

國立交通大學

資訊科學與工程研究所

碩士論文

運用 DCDC Model 來遞增性分析高階程式之異常使用



Detecting Artifact Usage Anomalies

in High-level Software Incrementally with DCDC Model

研究生：林建志

指導教授：王豐堅 教授

中華民國九十七年八月

運用 DCDC Model 來遞增性分析高階程式之異常使用

Detecting Artifact Usage Anomalies
in High-level Software Incrementally with DCDC Model

研究生：林建志

Student : Chien-Chih Lin

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
碩士論文



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

Master

in

Computer Science

August 2008

Hsinchu, Taiwan, Republic of China

中華民國九十一年八月

運用 DCDC Model

來遞增性分析高階程式之異常使用

研究生：林建志

指導教授：王豐堅 博士

國立交通大學
資訊科學與工程研究所
碩士論文



工作流程是由一組系統化組成的工作，經過特定的順序執行之後，能夠達成所要的目標或是產品。其中資料是實作和執行工作流程不可或缺的元素。然而工作流程可能會因為不適當的資料操作而產生預期外的結果，例如，遺失執行所需的資料或是收到多份同名資料產生衝突等等。因此在設計流程的過程中，資料分析結果便能提供相關的幫助。這篇論文提出了一個流程模組去描述一個 Well-formed 工作流程，這一名叫 DCDC Model 的模組顯示出四類的資料使用異常。因此我們提出了一些漸進式的演算法去偵測出這些資料使用異常，以輔助使用者在工作流程之編輯。

關鍵字：工作流程、商業流程、控制流、資料、資料流、遞增分析、異常

Detecting Artifact Usage Anomalies in High-level Software Incrementally with DCDC Model

Student: Chien-Chih Lin

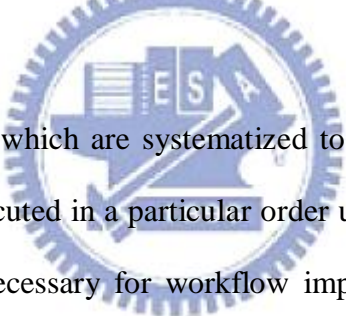
Advisor: Dr. Feng-Jian Wang

Institute of Computer Science and Engineering

National Chiao Tung University

1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC

Abstract



Workflow is a set of tasks which are systematized to achieve certain business goal(s), complete where each task is executed in a particular order under automatic control. Artifacts, collections of data items, are necessary for workflow implementation and support process execution. However, a workflow may yield unexpected results in execution due to improper artifact manipulation; e.g. activities miss artifact, or artifact conflict occurs at an activity in run time. Therefore, the analyses on artifact usage in design phase are very important. This thesis presents a process model, named DCDC model, to describe a well-formed workflow. There are four types of artifact usage anomalies in DCDC identified. To help the edit of a process, the corresponding incremental algorithms are presented to detect these anomalies. Their time complexities are also studied.

Keywords: work flow, business process, control flow, artifact, artifact flow, incremental analysis, anomalies.

誌謝

本篇論文的完成，首先要感謝我的指導教授王豐堅博士兩年來不斷的指導與鼓勵，讓我在軟體工程及工作流程的技術上得到很多豐富的知識，使我可以在偵測資料使用異常上找到靈感。另外，也非常感謝我的畢業口試評審委員朱治平博士以及朱正忠博士，提供許多寶貴的意見，補足我論文裡不足的部分。

其次，我要感謝實驗室的學長姊們在研究生涯上的指導與照顧，尤其是懷中學長的督促與幫忙，讓我學得許多做研究的方法和技巧，得以順利的撰寫論文。

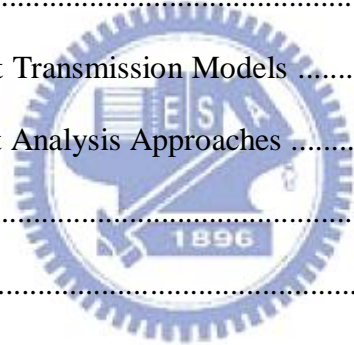
最後，我要感謝我的家人，由於有你們的支持，讓我能讀書、作研究到畢業，此外，謝謝同期碩士班好友們的打氣，在我遇到挫折時能互相勉勵並度過難關。由衷地感謝你們大家一路下來陪著我走過這段研究生歲月。



Table of Contents

摘要	i
Abstract	ii
誌謝	iii
Table of Contents	iv
List of Tables	vi
List of Figures	viii
Chapter 1. Introduction	1
Chapter 2. Related Work	3
2.1. Artifact Transmission Models	3
2.2. Previous Work	5
Chapter 3. A Process Model Based on Well-formed Workflow	7
3.1. Control Flow Specification	7
3.2. Relations between Activities and Control Blocks	15
3.3. Artifact Flow Diagram Specification	17
3.3.1. Artifact Operations and Usages	18
3.3.2. Artifact Flow Diagrams	21
3.3.3. An Example of an Artifact Flow Diagram	28
Chapter 4. Artifact Usage Anomalies	31
4.1. Missing Artifact Anomalies	31
4.2. Artifact Conflict Anomalies	33
4.3. Cross Passing Artifact Anomalies	36
4.4. Redundant Anomalies	38
Chapter 5. Incremental Algorithms for Anomalies Detection	41
5.1. Edit Operations for a Process	41

5.2. Incremental Algorithms to Detect Artifact Usage Anomalies	45
5.3. Algorithms to Update an Artifact Flow Diagram	49
5.3.1. Construction of XBNodes	49
5.3.2. Update of the Properties of Flows	55
5.3.3. An Example for Illustration of Updating an Artifact Flow Diagram	60
5.4. Algorithms to Detect Artifact Usage Anomalies	64
Chapter 6. Examples for Illustrating Incremental Algorithms	68
6.1. Activity Modification	70
6.2. Pass Insertion	74
6.3. Pass Deletion	78
Chapter 7. Comparisons	83
7.1. Comparison of Artifact Transmission Models	83
7.2. Comparison of Artifact Analysis Approaches	85
Chapter 8. Conclusion	89
Reference	90



List of Tables

Table 3.1: The States of Each Activity in Figure 3.7	29
Table 3.2: The States of Each XBNode in Figure 3.8	30
Table 5.1: Edit Operations for a Process	41
Table 5.2: Construction of XBNodes in $\text{BuildXBNodes}(POUT_{v_1}^d, v_1, XOUT_{v_1}^d)$	62
Table 5.3: The States of Each XBNode in $XOUT_{v_1}^d$	62
Table 6.1: The States of Each Activity in Figure 6.1	69
Table 6.2: The States of Each Activity in Figure 6.2	70
Table 6.3: The Anomalies Occur in Figure 6.2	70
Table 6.4: The Anomalies Occur in Figure 6.3	71
Table 6.5: The Anomalies Occur in Figure 6.4	72
Table 6.6: The Anomalies Occur in Figure 6.5	73
Table 6.7: The States of Activities v_9 and v_{10} in Figure 6.6	75
Table 6.8: The Anomalies Occur in Figure 6.6	75
Table 6.9: The States of Activities v_3 and v_6 in Figure 6.7	76
Table 6.10: The Anomalies Occur in Figure 6.7	77
Table 6.11: The States of Activities v_6 and v_8 in Figure 6.8	78
Table 6.12: The Anomalies Occur in Figure 6.8	78
Table 6.13: The States of Activities v_1 and v_8 in Figure 6.9	79
Table 6.14: The Anomalies Occur in Figure 6.9	79
Table 6.15: The States of Activities v_4 and v_7 in Figure 6.10	80
Table 6.16: The Anomalies Occur in Figure 6.10	81
Table 6.17: The States of Activities v_1 and v_3 in Figure 6.11	82
Table 6.18: The Anomalies Occur in Figure 6.11	82
Table 7.1: Comparison of Artifact Transmission Models	85

Table 7.2: Comparison of the Artifact Usage Anomalies Addressed 87

Table 7.3: Comparison with Previous Work 88



List of Figures

Figure 2.1: Three Major Artifact Transmission Models	3
Figure 3.1: Notations of Control Flow Graph	12
Figure 3.2: An Example of Control Flow Graph	12
Figure 3.3: Using a Loop Sub-Process Activity to Replace a Loop Structure	17
Figure 3.4: An Artifact Flow Diagram without XBNodes	23
Figure 3.5: An Example of Receiving and Sending Sets	25
Figure 3.6: An Artifact Flow Diagram with XBNode	26
Figure 3.7: An Example of a Control Flow Graph	28
Figure 3.8: The Corresponding Artifact Flow Diagram for d	29
Figure 4.1: An Example of an Explicit Missing Artifact Anomaly	32
Figure 4.2: An Example of an Implicit Missing Artifact Anomaly	32
Figure 4.3: An Example of a Destroyed Artifact Anomaly	33
Figure 4.4: An Example of an Explicit Artifact Conflict Anomaly	34
Figure 4.5: An Example of an Implicit Artifact Conflict Anomaly	35
Figure 4.6: An Example of a Production Conflict Anomaly	36
Figure 4.7: An Example of a Passing between Parallel Activities Anomaly	37
Figure 4.8: An Example of a Passing between Exclusive Activities Anomaly	38
Figure 4.9: An Example of a Redundant Update/Initialization Anomaly	39
Figure 4.10: An Example of a Redundant Pass Anomaly	40
Figure 5.1: An Example of Building XBNodes	60
Figure 5.2: The Corresponding Artifact Flow Diagram for d	61
Figure 6.1: An Example of a Control Flow Graph	68
Figure 6.2: The Artifact Flow Diagram for d	69
Figure 6.3: The Artifact Flow Diagram After Modifying v_9	71

Figure 6.4: The Artifact Flow Diagram After Modifying v_8 72

Figure 6.5: The Artifact Flow Diagram After Modifying v_2 73

Figure 6.6: The Artifact Flow Diagram After Adding $\text{Pass}(d, v_{10})$ into PassList_{v_9} 75

Figure 6.7: The Artifact Flow Diagram After Adding $\text{Pass}(d, v_6)$ into PassList_{v_3} 76

Figure 6.8: The Artifact Flow Diagram After Adding $\text{Pass}(d, v_6)$ into PassList_{v_8} 77

Figure 6.9: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_8)$ from PassList_{v_1} 79

Figure 6.10: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_7)$ from PassList_{v_4} 80

Figure 6.11: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_3)$ from PassList_{v_1} 81

Figure 7.1: A Control Flow in GDS 83

Figure 7.2: The Artifact Diagrams for Two Execution Orders 84



Chapter 1. Introduction

Workflow is a set of tasks which are systematized to achieve certain business goals by completing each task in a particular order under automatic control [1]. On the other hand, resources are necessary for workflow implementation and support process execution. Resource allocation and resource constraint analysis [2 - 6] are popular topics of workflow research. However, data flow within workflow is seldom addressed [7, 8].

Artifacts are collections of data items involved in a process. Introducing analysis of artifact usage into workflow designs might help maintain data consistency, as well as prevent the exceptions. In contrast to structural correctness, accuracy in artifact manipulation can help determine whether the execution result of a workflow is meaningful and desirable.

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 discussion about any implementation or formal method to demonstrate how to apply their researches and which types of workflow model are compatible with their activity-based data model.

Sun et al. [8] presented a data flow analysis framework for detecting data flow anomalies such as missing data, redundant data, and potential conflicts of data. In addition, several algorithms were provided to detect anomalies, however, the work is done only based on read and first initial write operations of an artifact.

Jin Hyun Son [14] defined a well-formed workflow based on the concepts of closure and control block. He claimed that a well-formed workflow is free from structure errors, and that complex control flows can be made with nested control blocks.

Aalst [9] identifies three major artifact transmission models in a workflow: (1) Global Data Store (GDS), (2) Integrated Control and Data Channels (ICDC), and (3) Distinct Control and Data Channels (DCDC). DCDC is more flexible for representing artifact transmission than GDS and ICDC. Therefore, this thesis proposes a process model for describing a well-formed workflow. The artifact transmissions in this model are based on DCDC model and four types of artifact usage anomalies are addressed.

Further, each artifact in this model has a corresponding artifact flow diagram for representing its transmissions and usages. This thesis presents a set of incremental algorithms to update the artifact flow diagrams when designer edits a workflow. By analyzing the updated artifact flow diagram, the artifact usage anomalies can be detected meanwhile. The warning messages are provided to the designers if artifact usage anomalies occur.

The remainder of this thesis is organized as follows. Chapter 2 presents the related work. Chapter 3 presents our process modeling based on well-formed workflow and the artifact flow diagram. Chapter 4 identifies four types of artifact usage anomalies. Chapter 5 proposes several incremental algorithms to update artifact flow diagram and detect artifact usage anomalies. Chapter 6 demonstrates our incremental algorithms with several scenarios. Chapter 7 compares this thesis with related work. Conclusions and future works are finally drawn in chapter 8.

Chapter 2. Related Work

To develop an effective and reliable workflow application, a well-defined workflow model is necessary. The correctness issues in a workflow might be classified into three dimensions: control flow, resource, and data flow. The analysis of control flow aspect includes structural correctness focuses on soundness of control logic [10], process model analysis, workflow patterns [11, 12], and automatic control of workflow process [13], etc. The analysis of resource aspect includes resource allocation constraints [2], resource availability [3], resource management [5], and resource modeling [6], etc. The analysis of data flow aspect includes data flow validation [7], data flow formulation [8], and artifact usage anomalies detection [8][15], etc,

2.1 Artifact Transmission Models

Aalst [11] identifies three major artifact transmission models in a workflow: (1) Global Data Store, (2) Integrated Control and Data Channels, and (3) Distinct Control and Data Channels. These transmission models are illustrated in Figure 2.1.

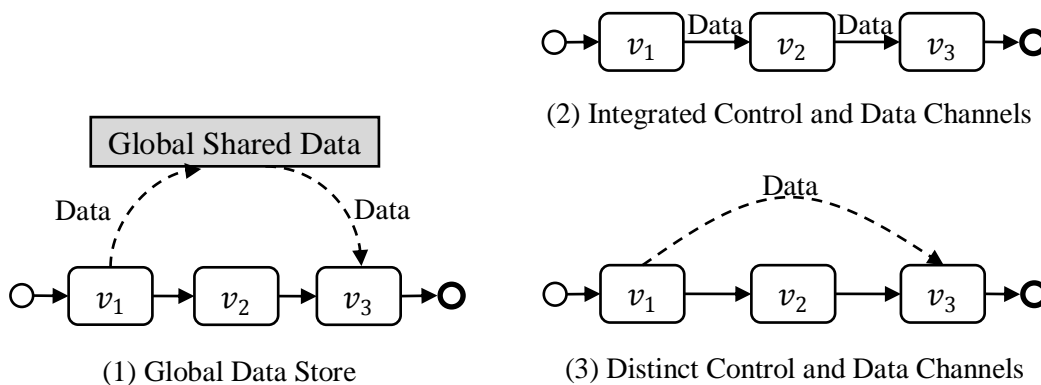


Figure 2.1: Three Major Artifact Transmission Models.

(1) Global Data Store (GDS):

Each artifact is only allowed to have one instance in a workflow and the instance is typically stored in a global shared data store. All activities share the same artifact instances in the global data store.

(2) Integrated Control and Data Channels (ICDC):

In this model, all artifacts are passed with control flows regardless of whether the next activity will use them or not.

(3) Distinct Control and Data Channels (DCDC):

Artifacts are passed between activities via explicit channels [16] which are distinct from control flows. Hence, each activity can decide where artifacts are passed.

Based on the introductions above, we discover that some behaviors or properties in DCDC are hard to represent in GDS and ICDC. For example in the aspect of artifacts security, let an artifact d in Figure 2.1 be updated by v_1 and only allowed to be read by v_3 . In GDS, each activity can access the artifact d because it is store in the global data store. Therefore, it is hard to limit other activities to read or update d before v_3 reads it.

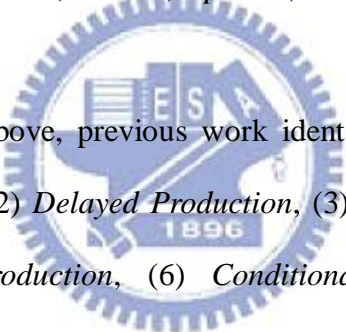
In ICDC, if v_3 is required to read the artifact d updated by v_1 , d has to be transferred through the activities between v_1 and v_3 in the control flow. In Figure 2.1 (2), v_2 will receive d from v_1 and pass it to v_3 regardless of whether v_2 requires d or not. Hence, it is also hard to limit v_2 to read or update d before v_3 receives it.

However, each activity in DCDC can decide where artifacts are passed. Activity v_1 in Figure 2.1 (3) can simply decide to pass d to v_3 and only v_3 will receive it. The other activities without receiving d can not access it. Therefore, DCDC is more flexible and is adapted to represent artifact transmission in this thesis.

2.2 Previous Work

Previous work presents a process model to describe workflow schemas. The definitions of a process, an activity, and a control block are proposed. The relations among activities and control blocks, e.g. paths, reachability, predecessors, successors, parallel activities, and exclusive activities, are also defined.

In order to simplify analysis the artifact usages in GDS, every artifact operation can be regarded as one of the following operations: *Initialize*, *Read*, *Updated*, and *Destroy*, regardless of its semantic meaning. Further, the usage relation between an activity and an artifact can be identified as: *Producer*, *Reader*, *Updater*, *Destroyer*.



Based on the definitions above, previous work identifies the following artifact usage anomalies: (1) *No Production*, (2) *Delayed Production*, (3) *Early Destruction*, (4) *Exclusive Production*, (5) *Uncertain Production*, (6) *Conditional Production*, (7) *Conditional Destruction*, (8) *Uncertain Destruction*, (9) *Explicit Redundant Update*, (10) *Potential Redundant Update*, (11) *Multiple Parallel Productions*, (12) *Multiple Parallel Updates*, and (13) *Parallel Read and Update*. After identifying the causing conditions of each anomaly, previous work proposes a batch algorithm to traversal a control flow and detect whether anomalies occur or not.

Since previous work only discusses artifact transmission in GDS, the activities share the same artifact instances stored in global data store and an artifact is only allowed to have one instance in a process. However, the activity in DCDC can decide where artifacts are passed and an artifact is allowed to have multiple instances in a process.

Therefore, we extend the process model of previous work and present an artifact flow diagram for each artifact to represent the artifact usages and transmissions in this thesis. Based on the information in artifact flow diagram, the artifact usage anomalies in DCDC can be identified. The differences of anomalies between GDS and DCDC are discussed in Section 7.1. The comparison of the anomalies addressed in previous work and our work is discussed in Section 7.2.

Furthermore, our model proposes several edit operations for editing a process. In order to update the artifact flow diagram after each edit operation incrementally, the incremental algorithms are introduced for each edit operation. The detailed algorithms about how to construct artifact flow diagram are also described in this thesis.



Chapter 3. A Process Model Based on Well-formed Workflow

3.1 Control Flow Specification

As discussed in Section 2.2, [15] does not concern the activities can decide where artifacts are passed and the artifacts can have multiple instances in a workflow. To solve these problems, there are several features introduced in addition in this thesis. The new features are described in the following paragraphs. First, Definition 3.1 formally defines a process specification and Definition 3.2 describes the fundamental properties contained in each activity.

Definition 3.1 (Process Specification)[15]

A process specification $p = (V_p, F_p, R_p, C_p, S_p, E_p)$, where

- V_p : The set of activities in p . The id of each activity is unique.
 - F_p : The set of flows in p .
 - R_p : The set of resources (artifacts) used in p .
 - C_p : The set of control blocks in p . The id of each control block is unique.
 - S_p : The start activity of p .
 - E_p : The end activity of p .
- $S_p, E_p \in V_p$.
- $Child_p = \{sp \mid \forall \text{compound activity } v \in V_p, v.subp = sp\}$.

Definition 3.2 (Activity Fundamental Properties)

$\forall v \in V_p$, there are two fundamental properties in v :

- $v.type \in \{\text{Task, ProcessStart, ProcessEnd, XorSplit, XorJoin, AndSplit, AndJoin, LoopSubProcess, SubProcess}\}$.
- $ABStack_v$: A stack stores all the control blocks and branches where v resides. (The control block and branch are defined further in Definition 3.4 and 3.6.)

For each activity $v \in V_p$, the property $ABStack_v$ is a new feature and its usage is illustrated in Definition 3.9. The property $v.type$ represents v 's type. If $v.type$ is SubProcess or LoopSubProcess, v is a compound activity. Definition 3.3 below introduces a new feature: $subp$ to a compound activity.

Definition 3.3 (Compound Activity)

$\forall v \in V_p$, v is a compound activity when $v.type \in \{\text{SubProcess, LoopSubProcess}\}$.

- $v.type \in \{\text{SubProcess, LoopSubProcess}\}$
- $v.subp$: The sub-process included within v .

In our model, there are three types control blocks, "ROOT Control Block", "AND Control Block", and "XOR Control Block", and the properties to record the execution order of these blocks are defined in Definition 3.4. Further, for an activity v , v is *contained* in c when v is reachable from $c.start$ and $c.end$ is reachable from v . For a flow $f = (u, v) \in F_p$, f is *contained* in control block c when u and v are both contained in c .

Definition 3.4 (Control Block)

A control block $c = (start, end, type) \in C_p$, where

- $c.start = v_s, v_s \in V_p$ and $v_s.type \in \{\text{ProcessStart, XorSplit, AndSplit}\}$
- $c.end = v_e, v_e \in V_p$ and $v_e.type = \begin{cases} \text{ProcessEnd} & \text{if } v_s.type = \text{ProcessStart} \\ \text{XorJoin} & \text{if } v_s.type = \text{XorSplit} \\ \text{AndJoin} & \text{if } v_s.type = \text{AndSplit} \end{cases}$
 - $ABStack_{v_s} = ABStack_{v_e}$
 - $\nexists v \in V_p, v.type \in \{\text{ProcessStart, ProcessEnd, XorSplit, XorJoin, AndSplit, AndJoin}\}, \text{IsReachable}(c.start, v) = \text{ture}, \text{IsReachable}(v, c.end) = \text{ture},$ and $ABStack_v = ABStack_{v_s}.$
- $c.type \in \{\text{ROOT, AND, XOR}\}$
 - $c.type = \begin{cases} \text{ROOT} & \text{if } v_s.type = \text{ProcessStart} \\ \text{XOR} & \text{if } v_s.type = \text{XorSplit} \\ \text{AND} & \text{if } v_s.type = \text{AndSplit} \end{cases}$
- $c.bcounter$ is a counter of branch id. The value of $bcounter$ for c is set to 0 at the beginning of c 's construction.
- $c.totalbranches = |\text{outflows of } c.start|.$ (The outflow is defined in Definition 3.7.)

For each control block $c \in C_p$, the properties $c.start$ and $c.end$ represent the start activity and end activity of c separately. If activity v_s is the start activity and v_e is the end activity, the following properties hold:

- (a) $ABStack_{v_s} = ABStack_{v_e}.$
- (b) There is no such an activity $v \in V_p, v.type \in \{\text{ProcessStart, ProcessEnd, XorSplit, XorJoin, AndSplit, AndJoin}\}, v$ is reachable from v_s, v_e is reachable from $v,$ and $ABStack_v = ABStack_{v_s}.$

The function $\text{IsReachable}(c.start, v)$ in Definition 3.4 represents whether v is reachable from $c.start$. If v is reachable from $c.start$, the function returns *true*. Otherwise, the function

returns *false*. The function is defined in Definition 3.11.

Based on the type of $c.start$, the property $c.type$ is one of the following types: “ROOT”, “AND”, and “XOR”. The property $c.bcounter$ is a new feature and is used to count branch ids. The value of $c.bcounter$ is increased by one when a new branch is added into c . The property $c.totalbranches$ represents the number of branches in c .

Each process is only allowed to contain one root control block R . For a process p , $p.start = ps$, $p.end = pe$, the root control block $R = (ps, pe, \text{ROOT})$ and $R.totalbranches$ is always 1.

Based on the Definition 3.4, a control activity is defined in Definition 3.5.

Definition 3.5 (Control Activity)

$\forall v \in V_p$, v is a control activity.

- $v.type \in \{\text{ProcessStart}, \text{ProcessEnd}, \text{AndSplit}, \text{AndJoin}, \text{XorSplit}, \text{XorJoin}\}$.
- v is the start activity of a control block if $v.type \in \{\text{ProcessStart}, \text{AndSplit}, \text{XorSplit}\}$.
- v is the end activity of a control block if $v.type \in \{\text{ProcessEnd}, \text{AndJoin}, \text{XorJoin}\}$.
- $v.cb$ represents the id of the block beginning from or ending at v .
- $v.cb = R$ if $v.type \in \{\text{ProcessStart}, \text{ProcessEnd}\}$.

In addition, if an activity $v \in V_p$ is neither a control activity nor a compound activity, v is a task activity and $v.type = \text{Task}$.

Based on Definition 3.5, a branch in a control block is basically a path. Also the path begins at the start activity of the control block, and ends at the end activity of the control

block. In order to identify different branches in a control block, each branch contains a property *id*. When a new branch *b* is added into a control block *c*, *c.bcounter* is added by one and is set value of *b.id*. A branch is defined in Definition 3.6.

Definition 3.6 (Branch)

In a control block *c* which is not sequence, a branch $b = (v_1, \dots, v_k), k \geq 2$, where

- $\exists \text{flow } (v_i, v_{i+1}) \in F_p, i = 1, 2, \dots, k-1$
- $v_1 = c.start$
- $v_k = c.end$
- *b.id* represents the id of *b*.

A flow in workflow specification represents the execution order of activities and is defined in Definition 3.7.



Definition 3.7 (Flow)

$\forall f = (u, v) \in F_p$, where $u, v \in V_p$.

- *u* is the *source activity* of *f*, and *v* is the *sink activity* of *f*.
- *f* is an *inflow* of *v* and an *outflow* of *u*.

Figure 3.1 shows the corresponding notations of control activities, task activity, sub-process activity, loop sup-process activity, and flow.

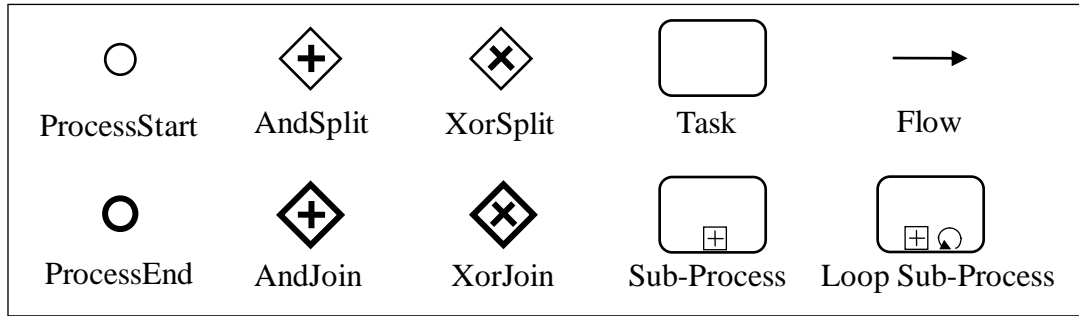


Figure 3.1: Notations of Control Flow Graph.

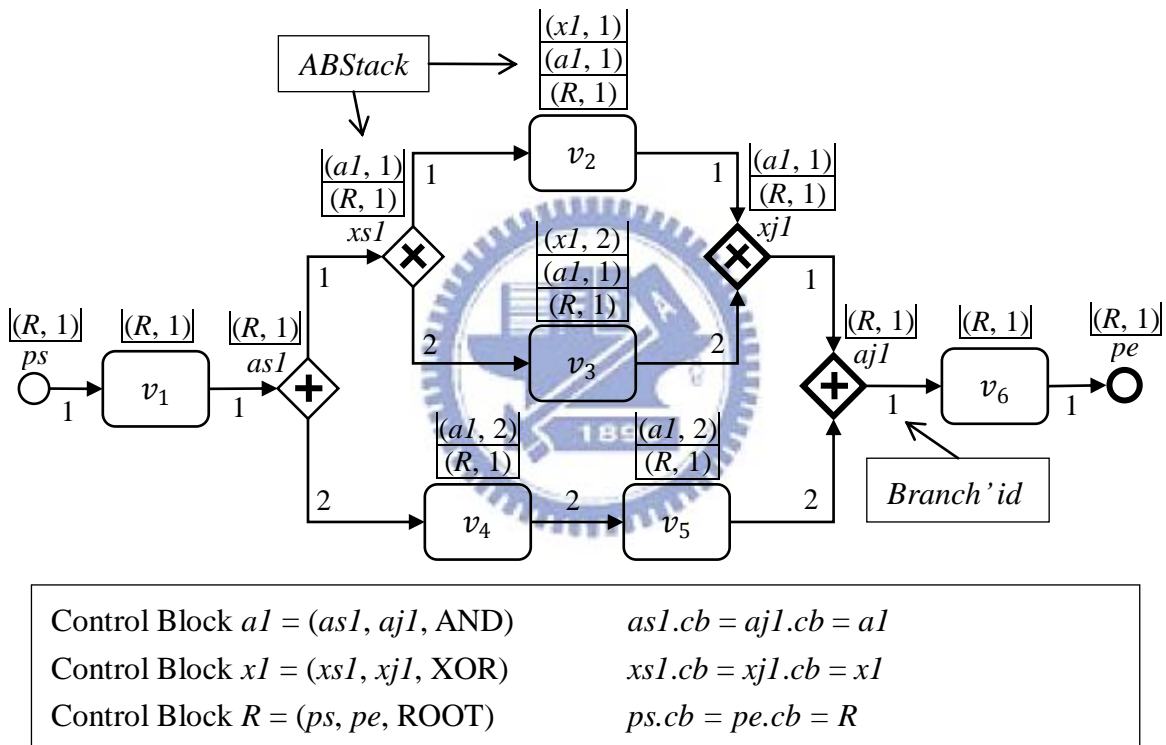


Figure 3.2: An Example of Control Flow Graph.

Figure 3.2 shows an example of a *well-formed* control flow. In Figure 3.2, each activity is associated with an Ancestor Block Stack, $ABStack$, showing the control blocks where the activity resides. The properties about the relations between elements in an $ABStack$ are described after illustrating the elements in Definition 3.8. Since all activities are contained in root control block R and $R.totalbranches$ is always 1, an element $(R, 1)$ is in the bottom of

each $ABStack$. For an activity v , each element in $ABStack_v$ marks a control block containing v . An element inside an $ABStack$ is formally defined in Definition 3.8.

Definition 3.8 (The element inside an $ABStack$)

An element e inside $ABStack_v$ contains two tuples $(blockID, branchID)$,

- $blockID$ represents a control block containing v .
- $branchID$ represents the id of the branch after the initial control activity of $blockID$.

An $ABStack$ has the following properties:

1. Assume two elements, $e1$ and $e2$, exist in an $ABStack$. Element $e1$ is above $e2$ if and only if the block represented by the first tuple of $e1$ is contained in that represented by $e2$. For example, $ABStack_{v_2}$ in Figure 3.2 has two elements $e1=(x1, 1)$ and $e2=(a1, 1)$. Because $x1$ is contained in $a1$, $e1$ is above $e2$. By reading the elements top-down in $ABStack_{v_2}$, it is found that v_2 is located on branch 1 of $x1$, $x1$ is located on branch 1 of $a1$, and $a1$ is located in R . Thus, the $ABStack_{v_2}$ can be used to represent the location of activity v_2 . In this case, because $x1$ is contained in $a1$, $x1$ is deeper than $a1$ in this nested control blocks structure. In the same reason, because $a1$ is contained in R , $a1$ is deeper than R . Therefore, $x1$ is the deepest control block which contains v_2 because $x1$ does not contain any control block.
2. Consider $ABStack_u$ and $ABStack_v$, if there are some elements contained in $ABStack_u$ but not in $ABStack_v$, u is contained in the blocks represented by the elements but v is not. For example, element $(x1, 1)$ is contained in $ABStack_B$ but not contained in $ABStack_{xj1}$ in Figure 3.2. Therefore, control block $x1$ contains activity v_2 but not activity $xj1$.

The operations associated with $ABStack$ are defined in Definition 3.9. The construction of an $ABStack$ is described in Algorithm 3.1.

Definition 3.9 (The operations associated with an $ABStack$)

The operations associated with an $ABStack$ include:

1. $push(E: \text{element})$: Pushing an element into the top of $ABStack$.
2. $pop()$: Popping an element out from the top of $ABStack$.
3. $isEmpty()$: If the $ABStack = \emptyset$, return *true*. Otherwise return *false*.
4. $getTopXOR()$: Getting the first element from the top of $ABStack$ and the type of the block associated with the element is XOR. For example, let $ABStack_v = \begin{matrix} \boxed{(a2, 1)} \\ \boxed{(x2, 2)} \\ \boxed{(x1, 1)} \end{matrix}$, $x1$ and $x2$ be XOR control blocks, $a2$ be an AND control block, then $ABStack_v.getTopXOR() = (x2, 2)$.
5. $removeIdenticalElements(A: ABStack)$: Removing the same elements in both A and the target $ABStack$. For example, $ABStack_v = \begin{matrix} \boxed{(a2, 1)} \\ \boxed{(x2, 2)} \\ \boxed{(R, 1)} \end{matrix}$ and $ABStack_u = \begin{matrix} \boxed{(x2, 2)} \\ \boxed{(R, 1)} \end{matrix}$, $ABStack_v.removeIdenticalElements(ABStack_u) = \boxed{(a2, 1)}$.

Algorithm 3.1 (Construction of an $ABStack$)

$\forall v \in V_p$, $ABStack_v$ is constructed by the following steps:

Algorithm Construct $ABStack(v)$ {

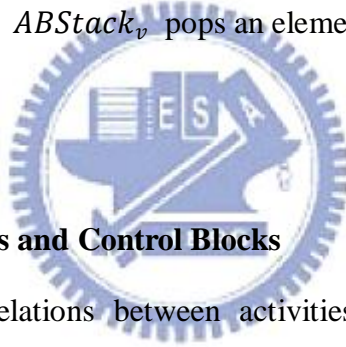
1. **if**($v.type == \text{ProcessStart}$){
2. $ABStack_v = \boxed{(R, 1)}$;
3. }**else if**($v.type \neq \text{ProcessStart}$ and $\exists \text{inflow } f=(u, v) \in F_p$ and f is located in branch b) {
4. **if**($u.type \notin \{\text{XorSplit}, \text{AndSplit}\}$) {
5. $ABStack_v = ABStack_u$;
6. }**else**{
7. Create an element $e1 = (u.cb, b.id)$;
8. $ABStack_v = ABStack_u.push(e1)$;
9. }

```

10.     if( $v.type \in \{XorJoin, AndJoin\}$ )
11.          $ABStack_v.pop()$ ;
12.     }
13. }
}

```

At line 1 to 2, if the type of v is ProcessStart, $ABStack_v$ only contains one element(R , 1). If the type of v is not ProcessStart, v has inflow(s). Let f be the inflow (u, v) and f be located in branch b . If v has multiple inflows, random one of them is assigned to f . At line 4 to 9, if the type of u is not XorSplit and AndSplit, $ABStack_v = ABStack_u$. Otherwise, u is the start activity of a control block. Hence, $ABStack_v$ is equal to $ABStack_u$ which is added an element ($u.cb, b.id$). At line 10 to 12, if the type of v is XorJoin or AndJoin, v is the end activity of a control block. Thus, $ABStack_v$ pops an element.



3.2 Relations between Activities and Control Blocks

In a process, there are relations between activities and control blocks; e.g. path, reachability, etc. These relations are defined in the following definitions.

Definition 3.10 (Path)[17]

A path $q = (v_1, \dots, v_k)$ in which $k \geq 2$ and the flow $f = (v_i, v_{i+1})$ for $i=1,2,\dots,k-1 \in F_p$.

The path from v_1 to v_k is denoted by $Path(v_1, v_k)$.

Definition 3.11 (Reachability)[17]

Given two activities u and v , $IsReachable(u, v)$ is a Boolean function that indicates whether there is a path from u to v . I.e.,

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

Definition 3.12 (Predecessors and Successors)

$$V_v^{isPredecessor} = \{u \in V_p \mid IsReachable(u, v) = true\}$$

$$V_v^{isSuccessor} = \{u \in V_p \mid IsReachable(v, u) = true\}$$

$V_v^{isPredecessor}$ is a set of activities and v is reachable from each activity in $V_v^{isPredecessor}$. Each activity u in $V_v^{isPredecessor}$ is called a *predecessor* of v . $V_v^{isSuccessor}$ denotes the transitive closure of $V_v^{isPredecessor}$.

Definition 3.13 (Parallel Activities)

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

$$IsParallel(u, v) = true \leftrightarrow$$

$$\exists e1 \in ABStack_u, e2 \in ABStack_v \wedge \begin{cases} e1.blockID = e2.blockID \\ e1.branchID \neq e2.branchID \\ e1.blockID.type = AND \end{cases}$$

Definition 3.14 (Exclusive Activities)

Given two activities u and v , $IsExclusive(u, v)$ is a Boolean function to represent if u is selected for execution then v won't be selected for execution and vice versa.

$$IsExclusive(u, v) = true \leftrightarrow$$

$$\exists e1 \in ABStack_u, e2 \in ABStack_v \wedge \begin{cases} e1.blockID = e2.blockID \\ e1.branchID \neq e2.branchID \\ e1.blockID.type = XOR \end{cases}$$

To simplify our analysis, the workflow specifications discussed in this thesis are well-formed. A well-formed workflow has four basic control structures: sequential, exclusive split, parallel split, and loop. The control blocks defined in Definition 3.4 can be used to represent exclusive split, and parallel split structures. As in BPMN [18], a loop structure can be replaced by a loop sub-process activity in our model.

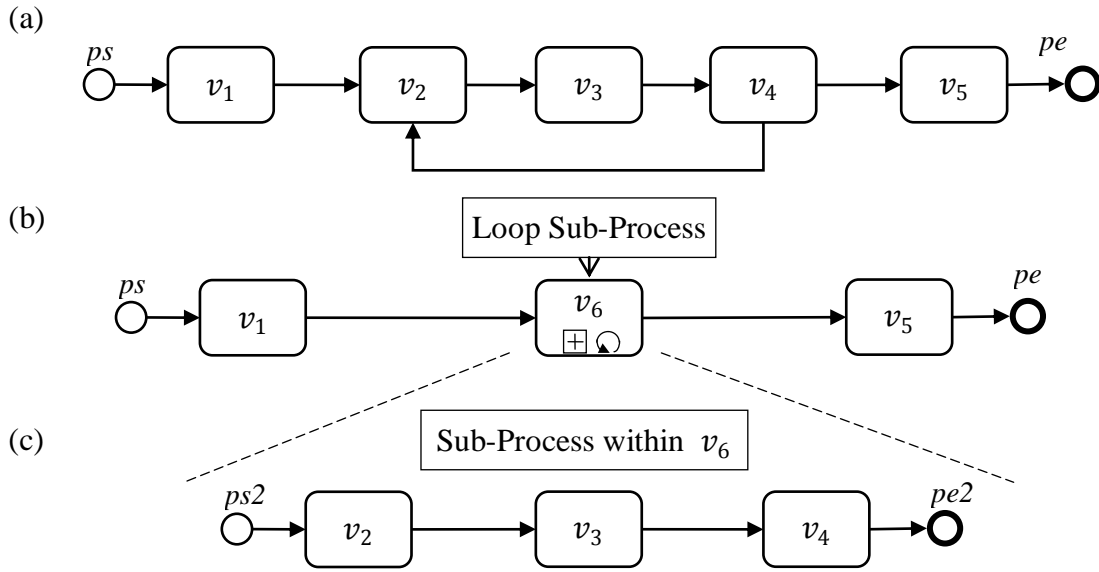


Figure 3.3: Using a Loop Sub-Process Activity to Replace a Loop Structure.

Figure 3.3 (a) shows an example of a well-formed control flow and activities v_2 , v_3 , and v_4 are located in a loop structure. In Figure 3.3 (b), a loop sub-process activity v_6 is used to replace the loop structure and v_6 contains a sub-process including v_2 , v_3 , and v_4 in Figure 3.3 (c). Therefore, there is no cycle in a control flow graph in our model; i.e. a control flow graph in our model is a directed acyclic graph (DAG).

3.3 Artifact Flow Diagram Specification

As identified in [11], there are three models used to define artifact flow transmission in a workflow: (1) Global Data Store (GDS), (2) Integrated Control and Data Channels (ICDC), and (3) Distinct Control and Data Channels (DCDC). Since DCDC is more flexible for representing artifact flow, DCDC is adapted to express artifact flow in this thesis.

In this thesis, a verb “*pass*” is used to represent the action about sending an artifact to another activity [11]. For example, if an artifact d is sent from an activity v to u , it is described as that activity v *passes* artifact d to u . In other words, activity u *receives* artifact d from v .

In DCDC, artifacts are passed between activities via explicit channels [16] which are distinct from control flows. Hence, each artifact has a corresponding artifact flow diagram representing the artifact usages and transmissions in a workflow. In addition, artifacts transfers are passed by value in our model and an activity starts execution when it receives all necessary artifacts.

3.3.1 Artifact Operations and Usages

Artifacts are collections of data items involved in a process. Intuitively, all artifacts participating in a workflow execution are pre-defined in the process specification. Each artifact contains a set of legal operations for its internal data, and is applied in an activity to perform them. In the aspect of data usage, artifact operations can be conceptually classified as *initialize*, *read*, *update*, and *delete* [15].

Because artifacts are passed between activities via explicit channels distinct from control flows, activities can decide where the artifacts are passed. Here, an operation $\text{Pass}()$, passing an artifact to another activity, is defined. Let PassList_v denote a linked list to record the pass operations in an activity v . Thus, $\text{Pass}(d, u) \in \text{PassList}_v$ if and only if v passes artifact d to activity u . In this case, v is a sender of d and u is a receiver of d .

Besides, the receiver can not receive the artifact if the receiver is executed before the sender. Hence, an activity can not pass artifacts to its predecessors.

For an activity v , the operations $\text{Pass}()$ in PassList_v are performed only when all the other operations, such as initialize, read, update, and delete, are completed.

In our model, artifact transfers are passed by value, more than one copy of an artifact may exist in a workflow. For example, given an artifact d and activities v, u, w , if $\text{Pass}(d, u)$ and $\text{Pass}(d, w)$ are contained in PassList_v , activities u and w will receive d separately. This case is called *artifact split* and there are two copies of d sent out after v .

In order to identify the artifact usages in an activity, each activity contains the following input and output artifacts sets: I, O, U^+ , and U^- . These sets are defined in Definition 3.15.

Definition 3.15 (Input and Output Artifacts Sets)

For an activity v , v contains the following sets:

- I_v is a set of artifacts, of which each is read, updated, or deleted in v .
- O_v is a set of artifacts, of which each is passed from v after v 's executing.
- U_v^+ is a set of artifacts, of which each is initialized or updated in v .
- U_v^- is a set of artifacts, of which each is deleted in v .

To simplify our discussion, an activity v can be classified into the following roles of d : Producer, Reader, Updater, Destroyer, Irrelevantor, and Relevantor, based on the memberships between an artifact d and I_v, O_v, U_v^+ , and U_v^- . Definition 3.16 shows how to identify these roles.

Definition 3.16 (Roles of an Activity in an Artifact Flow Diagram)

For an activity v , $Role_v^d$ denotes v 's role of d . $Roles(v, d)$ is a function to identify v 's role of d . All the possible usages are categorized as follows:

if($v.type \notin \{ProcessStart, ProcessEnd\}$){

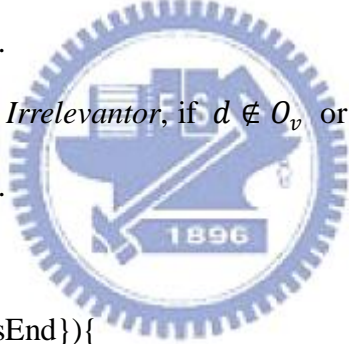
- $Role_v^d = Roles(v, d) = Producer$ when $d \notin I_v$ and $d \in U_v^+$.
- $Role_v^d = Roles(v, d) = Destroyer$ when $d \in I_v$ and $d \in U_v^-$.
- $Role_v^d = Roles(v, d) = Reader$ when $d \in I_v$ and $\begin{cases} d \notin U_v^+ \\ d \notin U_v^- \end{cases}$.
- $Role_v^d = Roles(v, d) = Updater$ when $d \in I_v$ and $d \in U_v^+$.
- When $d \notin I_v$ and $\begin{cases} d \notin U_v^+ \\ d \notin U_v^- \end{cases}$.
 - $Role_v^d = Roles(v, d) = Relevantor$, if $d \in O_v$ or $\exists u \in V_p$ and $Pass(d, v) \in PassList_u$.
 - $Role_v^d = Roles(v, d) = Irrelevantor$, if $d \notin O_v$ or $\nexists u \in V_p$ and $Pass(d, v) \in PassList_u$.

}

if($v.type \in \{ProcessStart, ProcessEnd\}$){

- For a ProcessStart Activity ps :
 - $Role_{ps}^d = Roles(ps, d) = Producer$ when $d \in O_{ps}$.
 - $Role_{ps}^d = Roles(ps, d) = Irrelevantor$ when $d \notin O_{ps}$.
- For a ProcessEnd Activity pe :
 - $Role_{pe}^d = Roles(pe, d) = Destroyer$ when $\exists u \in V_p$ and $Pass(d, pe) \in PassList_u$.
 - $Role_{pe}^d = Roles(pe, d) = Irrelevantor$ when $\nexists u \in V_p$ and $Pass(d, pe) \in PassList_u$.

}



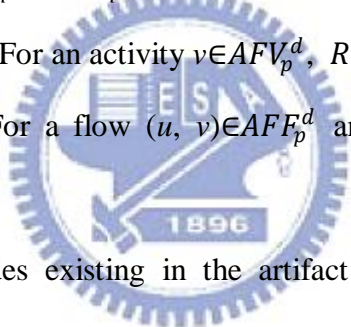
3.3.2 Artifact Flow Diagrams

Each artifact in a workflow has a corresponding artifact flow diagram to represent the usages and transmissions of it. For artifact d , an activity v is in the artifact flow diagram for d if d is used by v ; i.e. $Role_v^d \neq \text{Irrelevantor}$. The flows between activities in the artifact flow diagram for d represent the transmissions of d . Definition 3.17 defines an artifact flow diagram for an artifact.

Definition 3.17 (An Artifact Flow Diagram)

For an artifact d , artifact flow diagram for d in process p is denoted by AF_p^d which contains two tuples $(AFV_p^d, AFF_p^d, AFN_p^d)$, where

- AFV_p^d is a set of activities. For an activity $v \in AFV_p^d$, $Role_v^d \neq \text{Irrelevantor}$.
- AFF_p^d is a set of flows. For a flow $(u, v) \in AFF_p^d$ and $u, v \in AFV_p^d$, a $Pass(d, v) \in PassList_u$.
- AFN_p^d is a set of XBNodes existing in the artifact flow diagram. An XBNode is introduced in the following paragraphs and is defined in Definition 3.19.



In Definition 3.17, a flow (u, v) is added into AFF_p^d when a $Pass(d, v)$ is added into $PassList_u$. If there is a Path $(v, u) \in AF_p^d$, a loop structure is formed in the artifact flow diagram after adding flow (u, v) into AFF_p^d . The activities in the loop wait for the artifact cyclically and a deadlock occurs. In order to avoid the deadlock, the loop structure is not allowed to exist in an artifact flow diagram. Therefore, $Pass(d, v)$ can not be added into $PassList_v$ when a Path (v, u) exists in AF_p^d .

In order to detect some artifact usage anomalies, the number of an artifact passed/received from/in an activity is required to be identified. For an activity v , let

$Receive_v^d$ be a set of activities, of which each passes d to v . Let $Send_v^d$ be a set of activities, of which each receives d from v . Therefore, $|Receive_v^d|$ represents the number of inflows of v in the artifact flow diagram for d . On the contrary, $|Send_v^d|$ represents the number of outflows of v in the artifact flow diagram for d .

However, some activities in $Receive_v^d$ might not pass d to v in run time. Therefore, $|Receive_v^d|$ is not always equal to the number of d received in v . Similarly, $|Send_v^d|$ is not always equal to the number of d passed from v . In Figure 3.4 (a), (b), and (c), activity v_1 passes artifact d to v_2 and v_3 respectively. Hence, their artifact flow diagrams for d in Figure 3.4 (d) contain activities: v_1 , v_2 , and v_3 , and flows: (v_1, v_2) and (v_1, v_3) . $Send_{v_1}^d = \{v_2, v_3\}$ and $|Send_{v_1}^d| = 2$ in Figure 3.4 (a), (b), and (c).

In Figure 3.4 (a), because v_2 and v_3 are parallel activities, the number of d passed from v_1 is 2. In Figure 3.4 (b), because v_2 and v_3 are exclusive activities, the number of d passed from v_1 is 1. In Figure 3.4 (c), v_2 , v_3 , and v_5 are exclusive activities and v_1 does not pass d to v_5 . Therefore, the number of d passed from v_1 is 0 if v_5 is selected to execute. Otherwise, the number of d passed from v_1 is 1. Obviously, the number of d passed from v_1 is not always equal to $|Send_{v_1}^d|$. The artifact flow diagram in Figure 3.4 (d) is ambiguous to represent the number of an artifact passed/received from/in an activity.

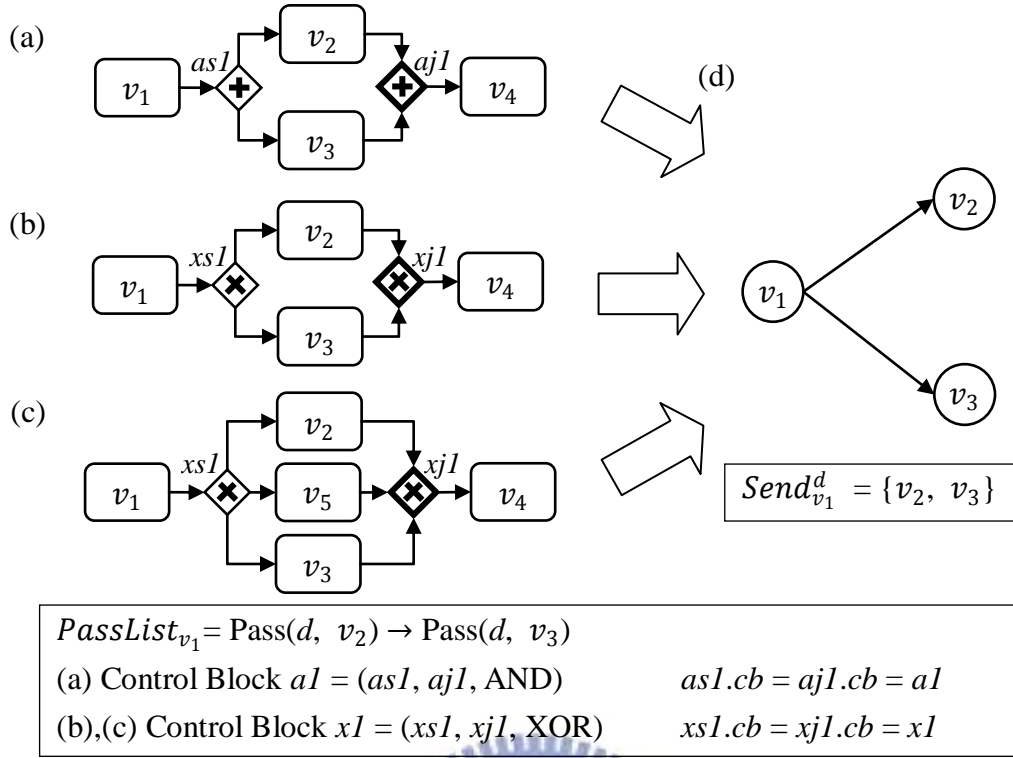


Figure 3.4: An Artifact Flow Diagram without XBNodes.

In order to solve this problem, the activities in $Receive_v^d$ are categorized into two sets: EIN_v^d and PIN_v^d . PIN_v^d contains the activities passing d to v potentially. EIN_v^d contains the activities passing d to v explicitly. Similarly, the activities in $Send_v^d$ are categorized into two sets, $POUT_v^d$ and $EOUT_v^d$. $POUT_v^d$ contains the activities receiving d from v potentially. $EOUT_v^d$ contains the activities receiving d from v explicitly.

For example in Figure 3.4 (a), because v_1 always passes d to v_2 and v_3 , $EOUT_{v_1}^d = \{v_2, v_3\}$ and $POUT_{v_1}^d = \emptyset$. In Figure 3.4 (b), activity v_1 does not pass d to v_2 when v_2 is not selected to execute in xl . Hence, v_1 passes d to v_2 potentially. For the same reason, v_1 also passes d to v_3 potentially. Therefore, $EOUT_{v_1}^d = \emptyset$ and $POUT_{v_1}^d = \{v_2, v_3\}$. The formal definitions of these sets are defined in Definition 3.18.

Definition 3.18 (Receiving/Sending Sets)

For activity $v \in AFV_p^d$, v contains the following sets:

- $Receive_v^d = \{u \in AFV_p^d \mid \exists \text{flow}(u, v) \in AFF_p^d\}$
 - $PIN_v^d =$
 $\{u \in Receive_v^d \mid (\exists e \in ABStack_u. \text{removeIdenticalElements}(ABStack_v)) \text{ and}$
 $e.blockID.type = XOR \text{ and } IsExclusive(u, v) \neq true\}$
 - $EIN_v^d = \{u \in (Receive_v^d \setminus PIN_v^d) \mid IsExclusive(u, v) \neq true\}$
- $Send_v^d = \{u \in AFV_p^d \mid \exists \text{flow}(v, u) \in AFF_p^d\}$
 - $POUT_v^d =$
 $\{u \in Send_v^d \mid (\exists e \in ABStack_u. \text{removeIdenticalElements}(ABStack_v)) \text{ and}$
 $e.blockID.type = XOR \text{ and } IsExclusive(v, u) \neq true\}$
 - $EOUT_v^d = \{u \in (Send_v^d \setminus POUT_v^d) \mid IsExclusive(v, u) \neq true\}$

For a flow $(u, v) \in AFF_p^d$, if u and v are exclusive activities, u and v can not both be executed. Therefore, v can not receive d from u . For this reason, u is not added into PIN_v^d and EIN_v^d when $IsExclusive(u, v) = true$. Similarly, u is not added into $POUT_v^d$ and $EOUT_v^d$ when $IsExclusive(v, u) = true$. Based on Definition 3.18, Figure 3.5 shows the elements in $Send_{v_1}^d$, $EOUT_{v_1}^d$, and $POUT_{v_1}^d$.

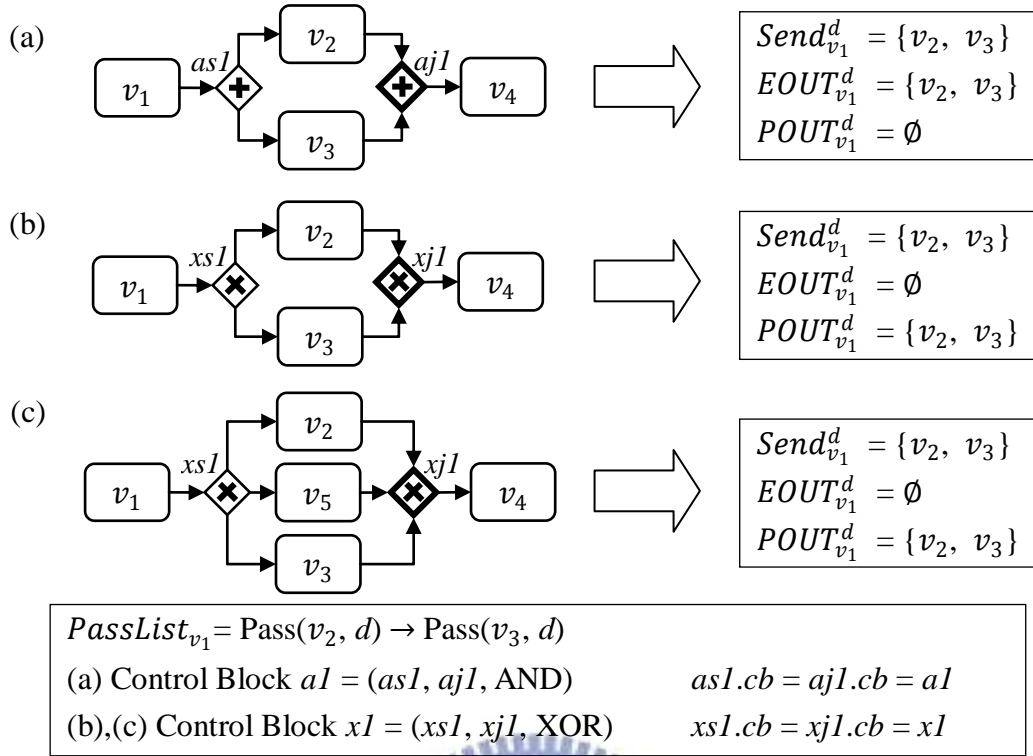


Figure 3.5: An Example of Receiving and Sending Sets.

Further, the *XOR Control Block Nodes*, $XBNodes$, are used to replace the activities located in the same XOR control block in $PIN_v^d/POUT_v^d$. For example in Figure 3.5 (b), v_2 and v_3 in $POUT_{v_1}^d$ are contained by XOR control block $x1$. Whether v_2 or v_3 is selected to execute in $x1$, v_1 always passes d to $x1$. Therefore, an *unconditional* $XBNode$ $n1$ is used to replace v_2 and v_3 in $POUT_{v_1}^d$. Because v_1 passes d to $x1$ explicitly, $n1$ is moved to $EOUT_{v_1}^d$ from $POUT_{v_1}^d$. In Figure 3.5 (c), v_2 and v_3 in $POUT_{v_1}^d$ are contained by XOR control block $x1$. However, v_1 does not pass d to $x1$ when v_5 is selected to execute in $x1$. Hence, a *conditional* $XBNode$ $n2$ is used to replace v_2 and v_3 in $POUT_{v_1}^d$. The detailed algorithms of construction of $XBNodes$ are illustrated in Section 5.3.

PIN'_v^d and EIN'_v^d are used to represent PIN_v^d and EIN_v^d whose activities are all replaced with $XBNodes$ and all unconditional $XBNodes$ are moved from PIN_v^d to EIN_v^d .

Therefore, the number of d passed/received from/in v can be identified. Let NIN_v^d be the number of d received in v , then $EIN_v^d \leq NIN_v^d \leq EIN_v^d + PIN_v^d$. Similarly, let $NOUT_v^d$ be the number of d passed from v , then $|EOUT_v^d| \leq NOUT_v^d \leq |EOUT_v^d| + |POUT_v^d|$. Figure 3.6 shows the artifact flow diagrams with XBNodes.

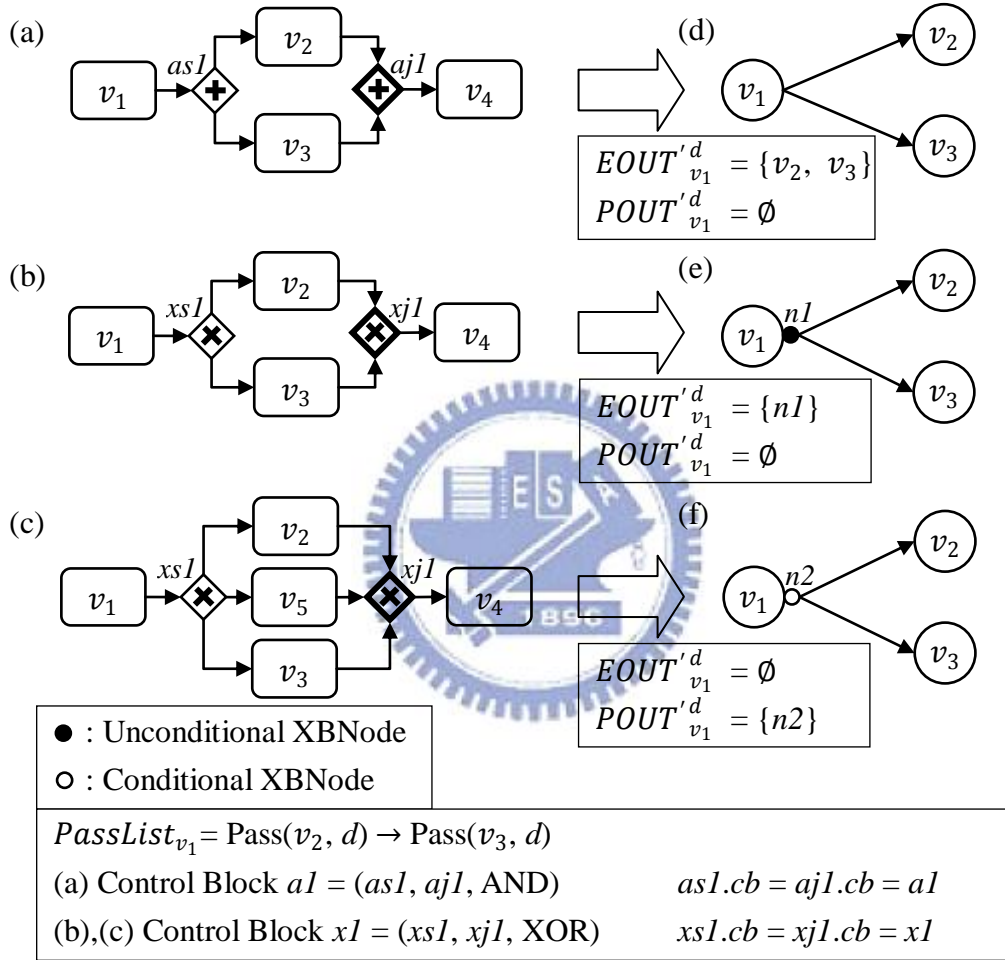


Figure 3.6: An Artifact Flow Diagram with XBNode.

In Figure 3.6 (d), because $|EOUT_{v_1}^d| = 2$ and $|POUT_{v_1}^d| = 0$, the number of d passed from v_1 is 2. In Figure 3.6 (e), because $|EOUT_{v_1}^d| = 1$ and $|POUT_{v_1}^d| = 0$, the number of d passed from v_1 is 1. In Figure 3.6 (f), because $|EOUT_{v_1}^d| = 0$ and $|POUT_{v_1}^d| = 1$, the number of d passed from v_1 is 0 or 1. Definition 3.19 defines an XBNode and its properties.

Definition 3.19 (An XOR Block Node)

Let an XBNode n be used to represent an XOR control block x . XBNode n contains four tuples $(blockID, cv_set, ABStack_n, isUncond)$,

- $blockID$ represents the id of an XOR control block which n expressed; thus, $blockID = x$.
- cv_set is a set of activities $\in PIN/POUT$ and the activities in cv_set are all located in different branches of x .
- $ABStack_n$ represents the location of x ; hence, $ABStack_n = ABStack_{x.start}$.
- $isUncond$ is a Boolean value. The value is *true* when n is unconditional. Otherwise, the value is *false*.
- $n.Parent = Null$ or the id of an XBNode containing n .
- $n.Attached = Null$ or the id of an activity where n is attached.

Based on above definitions, a flow in an artifact flow diagram contains the following properties: $outBlock$ and $inBlock$. The property $outBlock/inBlock$ is the id of an XBNode which contains sink/source activity. For example, let f be the flow (v_1, v_2) in Figure 3.6 (e). The $f.outBlock = n1$ because $v_3 \in n1.cv_set$. Definition 3.20 defines a flow in an artifact flow diagram.

Definition 3.20 (A Flow in an Artifact Flow Diagram)

For a flow $f = (u, v) \in AFF_p^d$ and activities $u, v \in AFV_p^d$, flow f has the following properties:

- $f.outBlock = Null$ or n if $v \in n.cv_set$.
- $f.inBlock = Null$ or n if $u \in n.cv_set$.

n is an XBNode.

3.3.3 An Example of an Artifact Flow Diagram

Figure 3.7 shows an example of a control flow graph including $PassList$, I , O , U^+ , and U^- for each activity.

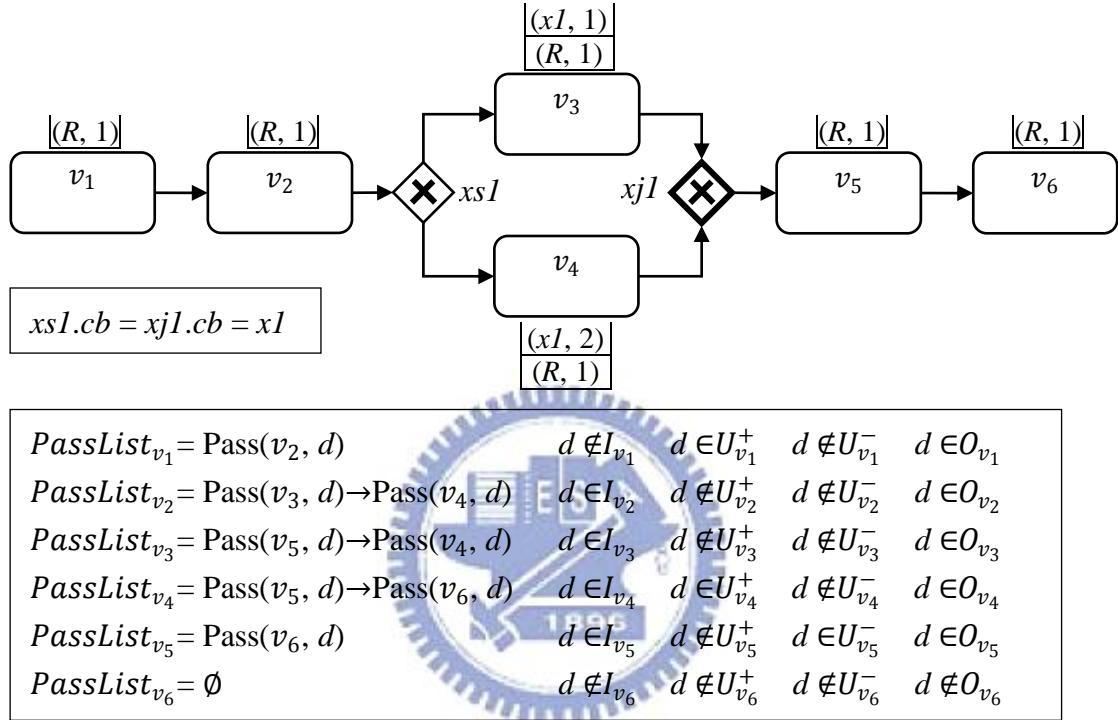


Figure 3.7: An Example of a Control Flow Graph.

Figure 3.8 shows the artifact flow diagram for d extracted from Figure 3.7. Because $Role_{xs1}^d = Irrelevantor$ and $Role_{xj1}^d = Irrelevantor$, control activities $xs1$ and $xj1$ are not added into the artifact flow diagram.

There are three types of flows in an artifact flow diagram:

- (1) Normal Flow: $\forall flow(u, v) \in AFF_p^d$, $IsParallel(u, v) = false$ and $IsExclusive(u, v) = false$.
- (2) AND Flow: $\forall flow(u, v) \in AFF_p^d$, $IsParallel(u, v) = true$.
- (3) XOR Flow: $\forall flow(u, v) \in AFF_p^d$, $IsExclusive(u, v) = true$.

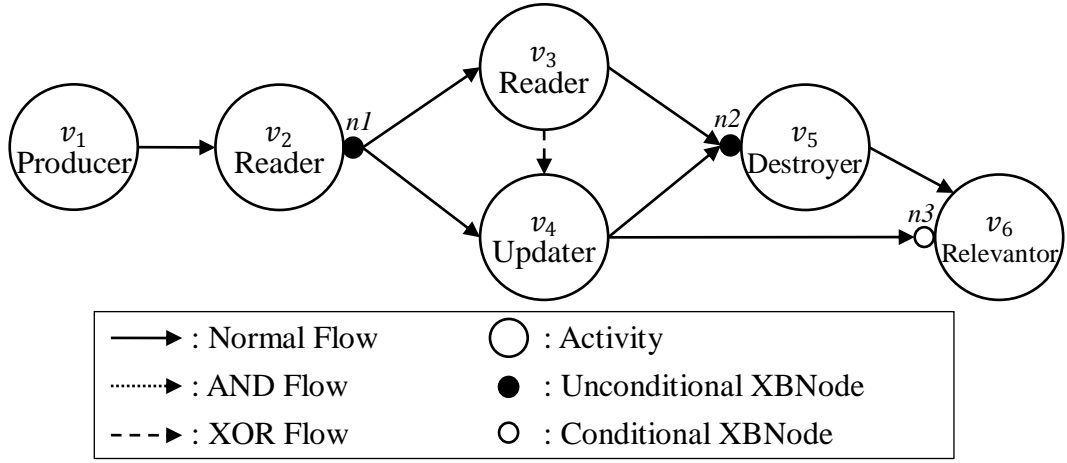


Figure 3.8: The Corresponding Artifact Flow Diagram for d .

Table 3.1 shows all states of each activity in Figure 3.7.

Activities	$Receive_v^d$	EIN_v^d	PIN_v^d	$Send_v^d$	$EOUT_v^d$	$POUT_v^d$
v_1	\emptyset	\emptyset	\emptyset	$\{v_2\}$	$\{v_2\}$	\emptyset
v_2	$\{v_1\}$	$\{v_1\}$	\emptyset	$\{v_3, v_4\}$	$\{n1\}$	\emptyset
v_3	$\{v_2\}$	$\{v_2\}$	\emptyset	$\{v_4, v_5\}$	$\{v_5\}$	\emptyset
v_4	$\{v_2, v_3\}$	$\{v_2\}$	\emptyset	$\{v_5, v_6\}$	$\{v_5, v_6\}$	\emptyset
v_5	$\{v_3, v_4\}$	$\{n2\}$	\emptyset	$\{v_6\}$	$\{v_6\}$	\emptyset
v_6	$\{v_4, v_5\}$	$\{v_5\}$	$\{n3\}$	\emptyset	\emptyset	\emptyset

Table 3.1: The States of Each Activity in Figure 3.7.

In Table 3.1, v_3 passes d to v_4 and v_5 ; thus, $Send_{v_3}^d = \{v_4, v_5\}$. However, v_3 and v_4 are exclusive activities. Activity v_4 is not added into $EOUT_{v_3}^d$ or $POUT_{v_3}^d$. Therefore, $EOUT_{v_3}^d = \{v_5\}$ and $POUT_{v_3}^d = \emptyset$. On the other hand, v_6 receives d from v_4 and v_5 ; thus, $Receive_{v_6}^d = \{v_4, v_5\}$. Activity v_6 can not receive d from v_4 when v_4 is not

selected to execute. Activity v_6 receives d from v_4 potentially. Hence, v_4 is added into $PIN_{v_6}^d$. Since v_6 can not receive d from $x1$ when v_3 is selected to execute, a conditional $n3$ is used to replace v_4 in $PIN_{v_6}^d$. Finally, $EIN'_{v_6} = \{v_5\}$ and $PIN'_{v_6} = \{n3\}$. Table 3.2 shows the states of each XBNode in Figure 3.7.

<i>XBNodes</i>	<i>blockID</i>	<i>cv_set</i>	<i>ABStack</i>	<i>isUncond</i>	<i>Parent</i>	<i>Attached</i>
<i>n1</i>	<i>x1</i>	$\{v_3, v_4\}$	$\langle R, 1 \rangle$	<i>true</i>	<i>Null</i>	v_2
<i>n2</i>	<i>x1</i>	$\{v_3, v_4\}$	$\langle R, 1 \rangle$	<i>true</i>	<i>Null</i>	v_5
<i>n3</i>	<i>x1</i>	$\{v_4\}$	$\langle R, 1 \rangle$	<i>false</i>	<i>Null</i>	v_6

Table 3.2: The States of Each XBNode in Figure 3.8.

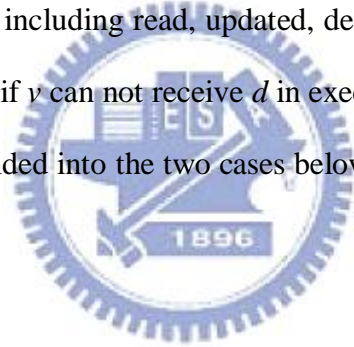


Chapter 4. Artifact Usage Anomalies

In a process specification, there might be four classes of anomalies: (1) Missing Artifact Anomalies, (2) Artifact Conflict Anomalies, (3) Cross Passing Artifact Anomalies, and (4) Redundant Anomalies. These anomalies are defined in the following subsections. Besides, every class indicates several types of anomalies of which each is illustrated with an example to show the scenario.

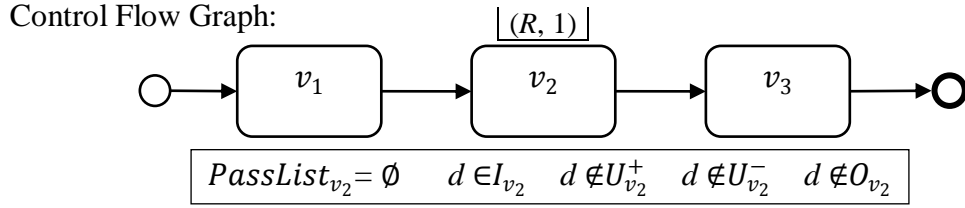
4.1 Missing Artifact Anomalies

While an artifact d is used, including read, updated, deleted, or passed, in an activity v , a missing artifact anomaly occurs if v can not receive d in execution time. For a receiver activity, missing of an artifact can be divided into the two cases below:



(1) Explicit Missing Artifacts:

- **Description:** Activity v requires artifact d but can not receive d .
- **Conditions:** $v \in AFV_p^d \wedge |EIN'_v{}^d| = 0 \wedge |PIN'_v{}^d| = 0$
 $\wedge Role_v^d \in \{\text{Reader, Updater, Destroyer, Relevantor}\}$
- **Example:** In the artifact flow of d in Figure 4.1, activity v_2 is a reader of d . Because no activity passes d to v_2 , v_2 can not receive d . Therefore, $|EIN'_{v_2}{}^d|$ and $|PIN'_{v_2}{}^d|$ are both 0. An explicit missing artifact anomaly occurs at v_2 .



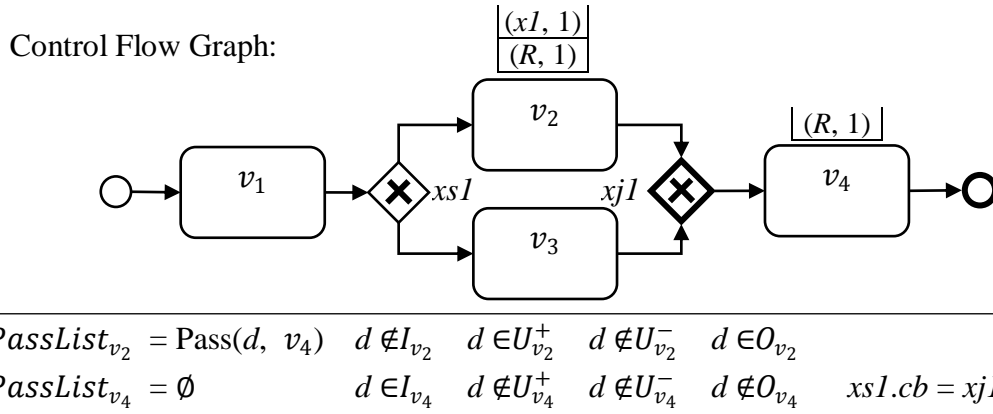
Artifact Flow Diagram for d :



Figure 4.1: An Example of an Explicit Missing Artifact Anomaly.

(2) Implicit Missing Artifact:

- **Description:** Activity v requires artifact d but receives d implicitly; i.e. v might not receive d for beginning execution.
- **Conditions:** $v \in AFV_p^d \wedge |EIN_v^d| = 0 \wedge |PIN_v^d| > 0$
 $\wedge Role_v^d \in \{\text{Reader, Updater, Destroyer, Relevantor}\}$
- **Example:** In Figure 4.2, v_2 is located in an XOR control block xI and v_2 passes d to v_4 . If v_2 is selected to execute in xI , v_4 receives d . Otherwise, v_4 can not receive d . Therefore, an implicit missing artifact anomaly occurs at v_4 .



Artifact Flow Diagram for d :

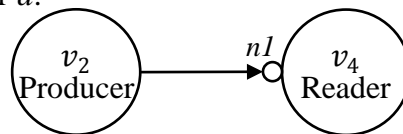


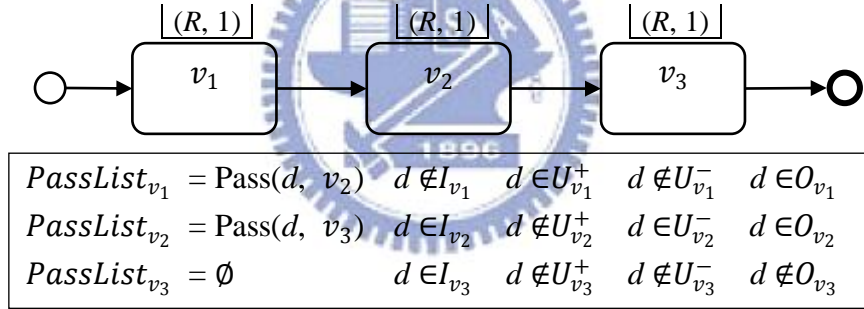
Figure 4.2: An Example of an Implicit Missing Artifact Anomaly.

The third case is observed from the sender and is named as *destroyed artifact anomaly*. For an artifact d , once a destroyer of d passes it to some other activities, d never reaches to its receivers since it has been destroyed. Thus, the receiver misses d . Because this missing is caused by the destroyer, a destroyed artifact anomaly occurs at the destroyer.

(3) Destroyed Artifact:

- **Description:** Activity v is a destroyer of artifact d and passes d to other activities.
- **Conditions:** $v \in AFV_p^d \wedge Role_v^d = \text{Destroyer} \wedge (|EOUT_v^d| > 0 \vee |POUT_v^d| > 0)$
- **Example:** In the artifact flow of d in Figure 4.3, activity v_2 is a destroyer of d and v_2 passes d to v_3 . Thus, $|EOUT_{v_2}^d|$ is 1 and a destroyed anomaly occurs at v_2 .

Control Flow Graph:



Artifact Flow Diagram for d :

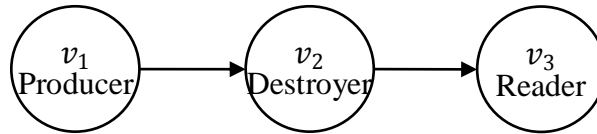


Figure 4.3: An Example of a Destroyed Artifact Anomaly.

4.2 Artifact Conflict Anomalies

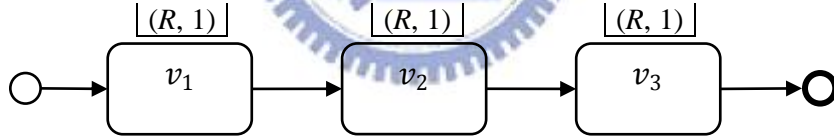
As describing in Section 3.3.1, because an activity can decide where artifacts are passed and the artifact transfers are passed by value, an artifact can have multiple copies in a

workflow. Hence, an activity may receive multiple copies of an artifact. For an artifact d , when an activity receives multiple copies of d concurrently, an artifact conflict occurs. In the case, the activity is not able to select the right copy [11]. Thus, an artifact conflict is concerned as an anomaly. Artifact conflict anomalies can be divided into the cases below:

(1) Explicit Artifact Conflict:

- **Description:** Activity v receives multiple copies of artifact d explicitly.
- **Conditions:** $v \in AFV_p^d \wedge |EIN_v^d| > 1$
- **Example:** In artifact flow of d in Figure 4.4, v_1 and v_2 both pass d to v_3 ; thus, v_3 receives two copies of d . Because v_1 and v_2 are predecessors of v_3 and are not located in any XOR control block, v_3 receives two copies of d explicitly. Therefore, $|EIN_{v_3}^d|$ is 2 and an explicit artifact conflict anomaly occurs at v_3 .

Control Flow Graph:



$PassList_{v_1} = Pass(d, v_2) \rightarrow Pass(d, v_3)$	$d \notin I_{v_1}$	$d \in U_{v_1}^+$	$d \notin U_{v_1}^-$	$d \in O_{v_1}$
$PassList_{v_2} = Pass(d, v_3)$	$d \in I_{v_2}$	$d \in U_{v_2}^+$	$d \notin U_{v_2}^-$	$d \in O_{v_2}$
$PassList_{v_3} = \emptyset$	$d \in I_{v_3}$	$d \notin U_{v_3}^+$	$d \notin U_{v_3}^-$	$d \notin O_{v_3}$

Artifact Flow Diagram for d :

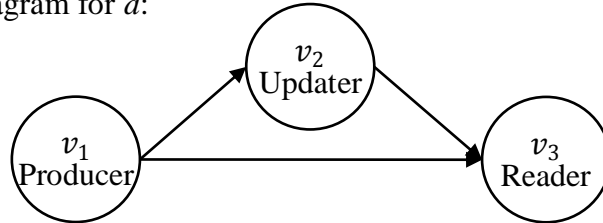
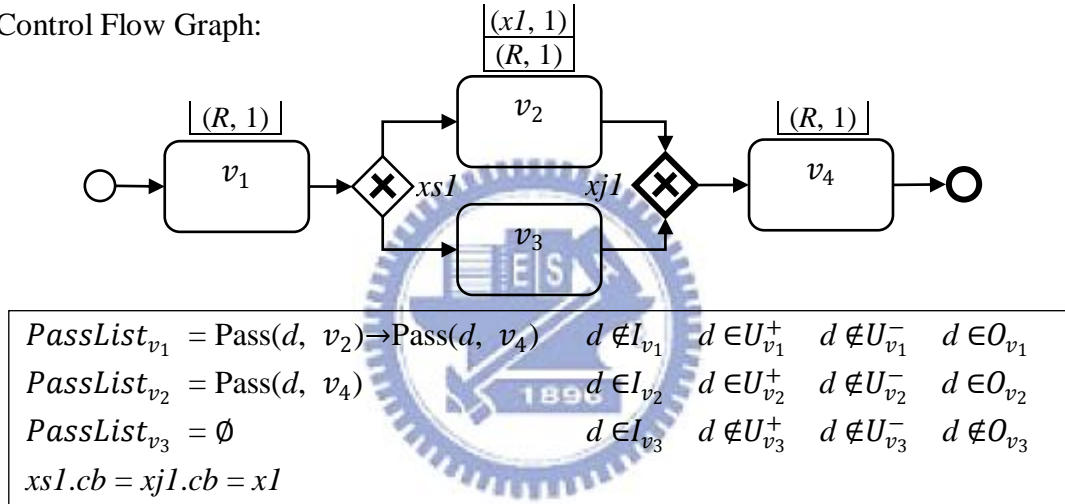


Figure 4.4: An Example of an Explicit Artifact Conflict Anomaly.

(2) Implicit Artifact Conflict:

- **Description:** Activity v receives multiple copies of artifact d implicitly.
- **Conditions:** $v \in AFV_p^d \wedge ((|EIN'_v{}^d| = 1 \wedge |PIN'_v{}^d| > 0) \vee (|EIN'_v{}^d| = 0 \wedge |PIN'_v{}^d| > 1))$
- **Example:** In Figure 4.5, activity v_4 receives d from v_1 explicitly; thus, $|EIN'_{v_4}{}^d|$ is 1. In addition, v_4 receives another d if v_2 is executed in XOR control block xI . Thus, v_4 receives d from v_2 implicitly and $|PIN'_{v_4}{}^d|$ is 1. Because $|EIN'_{v_4}{}^d|$ and $|PIN'_{v_4}{}^d|$ are both 1, an implicit artifact conflict anomaly occurs at v_4 .

Control Flow Graph:



Artifact Flow Diagram for d :

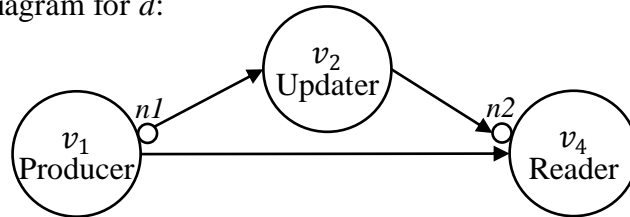


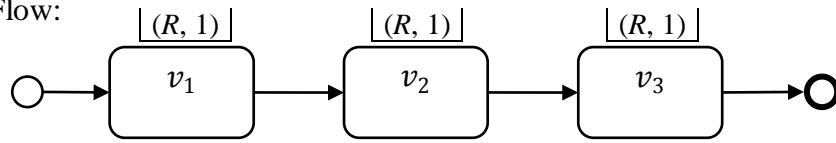
Figure 4.5: An Example of an Implicit Artifact Conflict Anomaly.

The third case is different from above cases because an artifact conflict may occur at an activity which does not receive multiple copies of an artifact. For an artifact d , if an activity receives one artifact d and produces another d , there are two copies of d in the same activity. Therefore, an anomaly called *production conflict anomaly* occurs.

(3) Production Conflict:

- **Description:** Activity v is a producer of artifact d and receives another d from other activity.
- **Conditions:** $v \in AFV_p^d \wedge Role_v^d = \text{Producer} \wedge (|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0)$
- **Example:** In Figure 4.6, activity v_3 receives artifact d from v_2 explicitly; thus, $|EIN'_{v_3}{}^d|$ is 1. Because v_3 is a producer of d , a production conflict anomaly occurs at v_3 .

Control Flow:



$PassList_{v_1} = \text{Pass}(d, v_2)$	$d \notin I_{v_1}$	$d \in U_{v_1}^+$	$d \notin U_{v_1}^-$	$d \in O_{v_1}$
$PassList_{v_2} = \text{Pass}(d, v_3)$	$d \in I_{v_2}$	$d \in U_{v_2}^+$	$d \notin U_{v_2}^-$	$d \in O_{v_2}$
$PassList_{v_3} = \emptyset$	$d \notin I_{v_3}$	$d \in U_{v_3}^+$	$d \notin U_{v_3}^-$	$d \notin O_{v_3}$

Artifact Flow Diagram for d :



Figure 4.6: An Example of a Production Conflict Anomaly.

4.3 Cross Passing Artifact Anomalies

A *cross passing artifact anomaly* occurs when an artifact is passed between branches of an XOR/AND control block. Given two parallel activities u and v , let u pass an artifact d to v . Due to the race hazard of parallel activities, v might be asked to execute before u in running time. Obviously, v can not start until d is received. This issue can be simply solved by ensuring that v waits for d until u completes its execution [11]. However, the designer may not allow v to wait. Hence, an artifact passed between parallel activities is concerned as an

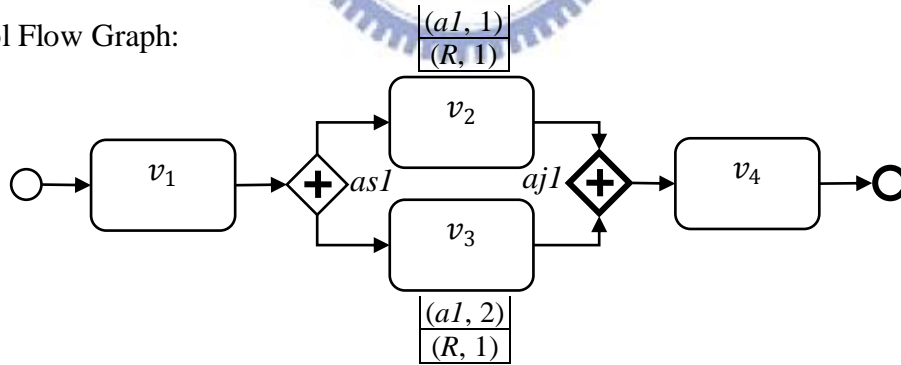
anomaly.

On the other hand, given two exclusive activities u and v , let u be designed to pass an artifact d to v . Since only one of u and v is executed, v may never receive d or u may pass d for nothing. Thus, an anomaly occurs in this case.

(1) Passing between Parallel Activities:

- **Description:** There are two parallel activities u and v , and u passes an artifact d to v .
- **Conditions:** $\text{flow}(u, v) \in \text{AFF}_p^d \wedge \text{IsParallel}(u, v) = \text{true}$
- **Example:** In the artifact flow of d in Figure 4.7, activity v_2 passes artifact d to v_3 ; thus, there is a flow (v_2, v_3) in the diagram. Because v_2 and v_3 are parallel activities, $\text{IsParallel}(v_2, v_3) = \text{true}$. Hence, a passing between parallel activities anomaly occurs at v_2 .

Control Flow Graph:



$PassList_{v_2} = \text{Pass}(d, v_3)$	$d \notin I_{v_2}$	$d \in U_{v_2}^+$	$d \notin U_{v_2}^-$	$d \in O_{v_2}$
$PassList_{v_3} = \emptyset$	$d \in I_{v_3}$	$d \notin U_{v_3}^+$	$d \notin U_{v_3}^-$	$d \notin O_{v_3}$
$asl.cb = ajl.cb = a1$				

Artifact Flow Diagram for d :

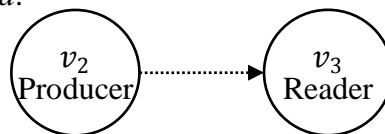


Figure 4.7: An Example of a Passing between Parallel Activities Anomaly.

(2) Passing between Exclusive Activities:

- **Description:** There are two exclusive activities u and v , and u passes an artifact d to v .
- **Conditions:** $\text{flow}(u, v) \in \text{AFF}_p^d \wedge \text{IsExclusive}(u, v) = \text{true}$
- **Example:** In the artifact flow of d in Figure 4.8, activity v_2 passes artifact d to v_3 ; thus, there is a flow (v_2, v_3) in the diagram. $\text{IsExclusive}(v_2, v_3) = \text{true}$ because v_2 and v_3 are exclusive activities. Therefore, a passing between exclusive activities anomaly occurs at v_2 .

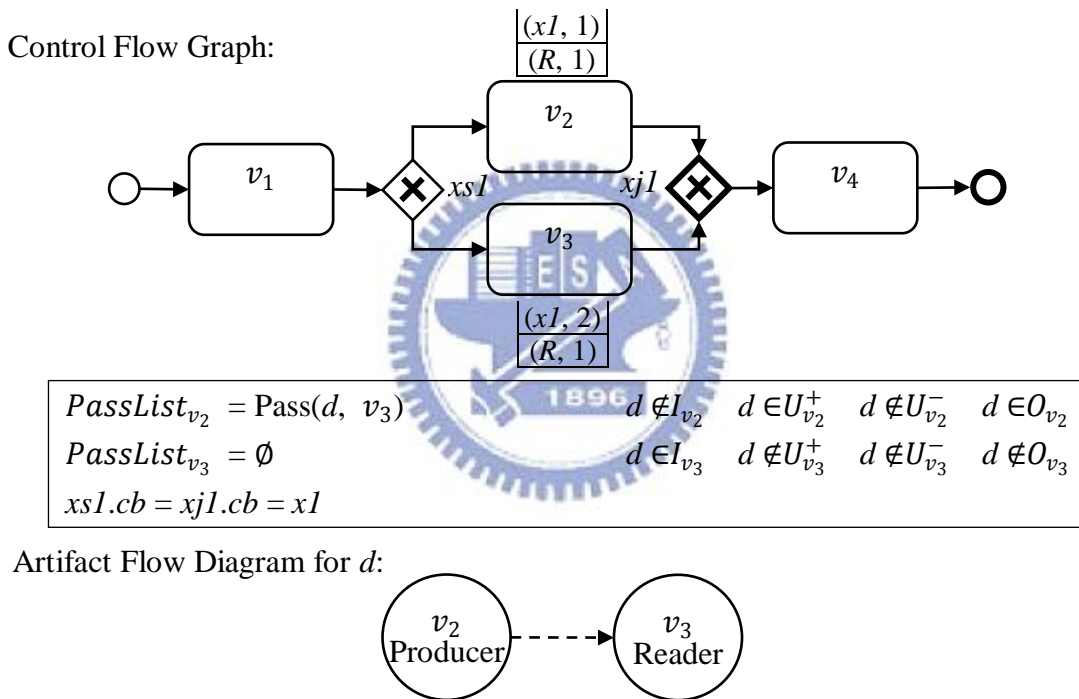


Figure 4.8: An Example of a Passing between Exclusive Activities Anomaly.

4.4 Redundant Anomalies

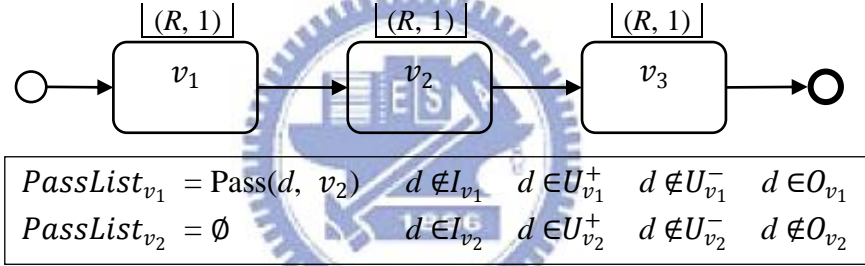
In a program, redundancy is not an error but makes inefficiency. For an activity v and an artifact d , a *redundant update/initialization anomaly* occurs when v updates or initializes d ,

but does not pass d to other activities. A *redundant pass anomaly* occurs when v receives d but does not use d .

(1) Redundant Update/Initialization:

- **Description:** Activity v initializes or updates d but does not pass d to other activities.
- **Conditions:** $v \in AFV_p^d \wedge |EOUT'_v{}^d| = 0 \wedge |POUT'_v{}^d| = 0$
 $\wedge Role_v^d \in \{\text{Producer, Updater}\}$
- **Example:** In Figure 4.9, activity v_2 is an updater of d in the artifact flow diagram and v_2 does not pass d to other activities. Hence, a redundant update anomaly occurs at v_2 .

Control Flow Graph:



Artifact Flow Diagram for d :

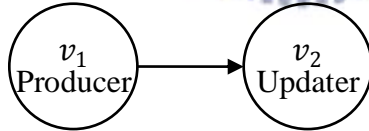


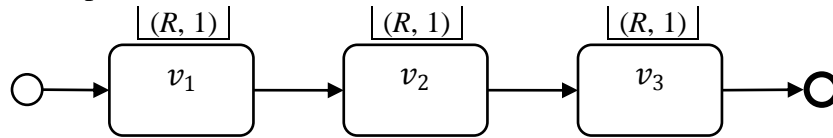
Figure 4.9: An Example of a Redundant Update/Initialization Anomaly.

(2) Redundant Pass:

- **Description:** Activity v receives an artifact d but does not use it.
- **Conditions:** $v \in AFV_p^d \wedge (|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0)$
 $\wedge Role_v^d = \text{Relevantor}$
- **Example:** In Figure 4.10, activity v_2 receives artifact d from v_1 . Because v_2 does

not use d , a redundant pass anomaly occurs at v_2 .

Control Flow Graph:



$PassList_{v_1} = Pass(d, v_2)$	$d \notin I_{v_1}$	$d \in U_{v_1}^+$	$d \notin U_{v_1}^-$	$d \in O_{v_1}$
$PassList_{v_2} = \emptyset$	$d \notin I_{v_2}$	$d \notin U_{v_2}^+$	$d \notin U_{v_2}^-$	$d \notin O_{v_2}$

Artifact Flow Diagram for d :

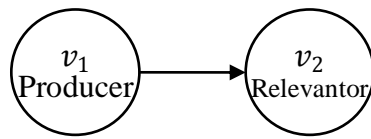


Figure 4.10: An Example of a Redundant Pass Anomaly.



Chapter 5. Incremental Algorithms for Anomalies Detection

5.1 Edit Operations for a Process

For introducing the incremental algorithms to detect artifact usage anomalies, we define the following edit operations for designer to edit a process: (1) Activity Insertion, (2) Activity Deletion, (3) Control Block Insertion, (4) Control Block Deletion, (5) Branch Insertion, (6) Branch Deletion, (7) Pass Insertion, (8) Pass Deletion, and (9) Activity Modification. Table 5.1 shows the edit operations discussed in this thesis.

	Activity	Control Block	Branch	Pass
Insertion	1	3	5	7
Deletion	2	4	6	8
Modification	9	-	-	-

Table 5.1: Edit Operations for a Process.

A process defined in Definition 3.1 can be described in more details below when being initialized.

An initial process $p = (V_p, F_p, R_p, C_p, ps, pe)$, where

- $V_p = \{ps, pe\}$, $ps.type = \text{ProcessStart}$, and $pe.type = \text{ProcessEnd}$.
- $F_p = \{(ps, pe)\}$.
- $R_p = \emptyset$.
- $C_p = \{(ps, pe, \text{ROOT})\}$.

The contents below indicate the definition of above edit operations. To simplify the incremental analysis, most operations are defined one or more constraints respectively. These constraints are simple and their effect for edit behavior is little.

1. Inserting a task/compound activity v :

- Constraints: $v \notin V_p$.
- Actions: Let v be inserted into flow (u, w) . Activity v is added into V_p and the flow (u, w) in F_p is replaced with flows, (u, v) and (v, w) .

2. Removing a task/compound activity v :

- Constraints: $v \in V_p$.
- Actions: Let I_v , U_v^+ , and U_v^- be empty by modifying v with Operation 9 automatically. $\forall u \in V_p$, all passes in $PassList_u$ and $PassList_v$ are also removed with Operation 8. Let flow (u, v) be v 's inflow, flow (v, w) be v 's outflow, and flow $(u, w) \notin F_p$. Activity v is removed from V_p . The inflow (u, v) and outflow (v, w) in F_p are replaced with flow (u, w) .

3. Inserting a control block c :

- Constraints: $c \notin C_p$.
- Actions: Let c be inserted into flow (u, w) . Property $c.start$ is initialized as a new control activity cs and property $c.end$ is initialized as a new control activity ce . The flow (u, w) in F_p is replaced with flows, (u, cs) , (cs, ce) , and (ce, w) . Property $c.bcounter$ is initialized as 1. Activities cs and ce are added into V_p . Control block c is added into C_p .

4. Removing a control block c :

- Constraints: $c \in C_p$.
- Actions: $\forall v \in V_p$, and $\text{IsReachable}(c.start, v) = \text{IsReachable}(v, c.end) = true$, v is removed with Operation 2. By modifying $c.start$ and $c.end$ with Operation 9, let $I_{c.start}$, $U_{c.start}^+$, $U_{c.start}^-$, $I_{c.end}$, $U_{c.end}^+$, and $U_{c.end}^-$ be empty. $\forall u \in V_p$, all passes in $PassList_u$, $PassList_{c.start}$, and $PassList_{c.end}$ are removed with Operation 8. Let flow $(u, c.start)$ be c 's inflow, flow $(c.end, w)$ be c 's outflow, and flow $(u, w) \notin F_p$. The flow $(c.start, c.end)$, inflow $(u, c.start)$, and outflow $(c.end, w)$ in F_p are replaced with flow (u, w) . Control activities $c.start$ and $c.end$ are removed from V_p . Control block c is removed from C_p .

5. Adding a branch into a control block c and:

- Constraints: flow $(c.start, c.end) \notin F_p$ and $c.type \neq \text{ROOT}$.
- Actions: The flow $(c.start, c.end)$ is added into F_p and $c.bcounter$ increases by 1.

6. Removing a branch from a control block c :

- Constraints: flow $(c.start, c.end) \in F_p$ and $c.totalbranches > 1$.
- Actions: The flow $(c.start, c.end)$ is removed from F_p .

7. Adding $\text{Pass}(d, u)$ into an activity v :

- Constraints:
 $\text{Pass}(d, u) \notin PassList_v$ and $\text{IsReachable}(u, v) = false$.
 $\nexists \text{Path}(u, v) \in AF_p^d$.
- Actions: $\text{Pass}(d, u)$ is added into $PassList_v$.

8. Removing $\text{Pass}(d, u)$ from an activity v :

- Constraints: $\text{Pass}(d, u) \in \text{PassList}_v$.
- Actions: $\text{Pass}(d, u)$ is removed from PassList_v .

9. Modifying an activity v :

- For an atomic activity v :
 - Actions: Modifying v 's specification. For example, let v initialize, read, update, or delete an artifact.
- For a compound activity v : Assigning a sub-process sp to $v.\text{subp}$.
 - Constraints: $\text{IsInstantiateRecursively}(p, sp) = \text{false}$.
 - Actions: $\text{Child}_p = \text{Child}_p \cup \{sp\}$. $\text{Parents}_{sp} = \text{Parents}_{sp} \cup \{p\}$.
 $v.\text{subp} = sp$.

Operation 9 updates the associated compound activity v by assigning $v.\text{subp}$ to be another sub-process. For example, if $\exists v_1 \in V_{p_1}$ and $v_1.\text{subp} = p_2$, p_2 is instantiated when v_1 is activated. On the other hand, if $\exists v_2 \in V_{p_2}$ and $v_2.\text{subp} = p_1$ exist at the same time, p_1 and p_2 might be instantiated recursively. In order to avoid this situation, function $\text{IsInstantiateRecursively}()$ is executed right after modifying a compound activity. If $\text{IsRecursiveCall}() = \text{false}$, recursive instantiating will not occur in this modification. Algorithm 5.1 shows how to identify whether a recursive instantiating occurs.

Algorithm 5.1 (Identifying Whether Recursive Instantiating Occurs)

Algorithm $\text{IsInstantiateRecursively}(p, sp)\{$

//Input: Identifying whether p is instantiated by sp or the processes in Child_{sp} recursively.

//Output: *true* is returned if a recursive instantiating occurs. Otherwise, *false* is returned.

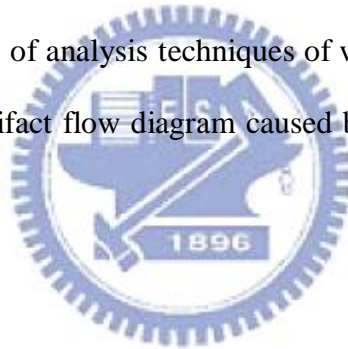
1. Boolean $flag = \text{false}$;
2. **if**($\text{Child}_{sp} \neq \emptyset$) {

```

3.   if( $p \in Child_{sp}$ ){
4.        $flag = true$ ;
5.   }else{
6.       for each process  $ssp \in Child_{sp}$ {
7.            $flag = flag \vee IsInstantiateRecursively(p, ssp)$ ;
8.       }
9.   }
10. }
11. return  $flag$ ;
}

```

Since Operation 1 inserts an activity without using an artifact, the artifact flow diagram is not necessary to update. It is similar for Operation 3. Operation 2 and 4 can be treated as a series of Operations 6, 8, and 9 correspondingly. Thus, the analysis based on Operation 2 or 4 can be done by applying a series of analysis techniques of which each is done after Operation 6, 8, or 9. The updates of an artifact flow diagram caused by Operations 5, 6, 7, 8, and 9 are described in Section 5.2.



5.2 Incremental Algorithms to Detect Artifact Usage Anomalies

In order to detect artifact usage anomalies incrementally, the artifact flow diagram has to be updated after each edit operation. The incremental algorithms for each edit operation are introduced in this section. For edit operations 7 to 9, Algorithms 5.2 to 5.4 show how to update the artifact flow diagram and detect anomalies. Algorithm 5.5 shows the updates and detections after operation 5 or 6.

Algorithm 5.2 (Updates and Detections After Inserting a Pass)**Algorithm** AfterAddingPass(u, d, v) {//Input: After Pass(d, v) is added into $PassList_u$.

//Output:

1. $AFV_p^d = AFV_p^d \cup \{u, v\}$;
 2. $AFF_p^d = AFF_p^d \cup \{\text{flow}(u, v)\}$;
 3. UpdateOutflows(u, d, v);
 4. UpdateInflows(v, d, u);
 5. DetectAnomalies(u, v, d);
- }

Algorithm 5.2 is executed after Pass(d, v) is added into $PassList_u$. At line 2, flow (u, v) is created and added into AFF_p^d . The properties of outflows of u and inflows of v are updated from lines 3 to 4. Finally, function DetectAnomalies(u, v, d) presented in Section 5.4 is executed to detect whether anomalies occur in u and v respectively at line 5.

Algorithm 5.3 (Updates and Detections After Removing a Pass)**Algorithm** AfterRemovingPass(u, d, v) {//Input: After Pass(d, v) is removed from $PassList_u$.

//Output:

1. $AFF_p^d = AFF_p^d \setminus \{\text{flow}(u, v)\}$;
2. $Role_u^d = \text{Roles}(u, d)$;
3. $Role_v^d = \text{Roles}(v, d)$;
4. **if**($Role_u^d == \text{Irrelevantor}$) {
5. $AFV_p^d = AFV_p^d \setminus \{u\}$;
6. }**else** {
7. UpdateOutflows(u, d, v);
8. DetectAnomalies(u, u, d);
9. }
10. **if**($Role_v^d == \text{Irrelevantor}$) {
11. $AFV_p^d = AFV_p^d \setminus \{v\}$;
12. }**else** {
13. UpdateInflows(v, d, u);
14. DetectAnomalies(v, v, d);

```

15. }
}

```

Algorithm 5.3 is executed after removing $\text{Pass}(d, v)$ from PassList_u . At lines 2 to 3, function $\text{Roles}(u, d)$ and $\text{Roles}(v, d)$ are used to update the roles of u and v for d . The condition at line 4 is used to check whether u becomes an irrelevantor of d or not. Otherwise, function $\text{UpdateOutflows}(u, d, v)$ presented in Section 5.3 is called to adjust the flows and update properties and function $\text{DetectAnomalies}(u, u, d)$ is executed to detect anomalies. It is similar for v at line 10.

Algorithm 5.4 (Updates and Detections After Modifying an Activity)

Algorithm $\text{RoleIsChanged}(v, d)\{$

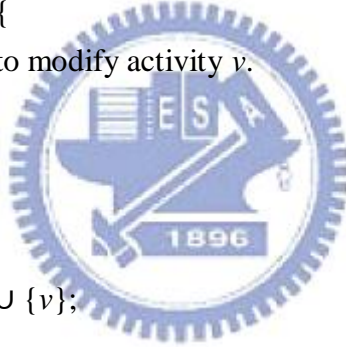
//Input: Role_v^d is changed due to modify activity v .

//Output:

```

1.  $\text{Role}_v^d = \text{Roles}(v, d);$ 
2. if( $\text{Role}_v^d \neq \text{Irrelevantor}$ ){
3.     if( $v \notin \text{AFV}_p^d$ ){
4.          $\text{AFV}_p^d = \text{AFV}_p^d \cup \{v\};$ 
5.     }
6.      $\text{DetectAnomalies}(v, v, d);$ 
7. }else{
8.      $\text{AFV}_p^d = \text{AFV}_p^d \setminus \{v\};$ 
9. }
}

```



Algorithm 5.4 is executed when activity v is changed its role for d . At line 1, function $\text{Roles}(v, d)$ is used to update v 's role of d . At line 2, if v is not an irrelevantor of d , v is added into AFV_p^d and $\text{DetectAnomalies}(v, v, d)$ is used to check whether anomalies occur at v . Otherwise, v is removed from AFV_p^d .

Algorithm 5.5 (Updates and Detections After Adding or Removing a Branch)

Algorithm AfterAddingorRemovingBranch(c){

//Input: After adding/removing a branch into/from c .

//Output:

```
1.  if( $c.type == XOR$ ){
2.      for each  $n \in AFN_p^d$  and  $n.blockID = c$ {
3.          UpdatePXBNNode( $n$ );
4.      }
5.  }
```

Algorithm 5.5 is performed when a branch is added/removed into/from a control block.

In our model, only the insertion or deletion of a branch in an XOR control block affects the artifact flow diagram. Let XBNNode n represent an XOR control block x in an artifact flow diagram. After adding/removing a branch of x , $n.isUncond$ may be changed. Therefore, in the for loop at lines 2 to 4, function UpdatePXBNNode(n) is executed to update each XBNNode which may be affected by this operation. Function UpdatePXBNNode() is described in Algorithm 5.6.

Algorithm 5.6 (Updating the Parent XBNNode)

Algorithm UpdatePXBNNode(n){

//Input: Update XBNNode n and its parent XBNNode.

//Output:

```
1.  if( $n.isUncond \neq IsUnconditional(n.cv\_set, n.blockID)$ ){
2.       $n.isUncond = IsUnconditional(n.cv\_set, n.blockID)$ ;
4.      if( $n.Parent \neq Null$ ){
5.          UpdatePXBNNode( $n.Parent$ );
6.      }else{
7.          DetectAnomalies( $n.Attached, n.Attached, d$ );
8.      }
9.  }
```

Since an XBNODE may be contained in another XBNODE, Algorithm 5.6 updates the XBNODE n and its parent XBNODE. The modification process starts up from n , level by level, until it has no parent XBNODE or $n.isUncond$ is not changed. The condition at line 1 identifies whether $n.isUncond$ is changed or not. If $n.isUncond$ is changed, $n.isUncond$ is updated to the result of function $IsUnconditional(n.cv_set, n.blockID)$ at line 2. At line 4 to 5, if n has parent XBNODE, its parent XBNODE is updated recursively. Otherwise, function $DetectAnomalies(n.Attached, n.Attached, d)$ is executed to detect whether anomalies occur at activity $n.Attached$.

5.3 Algorithms to Update an Artifact Flow Diagram

For an activity $v \in AFV_p^d$, the properties $inBlock$ of v 's inflows depend on the XBNODEs in EIN'_v/PIN'_v ; on the contrary, the properties $outBlock$ of v 's outflows depend on XBNODEs in $EOUT'_v/POUT'_v$. Our approach updates the XBNODEs affected before modifying the flows. The constructing methods are illustrated in Section 5.3.1 and the updating methods are introduced in Section 5.3.2. Section 5.3.3 presents an example for illustrating how to build XBNODEs and update flows.

5.3.1 Construction of XBNODEs

In this section, an algorithm for the construction of XBNODEs is proposed. The construction is done according to the activities in input set S . Since the algorithm for the construction is complex, three sub-functions are introduced before illustrating the algorithm.

Let an XBNODE n represent an XOR control block x . According to Definition 3.19, the

activities in $n.cv_set$ are not located in the same branch of x . Hence, the activities/XBNodes in S located in the same deepest XOR control blocks and branches are identified. In these identified activities/XBNodes, only one is selected to stay in S . The remainders are moved to another set for constructing another XBNode. Algorithm 5.7 shows how to identify whether the activities are located in the same deepest XOR control blocks and branches.

Algorithm 5.7 (Searching the Activities Located in the Same Deepest XOR Control Blocks and Branches)

Algorithm SearchIdenticalBranch(S) {

//**Input:** S is a set of activities to build XBNodes.

//**Output:** S_{same} : The set of activities/XBNodes located in the same deepest XOR control blocks and branches.

```

1.  $S_{same} = \emptyset$ ;
2. for each  $u \in S$  {
3.     for each  $w \in S$  and  $w \neq u$  {
4.         if( $ABStack_u.getTopXOR() == ABStack_w.getTopXOR()$ ) {
5.             if( $w$  is an XBNode and  $w.isUncond == false$ ) {
6.                  $S_{same} = S_{same} \cup \{w\}$ ;
7.                  $S = S \setminus \{w\}$ ;
8.             } else if( $u$  is an XBNode and  $u.isUncond == false$ ) {
9.                  $S_{same} = S_{same} \cup \{u\}$ ;
10.                 $S = S \setminus \{u\}$ ;
11.            } else {
12.                 $S_{same} = S_{same} \cup \{w\}$ ;
13.                 $S = S \setminus \{w\}$ ;
14.            }
15.        }
16.    }
17. }
18. return  $S_{same}$  ;
}

```

For an activity v , if a control block x is the deepest control block containing v , x is associated with the top element in $ABStack_v$. The function `getTopXOR()` gets the first element from the top of an $ABStack$ and the type of the block associated with the element is XOR. Hence, if the results of `getTopXOR()` of the activities are the same, they are located in the same deepest XOR control blocks and branches. In Algorithm 5.7, the condition at line 4 identifies whether the activities/XBNodes are located in the same deepest XOR control blocks and branches. The conditions at lines 5 to 14 decide who is added into S_{same} . If u is a conditional XBNode, u is moved to S_{same} from S . It is similar for w .

During construction of XBNodes, the activity contained in the deepest XOR control blocks is first selected to construct an XBNode. For an activity v , let $xlevel$ denote the number of XOR control blocks containing v ; hence, the activity with the maximal $xlevel$ is selected first. The computation of $xlevel$ of an activity/XBNode is illustrated in Algorithm 5.8.

Algorithm 5.8 (Computing $xlevel$ for an Activity/XBNode)

Algorithm XLevel(u, v) {

//Input: u is an activity/XBNode in S .

// v is the activity where XBNodes are attached.

//Output: $|xset|$ is the number of XOR control blocks which contain u but not v .

1. $ABStack_{u'} = ABStack_u.removeIdenticalElements(ABStack_v)$;
2. $xset = \{e \in ABStack_{u'} \mid e.blockID.type = XOR\}$;
3. **return** $|xset|$;

}

In Algorithm 5.8, u is an activities/XBNodes in S . At line 1, the same elements in $ABStack_u$ and $ABStack_v$ are removed because the XOR control blocks containing both u and v are not needed to build XBNodes. The rest elements are stored in $ABStack_{u'}$. The number of XOR control blocks containing u but not v is equal to the number of elements

whose $branchID.type$ is XOR in $ABStack_u'$. At lines 2 to 3, $|xset|$ is equal to the number of elements whose $branchID.type$ is XOR in $ABStack_u'$. Therefore, $|xset|$ represents the number of XOR control blocks containing u but not v .

The XBNodes are categorized into two types: *unconditional* and *conditional*. Let an XBNODE n is constructed in $PIN_v^d/POUT_v^d$ for expressing an XOR control block x . XBNODE n is unconditional if each branch of x contains an activity receiving/passing d from/to v . Otherwise, n is conditional. Algorithm 5.9 shows how to identify whether an XBNODE is conditional or not.

Algorithm 5.9 (Identifying an Unconditional/Conditional XBNODE)

Algorithm $IsUnconditional(cv_set, blockID)$ {

//**Input:** cv_set is a set of activities/XBNodes which are contained in an XBNODE.

// $blockID$ is the id of an XOR control block represented by an XBNODE.

//**Output:** $true$ is returned if the XBNODE is unconditional. Otherwise, $false$ is returned.

1. Boolean $flag = true$;
2. **for each** $u \in cv_set$ {
3. **if**(u is an XBNODE and $u.isUncond == false$){
4. $flag = false$;
5. }
6. }
7. **if**($blockID.totalbranches == |cv_set|$ and $flag == true$){
8. **return** $true$;
9. }**else**{
10. **return** $false$;
11. }
- }

In Algorithm 5.9, if an XBNODE contains another conditional XBNODE, it is also viewed as a conditional XBNODE. The for loop at lines 2 to 6 decides the Boolean value of $flag$ by checking each element in S . The $flag$ is $false$ if there is a conditional XBNODE in cv_set .

Otherwise, the *flag* is *true*. At lines 7 to 11, when *flag* is *true* and the number of elements in *cv_set* is equal to the number of branches in *blockID*, this function returns *true*; otherwise, *false* is returned.

Based on the sub-functions above, a complete algorithm of building XBNodes is proposed. Algorithm 5.10 shows how to build XBNodes.

Algorithm 5.10 (Building XBNodes)

Algorithm BuildXBNodes(S, v, XBN_v^d) {

//**Input:** S is a set of activities to build XBNodes which are attached to v .

// $XBN_v^d = XOUT_v^d$ or XIN_v^d . Each constructed XBNode is added into XBN_v^d .

//**Output:** A set of XBNodes which are built completely.

1. $S_{same} = \emptyset$;
2. $XLevelList = XLevelList \cup S$;
3. $m = XLevelList.getFirst()$;
4. **while**($XLevel(m, v) \neq 0$) {
5. $S_{same} = SearchIdenticalBranch(S)$;
6. $XLevelList = XLevelList \setminus S_{same}$;
7. XOR control block $x = ABStack_m.getTopXOR().blockID$;
8. Create an XBNode n ;
9. $n.blockID = x$;
10. $n.cv_set = \{u \mid u \in S \text{ and } ABStack_u.getTopXOR().blockID == x\}$;
11. for each $u \in n.cv_set$ {
12. if(u is an XBNode) {
13. $u.Parent = n$;
14. }
15. }
16. $n.isUncond = IsUnconditional(n.cv_set, x)$;
17. $ABStack_n = ABStack_{x.start}$;
18. $XLevelList = XLevelList \setminus n.cv_set$;
19. $XLevelList = XLevelList \cup \{n\}$;
20. $AFN_p^d = AFN_p^d \cup \{n\}$;
21. $XBN_v^d = XBN_v^d \cup \{n\}$;
22. $m = XLevelList.getFirst()$;


```

23. }
24. if( $S_{same} \neq \emptyset$ )
25.      $XLevelList = XLevelList \cup \text{BuildXBNodes}(S_{same}, v)$ ;
26.  $S = XLevelList$ ;
27. return  $S$ ;
}

```

In Algorithm 5.10, XBN_v^d is equal to XIN_v^d or $XOUT_v^d$. XIN_v^d is a set of XBNodes, of which each is contained in EIN_v^d/PIN_v^d or contained in the XBNode in EIN_v^d/PIN_v^d . On the other hand, $XOUT_v^d$ is a set of XBNodes, of which each is contained in $EOUT_v^d/POUT_v^d$ or contained in the XBNode in $EOUT_v^d/POUT_v^d$.

$XLevelList$ is a linked list and all elements in this list are sorted according to their $xlevel$. The element with large $xlevel$, it is closer to the front side in the list. Therefore, the element obtained by $XLevelList.getFirst()$ has the maximal $xlevel$ in the list. At line 2, all elements in S are added into $XLevelList$. At line 3, m is the element which has the maximal $xlevel$. The while loop at lines 4 to 23 repeats until the $xlevel$ of m becomes 0, which means all XBNodes in S are completely built. At lines 5 to 6, the activities/XBNodes identified by function $\text{SearchIdenticalBranch}()$ are moved to S_{same} from $XLevelList$. If the S_{same} is not empty, the activities/XBNodes in S_{same} are used to construct another XBNode recursively at lines 24 to 25.

At line 7, XOR control block x is the deepest XOR control block containing m . An XBNode n is created to represent x at lines 8 to 9. All activities/XBNodes located in x are added into $n.cv_set$ at line 10. The for loop at lines 11 to 15 updates the properties $Parent$ of XBNodes in $n.cv_set$. The function $\text{IsUnconditional}()$ is executed to identify whether n is conditional or not at line 16. $ABStack_n$ represents the location of x ; thus, $ABStack_n$ is equal to $ABStack_{x.start}$ at line 17. Finally, the activities/XBNodes in cv_set are removed

from S at line 18. At lines 19 to 21, n is added into $XLevelList$, AFN_p^d , and XBN_v^d respectively. While all XBNodes are constructed completely, S is returned.

5.3.2 Update of the Properties of Flows

After the XBNodes are built completely and put into EIN_v^d and PIN_v^d , v 's inflows can be updated according to the XBNodes in EIN_v^d and PIN_v^d . Algorithm 5.12 shows how to update *inBlock* of v 's inflows. On the other hand, Algorithm 5.13 shows how to update *outBlock* of v 's outflows. Before illustrating Algorithms 5.12 and 5.13, the sub-function, Algorithms 5.11, is introduced first.

Algorithm 5.11 (Getting All Activities in an XBNode)

Algorithm getAllActivities(n){

//**Input**: Getting all activities contained in n and n 's child XBNodes.

//**Output**: $AllA$ is a set of activities, of which each is contained in n or n 's child XBNodes.

```

1.   $AllA = \emptyset$ ;
2.  for each  $v \in n.cv\_set$ {
3.      if( $v$  is an activity){
4.           $AllA = AllA \cup \{v\}$ ;
5.      }else{
6.           $AllA = AllA \cup$  getAllActivities( $v$ );
7.      }
8.  }
9.  return  $AllA$ ;
}
```

Since an XBNode may contain other XBNodes in cv_set , Algorithm 5.11 is used to get all activities in cv_set and in child XBNodes recursively. At lines 2 to 7, if v is an activity, v is added into $AllA$. Otherwise, function getAllActivities(v) is used to get all activities in v . When

all activities are added into $AllA$, $AllA$ is returned at line 9.

Algorithm 5.12 (Updating Properties $inBlock$ of Inflows)

Algorithm UpdateInflows(v, d, u) {

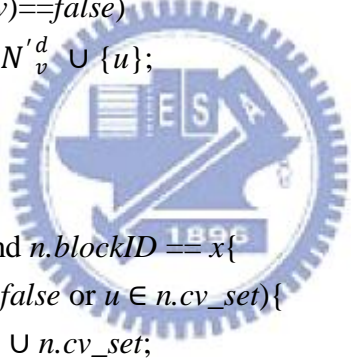
//Input: After an inflow (u, v) is added/removed into/from $AF F_p^d$.

//Output:

```

1.  $ABStack_u' = ABStack_u.removeIdenticalElements(ABStack_v)$ ;
2. XOR Control Block  $x = ABStack_u'.getTopXOR().blockID$ ;
3. if( $x == Null$ ) {
4.     if( $u \in Receive_v^d$ ) {
5.          $Receive_v^d = Receive_v^d \setminus \{u\}$ ;
6.          $EIN_v^d = EIN_v^d \setminus \{u\}$ ;
7.     } else {
8.          $Receive_v^d = Receive_v^d \cup \{u\}$ ;
9.         if(IsExclusive( $u, v$ ) == false)
10.             $EIN_v^d = EIN_v^d \cup \{u\}$ ;
11.     }
12. } else {
13.      $IN_v^d = \emptyset$ ;
14.     for each  $n \in XIN_v^d$  and  $n.blockID == x$  {
15.         if( $n.isUncond == false$  or  $u \in n.cv\_set$ ) {
16.              $IN_v^d = IN_v^d \cup n.cv\_set$ ;
17.              $AFN_p^d = AFN_p^d \setminus \{n\}$ ;
18.              $EIN_v^d = EIN_v^d \setminus \{n\}$ ;
19.              $PIN_v^d = PIN_v^d \setminus \{n\}$ ;
20.              $XIN_v^d = XIN_v^d \setminus \{n\}$ ;
21.             while( $n.Parent \neq Null$ ) {
22.                  $m = n.Parent$ ;
23.                  $IN_v^d = IN_v^d \cup (m.cv\_set \setminus \{n\})$ ;
24.                  $AFN_p^d = AFN_p^d \setminus \{m\}$ ;
25.                  $EIN_v^d = EIN_v^d \setminus \{m\}$ ;
26.                  $PIN_v^d = PIN_v^d \setminus \{m\}$ ;
27.                  $XIN_v^d = XIN_v^d \setminus \{m\}$ ;
28.                  $n = m$ ;
29.             }
30.         }
31.     }

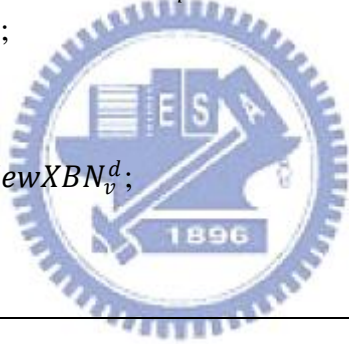
```



```

32.   if( $u \in Receive_v^d$ ){
33.        $Receive_v^d = Receive_v^d \setminus \{u\}$ ;
34.        $IN_v^d = IN_v^d \setminus \{u\}$ ;
35.   }else{
36.        $Receive_v^d = Receive_v^d \cup \{u\}$ ;
37.        $IN_v^d = IN_v^d \cup \{u\}$ ;
38.   }
39.    $NewXBN_v^d = BuildXBNodes(IN_v^d, v, XIN_v^d)$ ;
40.   for each  $n \in NewXBN_v^d$ {
41.        $n.Attached = v$ ;
42.       if( $n.isUncond == true$ ){
43.            $EIN'_v^d = EIN'_v^d \cup \{n\}$ ;
44.            $NewXBN_v^d = NewXBN_v^d \setminus \{n\}$ ;
45.       }
46.       for each  $u \in getAllActivities(n)$ {
47.           Let flow  $f$  be  $(u, v)$  in  $AFF_p^d$ ;
48.            $f.inBlock = n$ ;
49.       }
50.   }
51.    $PIN'_v^d = PIN'_v^d \cup NewXBN_v^d$ ;
52. }
}

```



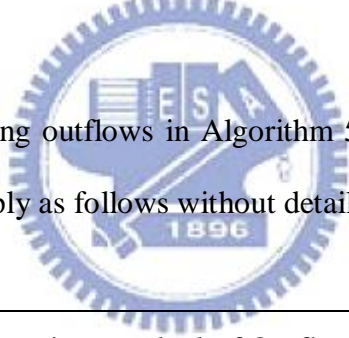
At lines 1 to 2, x is the deepest XOR control block containing u but not v . At line 3, if x is *Null*, the XBNodes in EIN'_v^d and PIN'_v^d are not affected after an inflow (u, v) is added/removed into/from AFF_p^d . At lines 4 to 11, u is added/removed into/from $Receive_v^d$ and EIN'_v^d according to whether u is in $Receive_v^d$ or not. If u is in $Receive_v^d$, u is removed from $Receive_v^d$ and EIN'_v^d . If u is not in $Receive_v^d$ and $IsExclusive(u, v) = false$, u is added into $Receive_v^d$ and EIN'_v^d .

If x is not *Null*, the XBNodes whose *blockID* is x in XIN_v^d may be affected. At line 13, IN_v^d is used to store all the affected activities/XBNodes. At lines 14 to 20, for each XBNode n whose *blockID* = x in XIN_v^d , if n is a conditional XBNode or u is contained in $n.cv_set$, n is

removed from AFN_p^d , EIN_v^d , PIN_v^d , and XIN_v^d . All elements in $n.cv_set$ are added into IN_v^d because these elements are used to rebuild XBNodes. At lines 21 to 29, if n has parent XBNNode, the elements in parent XBNNode are also added into IN_v^d and the parent XBNNode is removed from AFN_p^d , EIN_v^d , PIN_v^d , and XIN_v^d .

As in lines 4 to 11, u is added/removed into/from $Receive_v^d$ and IN_v^d according to whether u is in $Receive_v^d$ or not at lines 32 to 38. At line 39, $BuildXBNodes(IN_v^d, v, XIN_v^d)$ is used to rebuild XBNodes from the affected elements and the results are put into $NewXBN_v^d$. At lines 40 to 45, for each new XBNNode n , $n.Attached = v$. If n is unconditional, n is moved to EIN_v^d from $NewXBN_v^d$. The properties $inBlock$ of inflows of v are updated to n at lines 46 to 49. Finally, the conditional XBNodes in $NewXBN_v^d$ are added into PIN_v^d .

Since the method of updating outflows in Algorithm 5.13 is similar to Algorithm 5.12, Algorithm 5.13 is described simply as follows without detailed explanation.



Algorithm 5.13 (Updating Properties $outBlock$ of Outflows)

Algorithm UpdateOutflows(v, d, u) {

//Input: After an outflow (v, u) is added/removed into/from AFF_p^d .

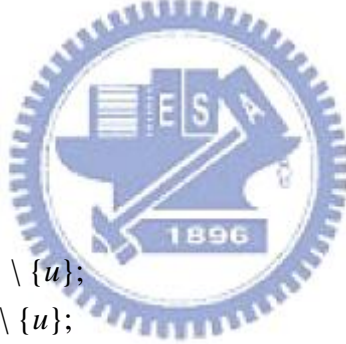
//Output:

1. $ABStack_{u'} = ABStack_u.removeIdenticalElements(ABStack_v)$;
2. XOR Control Block $x = ABStack_{u'}.getTopXOR().blockID$;
3. **if**($x == Null$) {
4. **if**($u \in Send_v^d$) {
5. $Send_v^d = Send_v^d \setminus \{u\}$;
6. $EOUT_v^d = EOUT_v^d \setminus \{u\}$;
7. **else** {
8. $Send_v^d = Send_v^d \cup \{u\}$;
9. **if**($IsExclusive(v, u) == false$)
10. $EOUT_v^d = EOUT_v^d \cup \{u\}$;
11. **}**

```

12. }else{
13.      $OUT_v^d = \emptyset;$ 
14.     for each  $n \in XOUT_v^d$  and  $n.blockID == x$ {
15.         if( $n.isUncond == false$  or  $u \in n.cv\_set$ ){
16.              $OUT_v^d = OUT_v^d \cup n.cv\_set;$ 
17.              $AFN_p^d = AFN_p^d \setminus \{n\};$ 
18.              $EOUT_v^d = EOUT_v^d \setminus \{n\};$ 
19.              $POUT_v^d = POUT_v^d \setminus \{n\};$ 
20.              $XOUT_v^d = XOUT_v^d \setminus \{n\};$ 
21.             while( $n.Parent \neq Null$ ){
22.                  $m = n.Parent;$ 
23.                  $OUT_v^d = OUT_v^d \cup (m.cv\_set \setminus \{n\});$ 
24.                  $AFN_p^d = AFN_p^d \setminus \{m\};$ 
25.                  $EOUT_v^d = EOUT_v^d \setminus \{m\};$ 
26.                  $POUT_v^d = POUT_v^d \setminus \{m\};$ 
27.                  $XOUT_v^d = XOUT_v^d \setminus \{m\};$ 
28.                  $n = m;$ 
29.             }
30.         }
31.     }
32.     if( $u \in Send_v^d$ ){
33.          $Send_v^d = Send_v^d \setminus \{u\};$ 
34.          $OUT_v^d = OUT_v^d \setminus \{u\};$ 
35.     }else{
36.          $Send_v^d = Send_v^d \cup \{u\};$ 
37.          $OUT_v^d = OUT_v^d \cup \{u\};$ 
38.     }
39.      $NewXBN_v^d = BuildXBNodes(OUT_v^d, v, XOUT_v^d);$ 
40.     for each  $n \in NewXBN_v^d$ {
41.          $n.Attached = v;$ 
42.         if( $n.isUncond == true$ ){
43.              $EOUT_v^d = EOUT_v^d \cup \{n\};$ 
44.              $NewXBN_v^d = NewXBN_v^d \setminus \{n\};$ 
45.         }
46.         for each  $u \in getAllActivities(n)$ {
47.             Let flow  $f$  be  $(v, u)$  in  $AFN_p^d$ ;
48.              $f.outBlock = n;$ 
49.         }

```



```

50.   }
51.    $POUT'_v^d = POUT'_v^d \cup NewXBN_v^d;$ 
52.   }
}

```

5.3.3 An Example for Illustration of Updating an Artifact Flow Diagram

Figure 5.1 shows a control flow and Figure 5.2 is the corresponding artifact flow diagrams for artifact d extracted from it.

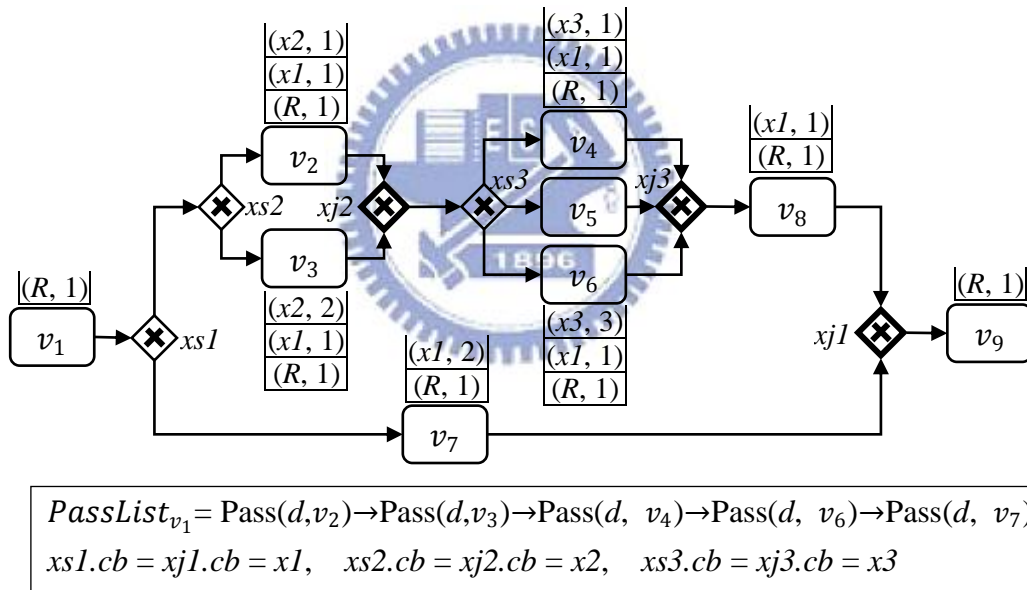


Figure 5.1: An Example of Building XBNodes.

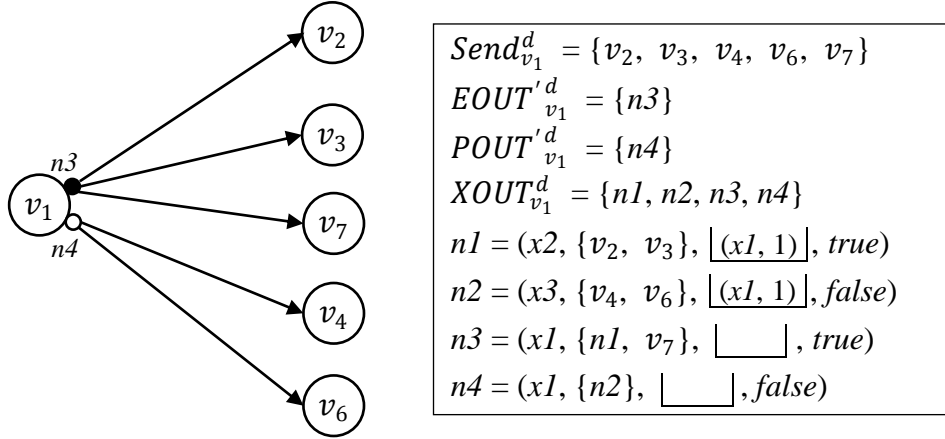


Figure 5.2: The Corresponding Artifact Flow Diagram for d .

In Figure 5.1, v_1 passes artifact d to $v_2, v_3, v_4, v_6,$ and v_7 . These activities are added into $POUT_{v_1}^d$ according to Definition 3.18. In order to update the properties of outflows, $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_6),$ and (v_1, v_7) , the XBNodes are built by $BuildXBNodes(POUT_{v_1}^d, v_1, XOUT_{v_1}^d)$.

The Table 5.2 shows the construction of XBNodes in $BuildXBNodes(POUT_{v_1}^d, v_1, XOUT_{v_1}^d)$ step by step. The element $(R, 1)$ in the bottom of each $ABStack$ is ignored in Table 5.2. Table 5.3 shows the states of each XBNode in $XOUT_{v_1}^d$.

Figure 5.1	$S=POUT_{v_1}^d$
Step 1	$S=\{v_2(\frac{(x2, 1)}{(x1, 1)}), v_3(\frac{(x2, 2)}{(x1, 1)}), v_4(\frac{(x3, 1)}{(x1, 1)}), v_6(\frac{(x3, 3)}{(x1, 1)}), v_7(\frac{(x1, 2)}{(x1, 1)})\}$
Step 2	$S=\{n1(x2, \{v_2, v_3\}, [(x1, 1)], true), v_4(\frac{(x3, 1)}{(x1, 1)}), v_6(\frac{(x3, 3)}{(x1, 1)}), v_7(\frac{(x1, 2)}{(x1, 1)})\}$
Step 3	$S=\{n1(x2, \{v_2, v_3\}, [(x1, 1)], true), n2(x3, \{v_4, v_6\}, [(x1, 1)], false), v_7(\frac{(x1, 2)}{(x1, 1)})\}$
Step 4	$S=\{n1(x2, \{v_2, v_3\}, [(x1, 1)], true), v_7(\frac{(x1, 2)}{(x1, 1)})\}$ $S_{same}=\{n2(x3, \{v_4, v_6\}, [(x1, 1)], false)\}$
Step 5	$S=\{n3(x1, \{n1, v_7\}, [], true)\}$

	$S=\{\mathbf{n4}(x1, \{n2\}, \underline{\quad\quad}), false\}$
Result	$POUT'_{v_1}^d=\{\mathbf{n4}(x1, \{n2\}, \underline{\quad\quad}), false\}$ $EOUT'_{v_1}^d=\{\mathbf{n3}(x1, \{n1, v_7\}, \underline{\quad\quad}), true\}$

Table 5.2: Construction of XBNodes in $BuildXBNodes(POUT_{v_1}^d, v_1, XOUT_{v_1}^d)$.

XBNodes	<i>blockID</i>	<i>cv_set</i>	<i>ABStack</i>	<i>isUncond</i>	<i>Parent</i>	<i>Attached</i>
<i>n1</i>	<i>x2</i>	$\{v_2, v_3\}$	$\frac{(x1,1)}{(R, 1)}$	<i>true</i>	<i>n3</i>	<i>Null</i>
<i>n2</i>	<i>x3</i>	$\{v_4, v_6\}$	$\frac{(x1,1)}{(R, 1)}$	<i>false</i>	<i>n4</i>	<i>Null</i>
<i>n3</i>	<i>x1</i>	$\{n1, v_7\}$	$(R, 1)$	<i>true</i>	<i>Null</i>	<i>v_1</i>
<i>n4</i>	<i>x1</i>	$\{n2\}$	$(R, 1)$	<i>false</i>	<i>Null</i>	<i>v_1</i>

Table 5.3: The States of Each XBNode in $XOUT_{v_1}^d$.

In Step 1 of Table 5.2, $S = \{v_2, v_3, v_4, v_6, v_7\}$. These activities located in the same deepest XOR control blocks and branches are identified. In these activities/XBNodes, only one is selected to stay in S and the remainders are moved to S_{same} from S . In Step 1, there is no activity moved to S_{same} .

Next, the activity with maximal *xlevel* is selected to construct an XBNode. In Step 1, the *xlevel* of $v_2, v_3, v_4,$ and v_6 are all 2; thus, v_2 is selected randomly to build an XBNode. Because $x2$ is the deepest XOR control block which contains v_2 , an XBNode $n1$ is created to represent $x2$. All the activities located in $x2$ are added into $n1$; thus, $n1.cv_set = \{v_2, v_3\}$. Because $ABStack_{n1}$ is used to express the location of $x2$, $ABStack_{n1} = ABStack_{x2.start}$. In addition, $n1$ is unconditional because $x2$ always receives artifact d from v_1 whether v_2 or v_3 is executed or not. Therefore, $n1.isUncond$ is *true*. The result of building $n1$ is showed in Step 2.

In Step 2, $S = \{n1, v_4, v_6, v_7\}$ and next XBNode is built based on these activities/XBNodes. The method of building an XBNode in Step 2 is the same with Step 1. In Step 2, there is no activity/XBNode moved to S_{same} . Because the $xlevel$ of v_4 and v_6 are 2, v_4 is selected to build an XBNode. Because $x3$ is the deepest XOR control block containing v_4 , XBNode $n2$ is created to represent $x3$. $n2.cv_set = \{v_4, v_6\}$ and $ABStack_{n2} = ABStack_{x3.start}$. If v_5 is executed, $x3$ can not receive d from v_1 ; thus, $n2.isUncond = false$. After $n2$ is built, the result is showed in Step 3.

In Step 3, $S = \{n1, n2, v_7\}$. XBNodes $n1$ and $n2$ are both located in branch 1 of XOR control block $x1$; thus, $n1$ or $n2$ is moved to S_{same} . Since $n1.isUncond = true$ and $n2.isUncond = false$, $n2$ is moved to S_{same} .

In Step 4, $S = \{n1, v_7\}$. Because the $xlevel$ of $n1$ and v_7 are 1, $n1$ is selected to build an XBNode. Because $x1$ is the deepest XOR control block containing $n1$, an XBNode $n3$ is created to represent $x1$. Activity v_7 is also located in $x1$; thus, $n3.cv_set = \{n1, v_7\}$. Since $n1$ is in $n3.cv_set$, $n3.Parent = n1$. The result of building $n3$ is showed in Step 5.

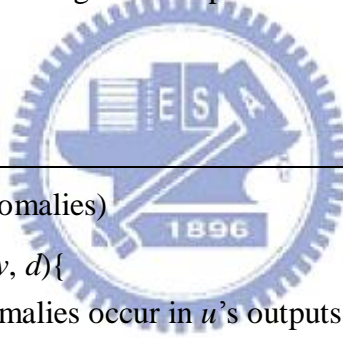
In Step 5, there are two S ; one contains $n3$ and another contains $n4$ which is built from S_{same} in Step 4. Because the $xlevel$ of $n3$ and $n4$ are both 0, they are not used to build another XBNode.

Finally, all the XBNodes are added into $POUT'_{v_1}^d$ if their $xlevel$ are all 0. Therefore, $POUT'_{v_1}^d = \{n3, n4\}$ and $n3.Attached = n4.Attached = v_1$. Since $n3$ is unconditional, $n3$ is moved to $EOUT'_{v_1}^d$ from $POUT'_{v_1}^d$. All XBNodes are built completely.

In $EOUT'_{v_1}$, since $getAllActivities(n3) = \{v_2, v_3, v_7\}$, the properties *outBlock* of outflows, (v_1, v_2) , (v_1, v_3) , and (v_1, v_7) , are updated to $n3$. In $POUT'_{v_1}$, because $getAllActivities(n4) = \{v_4, v_6\}$, the properties *outBlock* of outflows, (v_1, v_4) and (v_1, v_6) , are updated to $n4$. Therefore, the source of flows, (v_1, v_2) , (v_1, v_3) , and (v_1, v_7) , are $n3$ and the source of flows, (v_1, v_4) and (v_1, v_6) , are $n4$ in Figure 5.2.

5.4 Algorithms to Detect Artifact Usage Anomalies

As mention in Section 5.2, the function DetectAnomalies() is executed to detect anomalies after the artifact flow diagram is updated. The function DetectAnomalies() is described in Algorithm 5.14.



Algorithm 5.14 (Detecting Anomalies)

Algorithm DetectAnomalies(u, v, d) {

//Input: Identifying whether anomalies occur in u 's outputs and v 's inputs.

//Output:

1. Boolean $flag_u = false$;
2. Boolean $flag_v = false$;
3. //Detecting Missing Artifact Anomalies.
4. **if**($|EIN'_v|^d = 0 \wedge |PIN'_v|^d = 0 \wedge Role_v^d \in \{Reader, Updater, Destroyer, Relevantor\}$)
5. $flag_v = true$;
6. **if**($|EIN'_v|^d = 0 \wedge |PIN'_v|^d > 0 \wedge Role_v^d \in \{Reader, Updater, Destroyer, Relevantor\}$)
7. $flag_v = true$;
8. **if**($(|EOUT'_u|^d > 0 \vee |POUT'_u|^d > 0) \wedge Role_u^d == Destroyer$)
9. $flag_u = true$;
- 10.
11. //Detecting Artifact Conflict Anomalies.
12. **if**($|EIN'_v|^d > 1$)
13. $flag_v = true$;
14. **if**($(|EIN'_v|^d = 1 \wedge |PIN'_v|^d > 0) \vee (|EIN'_v|^d = 0 \wedge |PIN'_v|^d > 1)$)

```

15.   flag_v = true;
16.  if(( $|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0$ )  $\wedge$   $Role_v^d == \text{Producer}$ )
17.   flag_v = true;
18.
19.  //Detecting Redundant Anomalies.
20.  if( $|EOUT'_u{}^d| = 0 \wedge |POUT'_u{}^d| = 0 \wedge Role_u^d \in \{\text{Producer, Updater}\}$ )
21.   flag_u = true;
22.  if(( $|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0$ )  $\wedge$   $Role_v^d == \text{Relevantor}$ )
23.   flag_v = true;
24.
25.  if(flag_u == true)
26.    $Anomalies_d = Anomalies_d \cup \{u\}$ ;
27.  if(flag_v == true)
28.    $Anomalies_d = Anomalies_d \cup \{v\}$ ;
29.
30.  //Detecting Cross Passing Artifact Anomalies.
31.  if( $u \neq v$ ) {
32.   if( $IsParallel(u, v) == true$ )
33.     $PassAnomalies_d = PassAnomalies_d \cup \{\text{flow}(u, v)\}$ ;
34.   else if( $IsExclusive(u, v) == true$ ) {
35.     $PassAnomalies_d = PassAnomalies_d \cup \{\text{flow}(u, v)\}$ ;
36.   }
}

```

In Algorithm 5.14, the conditions at lines 3 to 9 check whether missing artifact anomalies occur at u and v . At lines 11 to 17, the conditions check whether artifact conflict anomalies occur at v . The redundant anomalies are checked at lines 19 to 23. At lines 25 to 26, if one or more anomalies occur at u , u is added into $Anomalies_d$. It is similar for v at lines 27 to 28. The cross passing artifact anomalies are checked at lines 30 to 36. If an anomaly occurs, flow (u, v) is added into $PassAnomalies_d$.

Algorithm 5.15 (Show Current Anomalies)

Algorithm ShowAnomalies(d){

//Input: Show all anomalies about d in the artifact flow diagram.

//Output:

```
1.  for each  $v \in Anomalies_d$  {
2.      Boolean  $flag = false$ ;
3.      if ( $|EIN'_v{}^d| = 0 \wedge |PIN'_v{}^d| = 0 \wedge Role_v^d \in \{Reader, Updater, Destroyer, Relevator\}$ ) {
4.          print "An Explicit Missing Artifact Anomaly occurs at Activity  $v!$ ";
5.           $flag = true$ ;
6.      }
7.      if ( $|EIN'_v{}^d| = 0 \wedge |PIN'_v{}^d| > 0 \wedge Role_v^d \in \{Reader, Updater, Destroyer, Relevator\}$ ) {
8.          print "An Implicit Missing Artifact Anomaly occurs at Activity  $v!$ ";
9.           $flag = true$ ;
10.     }
11.     if ( $|EOUT'_v{}^d| > 0 \vee |POUT'_v{}^d| > 0 \wedge Role_v^d == Destroyer$ ) {
12.         print "A Destroyed Artifact Anomaly occurs at Activity  $v!$ ";
13.          $flag = true$ ;
14.     }
15.     if ( $|EIN'_v{}^d| > 1$ ) {
16.         print "An Explicit Artifact Conflict Anomaly occurs at Activity  $v!$ ";
17.          $flag = true$ ;
18.     }
19.     if ( $(|EIN'_v{}^d| = 1 \wedge |PIN'_v{}^d| > 0) \vee (|EIN'_v{}^d| = 0 \wedge |PIN'_v{}^d| > 1)$ ) {
20.         print "An Implicit Artifact Conflict Anomaly occurs at Activity  $v!$ ";
21.          $flag = true$ ;
22.     }
23.     if ( $(|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0) \wedge Role_v^d == Producer$ ) {
24.         print "A Production Conflict Anomaly occurs at Activity  $v!$ ";
25.          $flag = true$ ;
26.     }
27.     if ( $|EOUT'_v{}^d| = 0 \wedge |POUT'_v{}^d| = 0 \wedge Role_v^d \in \{Producer, Updater\}$ ) {
28.         print "A Redundant Update/Initialize Anomaly occurs at Activity  $v!$ ";
29.          $flag = true$ ;
30.     }
31.     if ( $(|EIN'_v{}^d| > 0 \vee |PIN'_v{}^d| > 0) \wedge Role_v^d == Relevator$ ) {
32.         print "A Redundant Pass Anomaly occurs at Activity  $v!$ ";
33.          $flag = true$ ;
34.     }
```



```

35.     if(flag == false){
36.          $Anomalies_d = Anomalies_d \setminus \{v\}$ ;
37.     }
38. }
39. for each  $f=(u, v) \in PassAnomalies_d$ {
40.     if(IsParallel( $u, v$ ) == true){
41.         print “A Passing between Parallel Activities Anomaly occurs at  $u!$ ”;
42.     }else if(IsExclusive( $u, v$ ) == true){
43.         print “A Passing between Exclusive Activities Anomaly occurs at  $u!$ ”;
44.     }else{
45.          $PassAnomalies_d = PassAnomalies_d \setminus \{f\}$ ;
46.     }
47. }
}

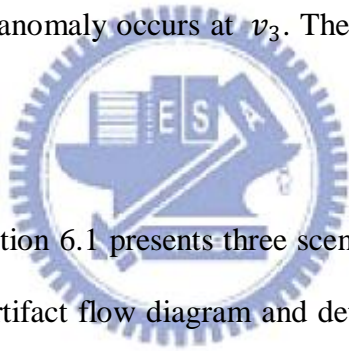
```

Algorithm 5.15 is executed when designer wants to know what anomalies about d occur in current process. At lines 1 to 38, each activity in $Anomalies_d$ is checked whether anomalies occur. If an anomaly occurs, the warning message is printed. Otherwise, the activity is removed from $Anomalies_d$. Similarly, each flow in $PassAnomalies_d$ is identified whether cross passing anomalies occur in lines 39 to 47. If no anomaly occurs in a flow, the flow is removed from $PassAnomalies_d$.

Chapter 6. Examples for Illustrating Incremental Algorithms

For illustrating and demonstrating our incremental algorithms, an example of a control flow graph and its artifact flow diagram for d are presented in this section. Figure 6.1 shows an example of a control flow graph, and the states of each activity in it are showed in Table 6.1. Figure 6.2 shows the artifact flow diagram for d extracted from Figure 6.1, and the states of each activity in it are displayed in Table 6.2.

In Figure 6.2, since v_9 is a producer of d and $|EOUT'_{v_9}| = |POUT'_{v_9}| = 0$, a redundant initialize anomaly occurs at v_9 . In addition, because v_3 is a relevantor and $|EIN'_{v_3}| = 1$, a redundant pass anomaly occurs at v_3 . These anomalies are showed in Table 6.3.



Based on this example, Section 6.1 presents three scenarios about activity modifications to illustrate the updates of the artifact flow diagram and detections of anomalies. Section 6.2 and Section 6.3 also show three scenarios about pass insertion and deletion respectively.

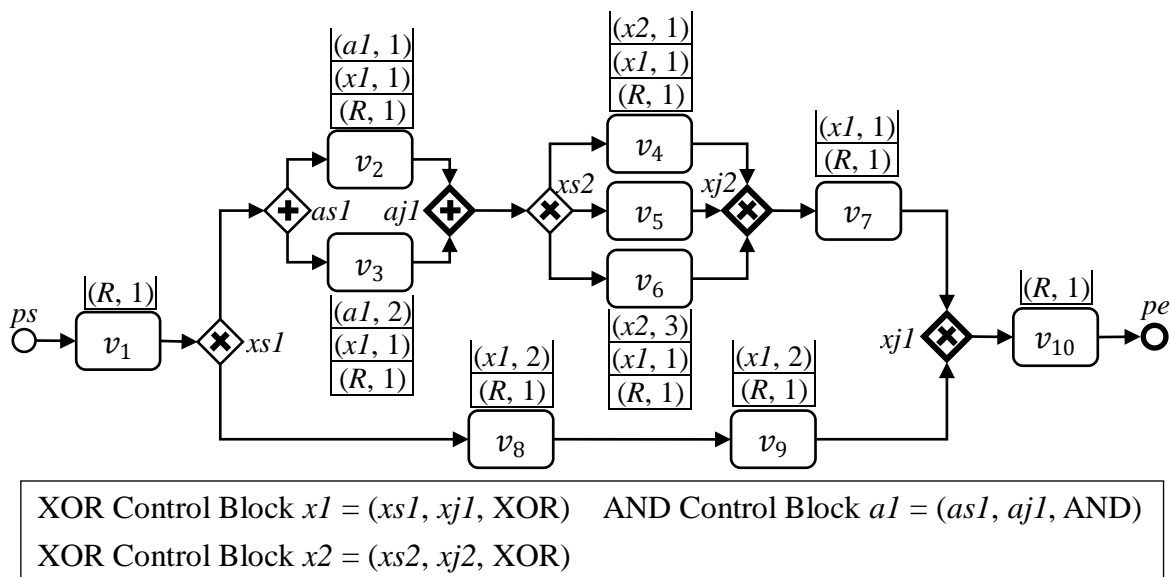


Figure 6.1: An Example of a Control Flow Graph.

	$PassList_v$	Input and Output Sets
v_1	$PassList_{v_1} = Pass(d, v_2) \rightarrow Pass(d, v_3) \rightarrow Pass(d, v_8)$	$d \notin I_{v_1} \quad d \in U_{v_1}^+ \quad d \notin U_{v_1}^- \quad d \in O_{v_1}$
v_2	$PassList_{v_2} = Pass(d, v_4) \rightarrow Pass(d, v_5) \rightarrow Pass(d, v_6)$	$d \in I_{v_2} \quad d \in U_{v_2}^+ \quad d \notin U_{v_2}^- \quad d \in O_{v_2}$
v_3	$PassList_{v_3} = \emptyset$	$d \notin I_{v_3} \quad d \notin U_{v_3}^+ \quad d \notin U_{v_3}^- \quad d \notin O_{v_3}$
v_4	$PassList_{v_4} = Pass(d, v_7)$	$d \in I_{v_4} \quad d \notin U_{v_4}^+ \quad d \notin U_{v_4}^- \quad d \in O_{v_4}$
v_5	$PassList_{v_5} = Pass(d, v_7)$	$d \in I_{v_5} \quad d \notin U_{v_5}^+ \quad d \notin U_{v_5}^- \quad d \in O_{v_5}$
v_6	$PassList_{v_6} = Pass(d, v_7)$	$d \in I_{v_6} \quad d \in U_{v_6}^+ \quad d \notin U_{v_6}^- \quad d \in O_{v_6}$
v_7	$PassList_{v_7} = \emptyset$	$d \in I_{v_7} \quad d \notin U_{v_7}^+ \quad d \in U_{v_7}^- \quad d \notin O_{v_7}$
v_8	$PassList_{v_8} = \emptyset$	$d \in I_{v_8} \quad d \notin U_{v_8}^+ \quad d \notin U_{v_8}^- \quad d \notin O_{v_8}$
v_9	$PassList_{v_9} = \emptyset$	$d \notin I_{v_9} \quad d \in U_{v_9}^+ \quad d \notin U_{v_9}^- \quad d \notin O_{v_9}$
v_{10}	$PassList_{v_{10}} = \emptyset$	$d \notin I_{v_{10}} \quad d \notin U_{v_{10}}^+ \quad d \notin U_{v_{10}}^- \quad d \notin O_{v_{10}}$

Table 6.1: The States of Each Activity in Figure 6.1.

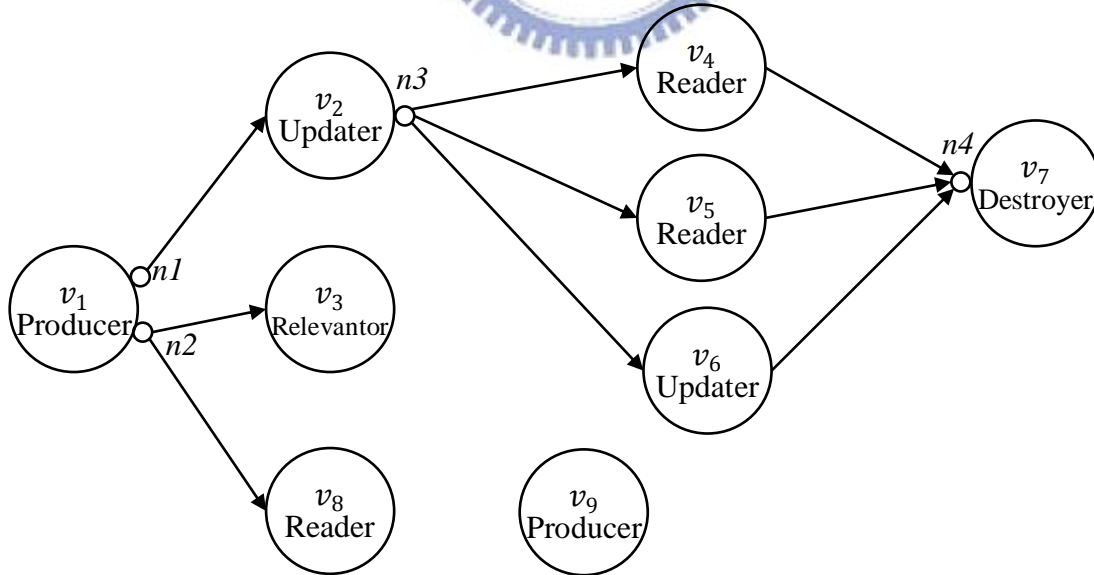


Figure 6.2: The Artifact Flow Diagram for d .

	$Receive_v^d$	EIN_v^d	PIN_v^d	$Send_v^d$	$EOUT_v^d$	$POUT_v^d$
v_1	\emptyset	\emptyset	\emptyset	$\{v_2, v_3, v_8\}$	$\{n2\}$	$\{n1\}$
v_2	$\{v_1\}$	$\{v_1\}$	\emptyset	$\{v_4, v_5, v_6\}$	$\{n3\}$	\emptyset
v_3	$\{v_1\}$	$\{v_1\}$	\emptyset	\emptyset	\emptyset	\emptyset
v_4	$\{v_2\}$	$\{v_2\}$	\emptyset	$\{v_7\}$	$\{v_7\}$	\emptyset
v_5	$\{v_2\}$	$\{v_2\}$	\emptyset	$\{v_7\}$	$\{v_7\}$	\emptyset
v_6	$\{v_2\}$	$\{v_2\}$	\emptyset	$\{v_7\}$	$\{v_7\}$	\emptyset
v_7	$\{v_4, v_5, v_6\}$	$\{n4\}$	\emptyset	\emptyset	\emptyset	\emptyset
v_8	$\{v_1\}$	$\{v_1\}$	\emptyset	\emptyset	\emptyset	\emptyset
v_9	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Table 6.2: The States of Each Activity in Figure 6.2.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN_{v_3}^d = 1$	Redundant Pass
v_9	$Role_{v_9}^d = \text{Producer} \wedge EOUT_{v_9}^d = POUT_{v_9}^d = 0$	Redundant Initialization

Table 6.3: The Anomalies Occur in Figure 6.2.

6.1 Activity Modification

In this section, the artifact flow diagram in Figure 6.2 is updated due to the following edit operations: (1) Modifying v_9 , (2) Modifying v_8 , and (3) Modifying v_2 , in order. After each edit operation, the updated artifact flow diagrams are showed in Figure 6.3, 6.4, and 6.5.

1. **Modifying v_9 :** $Role_{v_9}^d = \text{Producer} \rightarrow \text{Irrelevantor}$

After modification, activity v_9 becomes an irrelevantor of d . Therefore, v_9 is removed from the artifact flow diagram in Figure 6.3.

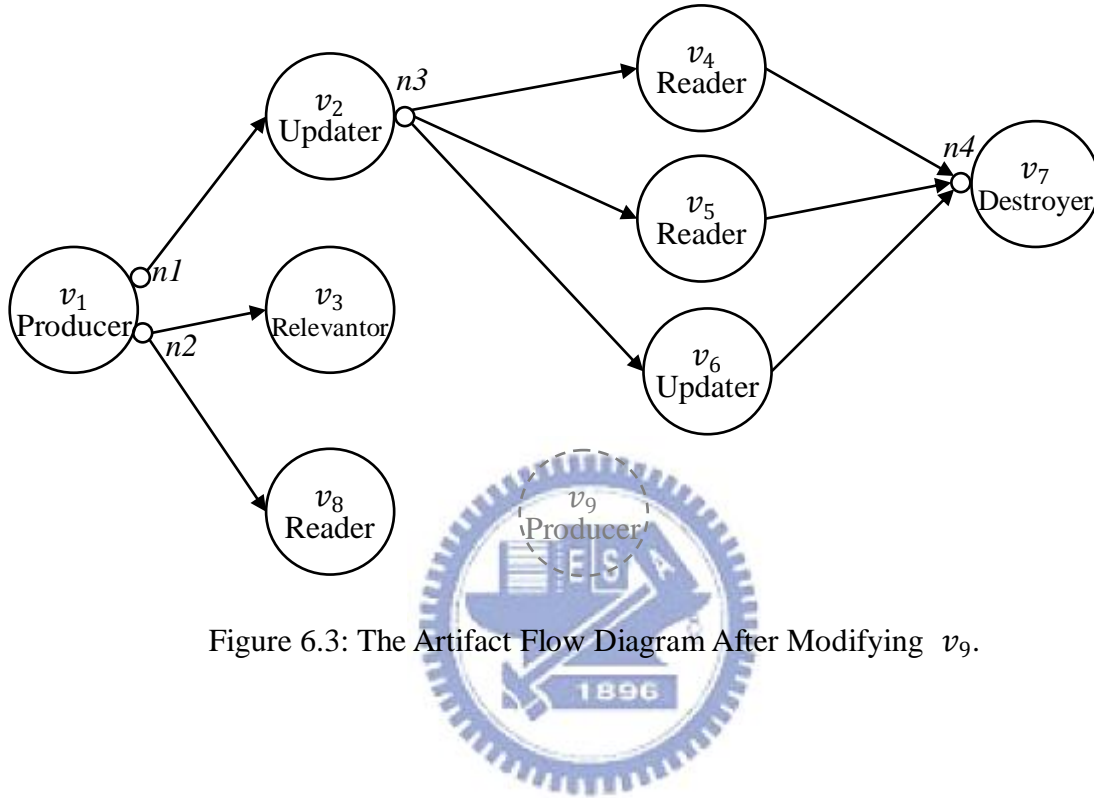


Figure 6.3: The Artifact Flow Diagram After Modifying v_9 .

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN'_{v_3}^d = 1$	Redundant Pass

Table 6.4: The Anomalies Occur in Figure 6.3.

2. **Modifying v_8 :** $Role_{v_8}^d = \text{Reader} \rightarrow \text{Relevantor}$

After modifying, v_8 becomes a relevantor of d and a redundant pass anomaly occurs at v_8 .

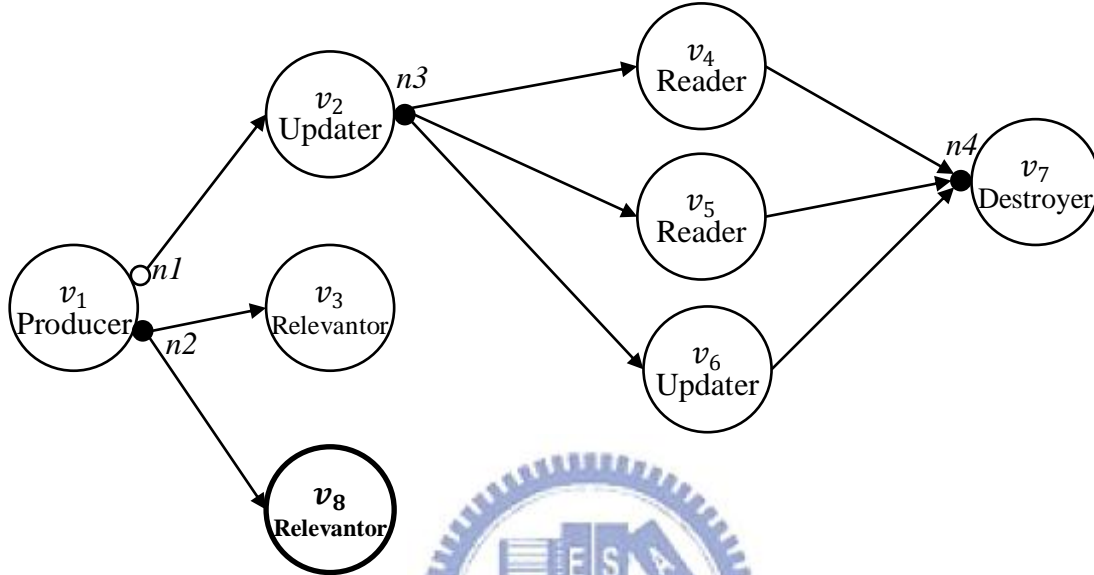


Figure 6.4: The Artifact Flow Diagram After Modifying v_8 .

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN'_{v_3}^d = 1$	Redundant Pass
v_8	$Role_{v_8}^d = \text{Relevantor} \wedge EIN'_{v_8}^d = 1$	Redundant Pass

Table 6.5: The Anomalies Occur in Figure 6.4.

3. Modifying v_2 : $Role_{v_2}^d = \text{Updater} \rightarrow \text{Destroyer}$

After modification, v_2 becomes a destroyer of d and a destroyed artifact anomaly occurs at v_2 .

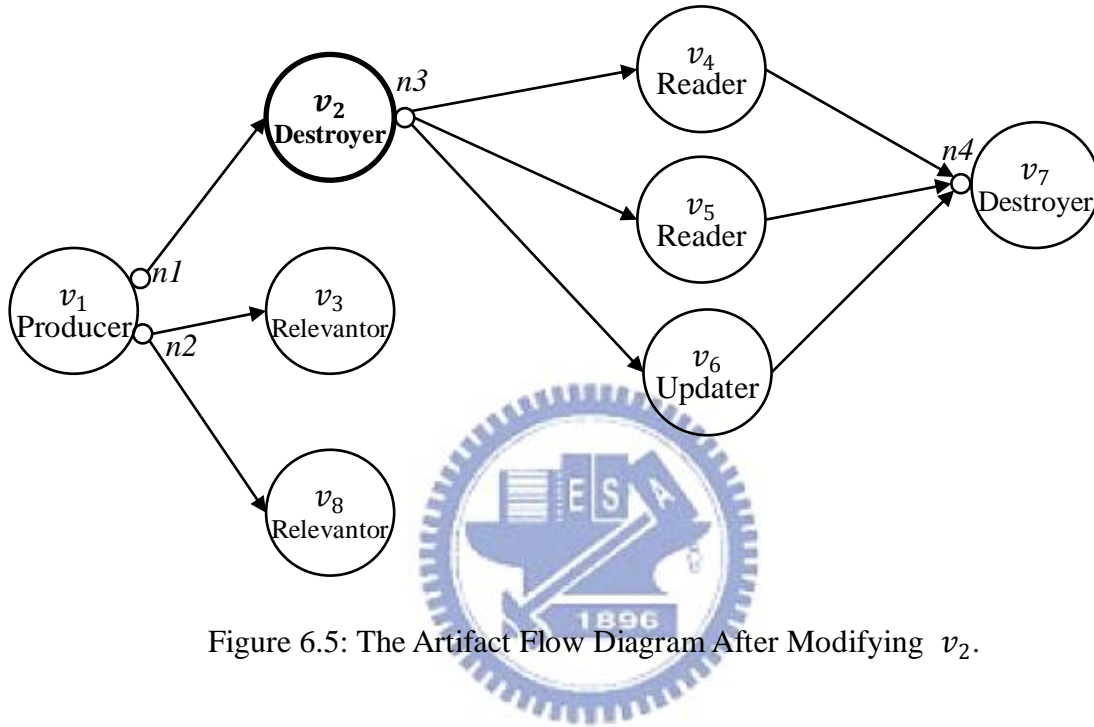


Figure 6.5: The Artifact Flow Diagram After Modifying v_2 .

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN'_{v_3}^d = 1$	Redundant Pass
v_8	$Role_{v_8}^d = \text{Relevantor} \wedge EIN'_{v_8}^d = 1$	Redundant Pass
v_2	$Role_{v_2}^d = \text{Destroyer} \wedge EOUT'_{v_2}^d = 1$	Destroyed Artifact

Table 6.6: The Anomalies Occur in Figure 6.5.

6.2 Pass Insertion

In this section, the artifact flow diagram in Figure 6.2 is updated due to the following edit operations: (1) Adding $\text{Pass}(d, v_{10})$ into PassList_{v_9} , (2) Adding $\text{Pass}(d, v_6)$ into PassList_{v_3} , and (3) Adding $\text{Pass}(d, v_6)$ into PassList_{v_8} , in order. After each edit operation, the updated artifact flow diagrams are showed in Figure 6.6, 6.7, and 6.8.

1. Adding $\text{Pass}(d, v_{10})$ into PassList_{v_9} :

After adding, flow (v_9, v_{10}) and activity v_{10} are added into the artifact flow diagram in Figure 6.6. Then $\text{UpdateOutflows}(v_9, d)$ is executed to construct XBNODE in $\text{POUT}_{v_9}^d$ and update $(v_9, v_{10}).\text{outBlock}$. $\text{UpdateInflows}(v_{10}, d)$ is executed to construct XBNODE in $\text{PIN}_{v_{10}}^d$ and update $(v_9, v_{10}).\text{inBlock}$. The methods of updating outBlock and inBlock and construction of XBNODEs have described in Section 5.3. Hence, the results of outBlock , inBlock , and XBNODEs are showed in Figure 6.6 without detailed explanation. In Figure 6.6, a conditional XBNODE $n5$ is constructed in $\text{PIN}_{v_{10}}^d$, $(v_9, v_{10}).\text{outBlock} = \text{Null}$, and $(v_9, v_{10}).\text{inBlock} = n5$.

Since v_{10} is a relevantor and $|\text{EIN}_{v_{10}}^d| = 0$ and $|\text{PIN}_{v_{10}}^d| = 1$, an implicit missing artifact anomaly occurs at v_{10} . In addition, because v_{10} is a relevantor and $|\text{PIN}_{v_{10}}^d| = 1$, a redundant pass anomaly also occurs at v_{10} . The anomalies occur in Figure 6.6 are shows in Table 6.8.

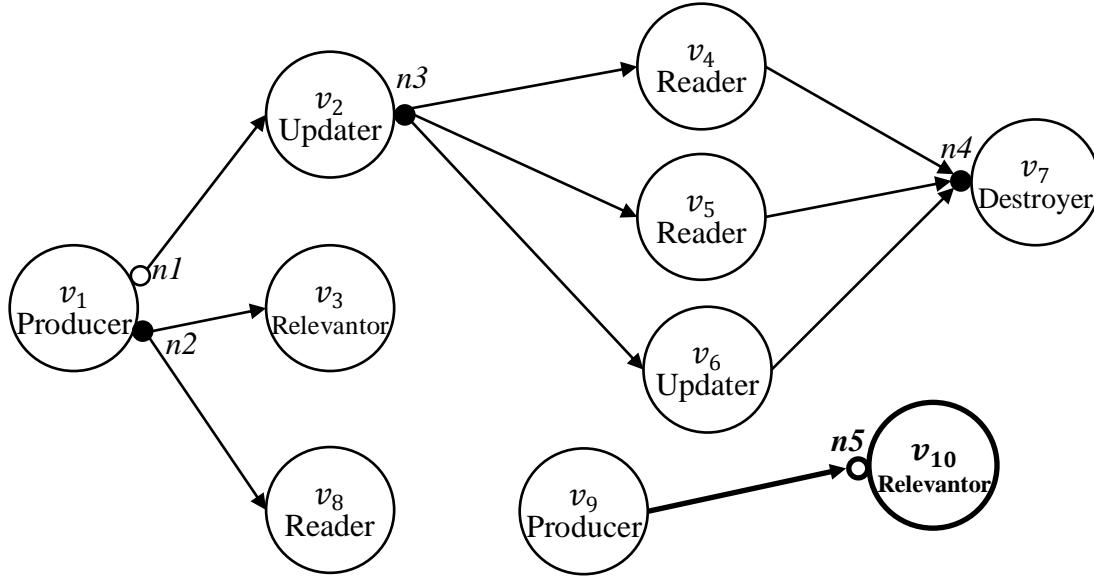


Figure 6.6: The Artifact Flow Diagram After Adding $Pass(d, v_{10})$ into $PassList_{v_9}$.

	$Receive_v^d$	EIN_v^d	PIN_v^d	$Send_v^d$	$EOUT_v^d$	$POUT_v^d$
v_9	\emptyset	\emptyset	\emptyset	$\{v_{10}\}$	$\{v_{10}\}$	\emptyset
v_{10}	$\{v_9\}$	\emptyset	$\{n5\}$	\emptyset	\emptyset	\emptyset

Table 6.7: The States of Activities v_9 and v_{10} in Figure 6.6.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN_{v_3}^d = 1$	Redundant Pass
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge EIN_{v_{10}}^d = 0 \wedge PIN_{v_{10}}^d = 1$	Implicit Missing Artifact
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge PIN_{v_{10}}^d = 1$	Redundant Pass

Table 6.8: The Anomalies Occur in Figure 6.6.

2. Adding $\text{Pass}(d, v_6)$ into PassList_{v_3} :

After insertion, flow (v_3, v_6) is added into the artifact flow diagram in Figure 6.7. Since $|\text{EIN}'_d| = 2$, an explicit artifact conflict anomaly occurs at v_6 .

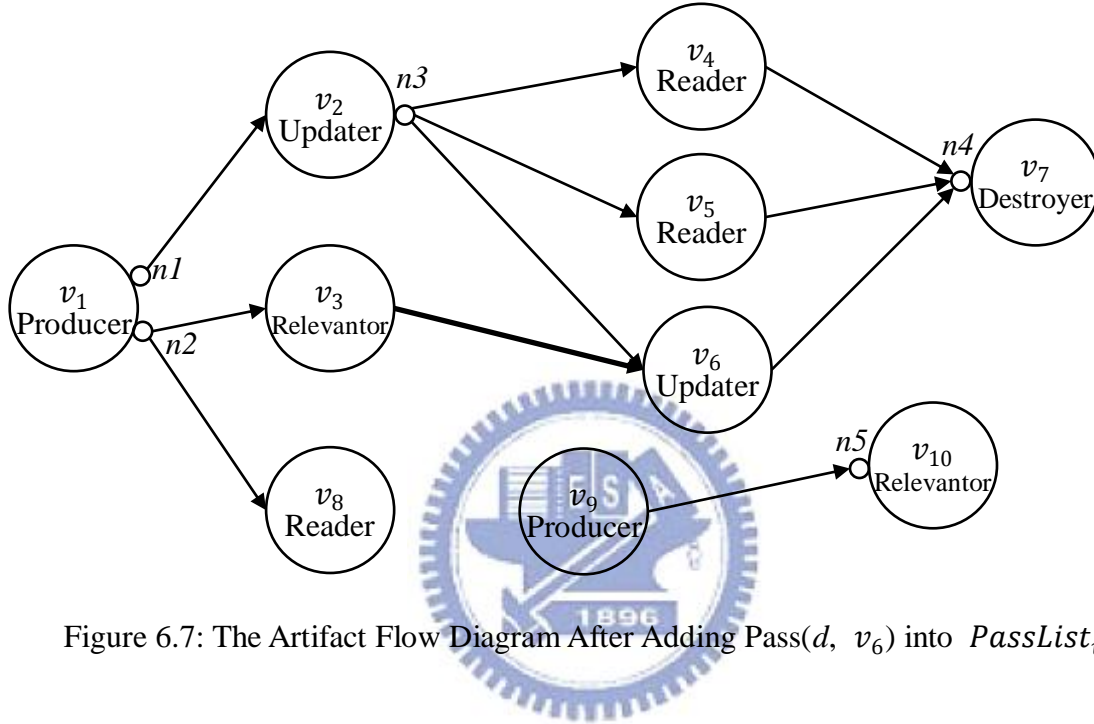


Figure 6.7: The Artifact Flow Diagram After Adding $\text{Pass}(d, v_6)$ into PassList_{v_3} .

	Receive_v^d	EIN'_v^d	PIN'_v^d	Send_v^d	EOUT'_v^d	POUT'_v^d
v_3	$\{v_1\}$	$\{v_1\}$	\emptyset	$\{v_6\}$	$\{v_6\}$	\emptyset
v_6	$\{v_2, v_3\}$	$\{v_2, v_3\}$	\emptyset	$\{v_7\}$	$\{v_7\}$	\emptyset

Table 6.9: The States of Activities v_3 and v_6 in Figure 6.7.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN'_{v_3}{}^d = 1$	Redundant Pass
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge EIN'_{v_{10}}{}^d = 0 \wedge PIN'_{v_{10}}{}^d = 1$	Implicit Missing Artifact
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge PIN'_{v_{10}}{}^d = 1$	Redundant Pass
v_6	$ EIN'_{v_6}{}^d = 2$	Explicit Artifact Conflict

Table 6.10: The Anomalies Occur in Figure 6.7.

3. Adding $\text{Pass}(d, v_6)$ into PassList_{v_8} :

Since v_6 and v_8 are exclusive activities, an XOR flow (v_6, v_8) is added into the artifact flow diagram in Figure 6.8 and a passing between exclusive activities anomaly occurs at v_8 .

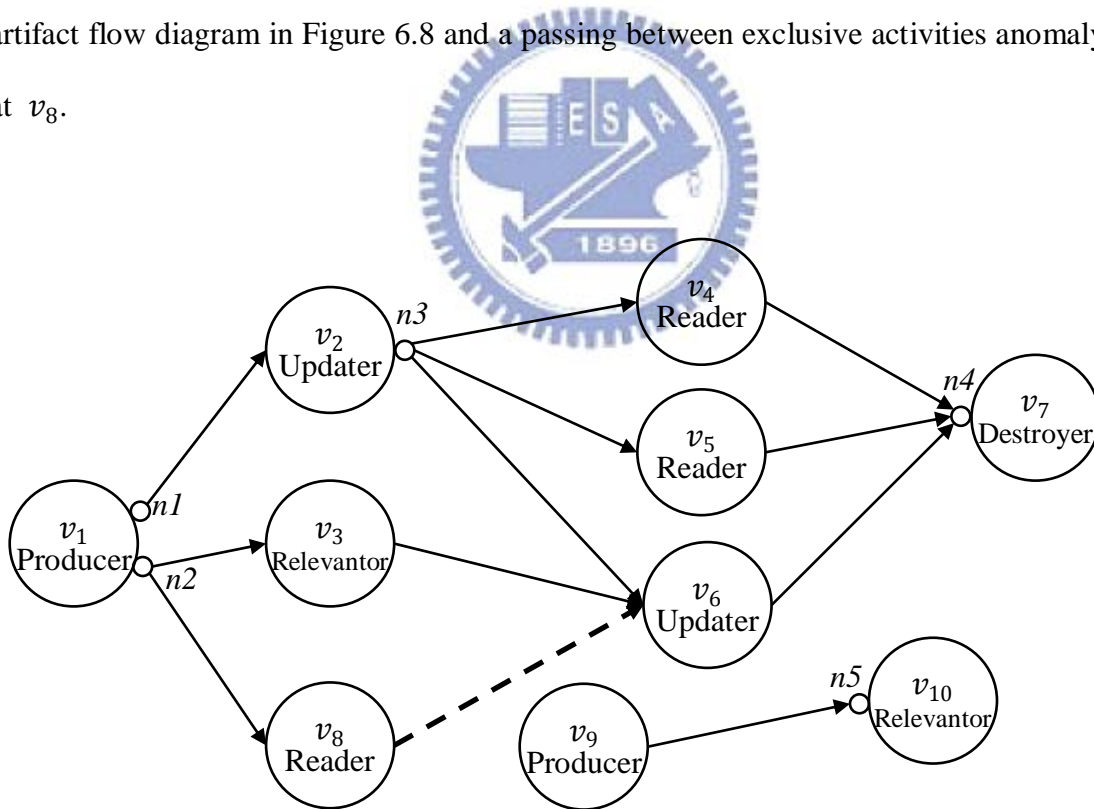


Figure 6.8: The Artifact Flow Diagram After Adding $\text{Pass}(d, v_6)$ into PassList_{v_8} .

	$Receive_v^d$	EIN_v^d	PIN_v^d	$Send_v^d$	$EOUT_v^d$	$POUT_v^d$
v_6	$\{v_2, v_8\}$	$\{v_2\}$	\emptyset	$\{v_7\}$	$\{v_7\}$	\emptyset
v_8	$\{v_1\}$	$\{v_1\}$	\emptyset	$\{v_6\}$	\emptyset	\emptyset

Table 6.11: The States of Activities v_6 and v_8 in Figure 6.8.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN_{v_3}^d = 1$	Redundant Pass
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge EIN_{v_{10}}^d = 0 \wedge PIN_{v_{10}}^d = 1$	Implicit Missing Artifact
v_{10}	$Role_{v_{10}}^d = \text{Relevantor} \wedge PIN_{v_{10}}^d = 1$	Redundant Pass
v_6	$ EIN_{v_6}^d = 2$	Explicit Artifact Conflict
v_8	$\text{IsExclusive}(v_8, v_6) = \text{true}$	Passing between Exclusive Activities

Table 6.12: The Anomalies Occur in Figure 6.8.



6.3 Pass Deletion

In this section, the artifact flow diagram in Figure 6.2 is updated due to the following edit operations: (1) Removing $\text{Pass}(d, v_8)$ from PassList_{v_1} , (2) Removing $\text{Pass}(d, v_7)$ from PassList_{v_4} , and (3) Removing $\text{Pass}(d, v_3)$ from PassList_{v_1} , in order. After each edit operation, the updated artifact flow diagrams are showed in Figure 6.9, 6.10, and 6.11.

1. Removing $\text{Pass}(d, v_8)$ from PassList_{v_1} :

After removing, flow (v_1, v_8) is removed from the artifact flow diagram in Figure 6.9. Then $\text{UpdateOutflows}(v_1, d)$ is executed to build XBNodes in $\text{POUT}_{v_1}^d$ and update outBlock

of outflows of v_1 . $\text{UpdateInflows}(v_8, d)$ is executed to build XBNode in $\text{PIN}_{v_8}^d$ and update inBlock of inflows of v_8 . In Figure 3.9, $n2$ becomes a conditional XBNode and is moved to $\text{EOUT}_{v_1}^d$ from $\text{POUT}_{v_1}^d$. In addition, because v_8 is a reader of d and $|\text{EIN}_{v_8}^d| = |\text{PIN}_{v_8}^d| = 0$, an explicit missing artifact anomaly occurs at v_8 .

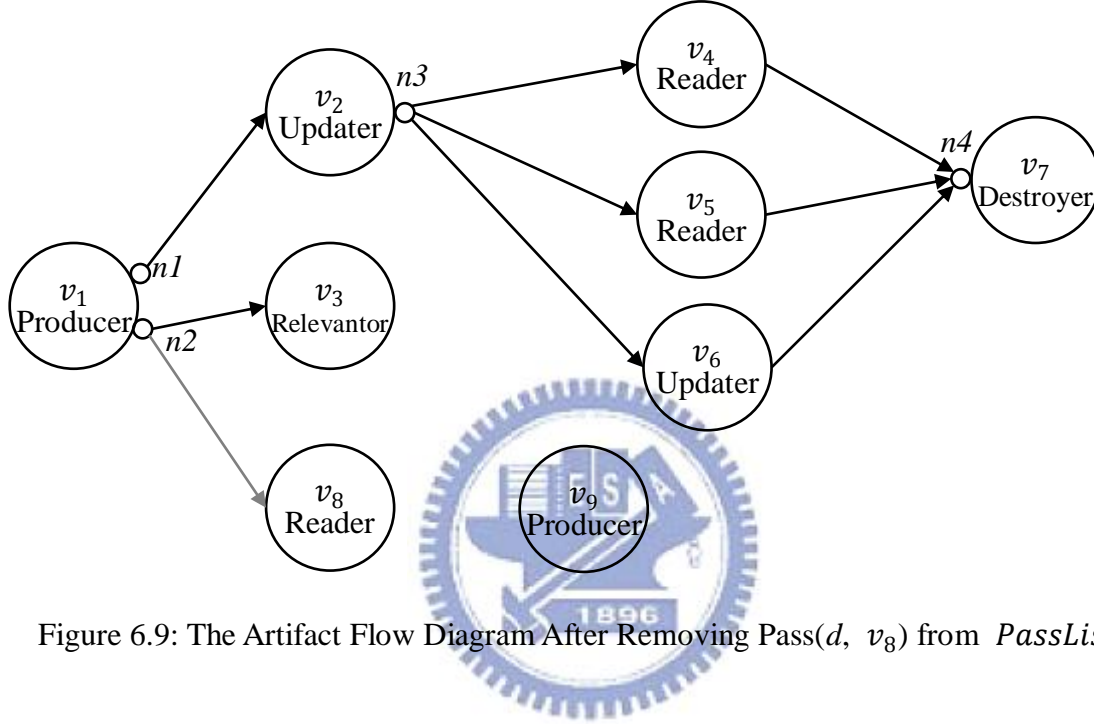


Figure 6.9: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_8)$ from PassList_{v_1} .

	Receive_v^d	EIN_v^d	PIN_v^d	Send_v^d	EOUT_v^d	POUT_v^d
v_1	\emptyset	\emptyset	\emptyset	$\{v_2, v_3\}$	\emptyset	$\{n1, n2\}$
v_8	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Table 6.13: The States of Activities v_1 and v_8 in Figure 6.9.

	Conditions	Artifact Usage Anomaly
v_3	$\text{Role}_{v_3}^d = \text{Relevantor} \wedge \text{EIN}_{v_3}^d = 1$	Redundant Pass
v_9	$\text{Role}_{v_9}^d = \text{Producer} \wedge \text{EOUT}_{v_9}^d = \text{POUT}_{v_9}^d = 0$	Redundant Initialization
v_8	$\text{Role}_{v_8}^d = \text{Reader} \wedge \text{EIN}_{v_8}^d = \text{PIN}_{v_8}^d = 0$	Explicit Missing Artifact

Table 6.14: The Anomalies Occur in Figure 6.9.

2. Removing $\text{Pass}(d, v_7)$ from PassList_{v_4} :

The flow (v_4, v_7) is removed from the artifact flow diagram in Figure 6.10. XBNode $n4$ becomes a conditional XBNode and is moved to $\text{PIN}'^d_{v_7}$ from $\text{EIN}'^d_{v_7}$. Since v_7 is a destroyer, $|\text{EIN}'^d_{v_7}| = 0$, and $|\text{PIN}'^d_{v_7}| = 1$, an implicit missing artifact anomaly occurs at v_7 .

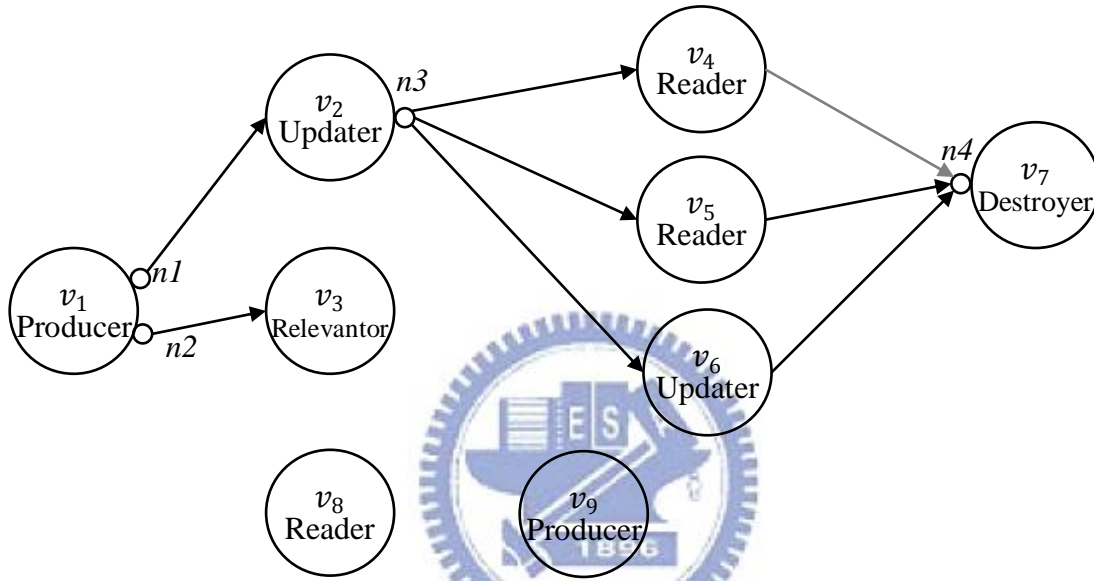


Figure 6.10: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_7)$ from PassList_{v_4} .

	Receive_v^d	EIN'^d_v	PIN'^d_v	Send_v^d	EOUT'^d_v	POUT'^d_v
v_4	$\{v_2\}$	$\{v_2\}$	\emptyset	\emptyset	\emptyset	\emptyset
v_7	$\{v_5, v_6\}$	\emptyset	$\{n4\}$	\emptyset	\emptyset	\emptyset

Table 6.15: The States of Activities v_4 and v_7 in Figure 6.10.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN'_{v_3}^d = 1$	Redundant Pass
v_9	$Role_{v_9}^d = \text{Producer} \wedge EOUT'_{v_9}^d = POUT'_{v_9}^d = 0$	Redundant Initialization
v_8	$Role_{v_8}^d = \text{Reader} \wedge EIN'_{v_8}^d = PIN'_{v_8}^d = 0$	Explicit Missing Artifact
v_7	$Role_{v_7}^d = \text{Destroyer} \wedge EIN'_{v_7}^d = 0 \wedge PIN'_{v_7}^d = 1$	Implicit Missing Artifact

Table 6.16: The Anomalies Occur in Figure 6.10.

3. Removing $\text{Pass}(d, v_3)$ from PassList_{v_1} :

After removing, flow (v_1, v_3) is removed from the artifact flow diagram in Figure 6.11. Since v_3 becomes an irrelevantor of d , v_3 is also removed. After construction XBNodes in $POUT_{v_1}^d$, only $n1$ exists in $POUT'_{v_1}^d$. There is no anomaly causing by this edit operation.

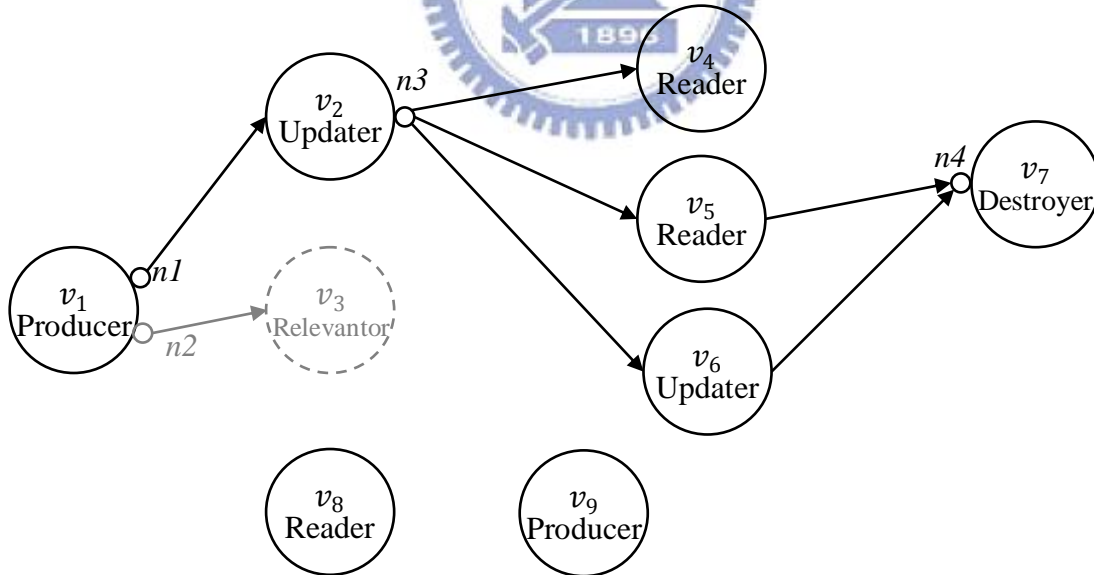


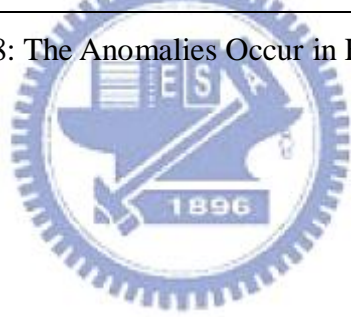
Figure 6.11: The Artifact Flow Diagram After Removing $\text{Pass}(d, v_3)$ from PassList_{v_1} .

	$Receive_v^d$	EIN_v^d	PIN_v^d	$Send_v^d$	$EOUT_v^d$	$POUT_v^d$
v_1	\emptyset	\emptyset	\emptyset	$\{v_2\}$	\emptyset	$\{nI\}$
v_3	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

Table 6.17: The States of Activities v_1 and v_3 in Figure 6.11.

	Conditions	Artifact Usage Anomaly
v_3	$Role_{v_3}^d = \text{Relevantor} \wedge EIN_{v_3}^d = 1$	Redundant Pass
v_9	$Role_{v_9}^d = \text{Producer} \wedge EOUT_{v_9}^d = POUT_{v_9}^d = 0$	Redundant Initialization
v_8	$Role_{v_8}^d = \text{Reader} \wedge EIN_{v_8}^d = PIN_{v_8}^d = 0$	Explicit Missing Artifact
v_7	$Role_{v_7}^d = \text{Destroyer} \wedge EIN_{v_7}^d = 0 \wedge PIN_{v_7}^d = 1$	Implicit Missing Artifact

Table 6.18: The Anomalies Occur in Figure 6.11.



Chapter 7. Comparisons

7.1 Comparison of Artifact Transmission Models

In GDS, there is a common anomaly called *race condition anomaly*. A race condition anomaly occurs when multiple activities try to use the same artifact in parallel. Each artifact is only allowed to have one instance in GDS. Therefore, different versions of an artifact exist when this anomaly occurs [15].

In DCDC and ICDC, this anomaly does not occur because the parallel activities use different copies of an artifact. Figure 7.1 shows an example of a race condition anomaly in GDS. Parallel activities v_2 and v_3 update artifact d which is initialized by v_1 . Then v_4 reads d which is updated by v_2 and v_3 . According to different execution orders of v_2 and v_3 , different versions of d exist. Therefore, a race condition anomaly occurs at v_2 and v_3 .

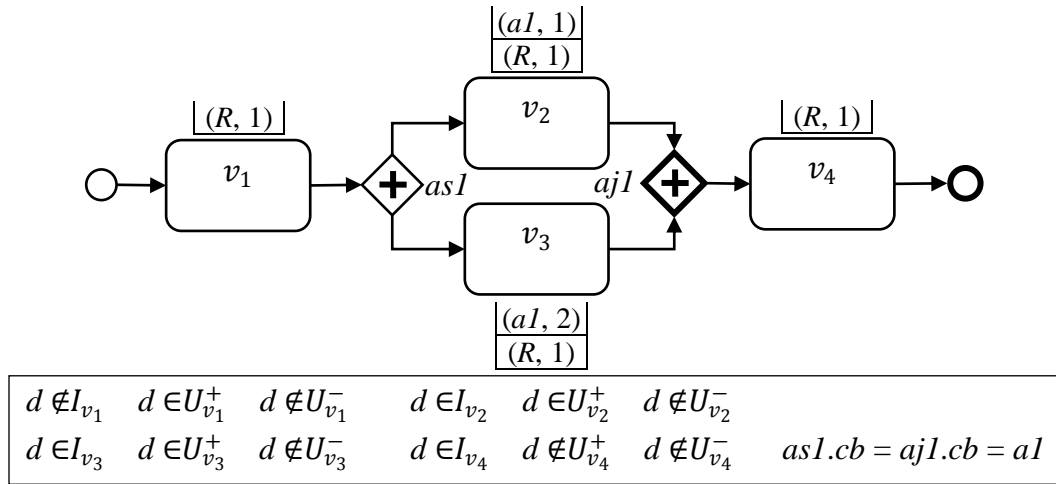


Figure 7.1: A Control Flow in GDS.

If GDS wants to be transferred into DCDC, the designer has to decide that v_2 or v_3 is executed first. Figure 7.2 (a) shows the artifact flow diagram if v_2 is selected to execute before v_3 . Figure 7.2 (b) shows the artifact flow when v_3 is selected to execute before v_2 . Since the execution order of v_2 and v_3 is decided in design time, the race condition anomaly does not occur in run time. Hence, the race condition anomaly does not occur in DCDC.

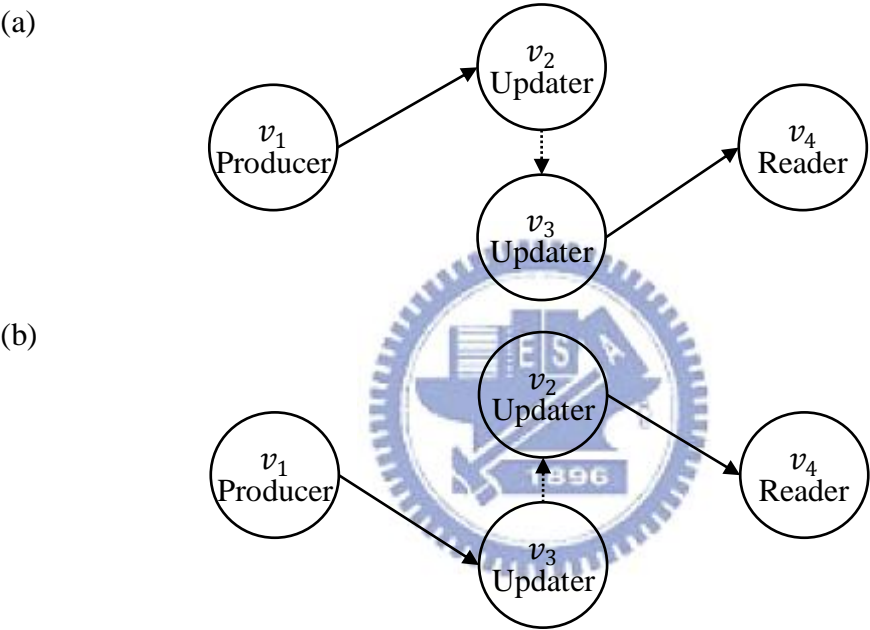


Figure 7.2: The Artifact Diagrams for Two Execution Orders.

In addition, because each artifact in GDS is allowed to have one instance in a workflow, the artifact conflict anomalies do not occur in GDS. Since the activities in ICDC always pass artifacts to their direct successors, the cross passing anomalies do not occur in ICDC. Table 7.1 shows the comparison of anomalies occurred in these artifact transmission models.

Artifact Usage Anomalies	DCDC	ICDC	GDS
1.1 Explicit Missing Artifact	○	○	○
1.2 Implicit Missing Artifact	○	○	○
1.3 Destroyed Artifact	○	×	○
2.1 Explicit Artifact Conflict	○	○	×
2.2 Implicit Artifact Conflict	○	○	×
2.3 Production Conflict	○	○	×
3.1 Passing Between Parallel Activities	○	×	○
3.2 Passing Between Exclusive Activities	○	×	○
4.1 Redundant Update/Initialization	○	○	○
4.2 Redundant Pass	○	×	×
Race Condition Anomalies	×	×	○

Table 7.1: Comparison of Artifact Transmission Models.

7.2 Comparison of Artifact Analysis Approaches

Since our previous work only discusses artifact usage anomalies in GDS, the following anomalies: (2.1) Explicit Artifact Conflict, (2.2) Implicit Artifact Conflict, (2.3) Production Conflict, and (4.2) Redundant Pass, do not occur.

In previous work, (1) *No Production* anomaly occurs when an artifact d is used by at least one activity; however, no producer of d exists in the process. (2) *Delayed Production*

anomaly occurs when an artifact d is used by an activity which precedes every producer of d . In the two cases, these activities are missing artifacts explicitly. Therefore, they belong to (1.1) Explicit Missing Artifact in this thesis.

(6) *Conditional Production* anomaly occurs when an artifact d is produced conditionally before an activity using d . The activity may miss the artifact d . Hence, this anomaly belongs to (1.2) Implicit Missing Artifact.

(3) *Early Destruction*, (7) *Conditional Destruction*, and (8) *Uncertain Destruction* anomalies occur when an artifact d is destroyed before an activity using it. Therefore, they belong to (1.3) Destroyed Artifact.

(5) *Uncertain Production* anomaly occurs when two parallel activities, one is a producer of artifact d and another uses d . The artifact d is transferred between these two parallel activities. Hence, this anomaly belongs to (3.1) Passing between Parallel Activities. On the other hand, (4) *Exclusive Production* anomaly belongs to (3.2) Passing between Exclusive Activities.

(9) *Explicit Redundant Update* and (10) *Potential Redundant Update* anomalies occur when an artifact d is updated by an activity and the result is unused for all succeeding activities. Hence, they belong to (4.1) Redundant Update/Initialization.

Finally, (11) *Multiple Parallel Productions*, (12) *Multiple Parallel Updates*, (13) *Parallel Read and Update* anomalies occur due to race condition. Therefore, they do not occur in our model. Table 7.2 shows the comparison of the anomalies addressed in [15] and our work.

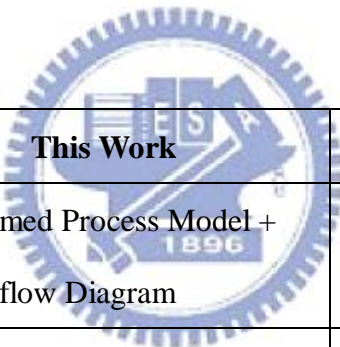
Artifact Anomalies in this thesis	Artifact Usage Anomalies in [15]
1.1 Explicit Missing Artifact	(1)No Production, (2)Delayed Production
1.2 Implicit Missing Artifact	(6)Conditional Production
1.3 Destroyed Artifact	(3)Early Destruction, (7)Conditional Destruction, (8)Uncertain Destruction
2.1 Explicit Artifact Conflict	-
2.2 Implicit Artifact Conflict	-
2.3 Production Conflict	-
3.1 Passing between Parallel Activities	(5)Uncertain Production
3.2 Passing between Exclusive Activities	(4)Exclusive Production
4.1 Redundant Update/Initialization	(9)Explicit Redundant Update, (10)Potential Redundant Update
4.2 Redundant Pass	-
Race Condition Anomalies	(11)Multiple Parallel Productions, (12)Multiple Parallel Updates, (13)Parallel Read and Update

Table 7.2: Comparison of the Artifact Usage Anomalies Addressed.

In previous work, a control flow diagram is proposed to represent a process. The concerned artifact operations include: Initialize, Read, Update, and Destroy. The artifact transmission is discussed in GDS. Besides, a batch algorithm is introduced to traversal a control flow and detects anomalies.

In our work, we not only propose a process model but also introduce an artifact flow diagram to represent artifact usages and transmissions. Since, the artifact transmissions are discussed in DCDC, each activity in DCDC can decide where artifacts are passed. Therefore, an additional artifact operation, pass, is concerned in our model.

Finally, this thesis presents several edit operations for editing a process. The effects on an artifact flow diagram for each edit operation are also discussed and the incremental algorithms are proposed to maintain the artifact flow diagram. Artifact usage anomalies can be identified according to the updated artifact flow diagram. Table 7.3 shows the summary of comparisons with previous work.



	This Work	Previous Work [15]
Fundamental Model	Well-formed Process Model + Artifact flow Diagram	Control flow Diagram
Artifact Transmission Model	Distinct Control and Data Channels (DCDC)	Global Data Store (GDS)
Artifact Operations	Initialize, Read, Update, Destroy, Pass	Initialize, Read, Update, Destroy
Detecting Methodology	Incremental	Batch

Table 7.3: Comparison with Previous Work.

Chapter 8. Conclusion

The main contribution of this thesis is to introduce an artifact usage analysis technique into workflow design phase. To achieve this goal, this thesis presents a process model for describing a well-formed workflow with DCDC and introduces an artifact flow diagram to represent artifact usages and transmissions in a workflow. According to the artifact flow diagram, the artifact usage anomalies observed are described. Finally, the incremental algorithms are proposed to maintain the artifact flow diagram and detect artifact usage anomalies to help the editing of such a workflow. Their time complexities are also studied.

In the future, we plan to design the formal algorithms for transferring the behaviors of artifact transmissions in GDS and ICDC to DCDC, so that we can detect artifact usage anomalies in GDS and ICDC with our work. In addition, we also plan to implement the proposed model and algorithms on current workflow management systems, so that our research result can be tested in real-world applications.

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, Volume 30, Issue 5, pp. 399-422, PERGAMON, July 2005.
- [3] Li, H., Yang, Y., and Chen, T.Y.: 'Resource constraints analysis of workflow specifications', The Journal of Systems and Software, Volume 73, Number 2, pp. 271-285, Elsevier Science, October 2004.
- [4] Liu, C., Lin X., Orłowska, M.E., and Zhou X.: 'Confirmation: increasing resource availability for transactional workflows', Information Sciences, Volume 153, Issue 1, pp. 37-53, Elsevier Science Inc, July 2003.
- [5] Du, W., and Shan, M.C.: '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] Muehlen, M.Z.: 'Resource modeling in workflow applications', Workflow Management Conference, Münster, Germany, 1999.
- [7] Sadiq, S., Orłowska, M.E., Sadiq, W., and Foulger C.: 'Data flow and validation in workflow modelling', Proceedings of the fifteenth conference on Australasian database, Volume 27, pp. 207-214, Australian Computer Society, 2004.
- [8] Sun, S.X., Zhao, J.L., Nunamaker, J.F., Sheng, O.R.L.: 'Formulating the data flow perspective for business process management', Information Systems Research, Vol. 17, No. 4, December 2006, pp. 374-391.
- [9] Russell, N., ter Hofstede, A. H.M., Edmond, D, and van der Aalst, W.M.P. : 'Workflow Data Patterns', QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004, <http://www.workflowpatterns.com/patterns/data/index.php>.

- [10] Lei Gong and Gai-yang Wang, 'A method to verify the soundness of workflow control logic', Computer Supported Cooperative Work in Design, Volume 1, pp.284-388, May 2004.
- [11] Russell, N., ter Hofstede, A.H.M., Edmond, D., and van der Aalst, W.M.P.: 'Workflow data patterns', QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004.
- [12] van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., and Barros, A.P.: 'Advanced workflow patterns', 7th International Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, pp. 18-29. Springer-Verlag, Berlin, 2000.
- [13] Joonsoo Bae, Hyerim Bae, Suk-Ho Kang, and Yeongho Kim, 'Automatic Control of Workflow Processes Using ECA Rules', IEEE Transaction on Knowledge and Data Engineering, Volume 14, Number 8, pp.1010-1023, IEEE Computer Society, August 2004.
- [14] 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, Volume 70, Issue 1, pp.86-106, Elsevier Science Publishers, February 2005.
- [15] Hsu, C.-L.: 'Detecting the Artifact Anomalies in Business Process Specifications with a Formal Model' D.S. Thesis, National Chiao-Tung University, 2007.
- [16] WebSphere MQ, IBM, <http://www.ibm.com/software/integration/wmq/>, accessed August 2008.
- [17] Hsu, H.-J.: 'An Incremental Analysis for Resource Conflicts to Workflow Specifications' D.S. Thesis, National Chiao-Tung University, 2008.
- [18] Object Management Group. 2006. Business Process Modeling Notation (BPMN). <http://www.bpmn.org/>.
- [19] The Workflow Management Coalition: 'Terminology & glossary', Document Number

WFMC-TC-1011, February 1999.

- [20] Wang, F.-J., Hsu, C.-L., Hsu, H.-J.: ‘Analyzing Inaccurate Artifact Usages in a Workflow Schema’, COMPSAC (2) 2006: 109-114.
- [21] Hsu, H.-J.: ‘Using State Diagrams to Validate Artifact Specifications on Primitive Workflow Schema’ M.S. Thesis, National Chiao-Tung University, 2005.
- [22] The Workflow Management Coalition: ‘Terminology & glossary’, Document Number WFMC-TC-1011, February 1999.
- [23] Sadiq, W. and Orłowska, M.E.: ‘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.

