol. 26, Issue 11, pp. 832–843, 1983

[40] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "Generalized Temporal Role-Based Access Control Model," IEEE Transactions on Knowledge and Data Engineering, Vol. 17, Issue 1, pp. 4-23, 2005

[41] M. Nyanchama, and S. Osborn, "The Role Graph Model and Conflict of Interest," in ACM Transactions on Information and System Security, Vol. 2, Issue 1, pp. 3-33, 1999

[42] P. H. Bammigatti, and P. R. Rao, “Delegation in Role Based Access Control Model for Workflow Systems,” in International Journal of Computer Science and Security, Vol. 2, Issue 2, pp. 1-10, 2008

[43] K. Gaaloul, and F. Charoy, “Task Delegation Based Access Control Models for Workflow Systems,” Software Services for e-Business and e-Society, IFIP Advances in Information and Communication Technology, Vol. 305, pp. 400-414, 2009

[44] D.-W. Zhang, X. Pei, J.-Q. Qiu, Y. Zhang, and J. Peng, “A Delegation Model For Time Constraints-Based TRBAC,” in the Proceedings of the 8th International Conference on Machine Learning and Cybernetics, pp. 2027-2032, 2009

[45] R. T. Simon, and M. E. Zurko, "Separation of Duty in Role-based Environments," in the Proceedings of 10th Computer Security Foundations Workshop, pp.183-195, 1997

[46] R. A. Botha, and J. H. P. Eloff, “Separation of Duties for Access Control Enforcement in Workflow Environments,” in IBM System Journal, Vol. 40, Issue 3, pp. 666-683, 2001

[47] M. J. Moyer, and M. Ahamad, “Generalized Role-based Access Control,” in the Proceedings of 21st IEEE International Conference on Distributed Computing Systems, pp. 391-398, 2001

[48] G. Ding, J. Chen, R. F. Lax, and P. P. Chen, “Graph-theoretic Method for Merging Security System Specifications,” in Information Sciences, Vol. 177, Issue 10, pp. 2152-2166, 2007

[49] C.-H. Chang and F.-J. Wang, "An Analysis of Delegation Mechanism in Workflow Management System," Master's Thesis, Institute of Computer Science and Engineering,

National Chiao Tung University, 2003

[50] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst, "Workflow Resource Patterns," BETA Working Paper Series, WP 127, Eindhoven University of Technology, Eindhoven, 2004

[51] S. X. Sun, J. L. Zhao, J. F. Nunamaker, and O. R. L. Sheng, "Formulating the data flow perspective for business process management," in Information Systems Research, Vol. 17, Issue 4, pp. 374-391, 2006

[52] P. M. Pardalos, and J. Xue, “The Maximum Clique Problem,” in Journal of Global Optimization, Vol. 4, No. 3, pp. 301-328, 1994

[53] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, “A new algorithm for generating all the maximal independent sets,” in Society for Industrial and Applied Mathematics (SIAM) Journal on Computing, Vol. 6, pp. 505–517, 1977.

[54] K. Makino, and T. Uno, “New Algorithms for Enumerating All Maximal Cliques,” in the Proceedings of 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science, Vol. 3111, pp. 260-272, 2004

[55] Y. Tang, L. Chen, K. He, and N. Jing, “SRN: an Extended Petri-net-based Workflow Model for Web Service Composition,” in the Proceedings of IEEE International Conference on Web Services, pp. 591-599, 2004

[56] P. Sun, J. Wang, X. Li, and C. Jiang, “Performance Analysis of Workflow Model with Resource Constraints,” in the Proceedings of the 1st International Multi Symposiums on Computer and Computational Sciences, Vol. 1, pp. 397-401, 2006

[57] J. Xiao, X. Wei, and S. Chen, "An Approach for Checking Resource Feasibility of Workflow Specifications," in the Proceedings of the 1st International Workshop on Education Technology and Computer Science, Vol. 1, pp.163-167, 2009

[58] H. Wang, and Q. Zeng, “Modeling and Analysis for Workflow Constrained by Resources and Nondetermined Time: An Approach Based on Petri Nets,” in IEEE Transactions on

Systems, Man, and Cybernetics – Part A: Systems and Humans, Vol. 38, Issue 4, 2008

[59] H. Li, and Y. Yang, “Dynamic Checking of Temporal Constraints for Concurrent Workflows,” in Electronic Commerce Research and Applications Vol. 4, pp. 124-142, 2005

[60] P. Delias, A. D. Doulamis, N. D. Doulamis, and N. Natsatsinis, “Optimizing Resource Conflicts in Workflow Management Systems,” in IEEE Transactions on Knowledge and Data Engineering, Vol. 23, Issue 3, 2011