國立交通大學

資訊科學與工程研究所

博 士 論 文

運用一正規化模式來偵測商業流程規格中
異常的Artifact使用

Detecting the Artifact Anomalies in
Business Process Specifications with a Formal Model

研 究 生：許嘉麟
指導教授：王豐堅　教授

中 華 民 國 九 十 六 年 十 月

運用一正規化模式來偵測商業流程規格中異常的 Artifact 使用

Detecting the Artifact Anomalies in Business Process Specifications with a Formal Model

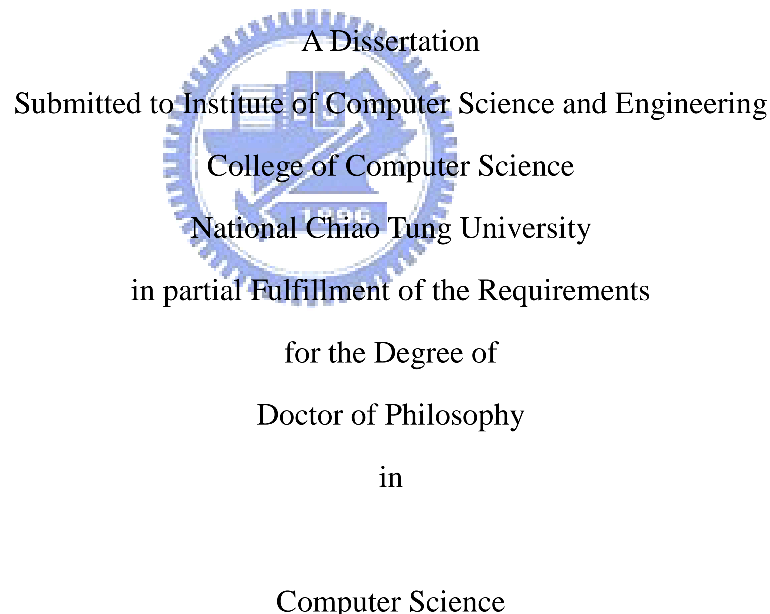
研 究 生：許嘉麟　　Student：Chia-Lin Hsu

指導教授：王豐堅　　Advisor：Feng-Jian Wang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 士 論 文



A Dissertation

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

October 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年十月

運用一正規化模式來偵測商業流程規格中異常的 Artifact 使用

學生：許嘉麟　　　　　　　　　　　　　　指導教授：王豐堅 博士

國立交通大學資訊工程與科學研究所 博士班

摘要

儘管已有許多商業流程模型被提出，卻鮮少針對 artifact 的使用進行分析。由於不適當的 artifact 操作，譬如說 artifact 流程與控制流程不一致或是相衝突的 artifact 運算，一個結構良好且擁有足夠資源的商業流程在執行時依然可能產生非預期的結果。因此，分析 artifact 的使用是很重要的畢竟活動無法在沒有精確的資訊的情況下執行正確。本論文提出一個流程模型來描述商業流程並且在此模型上分析 artifact 的使用。總共有三類(十三種狀況)會影響流程執行的異常 artifact 使用被確認出來並且使用系統化的方式來表達。除此之外，本論文提出偵測這些異常的演算法並以一個實際的例子作示範說明。

關鍵字: 工作流程，商業流程，分析，控制流程，資料流程，異常

Detecting the Artifact Anomalies in Business Process Specifications with a Formal Model

Student：Chia-Lin Hsu　　　　　　　　Advisor：Dr. Feng-Jian Wang

Institute of Computer Science and Engineering
National Chiao Tung University

Abstract

Although many business process models have been proposed, analyses on artifact usages are seldom discussed. A well-structured business process with sufficient resources may still fail or yield unexpected results during process execution due to inaccurate artifact specification e.g. inconsistency between artifact flow and control flow, or contradictions between artifact operations. Thus, the analyses on artifact usages are very important since activities cannot be executed properly without accurate information. This dissertation presents a process model for describing a business process and analyzes the artifact usages on this model. Three types with thirteen cases of artifact usage anomalies affecting process execution are identified and formulates and a set of algorithms to detect these anomalies in business process specifications is presented. Furthermore, an example is demonstrated to validate the usability of the proposed algorithms.

Keyword: workflow, business process, analysis, control flow, data flow, artifact, anomaly.

誌　　謝

本篇論文的完成，首先要萬分感謝指導教授王豐堅博士，王教授在我求學期間（從碩士班到博士班）持續不斷的指導與鼓勵，讓我不僅在論文研究方面學習到相當寶貴的經驗，在做人處事方面也獲益良多。如今學生若有些微的成就，王教授的指導實在功不可沒。

其次要感謝吳毅成教授，陳耀宗教授，朱治平教授，朱正忠教授，黃悅民教授，與楊鎮華教授，在百忙之中首肯擔任我博士論文的口試委員，並且提供了許多寶貴的意見，補足我論文裡不足的部分。其中吳、陳兩位教授亦是我的論文計畫書口試委員，在論文報告的方式上也給了我相當多的指導。此外，對於一起研究討論與互相鼓勵的實驗室學長與學弟妹們，包括楊基載、黃國展、王建偉、王靜慧、許懷中等等，在此一併加以感謝。

最後，我要與我的家人、同學、朋友、溶璘、徐媽媽與徐嬤嬤共同分享完成論文的喜悅，由於有您們的支持與關懷，陪伴我走過這漫長的求學過程。僅將此論文獻給我最敬愛的父母親與支持我的親友們。

Table of Contents

摘要 I

ABSTRACT II

誌　　謝 III

TABLE OF CONTENTS IV

LIST OF TABLES VI

LIST OF FIGURES VII

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. RELATED WORK AND BACKGROUND 3

CHAPTER 3. PROCESS MODELING 6

3.1. PROCESS SPECIFICATIONS 6

3.2. CONTROL FLOW SPECIFICATION 7

3.2.1. Activities and Control Blocks 7

3.2.2. Relations among Activities and Control Blocks 10

3.3. ARTIFACT FLOW SPECIFICATION 13

3.3.1. Artifacts and Artifact Operations 13

3.3.2. Artifact Flow and Artifact Usages 15

CHAPTER 4. ARTIFACT USAGE ANOMALIES 17

4.1. ARTIFACT USAGE ANOMALIES 17

4.1.1. Missing Production Anomalies 17

4.1.2. Redundant Write Anomalies 22

4.1.3. Conflict Write Anomalies 24

4.1.4. Summary of Usage Patterns Causing Artifact Usage Anomalies 25

CHAPTER 5. ALGORITHMS TO DETECTING ARTIFACT USAGE ANOMALIES .. 27

5.1. THE TRAVERSAL ALGORITHM 27

5.2. THE DETECTION ALGORITHM 30

5.2.1. Method for Detecting Missing Production Anomalies 30

5.2.2. Method for Detecting Redundant Production/Update Anomalies 35

5.2.3. Method for Detecting Conflict Writes Anomalies 39

5.2.4. Complexities of Traversal and Detection Algorithms 41

CHAPTER 6. ILLUSTRATIVE EXAMPLE 43

6.1. AN EXAMPLE: PROPERTY LOAN APPROVAL PROCESS 43

6.2. DETECTION OF MISSING PRODUCTION ANOMALIES 45

6.3. DETECTION OF REDUNDANT WRITE ANOMALIES 47

CHAPTER 7. COMPARISONS OF DATA-FLOW ANALYSIS APPROACHES 50
CHAPTER 8. CONCLUSION AND FUTURE WORK 54
REFERENCE 55

List of Tables

Table 4.1. Symbols Used in Usage Patterns 17
Table 4.2: Summary of Usage Patterns Causing Artifact Usage Anomalies 26
Table 6.1: Activities and Artifacts in the Property Loan Approval Process 43
Table 6.2: Artifacts Usages in the Property Loan Approval Process 44
Table 6.3. Steps to Detect Missing Production Anomalies.. 45
Table 6.4. Steps to Calculate the Unused Artifacts for Every Activity... 47
Table 7.1. Comparison of anomalies addressed. 51
Table 7.2. A summary of comparisons. 53

List of Figures

Figure 3.1. Notations of Control Flow Graph. 7
Figure 3.2. Four Primitive Types of Control Structures. 9
Figure 3.3. The State Diagram of an artifact. 14
Figure 5.1: Transform a Repeat-Until Loop. 28
Figure 5.2: Transform a While Loop. 28
Figure 6.1: Property Loan Approval Process. 44

Chapter 1. Introduction

Workflow can be viewed as a set of interrelated tasks that are systematized to achieve certain business goals by completing each task in a particular order under automatic control [1]. Resources are required for workflow implementation, and support process execution. Resource allocation and resource constraint analysis [2－6] are popular workflow research topics. However, data flow within workflow is seldom addressed [7－10].

Artifact is an abstraction of a data instance within a workflow. Introducing analysis of artifact usage into control-oriented workflow designs helps maintain consistency between execution order and data transition, as well as prevents the exceptions resulting from contradiction between data flow and control flow. In contrast to structural correctness, accuracy in artifact manipulation can help determine whether the execution result of a workflow is meaningful and desirable.

This dissertation proposes a process model for describing business processes and address three types of artifact usage anomalies. An artifact usage analysis procedure associated with the model is applied before deploying the workflow schema. Reports of consistency checking between data flow and control flow and information of manipulating artifacts are automatically provided to designers when they edit or adjust workflow specification. The model is based on component-based design technique [11, 12] and is compatible with existing control-oriented workflow design models. It provides an easier way to extract knowledge of artifact usages in a workflow. In our earlier work [13, 14], we have introduced the artifact usage analysis into workflow design phase and the improper artifact usages affecting workflow execution have been identified preliminary. In this dissertation, the artifact usages are formularized and the concrete algorithms to discovering the improper usages in workflow specifications are proposed. In addition, an example to demonstrate the contribution of our work and a comparison among related works and ours are presented.

The remainder of this dissertation is organized as follows. Chapter 2 presents the research background and related work. Chapter 3 presents our process modeling, including the control flow and artifact flow. Chapter 4 then defines three types with thirteen cases of artifact usage

anomalies. Next, chapter 5 proposes a set of algorithms to detect artifact usage anomalies in a process schema. Chapter 6 demonstrates the algorithms through an example. Chapter 7 compares our approach with related works. Conclusions are finally drawn in chapter 8, along with recommendations for future work.

Chapter 2. Related Work and Background

A workflow can be deemed as a collection of cooperating and coordinated activities designed to carry out a well-defined complex process, such as a trip planning, conference registration procedure, or business process in an enterprise. A workflow model is used to describe a workflow in terms of various elements, such as roles and resources, tools and applications, activities, and data, which represent different perspectives of a workflow [15, 16]. Roles and resources elements represent organizational perspective that describes where and by whom tasks are performed and available resources tasks can utilize in the organization. Tools and applications elements represent operational perspectives by specifying what tools and applications are used to execute a particular task. Activity elements are defined with two perspectives: 1) functional: what tasks a workflow performs; and 2) behavioral: when and how tasks are performed. Data elements represent the informational perspective, i.e., what information entities are produced or manipulated in the corresponding activities in a workflow.

A well-defined workflow model leads to the efficient development of an effective and reliable workflow application. The correctness issues in a workflow might be classified into three dimensions: control-flow, resource, and data-flow. Generally, the analyses in control-flow dimension are focused on correctness issues of control structure in a workflow. The common control-flow anomalies include deadlock, livelock (infinite loop), lack of synchronization, and dangling reference [17－28]. A deadlock anomaly occurs if it is no longer possible to make any progress for a workflow instance, e.g. synchronization on two mutually exclusive alternative paths. A livelock anomaly indicates an infinite loop, such as iteration without possible exit condition, which causes a workflow to make continuous progress, however, without progressing toward successful completion. A lack of synchronization anomaly represents the case of more than one incoming vertex merging into an or-join vertex. Activities without termination or without activation are two common cases of dangling reference anomaly.

Activities belonging to different workflows or parallel activities in the same workflow might access the same resources. A resource conflict occurs when these activities execute over the same time interval. Thus, the analyses in resource dimension include the identification of resource

conflicts under resource allocation constraints and/or under the temporal and/or causality constraints [2－6]. On the other hand, missing, redundancy, and conflict use of data are common anomalies in data-flow dimension [7－10]. A missing data anomaly occurs when an artifact is accessed before it is initialized. A redundant data anomaly occurs when an activity produces an intermediate data output but this data is not required by any succeeding activity. A conflicting data anomaly represents the existence of different versions of the same artifact.

Current workflow modeling and analyzing paradigms are mainly focused on the soundness of control logic, i.e., in the control-flow dimension, including process model analysis [19－30], workflow patterns [20－33] and automatic control of workflow process [34]. Aalst and ter Hofstede [19] proposed a WorkFlow net (WF-net), based on Petri nets, to model a workflow: transitions representing activities, places representing conditions, tokens representing cases, and directed arcs connecting transitions and places. Furthermore, control-flow anomalies, such as deadlock, livelock, and dangling reference (activities without termination or activation) have been identified through Petri net modeling and analysis. Son [35] defined a well-formed workflow based on the concepts of closure and control block. He claimed that a well-formed workflow is free from structural errors, and that complex control flows can be made with nested control blocks. Son [35] and Chang [36] identified and extracted the workflow critical path from the context of the workflow schema. They proposed extraction procedures from various non-sequential control structures to sequential paths, thus obtaining appropriate sub-critical paths in non-sequential control structures. Sadiq and Orlowska [30] proposed a visual verification approach and algorithm with a set of graph reduction rules to discover structural conflicts in process models for given workflow modeling languages.

There are several research topics discussed in resource dimension, including resource allocation constraints [2, 3], resource availability [4], resource management [5] and resource modeling [6]. Senkul [2] developed an architecture to model and schedule workflow with resource allocation constraints and traditional temporal/causality constraints. Li [3] concluded that a correct workflow specification should have resource consistence. His algorithms can verify resource consistency and detect the potential resource conflicts for workflow specifications. Both Pinar and Hongchen extended workflow specifications with constraint descriptions. Liu [4]

proposed a three-level bottom-up workflow design method to effectively incorporate confirmation and compensation in case of failure. In Liu's model, data resources are modeled as resource classes, and the only interface to a data resource is via a set of operations.

Current analysis techniques including above approaches pay little attention on the data-flow dimension, although the related analysis in data-flow dimension is very important since activities cannot be executed properly without sufficient data information. In the literature, there are two works in data-flow dimension found. Sadiq et al. [7] presented data flow validation issues in workflow modeling, including identifying requirements of data modeling and seven basic data validation problems: redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data. However, there is no concrete verification procedure presented. Sun et al. [8－10] presented a data-flow analysis framework for detecting data-flow anomalies such as missing data, redundant data, and potential conflicts of data. In addition, they provided several analysis algorithms; however, the work is done only based on read and initial write data operations.

Chapter 3. Process Modeling

3.1. Process Specifications

Based on BPMN, a process consists of a network of activities designed to produce a product or service for a particular customer or market. A process specification, a formalized view of a business process, defines a set of linked (parallel and/or sequential) activities across time and space, with a beginning and an end, associated with clear defined inputs and outputs respectively. Each activity takes a subset of process input(s) or output(s) of previous activity(ies) and transforms them to create the data for later use or as process outputs. The inputs or outputs of a process, as well as the intermediate outputs of activities, are called artifacts. Thus, a process specification contains not only the control flow but also the artifact flow of a business process. Definition 3.1 is a formal description of a business process.

1896

Definition 3.1. A process specification is a tuple $BP = (G, VT, D, I_W, O_W)$, where

— $G = (V, E)$, representing the control flow, is a directed, connected, and acyclic graph, where V is a set of vertices of which each represents an activity and $E \subset V \times V$ is a set of directed edges indicating the precedence relation between two activities.

— $VT : V \rightarrow T$ is a type function that maps each activity into one of the activity types defined as $T = \{Task, SubProcess, ProcessStart, ProcessEnd, AndSplit, AndJoin, XorSplit, XorJoin, LoopStart, LoopEnd\}$.

Activities whose types are $Task$ are called task activities while the others are called control activities.

— D is a set of artifacts used in the process.

— $I_W \subset D$, a subset of D, denotes the set of process inputs.

— $O_W \subset D$, a subset of D, denotes the set of process outputs.

3.2. Control Flow Specification

3.2.1. Activities and Control Blocks

An activity in a business process might be atomic or non-atomic (compound). An atomic activity is the smallest unit of work that is scheduled by a workflow engine during process enactment and cannot be decomposed. A *sub-process* included within a process is represented as a compound activity. Atomic activities are classified into two major types, *Task* activities and *control* activities, based on their functionalities. A task activity performs a piece of processing steps. Control activities are pairwise activities representing a group of activities, called a control block. There are eight types (four pairs) of primitive control activities in general: (1). ProcessStart (PS) and ProcessEnd (PE) are unique control activities of a process that represent the start and the end of the process respectively (2). AndSplit (AS) and AndJoin (AJ) are control activities for constructing a parallel structure (3). XorSplit (XS) and XorJoin (XJ) are control activities for constructing a branch structure. (4). LoopStart (LS) and LoopEnd (LE) are control activities representing an iteration structure.

Figure 3.1 shows the corresponding notations of control activities, task activity, sub-process activity, and the precedence relation [37].

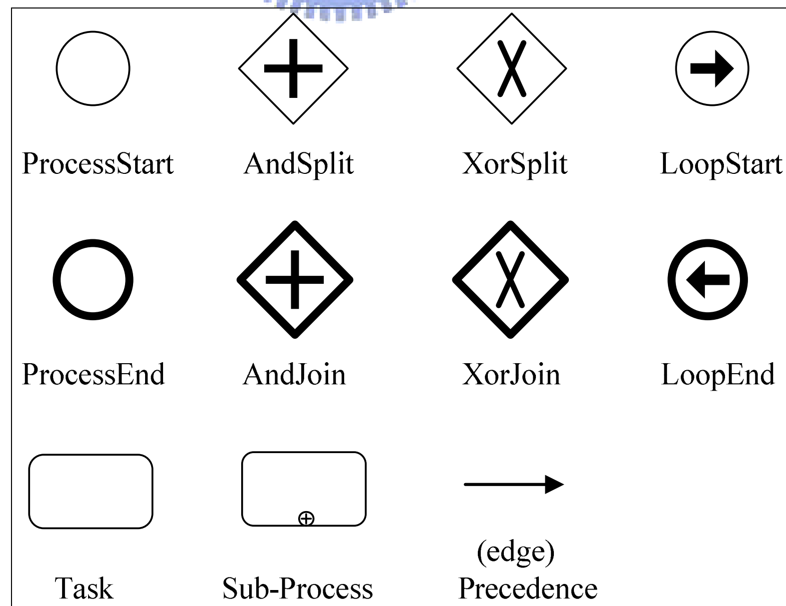


Figure 3.1. Notations of Control Flow Graph.

With typed activities and their precedence relation, various kinds of control structures can be constituted. In this dissertation, the four primitive control structures, "sequential", "parallel branch", "conditional branch" and "iterative structure", defined in [1] are concerned.

Figure 3.2 shows these control structures to construct a process respectively.

- *Sequential Block*: the activities within this structure are executed sequentially under a single thread. The main characteristic is that the target activity cannot execute until its preceding activity completes. In other words, the completion of a target activity triggers the execution of its succeeding activity.

- *Iteration Control Block*: The activities within the block enclosed by *LoopStart* and *LoopEnd* control activities are executed repetitively until certain conditions are met. There are two kinds of iteration control blocks: *while loop* and *repeat-until loop*. A while loop checks the conditions before the first activity within the block is executed and thus, it is often also known as a pre-test loop. On the contrary, a repeat-until loop, also known as a post-test loop, tests the conditions after the activities within the block are executed.

- *AND Control Block*: All outflows of an *AndSplit* activity are executed in parallel, and finally converge into an *AndJoin* activity synchronously.

- *XOR (eXclusive OR) Control Block*: An *XorSplit* activity decides one among multiple alternative outflows (process branches) to continue. These branches converge to a single *XorJoin* activity. No synchronization is required since only one thread is chosen for execution.

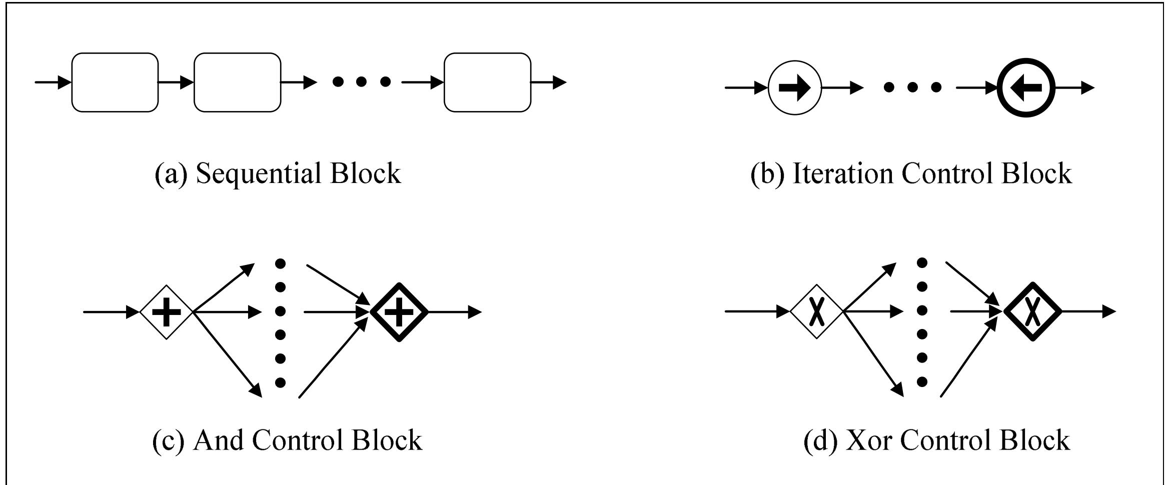


Figure 3.2. Four Primitive Types of Control Structures.

According to our notations, the control flow $G=(V,E)$ of a process specification is *well-formed* if the following constraints hold:

— G has a unique *Process Start* vertex v_{ps} of type *ProcessStart*, which has no incoming edge and one outgoing edge.

■ $\exists! v_{ps} : VT(v_{ps}) = ProcessStart \rightarrow InDegree(v_{ps}) = 0 \wedge OutDegree(v_{ps}) = 1$

— G has a unique *Process End* vertex v_{es} of type *ProcessEnd*, which has one incoming edge and no outgoing edge.

■ $\exists! v_{es} : VT(v_{es}) = ProcessStart \rightarrow InDegree(v_{es}) = 1 \wedge OutDegree(v_{es}) = 0$

— Vertices of type *Task*, *LoopStart*, and *LoopEnd* have one incoming edge and one outgoing edge.

■ $\forall v_i : (VT(v_i) = Task \vee LoopStart \vee LoopEnd) \rightarrow InDegree(v_i) = OutDegree(v_i) = 1$

— Vertices of type *AndSplit* and *XorSplit* have one incoming edge and more than one outgoing edge.

■ $\forall v_{bs} : (VT(v_{bs}) = AndSplit \vee XorSplit) \rightarrow InDegree(v_{bs}) = 1 \wedge OutDegree(v_{bs}) > 1$

— Vertices of type *AndJoin* and *XorJoin* have more than one incoming edge and one outgoing edge.

- $\forall v_{bj} \in (VT(v_{bj}) = AndJoin \vee XorJoin) \rightarrow InDegree(v_{bj}) > 1 \wedge OutDegree(v_{bj}) = 1$

— Any two control blocks can be nested but not overlapped.

- $\forall b_1 = [v_i, v_j], b_2 = [v_x, v_y], b_1 \neq b_2 \rightarrow b_1 \subset b_2 \vee b_1 \subset b_2 \vee b_1 \cap b_2 = \varnothing$

3.2.2. Relations among Activities and Control Blocks

In this session, relations among activities and control blocks are identified as follows.

Definition 3.2 (Paths).

A *path* from v_1 to v_k is a sequence of vertices $<v_1,...,v_k>$ in a control graph $G = (V, E)$ such that each node is connected to the next vertex in the sequence (the edges (v_i, v_{i+1}) for $i=1,2,...,k-1$ are in the edge set E). A *path* from v_1 to v_k is denoted by $Path(v_1, v_k)$.

Definition 3.3 (Reachability).

Given two vertices, u and v, $IsReachable(u,v)$ is a Boolean function that indicates whether if there exists a path from u to v.

$$\forall u, v \in V,\ IsReachable(u,v) = true \leftrightarrow \exists Path(u,v) \vee u = v$$

Definition 3.4 (Predecessors and Successors).

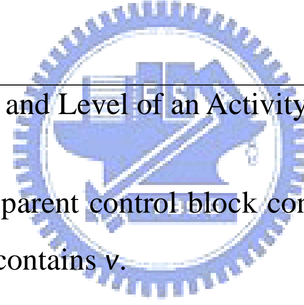
$$V_v^{IsPredecessor} = \{u \in V \mid (u,v) \in E\}$$

$$V_v^{\overline{IsPredecessor}} = \{t \in V \mid t \in V_v^{IsPredecessor} \vee (\exists u \in V_v^{IsPredecessor} : t \in V_u^{\overline{IsPredecessor}}\}$$

$$V_v^{IsSuccessor} = \{u \in V \mid (v,u) \in E\}$$

$$V_v^{\overline{IsSuccessor}} = \{t \in V \mid t \in V_v^{IsSuccessor} \vee (\exists u \in V_v^{IsSuccessor} : t \in V_u^{\overline{IsSuccessor}}\}$$

$V_v^{IsPredecessor}$ comprises the set of vertices which are the source of an edge with destination vertex $v \in V$. Each element u in $V_v^{IsPredecessor}$ is called a *direct predecessor* of the vertex and is denoted by $u \to v$. $V_v^{\overline{IsPredecessor}}$ denotes the transitive closure of $V_v^{IsPredecessor}$. $V_v^{\overline{IsPredecessor}}$ comprises those vertices that are reachable from v. Each element u in $V_v^{\overline{IsPredecessor}}$ is called a *predecessor* of v and is denoted by $u \twoheadrightarrow v$. $V_v^{IsSuccessor}$ and its transitive closure $V_v^{\overline{IsSuccessor}}$ are defined similarly.



Definition 3.5 (Ancestor Blocks and Level of an Activity).

$\forall v \in V$, let $v.PB$ denote the parent control block containing v. $\overline{AncestorBlock}$ comprises the set of all control blocks that contains v.

$$\overline{AncestorBlock}(v) = \{b \mid b = v.PB \vee (b \in \overline{AncestorBlock}(v.PB.startVertex)\}$$

In addition, the cardinality of $\overline{AncestorBlock}(v)$ identifies the nested level of v.

$$Level(v) = \begin{cases} \left|\overline{AncestorBlock}(v)\right| & \text{if } v \in V \\ \left|\overline{AncestorBlock}(v.StartVertex)\right| & \text{if } v \text{ represents a control block} \end{cases}$$

Definition 3.6 (Common Ancestor Blocks and Nearest Common Ancestor Blocks).

Given a set of vertices, $v_1, \ldots, v_n$, B_i is a *common ancestor block* of $v_1, \ldots, v_n$ if and only if the

following holds:

$$B_i \in \bigcap_{i=1}^{n} \overline{AncestorBlock}(v_i)\text{, denoted by } B_i \in CAB(v_1,\ldots,v_n)\,.$$

B_i is the *Nearest common ancestor* of $v_1,\ldots,v_n$ if and only if the following holds:

$$\forall B_j \in CAB(v_1,\ldots,v_n) \wedge B_j \neq B_i : Level(B_j) < Level(B_i)\text{, denoted by } NCAB(v_1,\ldots,v_n) = B_i\,.$$

Definition 3.7 (Parallel Activities).

Given two vertices, u and v, $IsParallel(u,v)$ is a Boolean function to represent if u and v might be executed in parallel within a workflow instance.

$$IsParallel(u,v) = true \Leftrightarrow NCAB(u,v).Type = \text{"}AND\text{"} \wedge \neg IsReachable(u,v) \wedge \neg IsReachable(v,u)$$

$IsParallel(u,v) = true$, denoted as $u \oplus v$, indicates that u and v might be executed in parallel and v is called a parallel activity of u.

Definition 3.8 (Exclusive Activities).

Given two vertices, u and v, $IsExclusive(u,v)$ is a Boolean function to represent some XOR characteristics of u and v. Within a workflow instance, if u is selected for execution then v won't be selected for execution and vice versa.

$$IsExclusive(u,v) = true \Leftrightarrow NCAB(u,v).Type = \text{"}XOR\text{"} \wedge \neg IsReachable(u,v) \wedge \neg IsReachable(v,u)$$

$IsExclusive(u,v) = true$, denoted as $u \otimes v$, indicates that at most one of u and v can be selected for execution and v is called an exclusive activity of u.

Definition 3.9 (Companion Activities).

Given two vertices, u and v, $IsCompanion(u,v)$ is a Boolean function which indicates whether if u is selected for execution then v will always be selected for execution and vice versa.

$$IsCompanion(u,v) = true \Leftrightarrow$$

$$\begin{cases} \forall b \in \overline{AncestorBlock(u)} \cup \overline{AncestorBlock(v)} \setminus CAB(u,v) : b.type = "AND" & \text{if } IsReachable(u,v) \vee IsReachable(v,u) \\ \forall b \in \overline{AncestorBlock(u)} \cup \overline{AncestorBlock(v)} \setminus CAB(u,v) \cup \{NCAB(u,v)\} : b.type = "AND" & \text{otherwise} \end{cases}$$

$IsCompanion(u,v) = true$, denoted as $u \odot v$, indicates that neither of u and v or both of them will be selected for execution and v is called a companion activity of u.

3.3. Artifact Flow Specification

Currently, as identified in [7], there are three major implementation models for artifact flow: explicit data flow, implicit data flow through control flow, and implicit data flow through a process data store. In this dissertation, we adopt the model of implicit data flow through a common process data store. The exchanges of artifacts between tasks are passed through global variables stored in a common database. In a workflow, some activities store their output artifacts in the database, and their following activities may access these artifacts later. The activities in our model are regarded as black boxes, i.e., their internal computations are not visible. Neither are the intermediate execution states. Thus, the artifact usages of an activity are identified through the inputs/outputs of the activity.

3.3.1. Artifacts and Artifact Operations

Artifacts are information entities involved in a process, including the input data to the process, the intermediate data produced within the process, and the final output data from the process. An artifact is an atomic data item (e.g. a number, a character string, or an image) or a collection of atomic data items (e.g. a document). Intuitively, all artifacts participating in a workflow execution must be pre-defined in process specifications. Each artifact contains a set of legal operations for its internal data. An activity designed to manipulate a certain artifact can work only with that

artifact's legal operations. From the data storage point of view, every artifact operation can be regarded as one of the following operations, regardless of its semantic meaning.

- *Initialize*: all definition operations, e.g. "fill in", "create", and "define" operations.
- *Read*: all reference operations, e.g. "use", "fetch", "select", and "retrieve" operations.
- *Update*: all modification operations, e.g. "write", "change", and "update" operations.
- *Destroy*: all deletion operations, e.g. "remove", "erase", "cancel", and "discard" operations.

In general, an *Initialize* operation is used to create an artifact instance in a process. *Read* and *Update* operations are then used to access the instance. Finally, a *Destroy* operation is used to delete the artifact instance. *Destroy* operations are applied for temporary artifacts created during in workflow execution, but may not strict for all artifacts.

Figure 3.3 shows the state diagram of an artifact with above four kinds of operations. There are four states, "Uninitialized", "Initialized", "Updated", and "Read". 'Uninitialized' represents the initial state of an artifact. "Initialized", "Updated", and "Read" represent the states after an *Initialize*, *Update*, and *Read* operation is performed respectively. In addition, the state of an artifact resets to "Uninitialized" after a *Destroy* operation.

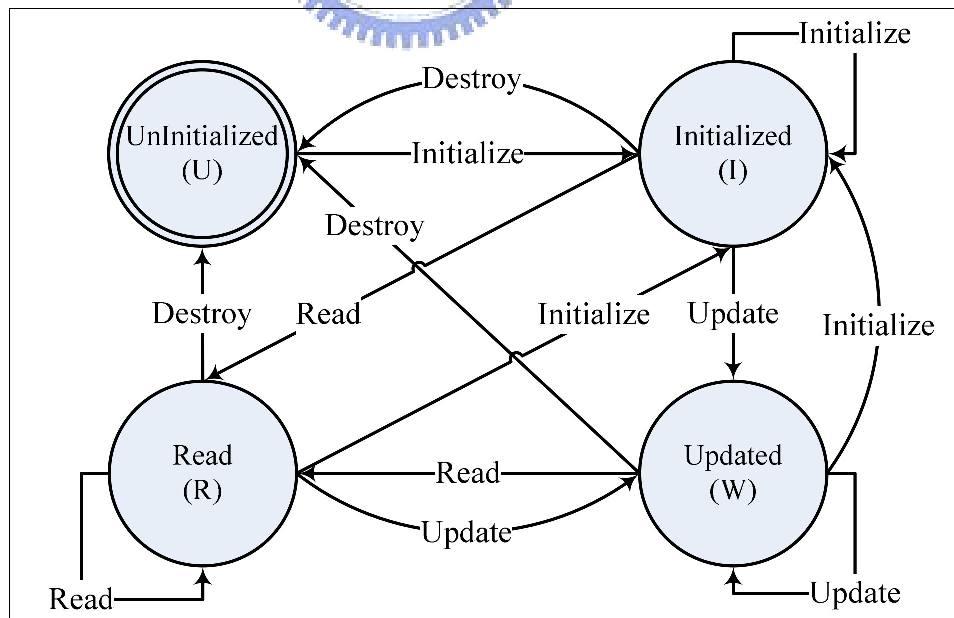


Figure 3.3. The State Diagram of an artifact.

3.3.2. Artifact Flow and Artifact Usages

To simplify the discussion of artifact usages, now a formal and complete definition of a task/control activity is shown below:

Definition 3.10 (Task/Control Activities).

An task/control activity is a tuple $v = (AT_v, SC_v, EC_v, RC_v, I_v, O_v, AS_v)$, where

- AT_v represents the type of the activity.
- SC_v, EC_v, and RC_v are sets of logical expressions which are evaluated by a workflow engine.
  - SC_v is the set of pre-conditions of which each is evaluated to decide whether an activity within a process instance can be started (only used by task activities).
  - EC_v is the set of post-conditions of which each is evaluated to decide whether an activity within a process instance is completed (only used by task activities).
  - RC_v is the set of routing conditions of which each is evaluated to decide the sequence of activity execution within a process (only used by control activities).
- I_v, the input set, identifies all the artifacts required to be accessed by the activity.
  - For a task activity, I_v contains all the artifacts required for computation.
  - For a control activity, I_v contains all the artifacts required for evaluating the routing conditions.
- O_v, the output set, identifies all the artifacts produced, updated, or destroyed after executing the activity. O_v is divided into two disjoint subsets, O_v^+ and O_v^-, where O_v^+ represents the set of the artifacts initialized or updated by t and O_v^- represents the set of the artifacts destroyed by t.
- AS_v is the activity specification (only used by task activities).

Based on Definition 3.10, a usage relation between an activity and an artifact can be defined as follows: an artifact usage representing the relation between an activity and an artifact is defined as follows:

Definition 3.11 (Consumer, Producer, Updator, and Destroyer Activities of an Artifact).

For a given artifact d, the memberships between artifact d and I_v, O_v^+, and O_v^- can be applied for identifying the usage of artifact d at activity v. All the possible usages are categorized as follows:

- if $d \in I_v$ and $\begin{cases} d \notin O_v^+ \\ d \notin O_v^- \end{cases}$, v is called a *Reader (Activity)* of artifact d.
- if $d \in I_v$ and $d \in O_v^+$, v is called an *Updator (Activity)* of artifact d.
- if $\begin{cases} d \in I_v \\ d \notin I_v \end{cases}$ and $d \in O_v^-$, v is called a *Destroyer (Activity)* of artifact d.
- if $d \notin I_v$ and $d \in O_v^+$, v is called a *Producer (Activity)* of artifact d.
- if $d \notin I_v$ and $\begin{cases} d \notin O_v^+ \\ d \notin O_v^- \end{cases}$, v is called an *Irrelevantor (Activity)* of artifact d.

In addition, if $d \in I_v$, v is generally called a *Consumer (Activity)* of artifact d and if $d \in O_v^+$, v is generally called a *Writer (Activity)* of artifact d.

Definition 3.12 (Consumer, Updator, Destroyer and Producer Activity Sets for an Artifact).

- $V_d^{IsConsumer} = \{v \in V \mid d \in I_v\}$ is called the *Consumer Activity Set* of artifact d.
- $V_d^{IsUpdator} = \{v \in V \mid d \in I_v \text{ and } d \in O_v^+\}$ is called the *Updator Activity Set* of artifact d.
- $V_d^{IsDestroyer} = \{v \in V \mid d \in O_v^-\}$ is called the *Destroyer Activity Set* of artifact d.
- $V_d^{IsProducer} = \{v \in V \mid d \notin I_v \text{ and } d \in O_v^+\}$ is called the *Producer Activity Set* of artifact d.

Chapter 4. Artifact Usage Anomalies

4.1. Artifact Usage Anomalies

In a process specification, some of the following three types of anomalies might occur: (1) Missing Production, (2) Redundant Write, and (3) Conflict Write. In the subsections, these anomalies are defined and the corresponding usage patterns that cause the anomalies are identified. Every usage pattern is given a name, description, and formulated detection conditions. Table 4.1 shows the symbols used in usage patterns.

Table 4.1. Symbols Used in Usage Patterns

P_d : a producer ($d \notin I_v$ and $d \in O_v^+$)	$\not{P_d}$: no producer of d exists
C_d : a consumer ($d \in I_v$)	$\not{C_d}$: no consumer of d exists
U_d : a updator ($d \in I_v$ and $d \in O_v^+$)	$\not{R_d}$: no reader of d exists
W_d : a updator ($d \in O_v^+$)	( ) : a control block
R_d : a reader ($d \in I_v$ and $\begin{cases} d \notin O_v^+ \\ d \notin O_v^- \end{cases}$)	($\otimes$): XOR control block
$\twoheadrightarrow$: reachable	($\oplus$): AND Control block

1896

4.1.1. Missing Production Anomalies

A missing production anomaly occurs when an artifact is consumed before it is produced or after it is destroyed. Formally speaking, given an activity v and an artifact d such that v is a consumer of d, a missing production anomaly occurs if d is not produced or is destroyed when v is selected for execution. To formulate this type of anomaly, the propagation of an artifact is introduced in Definition 4.1.

Definition 4.1 (Propagation of Artifacts to an Activity).

Given an activity v, let a preceding execution order to v denote an execution order leading to v without parallel activities of v, i.e., only consisting of the predecessors of v. Given an artifact d, if there exists at least one preceding execution order to v such that d is produced but not destroyed (i.e., d is not in *Uninitialized* state), we call d can be *propagated* from v's predecessors to v. The propagation of artifact d regarding only the preceding execution orders to v is called *preceding propagation* of d to v and can be classified into three cases: *no preceding propagation*, *conditional preceding propagation*, and *unconditional preceding propagation*.

No preceding propagation indicates that d is always *Uninitialized* for all preceding execution orders to v. Conditional preceding propagation indicates whether d is *Uninitialized* depends on the preceding execution orders to v taken. Unconditional preceding propagation denotes that d is *Uninitialized* for all preceding execution orders to v.

Based on Definition 4.1, let AA_v contains all the artifacts which can be propagated from the predecessors of v. AA_v can be divided into two disjoint subset, AA_v^u and AA_v^c, where AA_v^u contains the artifacts unconditional propagated from the predecessors of v and AA_v^c contains the artifacts propagated from the predecessors of v conditionally.

The causes of missing production anomalies can be classified into three categories: *No Preceding Propagation*, *Conditional Preceding Propagation*, and *Uncertain Preceding Propagation*. Intuitively, if $v \in V_d^{IsConsumer}$ and $d \notin AA_v$ hold, a missing production anomaly might occur due to *No Preceding Propagation* of d to v. Similarly, if $v \in V_d^{IsConsumer}$ and $d \in AA_v^c$ hold, a missing production anomaly might occur owning to *Conditional Preceding Propagation* of d to v. Furthermore, consider parallel activities of v, even though $v \in V_d^{IsConsumer}$ and $d \in AA_v^u$ hold, a missing production anomaly might still occur if there exists a parallel activity which destroys d and this cause is classified as *Uncertain Preceding Propagation*.

For each cause of the missing production anomaly, the possible usage patterns are characterized by its name, description, and required condition as followings:

(1). No Preceding Propagation: $v \in V_d^{IsConsumer} \wedge d \notin AA_v$

Usage Pattern 1: $\not{P_d} \twoheadrightarrow C_d \twoheadrightarrow \not{P_d}$

- **Name:** *No Production*
- **Description**: Artifact d has at least one consumer activity v; however, no producer activity of d exists in the process.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge V_d^{IsProducer} = \varnothing$

Usage Pattern 2: $\not{P_d} \twoheadrightarrow C_d \twoheadrightarrow P_d \twoheadrightarrow$

- **Name:** *Delayed Production*
- **Description:** Artifact d has a consumer activity v which precedes every producer activity of d.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge (V_v^{\overline{IsPredecessor}} \cap V_d^{IsProducer}) = \varnothing \wedge (V_v^{\overline{IsSuccessor}} \cap V_d^{IsProducer}) \neq \varnothing$

Usage Pattern 3: $\twoheadrightarrow P_d \twoheadrightarrow D_d \twoheadrightarrow C_d \twoheadrightarrow$

- **Name:** *Early Destruction*
- **Description:** Artifact d is produced and then destroyed before it is consumed.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \notin AA_v \wedge (V_v^{\overline{IsPredecessor}} \cap V_d^{IsProducer} \cap V_d^{IsDestroyer}) \neq \varnothing$

Usage Pattern 4: $\not{P_d} \twoheadrightarrow (C_d \otimes P_d) \twoheadrightarrow$

- **Name:** *Exclusive Production*
- **Description:** Given two exclusive activities v and u such that v is a consumer of artifact d and u is a producer of d. Due to the characteristic of exclusive activities, only one of v and u might be selected for execution. Although u is a producer of d, it makes no contribution to the propagation of d to v and thus a missing production anomaly occurs if artifact d cannot be propagated from the predecessors of v.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \notin AA_v \wedge (V_v^{IsExclusive} \cap V_d^{IsProducer}) \neq \varnothing$

Usage Pattern 5: $\not{P_d} \twoheadrightarrow (C_d \oplus P_d) \twoheadrightarrow$

- **Name:** *Uncertain Production*
- **Description:** Given two exclusive activities v and u such that v is a consumer of artifact d and u is a producer of d. Due to the race hazard of parallel activities, v might be executed before u. Therefore, u may not make contribution to the propagation of d for v and consequently, a missing production anomaly occurs if artifact d cannot be propagated from the predecessors of v.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \notin AA_v \wedge (V_v^{IsParallel} \cap V_d^{IsProducer}) \neq \varnothing$

(2). Conditional Preceding Propagation: $v \in V_d^{IsConsumer} \wedge d \in AA_v^c$

Whether d is propagated depends on the preceding path to v taken. Consequently, a missing production anomaly occurs when those preceding paths to v such that d is not propagated are taken.

Usage Pattern 6: $\not{P_d} \twoheadrightarrow (P_d \otimes \not{P_d}) \twoheadrightarrow C_d \twoheadrightarrow$

- **Name:** *Conditional Production*
- **Description:** Artifact d is produced conditionally before a consumer activity of d.

- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \in AA_v^c$

Usage Pattern 7: $\twoheadrightarrow P_d \twoheadrightarrow (D_d \otimes \not{D}_d) \twoheadrightarrow C_d \twoheadrightarrow$

- **Name:** *Conditional Destruction*
- **Description:** Artifact d is destroyed conditionally before a consumer activity of d**.**
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \in AA_v^c$

(3). Uncertain Preceding Propagation: $v \in V_d^{IsConsumer} \wedge d \in AA_v^u$

Usage Pattern 8: $\twoheadrightarrow P_d \twoheadrightarrow (D_d \oplus C_d) \twoheadrightarrow$

- **Name:** *Uncertain Destruction*
- **Description:** Given two parallel activities v and u such that v is a consumer of artifact d and u is a destroyer of d. Due to the race hazard of parallel activities, v might be executed before u. Therefore, even though d is unconditional propagated from the predecessors of v, d might be destroyed by u before v is executed and a missing production anomaly occurs.
- **Conditions:** $\exists v \in V_d^{IsConsumer} \wedge d \in AA_v^u \wedge (V_v^{IsParallel} \cap V_d^{IsDestroyer}) \neq \emptyset$

Theorem 1 (Missing Production Verification).

A process BP is free from missing production anomalies if the following condition holds: $\forall v \in V$, $\forall d \in I_v$: $d \in AA_v^u$ and $(V_v^{IsParallel} \cap V_d^{IsDestroyer}) = \emptyset$.

Proof: This theorem is proofed by contradiction as follows. Support that there exists a missing production anomaly in BP. It indicates that there exists an activity $v \in V$, an artifact $d \in I_v$, and an execution order Γ such that $v \in \Gamma$ and d is *Uninitialized* when v is selected for execution. However, $d \in AA_v^u$ implies that d will be always propagated from the predecessors

of v. Furthermore, $(V_v^{IsParallel} \cap V_d^{IsDestroyer}) = \varnothing$ implies that no parallel activity of v will affect the propagation of d from the predecessors of v. Thus, d will always be propagated to v regardless the execution order leading to v, that is, Γ does not exist. This contradicts the hypothesis and thus, Theorem 1 holds.

4.1.2. Redundant Write Anomalies

A redundant write anomaly occurs when an artifact is written (produced or updated) by an activity but the artifact is neither required by succeeding activities nor a member of the process outputs. Redundancy is not an error; nevertheless, it causes inefficiency. To formulate this type of anomaly, the set of artifacts *unused* to an activity is introduced in Definition 4.2.

Definition 4.2 (The Set of Artifacts Unused before an Activity).

Given an activity v and an artifact d, if there exists at least one preceding execution order to v such that d is written but not consumed when v is selected for execution, d is called *unused* for the predecessors of v or simply called *unused* before v. Intrusively, if artifact d is unused for the predecessors of the $Process\ End$ vertex and is not a member of the set of process outputs, a redundant write anomaly occurs. There are two cases: *completely unused* and *conditionally unused*. Completely unused indicates that d is unused for all preceding paths to v. Conditionally unused indicates whether d is unused depends on the preceding path to v taken.

Let NC_v contain all the artifacts unused for the predecessors of v. NC_v can be divided into two disjoint subset, NC_v^u and NC_v^c. NC_v^u contains the artifacts which are completely unused and NC_v^c contains the artifacts which are conditional unused.

Based on Definition 4.2, redundant update anomalies can be classified into two categories: *Explicit Redundant Update* and *Potential Redundant Update*. Intuitively, for every artifact

$d \in NC^{u}_{ProcessEnd}$ such that $d \notin O_w$, a redundant update anomaly always occurs for artifact d of the process. Similarly, for every artifact $d \in NC^{c}_{ProcessEnd}$ such that $d \notin O_w$, a redundant update anomaly might occur for artifact d depending on the execution paths taken.

For each category of the redundant write anomaly, the possible usage patterns are characterized by its name, description, and required condition as followings:

(1). Explicit Redundant Update

Usage Pattern 9:

$$
\begin{aligned}
&\not{C}_d \twoheadrightarrow W_d \twoheadrightarrow \not{C}_d \\
&\twoheadrightarrow C_d \twoheadrightarrow W_d \twoheadrightarrow \not{C}_d \\
&\twoheadrightarrow (C_d \otimes W_d) \twoheadrightarrow \not{C}_d
\end{aligned}
$$

- **Name:** *No Consumption After Last Write*
- **Description:** For an artifact d not belonging to the process outputs, when d is written by an activity v and the artifact is unused for all succeeding activities of v, a redundant update always occurs for the artifact.
- **Conditions:** $\exists d \in NC^{u}_{ProcessEnd} : d \notin O_w$

(2). Potential Redundant Update

Usage Pattern 10:

$$
\begin{aligned}
&\twoheadrightarrow W_d \twoheadrightarrow (C_d \otimes \not{C}_d) \\
&\twoheadrightarrow (C_d \oplus W_d) \twoheadrightarrow \not{C}_d
\end{aligned}
$$

- **Name:** *Conditional Consumption After Last Write*
- **Description:** For an artifact d not belonging to the process outputs, when d is written by an activity v and the artifact is conditionally unused for some succeeding activities of v, a redundant update might occurs.
- **Conditions:** $\exists d \in NC^{c}_{ProcessEnd} : d \notin O_w$

Theorem 2 (Redundant Write Verification).

A process BP is free from redundant write anomalies if $NC_{ProcessEnd} \setminus O_w = \varnothing$ holds:

Proof: $NC_{ProcessEnd} \setminus O_w = \varnothing$ indicates that every artifact d is a process output ($d \in O_w$) or is read after its last write for all possible (preceding) execution orders leading to *Process End* vertex. ($d \notin NC_{ProcessEnd}$). Therefore, no redundant write anomaly exists if $NC_{ProcessEnd} \setminus O_w = \varnothing$ holds.

4.1.3. Conflict Write Anomalies

A multiple parallel productions anomaly occurs when more than one activity tries to initialize the same artifact in parallel. When this anomaly occurs, different versions of an artifact will exist.

A conflict update anomaly occurs when more than one activity in parallel updates the same artifact.

Usage Pattern 11: $\twoheadrightarrow (P_d \oplus P_d) \twoheadrightarrow$

- **Name:** *Multiple Parallel Productions*
- **Description:** More than one activity initializes the same artifact in parallel.
- **Conditions:** $\exists v \in V_d^{IsProducer} \wedge u \in (V_d^{IsProducer} \cap V_v^{IsParallel})$

Usage Pattern 12: $\twoheadrightarrow (U_d \oplus U_d) \twoheadrightarrow$

- **Name:** *Multiple Parallel Updates*
- **Description:** More than one activity updates the same artifact in parallel.
- **Conditions:** $\exists v \in V_d^{IsUpdator} \wedge u \in (V_d^{IsUpdator} \cap V_v^{IsParallel})$

Usage Pattern 13: $\twoheadrightarrow (R_d \oplus U_d) \twoheadrightarrow$

- **Name:** *Parallel Read and Update*
- **Description:** Two activities perform read and update respectively on the same artifact concurrently.
- **Conditions:** $\exists v \in (V_d^{IsReader}) \wedge u \in (V_d^{IsUpdator} \cap V_v^{IsParallel})$

Theorem 3 (Conflict Writes Verification).

A process BP is free from conflict writes anomalies if for any two parallel activities v and u, $(O_v^+ \setminus I_v) \cap (O_u^+ \setminus I_u) = \emptyset$, $(O_v^+ \cap I_v) \cap (O_u^+ \cap I_u) = \emptyset$, $I_v \cap O_u^+ = \emptyset$, and $I_u \cap O_v^+ = \emptyset$ hold.

Proof: if for any two parallel activities v and u such that $(O_v^+ \setminus I_v) \cap (O_u^+ \setminus I_u) = \emptyset$, then no two activities initializes the same artifact in parallel. If $(O_v^+ \cap I_v) \cap (O_u^+ \cap I_u) = \emptyset$, then no two activities updates the same artifact in parallel. Furthermore, $I_v \cap O_u^+ = \emptyset$ and $I_u \cap O_v^+ = \emptyset$ indicate that no two activities perform read and update respectively on the same artifact. Thus, BP is free from conflict writes anomalies.

4.1.4. Summary of Usage Patterns Causing Artifact Usage Anomalies

Table 4.2 shows the summary of usage patterns for each type of artifact usage anomaly.

Table 4.2: Summary of Usage Patterns Causing Artifact Usage Anomalies

Type	**Case**	**Pattern**
Missing Production	*No Production*	$\cancel{P_d} \twoheadrightarrow C_d \twoheadrightarrow \cancel{P_d}$
	Delayed Production	$\cancel{P_d} \twoheadrightarrow C_d \twoheadrightarrow P_d \twoheadrightarrow$
	Early Destruction	$\twoheadrightarrow P_d \twoheadrightarrow D_d \twoheadrightarrow C_d \twoheadrightarrow$
	Exclusive Production	$\cancel{P_d} \twoheadrightarrow (C_d \otimes P_d) \twoheadrightarrow$
	Conditional Production	$\cancel{P_d} \twoheadrightarrow (P_d \otimes \cancel{P_d}) \twoheadrightarrow C_d \twoheadrightarrow$
	Conditional Destruction	$\twoheadrightarrow P_d \twoheadrightarrow (D_d \otimes \cancel{D_d}) \twoheadrightarrow C_d \twoheadrightarrow$
	Uncertain Production	$\cancel{P_d} \twoheadrightarrow (C_d \oplus P_d) \twoheadrightarrow$
	Uncertain Destruction	$\twoheadrightarrow P_d \twoheadrightarrow (D_d \oplus C_d) \twoheadrightarrow$
Redundant Write	*No Consumption After Last Write*	$\cancel{C_d} \twoheadrightarrow W_d \twoheadrightarrow \cancel{C_d}$ $\twoheadrightarrow C_d \twoheadrightarrow W_d \twoheadrightarrow \cancel{C_d}$ $\twoheadrightarrow (C_d \otimes W_d) \twoheadrightarrow \cancel{C_d}$
	Conditional Consumption After Last Write	$\twoheadrightarrow W_d \twoheadrightarrow (C_d \otimes \cancel{C_d})$ $\twoheadrightarrow (C_d \oplus W_d) \twoheadrightarrow \cancel{C_d}$
Conflict Write	*Multiple Parallel Productions*	$\twoheadrightarrow (P_d \oplus P_d) \twoheadrightarrow$
	Multiple Parallel Updates	$\twoheadrightarrow (U_d \oplus U_d) \twoheadrightarrow$
	Parallel Read and Update	$\twoheadrightarrow (R_d \oplus U_d) \twoheadrightarrow$

Chapter 5. Algorithms to Detecting Artifact Usage Anomalies

This chapter presents a solution for detecting artifact usage anomalies in a process specification. To simplify the discussion, our solution is divided into two algorithms: traversal algorithm and detection algorithm. The traversal algorithm is applied firstly to transform the control graph of a process for facilitating the presentation of the detection algorithm. The detection algorithm to artifact usage anomalies is then applied on the transformed structure.

5.1. The Traversal Algorithm

From the top-level of view, a well-formed control flow can be deemed as a sequence of task activity and top-level control blocks. Thus, an entire process can be deemed as a sequence of nodes, where each node may present a task activity or a control block. The same perspective can be applied to the branches of a control block. Based on this perspective, a control flow graph can be recursively transformed into a sequence of nodes.

Thus, for an input process schema, the traversal algorithm begins by traversing the main sequence enclosed by the start vertex and the end vertex of the process. The traversal algorithm is recursively applied until every task activity and control block in each level are processed.

Besides, the traversal algorithm also transforms each iteration control block into a corresponding XOR control block during the analysis of artifact usage anomalies. Figure 5.1 and Figure 5.2 show the transformation of a loop with at-least-once iteration and zero iteration respectively.

Transformation of a Repeat-Until Loop

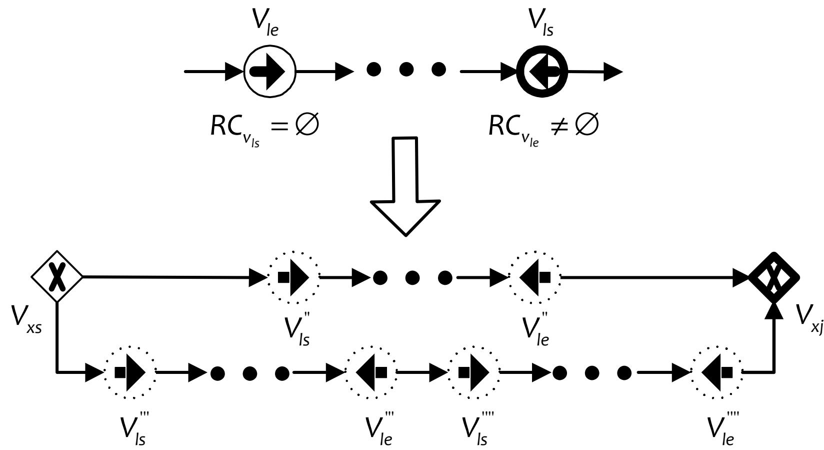


Figure 5.1: Transform a Repeat-Until Loop.

Transformation of a While Loop

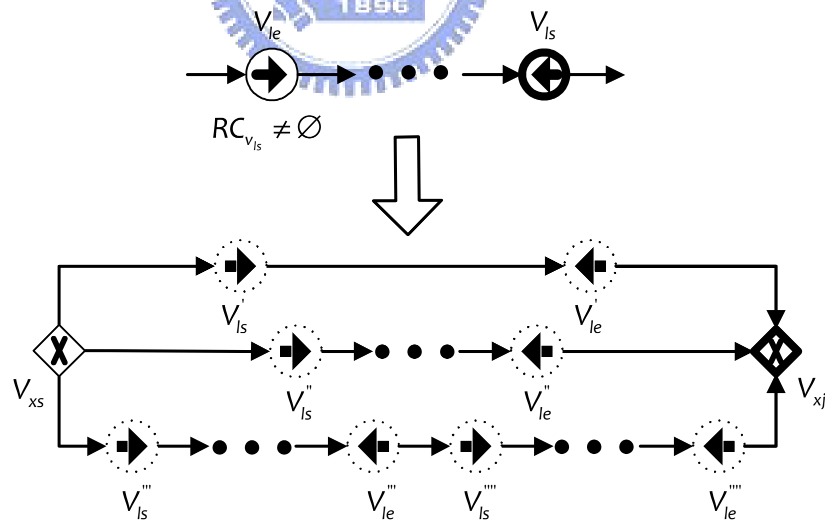


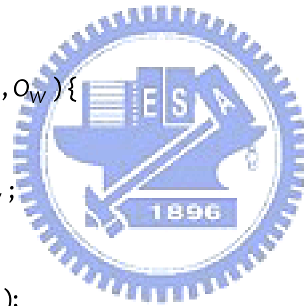
Figure 5.2: Transform a While Loop.

```
Algorithm ControlGraphToSequence(G, v, level) {
//Input:  G=(V,E): a directed connected graph
//        v: a vertex of G representing the next vertex to traverse
//        level: nested level
//Output:   S:   a structure containing a sequence of nodes (a node can represent a task or a control block)
//          S.startVertex     :corresponding vertex in G for the beginning of s
//          S.endVertex       :corresponding vertex in G for the end of s
//          S.nodes           :nodes collection (an ordered set of nodes)
    sequence.startVertex=currentVertex=v;
    sequence.level=level;
    while (currentVertex!=null) {
        currentVertex.ownerSequence=sequence;
        switch (currentVertex.type) {
            case "Task":
                sequence.nodes.append(currentVertex);
                nextVertex=currentVertex.next;
                break;
            case "ProcessStart", "AndSplit", "XorSplit", "LoopStart":
                if (currentVertex.type=="LoopStart") {
                    //Transform loop to corresponding XOR control block
                    //based on Figure 5.1 and 5.2
                }
                newNode.type=currentVertex.type;
                newNode.startVertex=currentVertex;
                for each edge (currentVertex, w)∈ E  {
                    //recursively transform every branch within a control block
                    subSequence= ControlGraph2Sequence(G, w, level+1);
                    subsequence.parentBlock = newNode;
                    //collect every subSequence (corresponding to each branch)
                    newNode.subSequences.append(subSequence);
                }
                newNode.endVertex=subSequence.endVertex.next;
                sequence.nodes.append(newNode);
                nextVertex=newNode.endVertex.next;
                break;
            case "ProcessEnd", "LoopEnd", "AndJoin", "XorJoin":
```

```
            exit while;
        }
        previousVertex=currentVertex;     //remember last traversed vertex
        currentVertex=nextVertex;           //continue to traverse next node
    }
    sequence.endVertex=previousVertex;
    return sequence
}
```

5.2. The Detection Algorithm

The detection algorithm is subdivided into several sub-algorithms described in subsections.



Algorithm AnalyzeProcess (G , D, I_W , O_W) {

 S = ControlGraphToSequence(G);

 // $O_{S.startVertex} = I_W$; // $I_{S.endVertex} = O_W$;

 DetectMissingProduction (S , I_W);

 DetectRedundantWrite(S , I_W , O_W);

 DetectConflictWrites(S , $\varnothing$);

}

5.2.1. Method for Detecting Missing Production Anomalies

5.2.1.1. Calculation of Propagated Artifacts from the Predecessors

Given a sequence S of the input process schema and an activity v of S, Let $S.AA_v$ denote the set of artifacts propagated from the predecessors of v and $S.AA'_v$ be the set of artifacts of which each can be propagated to the direct successors of v after execution of v. Initially, $S.AA_{S.startVertex} = I_w$ if S is the top level sequence. During the traversal of the sequence S, $S.AA$ is calculated after every traversed node n as follow.

- If n represents a task activity v, v has only one direct successor x. $S.AA'_v$ and $S.AA_x$ are calculated as follows:
  - For every destroyed artifact $d \in O_v^-$, remove d from $S.AA_v^u$ and $S.AA_v^c$
  - For every produced artifact $d \in (O_v^+ \setminus I_v)$, add d to $S.AA_v^u$ and remove d from $S.AA_v^c$.

$$S.AA_x = S.AA'_v = \begin{cases} S.AA'^u_v = S.AA_v^u \setminus O_v^- \cup (O_v^+ \setminus I_v) \\ S.AA'^c_v = S.AA_v^c \setminus O_v^- \setminus (O_v^+ \setminus I_v) \end{cases}$$

- If n represents a control block with subsequences $SS = \{SS_i \mid 1 \le i \le k\}$, every vertex within the block will be recursively traversed as follows:
  - *n.startVertex*, the start vertex of the control block, is traversed first.
    - $S.AA'_{n.startVertex} = S.AA_{n.startVertex} = S.AA_{n.startVertex}$ =since *n.startVertex* is a control node.
  - For every subsequence SS_i, recursively applied the same traversed algorithm to calculate each $SS_i.AA$
  - *n.endVertex* is traversed at last and each $SS_i.AA$ is merged according to the type of the control block.

    - If n is an XOR control block,

$$S.AA'_{n.endVertex} = S.AA_{n.endVertex} = \begin{cases} S.AA^u_{n.endVertex} = \bigcap_{i=1}^{k} SS_i.AA^u_{SS_i.endVertex} \\ S.AA^c_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.AA_{SS_i.endVertex} \setminus S.AA^u_{n.endVertex} \end{cases}$$

◆If n is an *And* control block,

$$S.AA'_{n.endVertex} = S.AA_{n.endVertex} = \begin{cases} S.AA^{u}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.AA^{u}_{SS_i.endVertex} \setminus \bigcup_{i=1}^{k} (SS_i.O^{-} \setminus SS_i.AA^{u}_{SS_i.endVertex}) \\ S.AA^{c}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.AA_{SS_i.endVertex} \setminus S.AA^{u}_{n.endVertex} \end{cases}$$

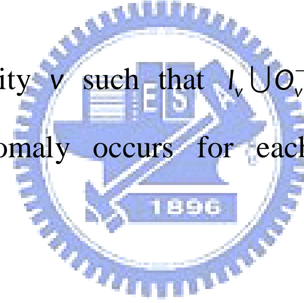
5.2.1.2. Rules for Detecting Missing Production Anomalies

- **No Propagation**

  - When visiting an activity v such that $I_v \cup O_v^- \neq \emptyset$, if $MA_v^u = (I_v \cup O_v^-) \setminus AA_v \neq \emptyset$ a missing production anomaly occurs for each artifact $d \in M_v^u$ due to *No Propagation*.

- **Conditional Propagation**

  - When visiting an activity v such that $I_v \cup O_v^- \neq \emptyset$, if $MA_v^c = (I_v \cup O_v^-) \cap AA_v^c \neq \emptyset$ a missing production anomaly occurs for each artifact $d \in M_v^c$ .due to *Conditional Propagation*.



- **Uncertain Propagation**

  - For an AND control block with subsequences $SS = \{SS_i \mid 1 \leq i \leq k, k \geq 2\}$, before merging $SS_i.AA_{SS_i.endVertex}$ from every subsequence, if $\exists i,j \wedge 1 \leq i,j \leq k \wedge i \neq j \wedge (UP_{SS_i,SS_j} = SS_i.I \cap SS_j.O^- \neq \emptyset)$, a missing production anomaly occurs for each artifact $d \in UP_{SS_i,SS_j}$ due to *Uncertain Propagation*.

5.2.1.3. Algorithm to Detect Missing Production Anomalies

Algorithm DetectMissingProduction (S ,AA_P) {
//**Input:** S: a structure containing a sequence of nodes (of which each is a task or a control block)
// AA_P: the set of *artifacts* propagated from preceding nodes.
//**Output:** messages for detected missing production anomalies.

```
S.AA = AA_P ;   //the set of artifacts propagated from preceding nodes.
S.I = ∅ ;   //the set of artifacts consumed by activities of this sequence.
S.O^- = ∅ ;   //the set of artifacts destroyed by activities of this sequence.
for each n ∈ S {   //process every task or control block
    switch (n.type) {
        case Task:   //a task activity
            v=n.startVertex;
            CheckMissingProduction( v , S.AA ); //check missing production on v
            //Calculate the set of artifacts that can be propagated to the successors of v.
            S.AA = UpdatePropagatedArtifactSet( v , S.AA );
            S.I = S.I ∪ I_v ;   //update the set of artifacts consumed after execution of v.
            S.O^- = S.O^- ∪ O_v^- ; //update the set of artifacts destroyed after execution of v.
            break;
        case default:   //control blocks
            CheckMissingProduction( n.startVertex , S.AA );//check missing production on n.startVertex
            SS = n.subSequences ; //the set of subsequences of the control block.
            for each SS_i ∈ SS {
                //recursively applied the algorithm on every subsequence by passing S.AA as an argument
                DetectMissingProduction ( SS_i , S.AA );
            }

            //check uncertain destruction before merging
            CheckUncertainDestruction( n , SS );
            //merge artifact sets propagated from all subsequences
            AA = MergePropagatedArtifactSets ( n , SS );
    }
}
}
```

Algorithm CheckMissingProduction(v , AA) {

if ($(I_v \cup O_v^-) \neq \emptyset$) {

$$MA_v = \begin{cases} MA_v^u = (I_v \cup O_v^-) \setminus AA \\ MA_v^c = (I_v \cup O_v^-) \cap AA^c \end{cases};$$

```
        for each  d∈ MA^u
            print "a missing production occurs on v due to no propagation of artifact d";
        for each  d∈ MA^c
            print "a missing production occurs on v due to conditional propagation of artifact d";
    }
}
```

Algorithm UpdatePropagatedArtifactSet (v , AA) {

//after traversing v, update the set of propagated artifacts.

//remove artifacts destroyed and add artifacts produced by v.

$$AA_v = \begin{cases} AA_v^u = AA^u \setminus O_v^- \cup (O_v^+ \setminus I_v) \\ AA_v^c = AA^c \setminus O_v^- \setminus (O_v^+ \setminus I_v) \end{cases};$$

return AA_v ;

}

Algorithm MergePropagatedArtifactSets(n , SS) {

if (n.endVetex.type=="XorJoin") {

$$AA^u_{n.endVertex} = \bigcap_{i=1}^{|SS|} SS_i.AA^u ;$$

$$AA^c_{n.endVertex} = \bigcup_{i=1}^{|SS|} SS_i.AA \setminus AA^u_{n.endVertex} ;$$

} **else** {

$$AA^u_{n.endVertex} = \bigcup_{i=1}^{|SS|} SS_i.AA^u \setminus \bigcup_{i=1}^{|SS|} (SS_i.DA \setminus SS_i.AA^u) ;$$

$$AA^c_{n.endVertex} = \bigcup_{i=1}^{|SS|} SS_i.AA \setminus AA^u_{n.endVertex} ;$$

}

$$AA_{n.endVertex} = AA^u_{n.endVertex} \cup AA^c_{n.endVertex} ;$$

return $AA_{n.endVertex}$;

}

Algorithm CheckUncertainDestruction (n , SS) {

```
for each SS_i ∈ SS {
    for each SS_j ∈ SS and SS_j ≠ SS_i {
        UP_{SS_i,SS_j} = SS_i.I ∩ SS_j.O^- ;
        for each d ∈ UP_{SS_i,SS_j} {
            print "A missing production anomaly for artifact d might occur due to uncertain production on
            parallel branches (SS_i, SS_j)."
        }
    }
}
}
```

5.2.2. Method for Detecting Redundant Production/Update Anomalies

5.2.2.1. Calculation of Redundant Production/Update

Given a sequence S of the input process schema and an activity v of S, Let $S.NC_v$ denote the set of artifact unused before v and $S.NC'_v$ denotes the set of artifact unused after executing v. During the traversal of the sequence S, $S.NC$ is calculated on every traversed node n as follow.

- If n represents a task activity v, $S.NC'_v = \begin{cases} S.NC'^u_v = (S.NC^u_v \setminus I_v \setminus O^-_v) \cup O^+_v \\ S.NC'^c_v = S.NC^c_v \setminus I_v \setminus O^-_v \setminus O^+_v \end{cases}$
  - For every read or destroyed artifact $d \in I_v$ or $d \in O^-_v$, remove d from NC^u_v and NC^c_v.
  - For every produced or updated artifact $d \in O^+_v$, add d to NC^u_v and remove d from NC^c_v.
- If n represents a control block with subsequences $SS = \{SS_i \mid 1 \le i \le k\}$, the same algorithm is recursively applied to calculate each $SS_i.NC$ and then merge them according to the type of the control block.
  - If n is an XOR control block,

$$S.NC'_{n.endVertex} = S.NC_{n.endVertex} = \begin{cases} S.NC^{u}_{n.endVertex} = \bigcap_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \\ S.NC^{c}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.NC_{SS_i.endVertex} \setminus S.NC^{u}_{n.endVertex} \end{cases}$$

■ If n is an *And* control block,

$$S.NC'_{n.endVertex} = S.NC_{n.endVertex} = \begin{cases} S.NC^{u}_{n.endVertex} = \bigcap_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \cup (\bigcup_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \setminus S.NC^{u}_{n.startVertex}) \\ \qquad \setminus \bigcup_{i=1}^{k} ((SS_i.I \cup SS_i.O^{-}) \setminus SS_i.NC^{u}_{SS_i.endVertex}) \\ S.NC^{c}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.NC_{SS_i.endVertex} \setminus S.NC^{u}_{n.endVertex} \end{cases}$$

5.2.2.2. Rules for Detecting Redundant Production/Update Anomalies

● **Explicit Redundant Update**

■ After visiting the *endVertex* of the top level sequence, i.e. the end vertex of the process, if $EC = NC^{u}_{S.endVertex} \setminus O_w \neq \varnothing$, a redundant update anomaly occurs for every artifact $d \in EC$ due to *No Consumption After Last Write*.

● **Potential Redundant Update**

■ After visiting the *endVertex* of the top level sequence, i.e. the end vertex of the process, if $CC = NC^{c}_{S.endVertex} \setminus O_w \neq \varnothing$, a redundant update anomaly occurs for every artifact $d \in CC$ due to *Conditional Consumption After Last Write*.

5.2.2.3. Algorithm to Detect Redundant Production and Update Anomalies

```
Algorithm DetectRedundantWrite ( S , NC_P, O_W ) {
//Input:  S: a structure containing a sequence of nodes (of which each is a task or a control block)
//        NC_P: the set of artifacts unused after preceding nodes.
//Output:   messages for detected missing production anomalies.
```

```
S.NC = NC_P ;     //the set of artifacts not unused by preceding nodes.
S.I = ∅ ;     //the set of artifacts consumed by activities of this sequence.
S.O⁻ = ∅ ;     //the set of artifacts destroyed by activities of this sequence.
for each  n∈S  {     //process every task or control block
    switch (n.type) {
        case Task:      //a task activity
            v=n.startVertex;
            S.NC = UpdateUnusedArtifactSet( v , S.NC );
            S.I = S.I ∪ I_v ; //update the set of artifacts consumed.
            S.O⁻ = S.O⁻ ∪ O_v⁻ ; //update the set of artifacts destroyed.
            break;
        case default:   //control blocks
            SS = n.subSequences ;
            for each  SS_i ∈ SS  {
                // for every subsequence, recursively
                //calculate the set of artifacts unused and detect redundant update
                DetectRedundantWrite ( SS_i , S.NC , O_W );
            }

            //merge unused artifact sets from all subsequences
            AA =MergeUnusedArtifactSets( n , SS );
    }
}

If (S.level==0) // top level sequence {
    CheckRedundantWrite (NC, O_W );
}
}
```

Algorithm UpdateUnusedArtifactSet (v , NC) {

//update the set of artifacts unused

//remove artifacts read or destroyed and add artifacts updated by v.

$$NC_v = \begin{cases} NC_v^u = NC^u \setminus I_v \setminus O_v^-) \cup O_v^+ \\ NC_v^c = NC_{P(v)}^c \setminus I_v \setminus O_v^- \setminus O_v^+ \end{cases};$$

}

Algorithm MergeUnusedArtifactSets (n , SS){

if (n.endVetex.type==”XorJoin”){

$$NC_{n.endVertex} = \begin{cases} NC^{u}_{n.endVertex} = \bigcap_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \\ NC^{c}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.NC_{SS_i.endVertex} \setminus NC^{u}_{n.endVertex} \end{cases};$$

} **else** {

$$NC_{n.endVertex} = \begin{cases} NC^{u}_{n.endVertex} = \bigcap_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \cup (\bigcup_{i=1}^{k} SS_i.NC^{u}_{SS_i.endVertex} \setminus S.NC^{u}_{n.startVertex}) \\ \qquad \setminus \bigcup_{i=1}^{k} ((SS_i.I \cup SS_i.O^{-}) \setminus SS_i.NC^{u}_{SS_i.endVertex}) \\ NC^{c}_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.NC_{SS_i.endVertex} \setminus NC^{u}_{n.endVertex} \end{cases};$$

}

return $NC_{n.endVertex}$;

}

Algorithm CheckRedundantWrite(NC , O_W) {

for each $d \in (NC^{c} \setminus O_W)$ {

print “A potential redundant update anomaly occurs for artifact d due to Conditional Unused for the Process”;

}

for each $d \in (NC^{u} \setminus O_W)$ {

print “A potential redundant update anomaly occurs for artifact d due to Completely Unused for the Process”;

}

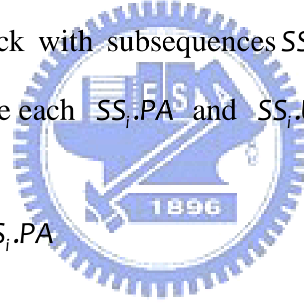
}

5.2.3. Method for Detecting Conflict Writes Anomalies

5.2.3.1. Calculation of the Set of Artifact Produced and Updated of a Sequence

Given a sequence S of the input process schema, let $S.PA$ and $S.UA$ denote the set of artifacts produced and updated respectively within the sequence S. Initially, $S.PA$ and $S.UA$ are empty. During the traversal of the sequence S, $S.PA$ and $S.UA$ are calculated on every traversed node n as follow.

- If n represents a task activity v, $\begin{cases} S.PA = S.PA \cup (O_v^+ \setminus I_v) \\ S.UA = S.UA \cup (O_v^+ \cap I_v) \end{cases}$
  - For every artifact d produced by v, i.e. $d \in (O_v^+ \setminus I_v)$, add v to $S.PA$.
  - For every artifact d updated by v, i.e. $d \in (O_v^+ \cap I_v)$, add v to $S.UA$.
- If n represents a control block with subsequences $SS = \{SS_i \mid 1 \le i \le k\}$, the same algorithm is recursively applied to calculate each $SS_i.PA$ and $SS_i.UA$ and then merge them.



$$S.PA = S.PA \cup \bigcup_{i=1}^{k} SS_i.PA$$

$$S.UA = S.UA \cup \bigcup_{i=1}^{k} SS_i.UA$$

5.2.3.2. Rules for Detecting Conflict Production/Update Anomalies

- Multiple Parallel Productions
  - For an AND control block with subsequences $SS = \{SS_i \mid 1 \le i \le k, k \ge 2\}$, before merging $SS_i.PA$ from every subsequence, if $MPA = \bigcap_{i=1}^{k} SS_i.PA \ne \varnothing$ then a conflict writes anomaly occurs for every artifact $d \in MPA$ due to multiple parallel productions.
- Multiple Parallel Updates
  - For an AND control block with subsequences $SS = \{SS_i \mid 1 \le i \le k, k \ge 2\}$, before merging

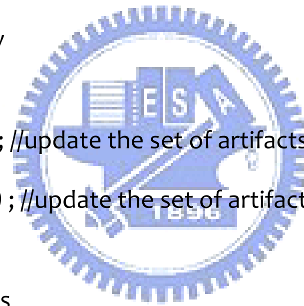
$SS_i.UA$ from every subsequence, if $MUA = \bigcap_{i=1}^{k} SS_i.UA \neq \emptyset$ then a conflict writes anomaly occurs for every artifact $d \in MUA$ due to multiple parallel updates.

5.2.3.3. Algorithm to Detect Conflict Production and Update Anomalies

```
Algorithm DetectConflictWrites ( S ) {
  S.PA = ∅;    //the set of artifacts produced by activities of sequence S.
  S.UA = ∅;    //the set of artifacts updated by activities of sequence S.
  S.RA = ∅;    //the set of artifacts read by activities of sequence S.

  for each  n∈ S  {    //process every task or control block
    switch (n.type) {
      case Task:     //a task activity
        v=n.startVertex;
        S.PA^u = S.PA^u ∪ (O_v^+ \ I_v) ; //update the set of artifacts consumed.
        S.UA^u = S.UA^u ∪ (O_v^+ ∩ I_v) ; //update the set of artifacts destroyed.
        break;
      case default:  //control blocks
        SS = n.subSequences;
        for each  SS_i ∈ SS  {
          // for every subsequence, recursively
          //calculate the set of artifacts produced/updated and detect conflict write
          DetectConflictWrites ( SS_i );
        }

        //detect conflict writes before merging
        DetectConflictWrites ( n , SS );
        //merge from all subsequences
        AA =MergeWriteArtifactSets( n , SS );
    }
  }
```



}

Algorithm DetectConflictWrites (n , SS) {

$MPP = \bigcap_{i=1}^{|SS|} SS_i.PA$; $MPU = \bigcap_{i=1}^{|SS|} SS_i.UA$;

for each $d \in MPP$ {

print "A conflict writes anomaly for artifact d might occur due to multiple parallel productions."

}

for each $d \in MPU$ {

print "A conflict writes anomaly for artifact d might occur due to multiple parallel updates."

}

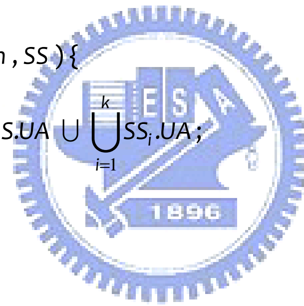
}

Algorithm MergeWriteArtifactSets (n , SS) {

$S.PA = S.PA \cup \bigcup_{i=1}^{k} SS_i.PA$; $S.UA = S.UA \cup \bigcup_{i=1}^{k} SS_i.UA$;

return $NC_{n.endVertex}$;

}



5.2.4. Complexities of Traversal and Detection Algorithms

Given an input process schema $BP = (G, VT, D, I_W, O_W)$, as shown in 5.2, four main algorithms are used to check the artifact usage anomalies: *ControlGraphToSequce*, *DetectMissingProduction*, *DetectRedundantWrite*, and *DetectConflictWrites*.

Algorithm *ControlGraphToSequce* walks through every vertex and edge in $G(V,E)$. Every edge is walked through exactly once and every vertex is processed once. Thus, the complexity of algorithm *ControlGraphToSequce* is $O(|V|+|E|)$.

The detection algorithms, including algorithm *DetectMissingProduction*, *DetectRedundantWrite*, and *DetectConflictWrites*, handles every activity only once. Two or

three sets of artifacts are required to calculate for every activity. Since the maximum size of each set of artifacts is fixed ($|D|$), we can use a bit-vector representation of an element of a set. The advantages of this representation are that operations are very fast and the required memory space can be small. Operations such as unions, intersections, and complements can be considered as constant time. Thus, given $n = |V|$, algorithm *DetectRedundantWrite* and *DetectConflictWrites* are $O(n)$. For algorithm *DetectMissingProduction,* if *destroy* operation is not considered, the complexity is $O(n)$. When *destroy* operation is involved, it is necessary to check every pair of parallel branches for missing production anomalies inside an *AND* control block. Thus, with *destroy* operations, the complexity of algorithm *DetectMissingProduction* is $O(n + n^2) = O(n^2)$.

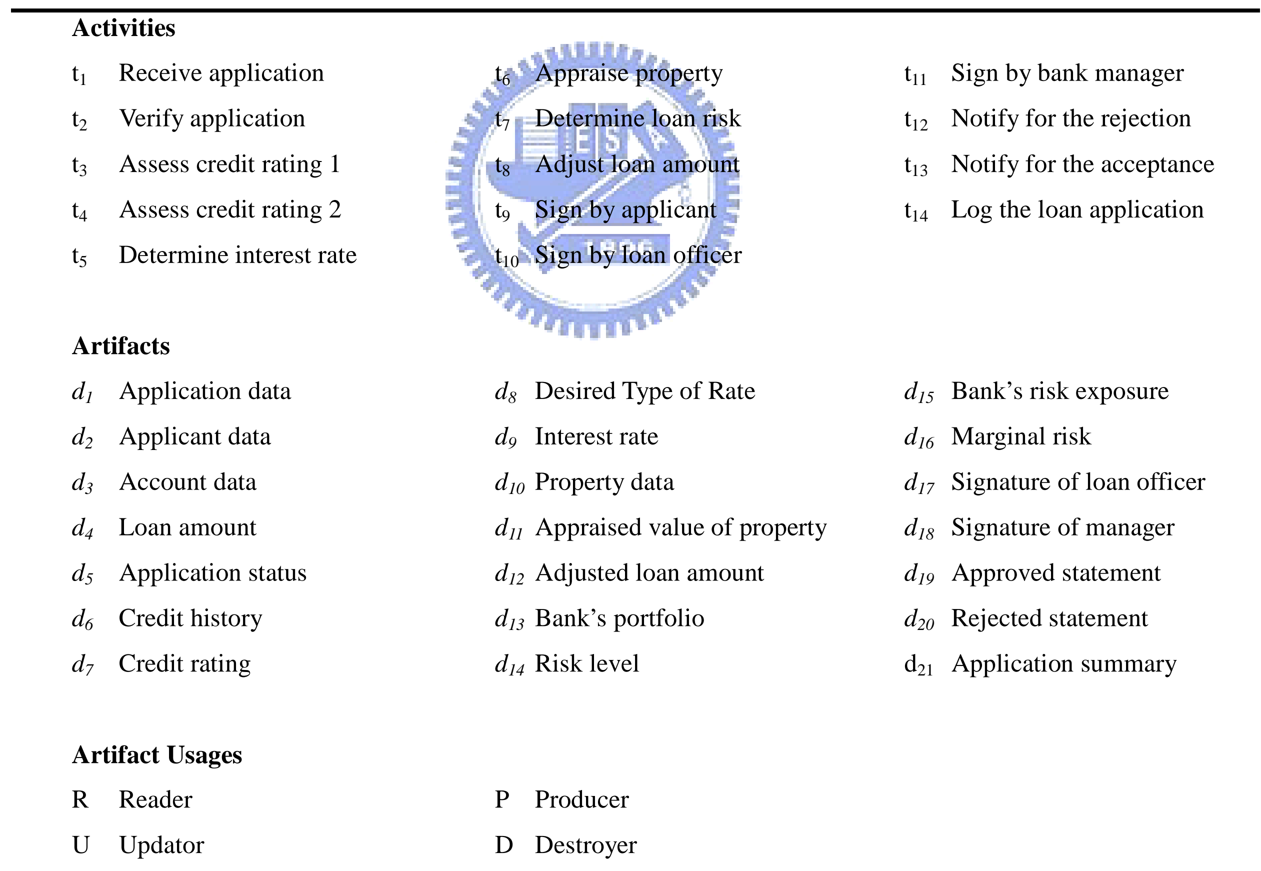
Chapter 6. Illustrative Example

To demonstrate the proposed analysis algorithms, this chapter introduces a property loan approval process [10] as an example. The proposed algorithms are applied on this example to illustrate the steps to detect the artifact usage anomalies.

6.1. An Example: Property Loan Approval Process

Figure 6.1 shows the control flow graph of the approval process and Table 6.2 presents the artifact usages.

Table 6.1: Activities and Artifacts in the Property Loan Approval Process



Activities					
t_1	Receive application	t_6	Appraise property	t_{11}	Sign by bank manager
t_2	Verify application	t_7	Determine loan risk	t_{12}	Notify for the rejection
t_3	Assess credit rating 1	t_8	Adjust loan amount	t_{13}	Notify for the acceptance
t_4	Assess credit rating 2	t_9	Sign by applicant	t_{14}	Log the loan application
t_5	Determine interest rate	t_{10}	Sign by loan officer		
Artifacts					
d_1	Application data	d_8	Desired Type of Rate	d_{15}	Bank's risk exposure
d_2	Applicant data	d_9	Interest rate	d_{16}	Marginal risk
d_3	Account data	d_{10}	Property data	d_{17}	Signature of loan officer
d_4	Loan amount	d_{11}	Appraised value of property	d_{18}	Signature of manager
d_5	Application status	d_{12}	Adjusted loan amount	d_{19}	Approved statement
d_6	Credit history	d_{13}	Bank's portfolio	d_{20}	Rejected statement
d_7	Credit rating	d_{14}	Risk level	d_{21}	Application summary
Artifact Usages					
R	Reader	P	Producer		
U	Updator	D	Destroyer		

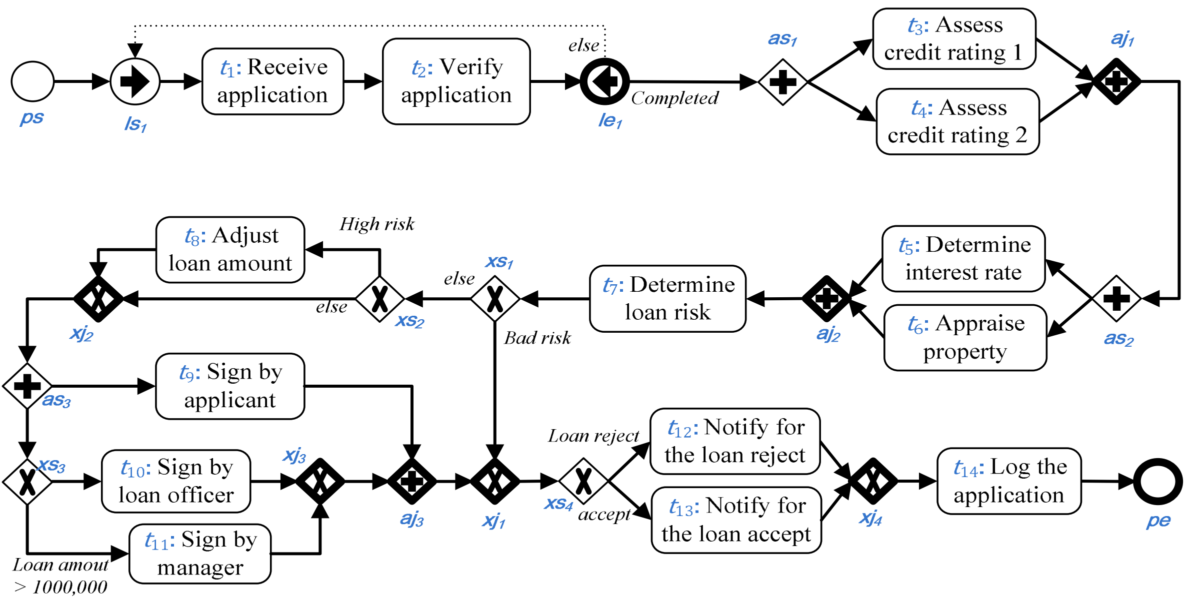


Figure 6.1: Property Loan Approval Process.

Table 6.2: Artifacts Usages in the Property Loan Approval Process

	ps	ls_1	t_1	t_2	le_1	as_1	t_3	t_4	aj_1	as_2	t_5	t_6	aj_2	t_7	xs_1	xs_2	t_8	xj_2	as_3	t_9	xs_3	t_{10}	t_{11}	xj_3	aj_3	xj_1	xs_4	t_{12}	t_{13}	xj_4	t_{14}	pe
d_1	P		R																													
d_2			P	R			R	R			R						R			D		R	R				R					
d_3			P	R			R				R																					
d_4			P	R							R			R			R				R	R	R									
d_5			P	U	R		U	U						U			U					U	U					U	U		U	
d_6							U	R																								
d_7							P	P			R			D								R	R									
d_8											R																					
d_9											P						D			R		R	R									
d_{10}			P									R		R																		
d_{11}											R	P		R			R															
d_{12}														R			P			R	R	R	R									
d_{13}	P																															
d_{14}														P	R	R	R				R	R	R				R					
d_{15}																R					R		R				R					
d_{16}															R												R					
d_{17}																						P	R									
d_{18}																							P				R					
d_{19}																													P		R	
d_{20}																												P			R	
d_{21}																							R									R

1896

6.2. Detection of Missing Production Anomalies

Table 6.3 shows the steps of the calculation of propagated artifacts and the detection of missing production anomalies.

Table 6.3. Steps to Detect Missing Production Anomalies.

ps, ls₁	$AA^u = \{d_1, d_{13}\}$, $AA^c = \varnothing$
t_1	$AA^u = \{d_1, d_{13}, (d_2), (d_3), (d_4), (d_5), (d_{10})\}$, $AA^c = \varnothing$, $I_{t_1} = \{d_1\}$, $I_{t_1} \setminus AA = \varnothing$
t_2	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}\}$, $AA^c = \varnothing$, $I_{t_2} = \{d_2, d_3, d_4, d_5\}$, $I_{t_2} \setminus AA = \varnothing$
le_1, as_1	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}\}$, $AA^c = \varnothing$, $I_{le_1} = \{d_5\}$, $I_{le_1} \setminus AA = \varnothing$
t_3	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, (d_7)\}$, $AA^c = \varnothing$, $I_{t_3} = \{d_2, d_3, d_5, d_6\}$ $I_{t_3} \setminus AA = \{d_6\}$ => no preceding propagation
t_4	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, (d_7)\}$, $AA^c = \varnothing$, $I_{t_4} = \{d_2, d_5, d_6\}$ $I_{t_4} \setminus AA = \{d_6\}$ => no preceding propagation
aj_1, as_2	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_7\}$, $AA^c = \varnothing$
t_5	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_7, (d_9)\}$, $AA^c = \varnothing$, $I_{t_5} = \{d_2, d_3, d_4, d_7, d_8, d_{11}\}$ $I_{t_5} \setminus AA = \{d_8, d_{11}\}$ => no preceding propagation
t_6	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_7, (d_{11})\}$, $AA^c = \varnothing$, $I_{t_6} = \{d_{10}\}$, $I_{t_6} \setminus AA = \varnothing$
aj_2	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_7, d_9, d_{11}\}$, $AA^c = \varnothing$

t_7	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, \cancel{d_7}, d_9, d_{11}, (d_{14})\}$, $AA^c = \varnothing$, $I_{t_7} = \{d_4, d_5, d_7, d_{10}, d_{11}, d_{12}\}$ $I_{t_7} \setminus AA = \{d_{12}\}$ => no preceding propagation
xs_1, xs_2	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_9, d_{11}, d_{14}\}$, $AA^c = \varnothing$, $I_{xs_1} = \{d_{14}, d_{16}\}$, $I_{xs_2} = \{d_{14}, d_{15}\}$ $I_{xs_1} \setminus AA = \{d_{16}\}$ => no preceding propagation $I_{xs_2} \setminus AA = \{d_{15}\}$ => no preceding propagation
t_8	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, \cancel{d_9}, d_{11}, d_{14}, (d_{12})\}$, $AA^c = \varnothing$, $I_{t_8} = \{d_2, d_4, d_5, d_9, d_{11}, d_{14}\}$ $I_{t_8} \setminus AA = \varnothing$
xj_2, as_3	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}\}$
t_9	$AA^u = \{d_1, d_{13}, \cancel{d_2}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}\}$, $I_{t_9} = \{d_2, d_9, d_{12}\}$ $I_{t_9} \setminus AA = \varnothing$, $I_{t_9} \cap AA^c = \{d_9, d_{12}\}$ => *conditional* preceding propagation
xs_3	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}\}$, $I_{xs_3} = \{d_4, d_{12}, d_{14}, d_{15}\}$ $I_{xs_3} \setminus AA = \{d_{15}\}$ => *no* preceding propagation $I_{xs_3} \cap AA^c = \{d_{12}\}$ => *conditional* preceding propagation
t_{10}	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}, (d_{17})\}$, $AA^c = \{d_9, d_{12}\}$, $I_{t_{10}} = \{d_2, d_4, d_5, d_7, d_9, d_{12}, d_{14}\}$ $I_{t_{10}} \setminus AA = \{d_7\}$ => no preceding propagation $I_{t_{10}} \cap AA^c = \{d_9, d_{12}\}$ => conditional preceding propagation
t_{11}	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}, (d_{18})\}$, $AA^c = \{d_9, d_{12}\}$ $I_{t_{11}} = \{d_2, d_4, d_5, d_7, d_9, d_{12}, d_{14}, d_{15}, d_{17}, d_{21}\}$ $I_{t_{11}} \setminus AA = \{d_7, d_{15}, d_{17}, d_{21}\}$ => no preceding propagation $I_{t_{11}} \cap AA^c = \{d_9, d_{12}\}$ => conditional preceding propagation
xj_3	$AA^u = \{d_1, d_{13}, d_2, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}\}$

aj_3	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}\}$ $I_{t_{10}} \cap O^-_{t_9} = \{d_2\}$ =>uncertain preceding propagation $I_{t_{11}} \cap O^-_{t_9} = \{d_2\}$ =>uncertain preceding propagation
xj_1, xs_4	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}, d_2\}$ $I_{xs_4} = \{d_2, d_{14}, d_{15}, d_{16}, d_{18}\}$ $I_{xs_4} \setminus AA = \{d_{15}\}$ => no preceding propagation $I_{xs_4} \cap AA^c = \{d_{12}\}$ => conditional preceding propagation
t_{12}	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}, (d_{20})\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}, d_2\}$, $I_{t_{12}} = \{d_5\}$, $I_{t_{12}} \setminus AA = \varnothing$
t_{13}	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}, (d_{19})\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}, d_2\}$, $I_{t_{13}} = \{d_5\}$, $I_{t_{13}} \setminus AA = \varnothing$
xj_4, t_{14}	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}, d_2, d_{20}, d_{19}\}$, $I_{t_{14}} = \{d_5, d_{19}, d_{20}\}$ $I_{t_{14}} \setminus AA = \varnothing$, $I_{t_{14}} \cap AA^c = \{d_{19}, d_{20}\}$ => conditional preceding propagation
pe	$AA^u = \{d_1, d_{13}, d_3, d_4, d_5, d_{10}, d_{11}, d_{14}\}$, $AA^c = \{d_9, d_{12}, d_{17}, d_{18}, d_2, d_{20}, d_{19}\}$

6.3. Detection of Redundant Write Anomalies

Table 6.4 shows the steps to calculate the set of unused artifacts for every activity.

Table 6.4. Steps to Calculate the Unused Artifacts for Every Activity.

ps, ls_1	$NC^u = \{d_1, d_{13}\}$, $NC^c = \varnothing$
t_1	$NC^u = \{\cancel{d_1}, d_{13}, (d_2), (d_3), (d_4), (d_5), (d_{10})\}$, $NC^c = \varnothing$
t_2	$NC^u = \{d_{13}, \cancel{d_2}, \cancel{d_3}, \cancel{d_4}, d_5, d_{10}\}$, $NC^c = \varnothing$

le_1, as_1	$NC^u = \{d_{13}, \cancel{d_5}, d_{10}\}$, $NC^c = \varnothing$
t_3	$NC^u = \{d_{13}, d_{10}, (d_5), (d_6), (d_7)\}$, $NC^c = \varnothing$
t_4	$NC^u = \{d_{13}, d_{10}, (d_5), (d_7)\}$, $NC^c = \varnothing$
aj_1, as_2	$NC^u = \{d_{13}, d_{10}, d_5, d_7\}$, $NC^c = \{d_6\}$
t_5	$NC^u = \{d_{13}, d_{10}, d_5, \cancel{d_7}, (d_9)\}$, $NC^c = \{d_6\}$
t_6	$NC^u = \{d_{13}, \cancel{d_{10}}, d_5, d_7, (d_{11})\}$, $NC^c = \{d_6\}$
aj_2	$NC^u = \{d_{13}, d_5, d_9\}$, $NC^c = \{d_6, d_{11}\}$
t_7	$NC^u = \{d_{13}, d_5, d_9, (d_{14})\}$, $NC^c = \{d_6, \cancel{d_{11}}\}$
xs_1	$NC^u = \{d_{13}, d_5, d_9, \cancel{d_{14}}\}$, $NC^c = \{d_6\}$
xs_2	$NC^u = \{d_{13}, d_5, d_9\}$, $NC^c = \{d_6\}$
t_8	$NC^u = \{d_{13}, d_5, \cancel{d_9}, (d_{12})\}$, $NC^c = \{d_6\}$
xj_2	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_9, d_{12}\}$
t_9	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, \cancel{d_9}, \cancel{d_{12}}\}$
xs_3	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_9, \cancel{d_{12}}\}$
t_{10}	$NC^u = \{d_{13}, d_5, (d_{17})\}$, $NC^c = \{d_6, \cancel{d_9}\}$

t_{11}	$NC^u = \{d_{13}, d_5, (d_{18})\}$, $NC^c = \{d_6, \cancel{d_9}\}$
xj_3	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}, d_{18}\}$
aj_3	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}, d_{18}\}$
xj_1, xs_4	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}, \cancel{d_{18}}\}$
t_{12}	$NC^u = \{d_{13}, d_5, (d_{20})\}$, $NC^c = \{d_6, d_{17}\}$
t_{13}	$NC^u = \{d_{13}, d_5, (d_{19})\}$, $NC^c = \{d_6, d_{17}\}$
xj_4	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}, d_{20}, d_{19}\}$
t_{14}	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}, \cancel{d_{20}}, \cancel{d_{19}}\}$
pe	$NC^u = \{d_{13}, d_5\}$, $NC^c = \{d_6, d_{17}\}$

After visiting *Process End* vertex, the redundant update anomalies are detected as follows:

- **Explicit Redundant Update**

$EC = NC^u \setminus O_w = \{d_{13}, d_5\} \setminus \{d_{21}\} = \{d_{13}, d_5\}$ is not empty and thus, a redundant update anomaly occurs for every artifact $d \in EC$ due to *Completely Unused for the Process*.

- **Potential Redundant Update**

$CC = NC^c \setminus O_w = \{d_6, d_{17}\} \setminus \{d_{21}\} = \{d_6, d_{17}\}$ is not empty and thus, a redundant update anomaly occurs for every artifact $d \in CC$ due to *Conditional Unused for the Process*.

Chapter 7. Comparisons of Data-flow Analysis Approaches

Current workflow modeling and analyzing paradigms are mainly focused on the control-flow and resource dimensions. Literature reports indicate little work in data-flow dimension such as those in conventional programming languages. Sadiq et al. [7] and Sun et al. [8－10] are two groups working on the analysis in data-flow dimension. In this chapter, we compare our work with them.

Regarding the anomalies addressed, Sun et al. [8－10] claimed that seven types of data-flow anomalies proposed by Sadiq et al. [7] can be either represented by their three basic data-flow anomalies or not a problem at the conceptual level. The anomalies in [7] can be found with our model as follows.

A *redundant data* anomaly occurs when an activity produces an intermediate data output but this data is not required by any succeeding activity. This anomaly is classified as *Redundant Write* in our approach.

A *lost data* anomaly occurs when two parallel activities perform non-read operations on an artifact. This anomaly is classified as *Conflict Write* (*Multiple Parallel Production/Updates)* in our approach.

A *missing data* anomaly occurs when an artifact is accessed before it is initialized. This anomaly is classified as *Missing Production* in our approach.

A *mismatched data* anomaly occurs when the structure of the data produced by the source is incompatible with the structure required by the activity that uses the artifact. This anomaly can be regarded as the occurrence of both *Missing Production* and *Redundant Write* in our approach.

An *inconsistent data* anomaly occurs when an initial input artifact of a workflow is updated externally during the execution time of the workflow. As stated by Sun et al., this anomaly is not a problem at the conceptual level.

A *misdirected data* anomaly occurs when a data-flow direction conflicts with the control flow in a workflow schema. This anomaly is classified as *Missing Production (Conditional Production)* in our approach.

An *insufficient data* anomaly occurs when data specified are not sufficient to complete an activity successfully. This anomaly results from ill-designed activity and can be classified as *Missing Production* in our approach at the semantic level. Table 7.1 summaries the comparison of anomalies addressed among Sadiq et al., Sun et al., and our work.

Table 7.1. Comparison of anomalies addressed.

<table>
<tr><th colspan="2">Our approach</th><th>Sadiq et al.</th><th>Sun et al.</th></tr>
<tr><td rowspan="8">Missing Production</td><td>No Production</td><td>Missing data
Insufficient data
Mismatched data</td><td rowspan="4">Missing data</td></tr>
<tr><td>Delayed Production</td><td rowspan="3">Misdirected data</td></tr>
<tr><td>Conditional Production</td></tr>
<tr><td>Uncertain Production</td></tr>
<tr><td>Exclusive Production</td><td rowspan="4">N/A</td><td rowspan="4">N/A</td></tr>
<tr><td>Early Destruction</td></tr>
<tr><td>Conditional Destruction</td></tr>
<tr><td>Uncertain Destruction</td></tr>
<tr><td rowspan="2">Redundant Write</td><td>Conditional Consumption After Last Write</td><td rowspan="2">Redundant data
Mismatched data</td><td rowspan="2">Redundant data</td></tr>
<tr><td>No Consumption After Last Write</td></tr>
</table>

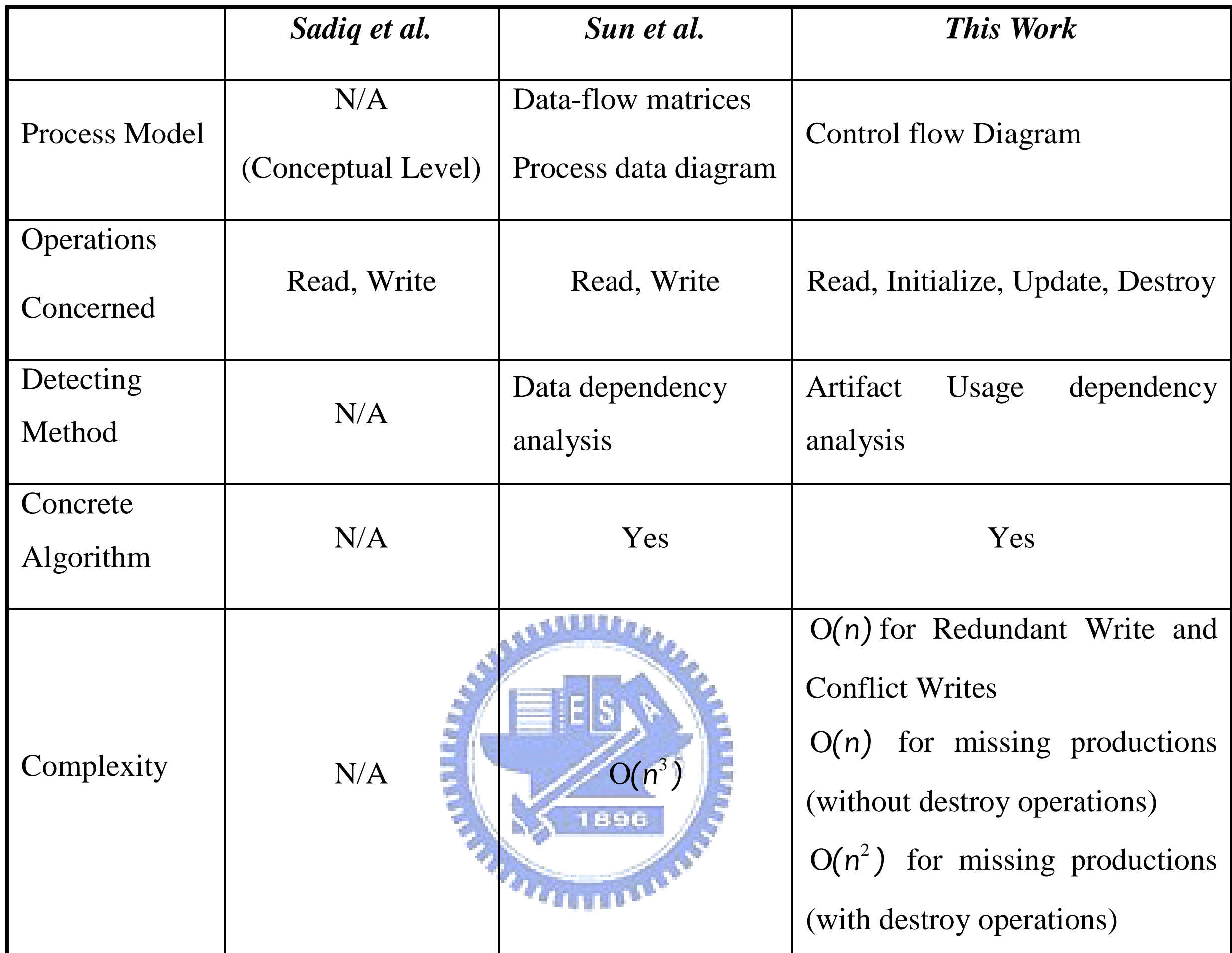
<table>
<tr><td rowspan="3">Conflict Write</td><td>Multiple Parallel Productions</td><td rowspan="2">Lost data</td><td>Conflicting data</td></tr>
<tr><td>Multiple Parallel Updates</td><td rowspan="2">N/A</td></tr>
<tr><td>Parallel Read and Update</td><td>N/A</td></tr>
</table>

Sadiq et al. [7] identify and justify the importance of data modeling in overall workflow design process. In addition, data-flow validation issues and essential requirements of data-flow modeling in workflow specifications are identified. They illustrate and define seven potential data-flow anomalies in above table. However, Sadiq's work is discussed only on the conceptual level and thus, neither concrete data-flow model nor detecting algorithms are proposed. Furthermore, operations on data are only classified into read and write type.

Sun et al. [8－10] formulate the data-flow perspective by means of dependency analysis. The data-flow matrix and an extension of the unified modeling language (UML) activity diagram are proposed to specify the data flow in a business process. Then, three basic types of data-flow anomalies, missing data, redundant data, and conflicting data, are defined. Based on the dependency analysis, algorithms to data-flow analysis for discovering the data-flow anomalies are presented. However, as Sadiq's work, no explicit model is proposed to characterize the behaviors of data. Also, read and initial write operation types are considered only.

Our approach presents a process model to describe workflow schemas. The behaviors of an artifact are explicitly modeled by a finite state machine. The operation types including Initialize, Read, Update, and Destroy, are concerned in this dissertation. Table 7.2 summaries the comparisons.

Table 7.2. A summary of comparisons.

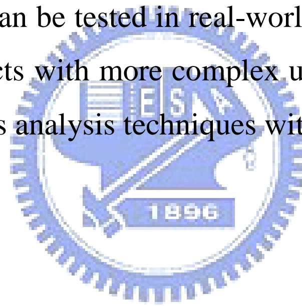


	Sadiq et al.	***Sun et al.***	***This Work***
Process Model	N/A (Conceptual Level)	Data-flow matrices Process data diagram	Control flow Diagram
Operations Concerned	Read, Write	Read, Write	Read, Initialize, Update, Destroy
Detecting Method	N/A	Data dependency analysis	Artifact Usage dependency analysis
Concrete Algorithm	N/A	Yes	Yes
Complexity	N/A	$O(n^3)$	$O(n)$ for Redundant Write and Conflict Writes $O(n)$ for missing productions (without destroy operations) $O(n^2)$ for missing productions (with destroy operations)

Chapter 8. Conclusion and Future Work

The main contribution of this dissertation is to introduce an artifact usage analysis technique into workflow design phase. To achieve this goal, this dissertation presents a business process model for describing a business process and analyzes the artifact usages on this model. In our model, the usages of an artifact are characterized by its state transition diagram. Among the usages of artifacts, three types with thirteen cases of improper artifact usage affecting workflow execution are identified and formulated and a set of algorithms to discovering these anomalies is presented. An example is demonstrated to validate the usability of the proposed algorithms.

We are currently continuing our research in several directions. First, we plan to implement the proposed model and algorithms on current workflow management systems, such as Agentflow [38], so that our research result can be tested in real-world applications. Second, we will continue the analysis on composite artifacts with more complex usages using *Revise* operations. The third is to integrate resource constrains analysis techniques with our work to build a practical workflow design methodology.


1896

Reference

1 The Workflow Management Coalition, “The workflow reference model”, Document Number TC00-1003, January 1995.

2 P. Senkul and I.H. Toroslu, “An architecture for workflow scheduling under resource allocation constraints”, Information Systems, Vol. 30, Issue 5, pp. 399-422, PERGAMON, July 2005.

3 H. Li, Y. Yang and T.Y. Chen, “Resource constraints analysis of workflow specifications”, Journal of Systems and Software, Vol. 73, No. 2, pp. 271–285, Elsevier Science, October 2004.

4 C. Liu, X. Lin, M.E. Orlowska, and X. Zhou, “Confirmation: increasing resource availability for transactional workflows”, Information Sciences, Vol. 153, Issue 1, pp. 37-53, Elsevier Science Inc, July 2003.

5 W. Du and M.C. Shan, “Enterprise workflow resource management”, Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, pp. 108-115, IEEE Computer Society, March 1999.

6 M.Z. Muehlen, “Resource modeling in workflow applications”, Proceedings of the 1999 Workflow Management Conference, pp. 137-153, Münster, Germany, November 1999.

7 S. Sadiq, M.E. Orlowska, W. Sadiq, and C. Foulger, ”Data flow and validation in workflow modeling”, Proceedings of the 15th Australasian database conference, pp. 207-214, Dunedin, New Zealand, January 2004.

8 S.X. Sun, and J.L. Zhao, "A data flow approach to workflow design", Proceedings of the 14th Workshop on Information Technology and Systems (WITS’04), pp. 80-85, 2004.

9 S.X. Sun, J.L. Zhao, and O.R. Sheng, "Data flow modeling and verification in business process management", Proceedings of the AIS Americas Conference on Information Systems, pp. 4064-4073, New York, August 5-8, 2004.

10 S.X. Sun, J.L. Zhao, J.F. Nunamaker, and O.R.L. Sheng, “Formulating the data flow perspective for business process management”, Information Systems Research, Vol. 17, No. 4, pp. 374-391, December 2006.

11 H. Zhuge, "Component-based workflow systems development", Decision Support Systems, Vol. 35, Issue 4, pp. 517-536, Elsevier Science Publishers, July 2003.

12 A.S. Hitomi and D. Le, "Endeavors and component reuse in web-driven process workflow", Proceedings of the California Software Symposium, pp. 15-20, Irvine, CA, USA, October 1998.

13 H.-J Hsu, "Using state diagrams to validate artifact specifications on primitive workflow schema", National Chiao-Tung University, M.S. Thesis, 2005.

14 F.-J. Wang, C.-L. Hsu, and H.-J. Hsu, "Analyzing inaccurate artifact usages in a workflow schema", Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Vol. 2, pp. 109-114, September 17-21, 2006.

15 B. Curtis, M.I. Kellner, and J. Over, "Process modeling", CACM Vol. 35, No. 9, pp. 75–90.

16 S. Jablonski and C. Bussler, Workflow management: modeling concepts, architecture, and implementation, International Thomson Computer Press, London, UK, 1996.

17 C. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater, "Model checking of workflow schemas", Proceeding of Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), pp. 170-179, IEEE Computer Society, September 2000.

18 W.M.P. van der Aalst, "The application of petri nets to workflow management", Journal of Circuits, Systems and Computers, Vol. 8, No. 1, pp. 21-66, 1998.

19 W.M.P. van der Aalst and A.H.M. ter Hofstede, "Verification of workflow task structures: a petri-net-based approach", Information Systems, Vol. 25, No. 1, pp. 43-69, 2000.

20 W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns", BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000.

21 W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Advanced workflow patterns", Proceeding of 7th International Conference on Cooperative Information Systems (CoopIS 2000), Vol. 1901 of Lecture Notes in Computer Science, pp. 18-29. Springer-Verlag, Berlin, 2000.

22 W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow

patterns", Distributed and Parallel Databases, Vol. 14, No. 1, pp. 5-51, July 2003.

23 W.M.P. van der Aalst, "Verification of workflow nets", Proceedings of the 18th International Conference on Application and Theory of Petri Nets, pp. 407-426, Toulouse, France, June 23-27, 1997.

24 W.M.P. van der Aalst, "The application of petri-nets to workflow management", Journal of Circuits, Systems and Computers, Vol. 8, No. 1, pp. 21-66, 1998.

25 W.M.P. van der Aalst and T. Basten, "Inheritance of workflows: an approach to tackling problems related to change", Theoretical Computer Science, Vol. 270, No. 1-2, pp. 125-203, 2002.

26 H.M.W. Verbeek and W.M.P. van der Aalst, "Woflan 2.0: a petri-net-based workflow diagnosis tool", Proceedings of the 21st International Conference of Application and Theory of Petri Nets (ICATPN 2000), pp. 475-484, Aarhus, Denmark, June 26-30, 2000.

27 C. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater, "Formal verification of workflow schemas", Technical Report, Control and Coordination of Complex Distributed Services, ESPRIT Long Term Research Project, 2000.

28 H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst, "Diagnosing workflow processes using woflan", The Computer Journal, Vol. 44, No. 4, pp. 246-279, 2001.

29 L. Gong and H.-Y. Wang, "A method to verify the soundness of workflow control logic", Computer Supported Cooperative Work in Design, Vol. 1, pp. 284-388, May 2004.

30 W. Sadiq and M.E. Orlowska, "Analyzing process models using graph reduction techniques", Information Systems, Vol. 25, No. 2, pp. 117-134, Elsevier Science Publishers, 2000.

31 W. Sadiq and M.E. Orlowska, "On correctness issues in conceptual modeling of workflows", Proceedings of the 5th European Conference on Information Systems (ECIS '97), Cork, Ireland, June 19-21, 1997.

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

33 N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst, "Workflow data patterns", QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.

34 J. Bae, H. Bae, S.-H. Kang, and Y. Kim, "Automatic control of workflow processes using ECA rules", IEEE Transaction on Knowledge and Date Engineering, Vol. 14, No. 8, pp. 1010-1023, IEEE Computer Society, August 2004.

35 J.H. Son, and M.H. Kim, "Extracting the workflow critical path from the extended well-formed workflow schema", Journal of Computer and System Sciences, Vol. 70, Issue 1, pp. 86-106, Elsevier Science Publishers, February 2005.

36 D.-H. Chang, J.H. Son, and M.H. Kim, "Critical path identification in the context of a workflow", Information & Software Technology, Vol. 44, No. 7, pp. 405-417, Elsevier Science Publishers, May 2002.

37 Object Management Group. 2006, Business Process Modeling Notation (BPMN), http://www.bpmn.org/.

38 Flowring Technology Corp., http://www.flowring.com, accessed May 2006.

39 The Workflow Management Coalition, "Terminology & glossary", Document Number WFMC-TC-1011, February 1999.

40 N.R. Adam, V. Atluri, and W.K. Huang, "Modeling and analysis of workflows using petri nets", Journal of Intelligent Information Systems, Vol. 10, No. 2, pp. 131-158, March/April 1998.

41 M. Koubarakis and D. Plexousakis, "A formal framework for business process modelling and design", Information Systems, Vol. 27, No. 5, pp. 299-319, July 2002.

42 H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan, "Logic based modeling and analysis of workflows", Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pp. 25-33, Seattle, Washington, United States, June 01-04, 1998.

43 P. Senkul, M. Kifer and I.H. Toroslu, "A logical framework for scheduling workflows under resource allocation constraints", Proceeding of 28th International Conference on Very Large

Data Bases (VLDB'02), pp. 694–702, August 2002.

44 G. Trajcevski, C. Baral, and J. Lobo, "Formalizing (and Reasoning About) the specifications of workflows", Proceedings of the 7th International Conference on Cooperative Information Systems, pp. 1-17, September 06-08, 2000.

45 N. Tatbul, S. Nural, P. Karagoz, I. Cingil, E. Gokkoca, M. Altinel, P. Koksal, and A. Dogac, "A workflow specification language and its scheduler", Proceedings of International Conference on Computer and Information Sciences, pp. 163-170, Antalya, Turkey, November 1997.

46 S. Chinn, and G. Madey, "Temporal representation and reasoning for workflow in engineering design change review", IEEE Transactions on Engineering Management, Vol. 47, No. 4, pp. 485-492, 2000.

47 A.H.M. ter Hofstede and M.E. Orlowska, "On the complexity of some verification problems in process control specifications", The Computer Journal, Vol. 42, No. 5, pp. 349-359, 1999.

48 A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse, "Verification problems in conceptual workflow specifications", Data and Knowledge Engineering, Vol. 24, Iss. 3, pp. 239-256, 1998.

49 C. Karamanolis, D. Giannakopoulou, J. Magee, and S. Wheater, "Modeling and analysis of workflow processes". Technical Report 99/2, Department of Computing, Imperial College.

50 K. Kim, C.A. Ellis, "Performance analytic models and analyses for workflow architectures", Journal of Information Systems Frontiers, Vol. 3, No. 3, pp. 339-355, September 2001.

51 A. Zaidi, "On temporal logic programming using petri nets", IEEE Transactions on Systems, Man, and Cybernetics, Part A: Systems and Humans, Vol. 29, Issue 3, pp. 245-254, 1999.

52 M. Rosemann and M. zur Mühlen, "Evaluation of workflow management systems: a meta-model approach",: Australian Journal of Information Systems, Vol. 6, No. 1, pp. 103-116, 1998.

53 E. Bertino; E. Ferrari, and V. Atluri, "The specification and enforcement of authorization constraints in workflow management systems", ACM Transactions on Information and System Security, Vol. 2 , Iss. 1, pp. 65-104, 1999.

54 O. Marjanovic, "Dynamic verification of temporal constraints in production workflows",

Proceedings of the Australian Database Conference ADC'2000, pp. 74-81, January 31-February 03, 2000.

55 M. Reichert and P. Dadam, “Adept$_{flex}$—Supporting Dynamic Changes of Workflows Without Losing Control”, Journal of Intelligent Information Systems, Vol. 10, No. 2, pp. 93-129, March/April 1998.

56 A. Bajaj and S. Ram, "Seam: a state-entity-activity-model for a well-defined workflow development methodology", IEEE Transactions on Knowledge and Data Engineering, Vol. 14, No. 2, pp. 415-431, March/April 2002.

57 A. Kumar and J.L. Zhao, "Dynamic routing and operational control in workflow management systems", Management Science, Vol. 35, No. 2, pp. 253-272, February 1999.

58 M.M. Kwan and P.R. Balasubramanian, "Dynamic workflow management: a framework for modeling workflows", Proceedings of the 13th Hawaii International Conference on System Sciences, Vol. 4, pp. 367-376, January 7-10, 1997.

59 H.B. Luo, Y.S. Fan, and C. Wu, “Analysis of event balance in the verification of workflow soundness”, Journal of Software, Vo1. 13, No. 8, pp. 1686-1691, 2002.