國科會計畫：NSC 98-2221-E-009-081-MY3

一商業流程規格中 Artifacts 異常使用偵測之研究

A Study to Detecting Artifact Anomalies in Workflow Specifications

摘要

一個結構良好的工作流程仍然可能由於錯誤的資料存取而發生執行期錯誤，在過去，由於結構衝突造成的資料存取錯誤已經被充分地研究，然而混合了時序與結構因素所造成的資料存取錯誤，也就是 artifact 異常使用，則仍然缺乏討論，在本計畫中，我們發展了一個可以針對時序結構化工作流程規格進行分析，找出隱藏於其中，由於時序與結構化因素同時作用而造成的資料存取錯誤，我們嶄新的分析方法，藉由將工作流程中的迴圈簡化可以達成快速的分析，幫助工作流程設計者除去規格中隱藏的資料存取錯誤，進一步減少工作流程在執行期可能發生的問題。

關鍵字：工作流程、時序結構化工作流程、工作流程規格分析、artifact 異常使用、資料存取錯誤

Abstract

A well-structured workflow may still fail or produce unanticipated run-time behavior because of abnormal data manipulation generated from twisted temporal and structural relationships between activities. These abnormal data manipulations, notated as artifact anomalies, should be analyzed and eliminated from workflow schemas before execution. The methodologies for eliminating artifact anomalies caused by structural factors have been studied in several previous works; however, the temporal issues are still seldom addressed in such analyses. In this project, we develop an approach discovering artifact anomalies caused by twisted temporal and structural relationships between activities in temporal structured workflow (TS workflow). With reasonable loop reduction, the methodology for acyclic workflows is adopted in our approach without loss of generality. Several novel techniques are developed for efficient analysis of complex temporal and structural relationships in TS workflow specifications. With our approach, workflow designers may eliminate artifact anomalies from workflow specifications, and prevent potential run-time problems resulted from abnormal data manipulation.

Keywords: *workflow, temporal structured workflow, analysis of workflow schema, artifact anomaly, abonormal data manipulation*

1. Introduction

A workflow is a set of tasks systematized to achieve certain business goals by completing the tasks in a particular order under automatic control (WfMC, 1999). Structural conflicts among tasks such as deadlocks might cause run-time errors, and should be eliminated. Analyses of workflow specifications are helpful in reducing such conflicts. By mapping workflow specifications into petri-nets, Adam's methodology can detect inconsistent dependencies among tasks to assure the safety of a workflow (Adam et al., 1998). Van der Aalst develops a petri-net based methodology to verify deadlocks, livelocks (infinite loops), and dead tasks in workflow specifications (van der Aalst and ter Hofstede, 2000; van der Aalst et al., 1999). Kiepuszewski et al. (2000) define the structured workflow model which is free from deadlock and multiple active instances of the same activity, i.e. the structured workflow is well-behaved. Kiepuszewski et al. claim that although structured workflow model is less expressive, most arbitrary well-behaved workflows can be transformed into a structured workflow, and structured workflow is a good tool for workflow analysis (Kiepuszewski et al, 2000).

However, a well-structured workflow may still fail or produce unanticipated run-time behavior because of abnormal data manipulation, the artifact anomalies. Detect artifact anomalies in workflows checks possible data misuse buried in workflow specifications. Various methodologies have been developed for detection of artifact anomalies generated from structural relationships between activities in a workflow (Sadiq et al., 2004; Wang et al., 2006; Hsu et al., 2007; Wang et al., 2009; Hsu et al, 2009). Sadiq et al. (2004) present seven basic data validation problems, redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data in structured workflow model. Hsu et al. define preliminary improper artifact usages anomalies, and introduce the analysis of such anomalies in design phase of a structured workflow (Wang et al., 2006; Hsu et al., 2007). Wang et al. (2009) introduce a behavior model to describe the data behavior in a workflow and refine the work accomplished by Hsu et al. (2007) through improving its efficiency. Hsu et al. (2009) raise the issues about analyzing artifact anomalies in workflows adopting message passing data models, and describe a formal description for such anomalies. Nevertheless, how temporal factors may affect the analysis of artifact anomalies is still seldom addressed. The methodology detecting artifact anomalies generated from twisted temporal and structural relationships between activities in workflows should be further discussed on the basis of the previous studies.

Besides, analysis of workflows may not be completed without considering temporal issues. Li et al. (2004a) indicate that analysis of temporal factors is essential for validation of the interval dependencies with temporal constraints in a workflow schema. Adam et al. (1998) consider timing constraints as the external conditions for structural correctness of a Petri-net based workflow model. Chen and Yang (2008) develop an approach for dynamic verification of fixed-time constraints in grid workflow syste. From a graph based workflow model, Eder et al.

(1999a; 1999b) develop a timed graph model to illustrate the working duration of activities among workflows with the corresponding earliest and latest finish time, and calculate the deadlines among internal activities to meet the overall temporal constraints on the basis of the model. Marjanovic (2000) build the timing model based on duration and instantiation space, and model the absolute and relative deadline constraints for dynamic verification.

Analysis of temporal relationships between processes in a workflow is another important issue. For example, two processes which are expected to be executed in parallel might never be activated at the same time because their active durations do not overlap. Zhuge et al. (2001) consider durations of activities for temporal checking in both design-time and run-time. On the basis of the work made by Zhuge et al. (2001), Li et al. (2004b) estimate the active intervals of activities, and develops an algorithm to detect and remove resource conflicts with respect to both temporal and structural issue. Besides, Li and Yang (2005) also analyze the resource and temporal constraints between distinct workflow instances dynamically. On the basis of pre-specified reference points in workflows, Li's methodology may adjust the temporal dependencies between the processes involved in some resource conflicts. Hsu et al. (2005) construct an incremental algorithm to notify the workflow designers about the resource conflicts generated from twisted temporal and structural relationships between activities. Nevertheless, there is still little discussion about how temporal factors may affect the analysis of artifact anomalies.

In this project, structured workflow modeled in (van der Aalst et al., 1999) is extended as temporal structured workflow (*TS workflow*) model with the temporal issues considered in the studies made by Li et al. (2004b) and Li and Yang (2005). Based on define-use-kill operations, a formal model describing artifact anomalies in TS workflow is established, and the algorithms detecting the artifact anomalies produced from twisted structural and temporal relationships between activities are developed. The rest parts of this report are organized as following. In section 2, TS workflow is modeled, and the methodology for analysis of structural and temporal relationships between processes in a TS workflow is depicted. In section 3, artifact operations and corresponding artifact anomalies are introduced, and the methodology detecting artifact anomalies in TS workflow is developed in section 4. In section 5, several case studies are discussed to illustrate the feasibility of our methodology. The related works are described and compared with our approach in section 6. Finally, the conclusion and future works are described in section 7.

2. Temporal Structured Workflow Model

2.1 Basic Elements

A workflow is composed of a *start process*, an *end process*, some *activity processes* and some *control processes*. The start (*ST*) process represents the entry point of a workflow, and the end (*END*) process indicates the termination point. An activity (*ACT*) process stands for a piece of work to be performed and describes one logical step within a workflow (WfMC, 1999).

A control process is a routing construct used to control the divergence and convergence of sequence flows. The control processes can be classified as *AND-split* (*AS*), *AND-join* (*AJ*), *XOR-split* (*XS*), and *XOR-join* (*XJ*). An AND-split process within a workflow splits a single sequence of control into two or more sequences to allow simultaneous execution of activities; on the contrary, an AND-join process merges multiple parallel executing sequences into a single common sequence of control (WfMC, 1999). An XOR-split process within a workflow is the point where a single sequence of control decides a branch to take from multiple alternative branches, and an XOR-join process converges multiple alternative branches in a workflow (WfMC, 1999).

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

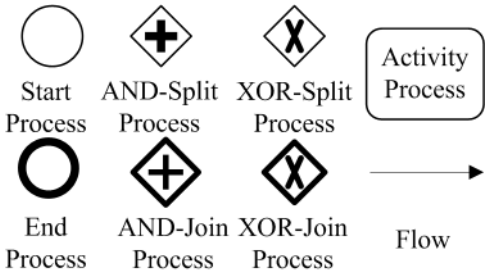

Figure 1 The Graphic Notations of the Basic Workflow Elements

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

Definition 1 (Workflow Model)
A workflow w, $w = (P_w, F_w, s, e)$. and
P_w represents the set of the processes in w, and
$\forall p \in P_w, p.type \in \{ACT, AS, AJ, XS, XJ, ST, END\}$
$F_w \subseteq P_w \times P_w$ represents the set of flows in w.

$\forall f \in F_w, f = (p, q)$ is the in-flow of process q and the out-flow of process p, and p is the source process of f, and q is the sink process of f.
$s \in P_w$ represents the start process of w, $s.type$ = ST, $\exists$ no in-flow to s.
$e \in P_w$ represents the end process of w, $e.type$ = END, $\exists$ no out-flow from e.

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

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

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

2.2 Structured Workflow

A structured workflow is a workflow that is syntactically restricted in a number of ways. Control processes are organized in pair, an XOR-split process is paired with an XOR-join process, and an AND-split process is paired with an AND-join process. A control block is composed of a pair of control processes and the processes placed in between the pair of control processes. According to the type of the control processes, the control blocks can be classified as parallel structures, decision structures, and structured loops as Figure 2 illustrates. Each process in a structured workflow has at least one path from the start process to it, and at least one path from it to the end process. Such restriction keeps a structured workflow well-behaved (Kiepuszewski et al., 2000), i.e. a structured workflow is free from deadlocks and multiple active instances. Most arbitrary well-behaved workflows can be transformed to be structured without loss of their contexts (Kiepuszewski et al., 2000). Figure 2 shows the building blocks of a structured workflow according to the basic elements and constraints described above.

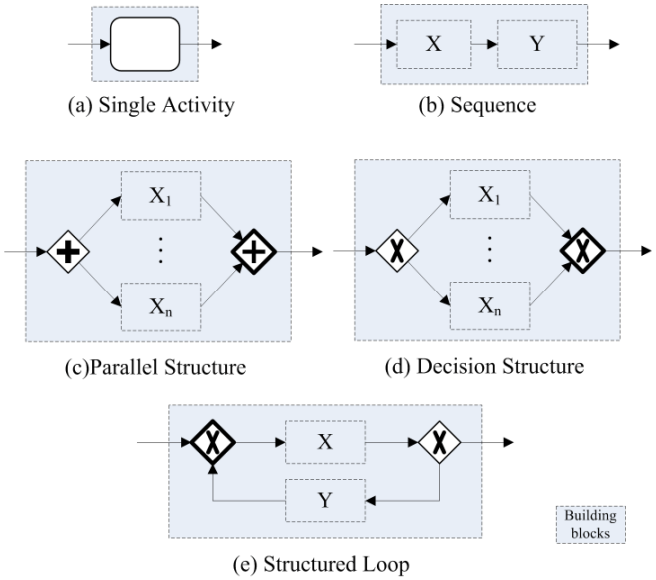

Figure 2 Building Blocks of a Structured Workflow

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

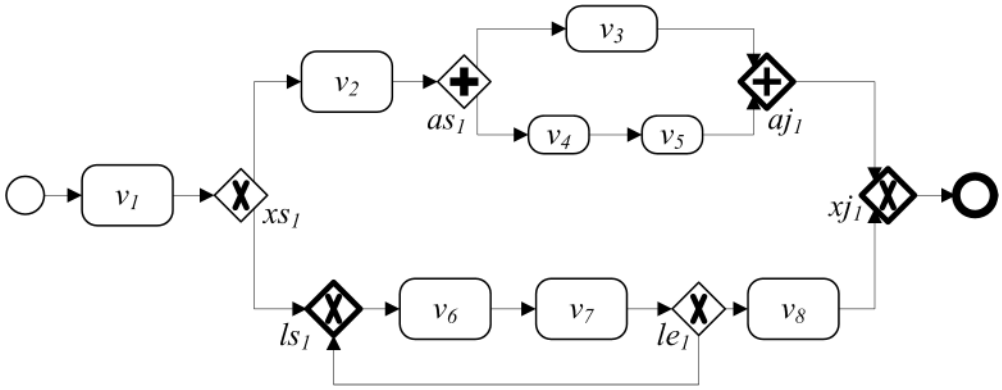

Figure 3 A Sample Structured Workflow

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

Definition 3 (Structural Relationships in a Structured Workflow)
For a structured workflow *w*,

Reachable: $P_w \times P_w \Rightarrow$ {true, false}

Reachable(*p*, *q*) holds if and only if there exists a path from *p* to *q*.

Parallel: $P_w \times P_w \Rightarrow$ {true, false}

Parallel(*p*, *q*) holds if and only if *p* and *q* reside in different branches of a parallel structure.

Exclusive: $P_w \times P_w \Rightarrow$ {true, false}

Exclusive(*p*, *q*) holds if and only if *p* and *q* reside in different branches of a decision structure.

2.3 Temporal Structured Workflow

A timed workflow is modeled by describing the maximal and minimum working durations for each activity (Zhuge et al., 2001). In this report, a timed and structured workflow named as *Temporal Structured Workflow* (*TS workflow*) is formally modeled as following:

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

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

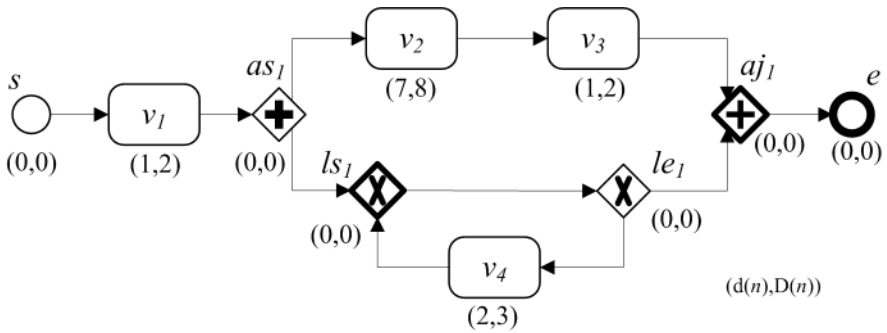

Figure 4 A Sample TS workflow

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

2.4.1 Loop Reduction

The structural and temporal relationships between processes are the bases of any further analysis of a TS workflow. Hsu et al. (2007) give several approaches to reveal the structural and temporal relationships between processes in acyclic structured and timed workflows. Hsu et al. (2007) and Wang et al. (2009) claim that in a structured workflow, all the possible state variations of the artifact operated in loops with more than two iterations are the same as those with exact two iterations. Therefore, they reduce a structured loop into a decision structure with three branches representing for no iteration, a single iteration, and two iterations for the analysis of artifact anomalies with better efficiency. In this project, we adopt an approach similar to the methods developed by Sadiq et al. (2004) and Wang et al. (2006) to reduce the structured loops in a TS workflow as decision structures to retrieve structural and temporal information in a TS workflow as in Hsu et al. (2007) did.

In a TS workflow, the number of iterations of a loop affects the active timing of processes succeeding to the loop. The loop reduction introduced in the study made by Sadiq et al. (2004) and Wang et al. (2006) may bring inaccuracy to the analysis of temporal factors, and is therefore not feasible for TS workflow. Leong and Si (2009) consider the worst case scenarios for loops in a workflow and develops a methodology to detect whether the workflow possibly exceeds its deadline during run-time. Here, we combine Leong and Si's (2009) concept and the methods developed by Sadiq et al. (2004) and Wang et al. (2006) to describe a refined loop

reduction method for the analysis of TS workflow.

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

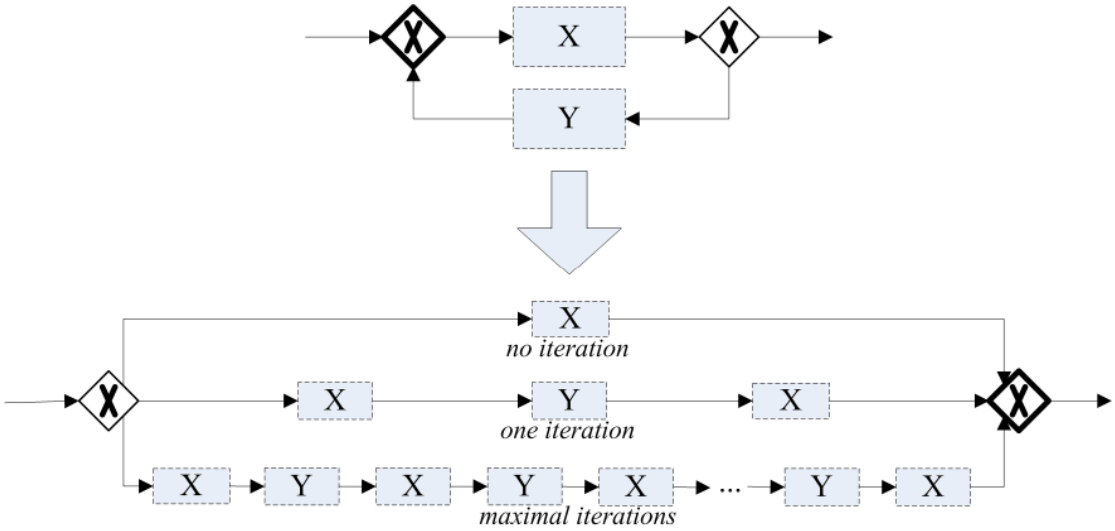

Figure 5 Refined Loop Reduction for TS Workflow Model

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

2.4.2 Analysis of Structural Relationships between Processes in LRTS workflow

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

Lemma 1

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

Proof:

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

structure illustrated in Figure 2.

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

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

Hsu et al. (2009) use a data structure, ABStack, to record the structural information of processes, and achieve an efficient analysis of the structural relationships between processes in an acyclic structured workflow. In this project, the similar approach is adopted. All the flows in an LRTS workflow are tagged with a branch mark. The branch mark is a natural number ID for each out-flow split from an XOR/AND split process, and is -1 for any other flow in the LRTS workflow. The branch mark in this project is formally defined as following.

Definition 5 (Branch Mark)
For an LRTS workflow w,

BM_w: $F_w \Rightarrow$ INTEGER

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

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

$BM_w((p, q)) \neq BM_w((p, q'))$

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

Definition 6 (ABStack)

$\forall p \in P_w$, *p.abstack* represents the ABStack corresponding to *p*.

A structural item, *stitem* = (*sp*, *bm*), is included in *p.abstack* if and only if

(1) $sp \in P_w$, $sp.type \in \{AS, XS\}$, and $\exists$ a path $<sp, \ldots, p, \ldots, jn>$ in *w* where *jn* is the corresponding join process of *sp*.

(2) *bm* = BM((*sp*, *p'*)) where *p'* == *p* or Reachable(*p'*, *p*) == true.

p.abstack == « » if and only if *p* resides in no decision/parallel structure.

p.abstack == «(sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)» exists if and only if a path $<sp_k, \ldots, sp_2, \ldots, sp_1, \ldots, p, \ldots, jn_1, \ldots, jn_2, \ldots, jn_k>$ exists.

To calculate ABStacks of the processes in an LRTS workflow, *push* and *pop* functions associated with ABStack are defined as following:

Let an ABStack *abs* == «(sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)»

Push(*abs*, (*sp*, *bm*)) returns a new ABStack *abs'*, where

abs' == «(*sp*, *bm*), (sp_1, bm_1), (sp_2, bm_2), …, (sp_k, bm_k)»

Pop(*abs*) returns a new ABStack *abs'*, where

abs' == «(sp_2, bm_2), …, (sp_k, bm_k)»

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

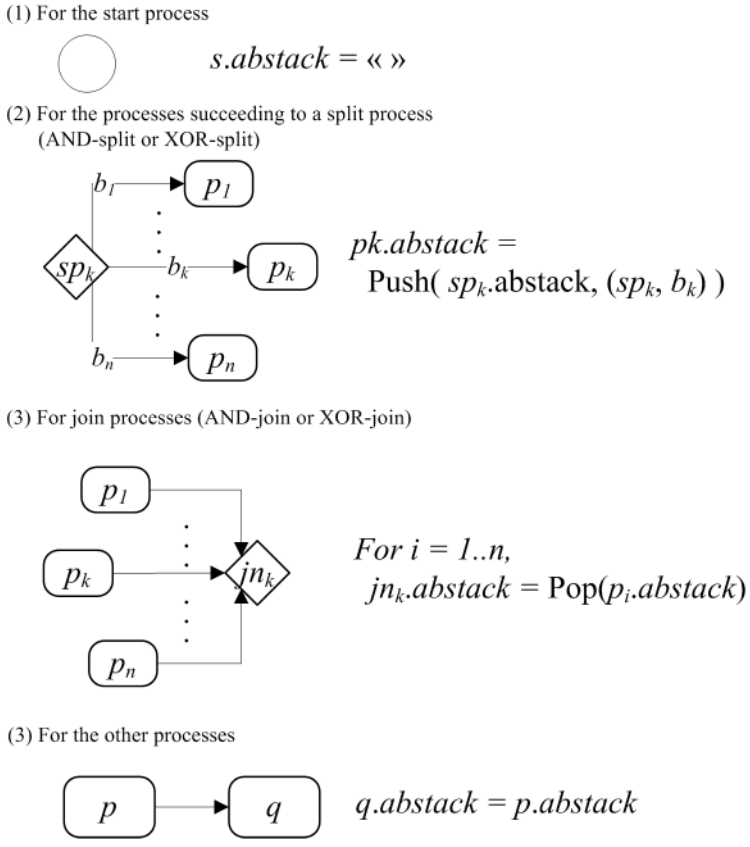

Figure 6 Calculation of ABStacks for Processes in an LRTS Workflow

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

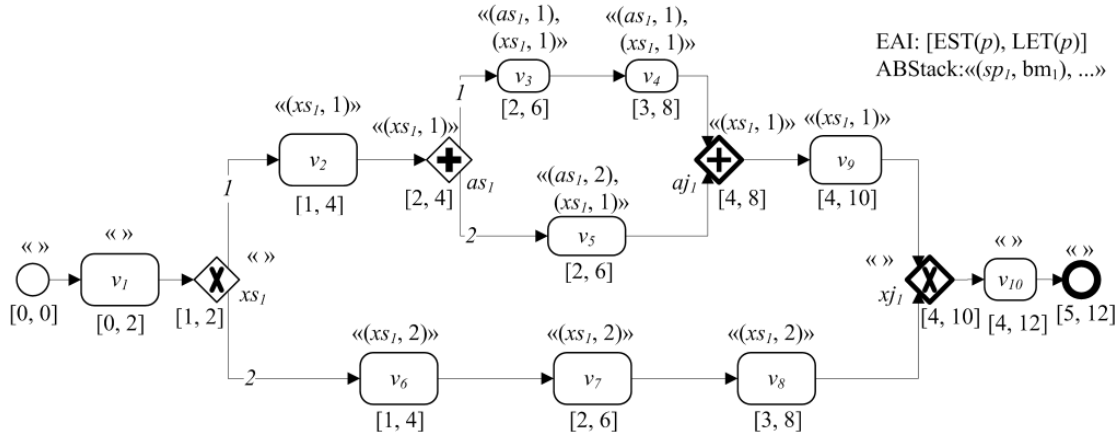

Figure 7 A Sample TS workflow with ABStacks and EAIs

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

Lemma 2

For an LRTS workflow w, and $p, q \in P_w$

(1) Parallel(p, q) holds if and only if

$\exists (sp, bm) \in p.abstack$ and $(sp, bm') \in q.abstack$ where $sp \in P_w$, $sp.type$ == AS and $bm \neq bm'$.

(2) Exclusive(p, q) holds if and only if

$\exists (sp, bm) \in p.abstack$ and $(sp, bm') \in q.abstack$ where $sp \in P_w$, $sp.type$ == XS and $bm \neq bm'$.

Proof:

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

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

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

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

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

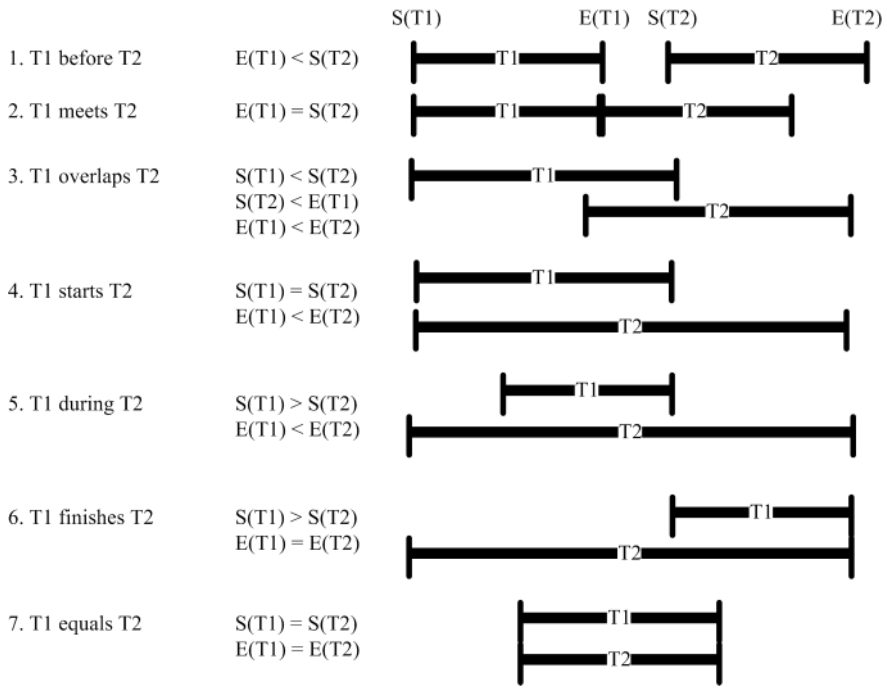

Figure 8 The Temporal Relationships between Time Intervals (Allen, 1983)

A time interval is duration of a segment of time. Allen (1983) defines seven reasoning relationships between time intervals. Figure 8 illustrates the temporal relationships adopted in this project on the basis of Allen's definition, and Definition 7 describes the formal definition of time intervals and the temporal relationships between time intervals adopted in this project.

Definition 7 (Time Intervals)

A time interval $ti = [S(ti), E(ti)]$ indicates a duration from the time point $S(ti)$ to $E(ti)$, $E(ti) \geq S(ti)$.

A time point *tp* can be represented as a time interval [*tp*, *tp*], and *ctime* is the time point indicating the current time.

For any two time intervals ti_1 and ti_2,

ti_1 is *before* ti_2, notated as $ti_1 \prec_{TI} ti_2$, if and only if $E(ti_1) \leq S(ti_2)$.

ti_1 is after ti_2, notated as $ti_1 \succ_{TI} ti_2$, if and only if ti_2 is before ti_1.

ti_1 overlaps ti_2, notated as $ti_1 \approx_{TI} ti_2$, if and only if

$MIN(\{E(ti_1), E(ti_2)\}) - MAX(\{S(ti_1), S(ti_2)\}) > 0$

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

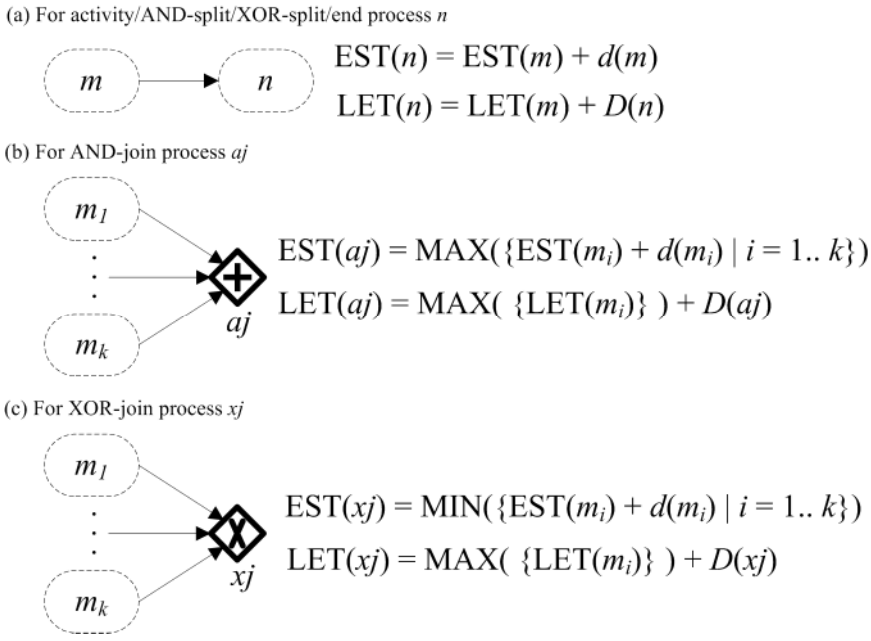

Figure 9 Calculation of EAIs in an LRTS Workflow (Li et al., 2004b; Hsu and Wang, 2008)

Minimum and maximum working durations are used to estimate the active duration of a process corresponding to the start of workflow (Li et al., 2004b; Hsu and Wang, 2008). The *Estimated Active Interval* (EAI) of a process is a time interval indicating when the process can be initialized and when it should be terminated. In this project, the Estimated Active Interval of a process *p*, notated as EAI(*p*) is defined as following:

Definition 8 (Estimated Active Interval)

For a TS workflow *w* and a process $p \in P_w$,

EAI(*p*) = [EST(*p*), LET(*p*)], and corresponding to when *w* starts:

EST(p) indicates the earliest time that p can be initialized.
LET(p) indicates the latest time that p must terminate.

With the assumption that the EST and LET of the start process of a TS workflow are zero, the methodologies made by Li et al. (2004b) and Hsu and Wang (2008) can be adopted to calculate the EAIs of processes in an LRTS workflow as Figure 9 illustrates.

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

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

Proof:

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

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

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

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

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

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

parallel, exclusive, or reachable from one to the other. Therefore, Lemma 4 can be re-stated as Lemma 5 that if two activity processes in an LRTS workflow are reachable from one to the other, the activity processes with larger LET is reachable from the one with smaller LET.

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

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

Definition 9 (Structural and Temporal Relationships in LRTS workflow)
For an LRTS workflow w,
Concurrent: $P_w \times P_w \Rightarrow$ {true, false}
Concurrent(p, q) == true if and only if
(Parallel(p, q) $\wedge$ EAI(p) $\approx_{TI}$ EAI(q)) == true.
Before: $P_w \times P_w \Rightarrow$ {true, false}
Before(p, q) == true if and only if
(Reachable(p, q) $\vee$ (Parallel(p, q) $\wedge$ EAI(p) $\prec_{TI}$ EAI(q))) == true.
After: $P_w \times P_w \Rightarrow$ {true, false}
After(p, q) == true if and only if Before(q, p) == true.

3. Artifact Anomalies in TS workflow

3.1 Artifact Operations

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

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

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

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

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

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

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

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

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

usages only.

As the example mentioned above, an RDK or an RDS introduces state ambiguous (AB) to the artifact. Besides, an artifact transits to state UD after an RKS or an RUK, and state DR after an RDU. Since the artifact state after a usage varies based on the input state of the artifact, the artifact state after an RUS requires additional consideration in merging the input states of the usages involved in the RUS. The artifact and its related operations are modeled in Definition 10, and Figure 10 illustrates how artifact transits its state with different artifact operations.

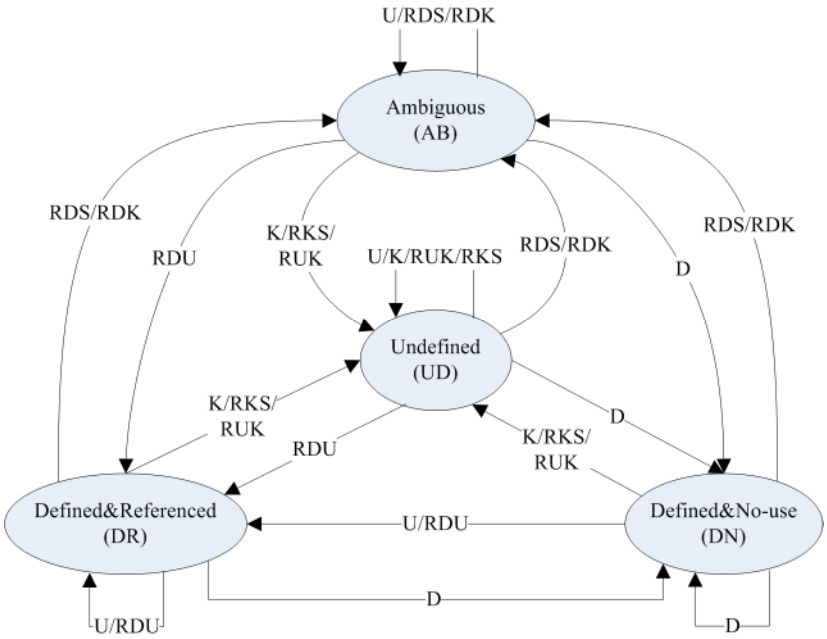

Figure 10 The Artifact State Transit Diagram

Definition 10 (Artifact Model in TS workflow)
For an LRTS workflow *w*,
The set of all the artifacts operated in *w* is notated as A_w.
$\forall a \in A_w$, $a.\text{state} \in \{\text{UD, DN, DR, AB}\}$.
The artifact operation made by processes in *w* is described as a relationship AOP:
$\text{AOP: } \{p \mid p \in P_w, p.type == \text{ACT}\} \times A_w \Rightarrow \{\text{Nop, Def, Use, Kill}\}$
$\{p \mid p \in P_w, p.type \neq \text{ACT}\} \times A_w \Rightarrow \{\text{Nop}\}$

3.2 Artifact Anomalies

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

(1) Useless Definition:

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

(2) Undefined Usage:

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

(3) Null Kill:

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

(4) Ambiguous Usage:

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

4. The Methodology to Analyzing Artifact Anomalies in LRTS workflow

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

Our methodology is divided into three parts. In section 4.1, we first describe how to traverse an LRTS workflow to collect the structural and temporal relationships between the processes and the artifact operations. In section 4.2, according to the structural and temporal relationships gathered in the first part, the methodology analyzing relationships between the artifact operations are described. Finally, on the basis of the analysis made in the second part, the methodology detecting artifact anomalies in an LRTS workflow is concluded in section 4.3

4.1 Gathering Structural, Temporal, Artifact Information in LRTS workflow

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

Definition 11 (Artifact Operation List)
For an LRTS workflow *w* and $\forall a \in A_w$,
$AOPL_a$ is the list of artifact operations working on *a*,
$\forall op \in AOPL_a$, $op = (p, a, est, let, type)$,
$p \in P_w$, $p.type \in \{ACT, END\}$,
$est = EST(p)$, and $let = LET(p)$, and
$type = AOP(p, a)$.

With the definition, the algorithm gathering structural, temporal, artifact information in LRTS workflow is described as following:

Algorithm 1 Information Gathering - *IG*
Input: an LRTS workflow *w*
Pre-Condition: *w.s.mark* == true, EAI(*w.s*) == [0, 0], *w.s.abstack* == « »
$\forall p \in P_w \backslash \{w.s\}$, *p.mark* == false
IG {
01: Queue *tq*;
02: $\forall (w.s, n) \in F_w$,
03: *tq*.enqueue(*n*);

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

In Algorithm 1, a traverse queue is introduced to hold the order of traversal of processes in an LRTS workflow. Starting from the start process, the processes in a TS workflow is traversed along with flows. The EAIs, ABStacks, and artifact operations lists are calculated and collected correspondingly. To prevent unnecessary redundancy, a Boolean flag *mark* is given to each process. Besides the start process, the mark of each process in *w* is initialized as false, and when a process is calculated, its mark turned to true. Since a join process may have several in-flows, a Boolean expression is checked at line 6 to assure that the join process is calculated only when each of its source process is calculated. Algorithm 1 records the artifact operation made by each activity process at line 12 and the "no operation" made by the end process in $AOPL_a$ at line 14 for further analysis of the definitions remaining useless at the end of *w*. At line 19, artifact operation list corresponding to each artifact is sorted by LET.

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

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

Definition 12 (Relationships between Artifact Operations)
For an LRTS workflow w and $\forall a \in A_w$,
$\forall op_i, op_j \in AOPL_a$,
Before(op_i, op_j) == true if and only if Before($op_i.p$, $op_j.p$) == true.
After(op_i, op_j) == true if and only if After($op_i.p$, $op_j.p$) == true.
Concurrent(op_i, op_j) == true if and only if Concurrent($op_i.p$, $op_j.p$) == true.
Exclusive(op_i, op_j) == true if and only if Exclusive($op_i.p$, $op_j.p$) == true.

According to Definition 9, Definition 12, Lemma 1, and Lemma 3, the following lemma holds.

Lemma 6
For two operation op and $op' \in AOPL_a$,
(1) If Before(op, op'), $op.let < op'.let$
(2) If $op.let < op'.let$, After(op, op') == false

Algorithm 2 is introduced to collect operations concurrent to each operation in an $AOPL_a$. To facilitate our discussion, it is assumed that each $AOPL_a$ is indexed, and $op_i \in AOPL_a$ indicates the ith operation in the list. Because $AOPL_a$ is sorted by LETs at the last part of Algorithm 1, for $0 < i < j$, $LET(op_i.p) \leq LET(op_j.p)$. Besides, for any op_i in $AOPL_a$, ConcD_op_i, is the set collecting the definitions concurrent to op_i, and ConcK_op_i collects kills correspondingly. These sets are defined as following:

Definition 13 (Records of Relationships between Artifact Operations)
For an LRTS workflow w and $\forall a \in A_w$,
$\forall op_i \in AOPL_a$,
ConcD_op_i =
{op | $op \in AOPL_a$, $op.type$ == Def, Concurrent($op_i.p$, $op.p$) == true}
ConcK_op_i =
{op | $op \in AOPL_a$, $op.type$ == Kill, Concurrent($op_i.p$, $op.p$) == true}

With the records, Algorithm 2 is constructed as following:

Algorithm 2 Identifying Concurrent Operations - *ICO*
Input: an artifact a
Pre-Condition: $a \in A_w$, and w is manipulated by Algorithm 1
ICO {
01: for(i = 1 to $|AOPL_a|$) {
02: if($op_i.p.abstack \neq$ « » {
03: $j = i + 1$;
04: while($j \leq |AOPL_a|$) {

```
05:          if ( Concurrent(op_i.p, op_j.p) ){
06:              if( op_i.type == Def ) add op_i to ConcD_op_j;
07:              else if( op_i.type == Kill ) add op_i to ConcK_op_j;
08:              if( op_j.type == Def ) add op_j to ConcD_op_i;
09:              else if( op_j.type == Kill ) add op_j to ConcK_op_i;
10:          }
11:          j++;
12:       }
13:    }
14: }
}
```

Because $AOPL_a$ is sorted by LETs in Algorithm 1, Algorithm 2 checks each operation in $AOPL_a$ in order. For any $op_i \in AOPL_a$, Algorithm 2 first checks if it resides in some parallel or decision structure(s) at line 2. If not, op_i can not be concurrent or exclusive to any other operations. From line 3 to line 14, the algorithm checks the operations which are succeeding to op_i in $AOPL_a$ in order. If the operation under checking is concurrent to op_i, the records for both operations are updated.

For an artifact operation, the operations directly before it generate/carry its input artifact state, and might make it an artifact anomaly. For example, when a kill directly before a usage, i.e. no other operations between them, the usage is an undefined usage. We define the relationship *directly before* between artifact operations on the basis of Definition 12 as following:

Definition 14 (Directly Before)
For an LRTS workflow w and $\forall a \in A_w$,
$\forall op, op' \in AOPL_a$, op is *directly before* op' if and only if op is before op', and $\exists$ no $op'' \in AOPL_a$ that op'' is after op and before op'.
$\forall op \in AOPL_a$, $DB4_{op} = \{ op' \mid op' \in AOPL_a$ and op' is directly before $op\}$

According to Definition 9, Definition 12, and Lemma 5, for any two artifact operations effective on artifact a, op and op', if op' is before op, $op'.let < op.let$. Therefore, the operations directly before op can be identified by analyzing the sub-list of $AOPL_a$ where the operations in the sub-list are all with smaller index in $AOPL_a$ than op. The sub-list is defined as following:

Definition 15 (The List of Operations with Smaller LET than Operation op)
$\forall op \in AOPL_a$,
$OPL_{op} = \{op' \mid op' \in AOPL_a$, and $op'.let < op.let\}$
Similar to $AOPL_a$, OPL_{op} is sorted and indexed with LETs

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

Algorithm 3 Collecting Directly Before Operations – *CDBO*
Input: an artifact operation *op*,
Pre-Condition: $AOPL_a$ has been produced by Algorithm 1, $op \in AOPL_a$
Operation Set CDBO {
01: $DB4_{op} = \emptyset$;
02: for(i = $|OPL_{op}|$ to 1) {
03: if ((Concurrent(op, op_i) || Exclusive(op, op_i)) == false) {
04: if ($DB4_{op} == \emptyset$) add op_i to $DB4_{op}$;
05: else if($\exists$ no $op' \in$ $DB4_{op}$ that Before(op_i, op') == true)
06: add op_i to $DB4_{op}$;
07: }
08: }
09: return $DB4_{op}$;
}

For the input artifact operation *op*, Algorithm 3 calculates $DB4_{op}$ from its corresponding artifact operation list. Algorithm 3 checks the operations in OPL_{op} with reverse order. According to Lemma 1 and Definition 9, the processes in an LRTS workflow are either before, after, concurrent or exclusive to each other, and so are the operations. The operations concurrent or exclusive to *op* are excluded at line 3. With Lemma 4, the first operation found passing the checking at line 3 is directly before *op*. According to Definition 14, if *op'* and *op"* are both directly before *op*, *op'* can not be before *op"* and vice versa. Therefore, the algorithm continues gathering the other directly before operations with the statement at line 6 after the first one is found.

To show Algorithm 3 is correct, the following lemma is depicted and proved.

Lemma 7
For any artifact operation *op* and *op'*, *op'* is directly before *op* if and only if $op' \in CDBO(op)$

Proof:

We first show the if-part is correct. B.W.O.C, it is assumed that $op' \in CDBO(op)$, but is not directly before *op*. According to the algorithm, the result set of Algorithm 3 is a sub-set of OPL_{op}. Therefore, $op' \in OPL_{op}$, $op'.let < op.let$, and *op'* can not be after *op* on the basis of Lemma 6. Besides, *op'* must pass the checking at line 3, *op'* is not concurrent or exclusive to *op*. Based on Lemma 4 and Definition 12, *op'* is before *op*. Since *op'* is not directly before *op*, according to Definition 14, there must exist another operation *op"* which is after *op'* and

before *op*. Because *op'* $\in$ CDBO(*op*), *op'* must be collected in the result set at line 4 or line 6 in Algorithm 3. Since *op"* is after *op'*, *op'.let* < *op".let*. *op"* has a larger index than *op'* in OPL_{op}, and is touched by Algorithm 3 earlier than *op'* does. Therefore, either *op"* is directly before *op* or not, *op'* can not be collected in the result set at line 4 or line 6. *op'* $\notin$ CDBO(*op*) which is a contradiction, and the if-part of Lemma 7 is shown correct.

As for the only-if-part, B.W.O.C, we assume that *op'* is directly before *op* and *op'* $\notin$ CDBO(*op*). The assumption indicates that *op'* is before *op*. According to Lemma 6, *op'.let* < *op.let*, and thus *op'* belongs to OPL_{op}. *op'* also passes the checking at line 3 based on Lemma 4 and Definition 12. If the result set is empty when Algorithm 3 touches *op'*, *op'* is inserted into the result set at line 4 because *op'* is before *op*. Otherwise, *op'* is added into the result set at line 6 because *op'* is directly before *op* and there exist no other operations after *op'* in OPL_{op}. Therefore, *op'* $\in$ CDBO(*op*) which is a contradiction, and the only-if-part of Lemma 7 is shown correct. With the proofs of the both direction, Lemma 7 is proved. □

For an artifact operation *op*, multiple operations directly before it might exist. According to Definition 14, the operations are not before or after each other. On the basis of Lemma 1 and Definition 9, the operations are mutually concurrent or exclusive, and are possibly organized as the following cases:

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

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

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

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

The operations in case (1) compose a racing operation. In case (2), each operation is considered separately during analysis because only one of the operations is executed during run-time. Case (3) and (4) happen when the operations are made by processes reside in nestedly organized decision and parallel structures. Since only one of the branches in a decision structure is taken during run-time, the operations reside in different branches of a decision structure are separately analyzed with the operations concurrent to them. Figure 11 illustrates two partial LRTS workflow schemas as the examples of case (3) and (4).

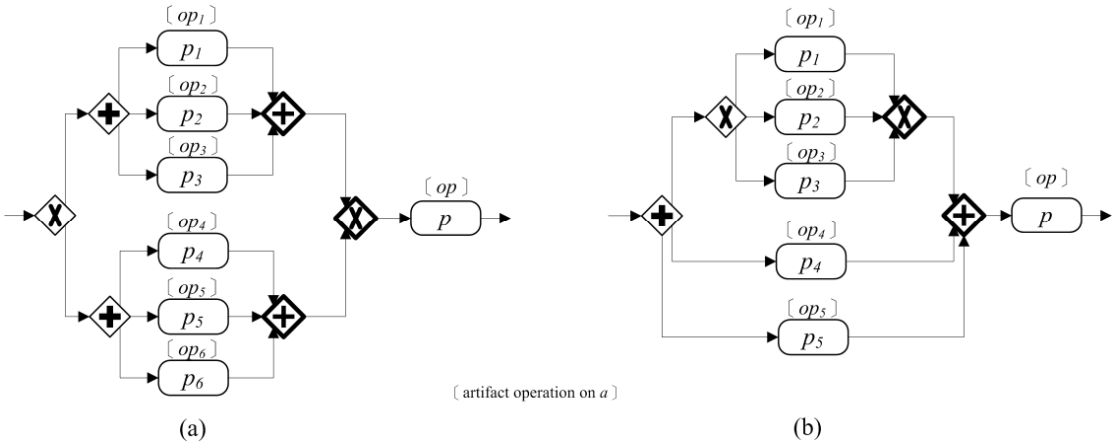

Figure 11 Examples for Nestedly Organized Decision and Parallel Structures

In Figure 11, we assume that the EAIs of the activity processes with footnotes are all overlapped, and all the operations made by them are thus directly before *op*. Figure 11(a) illustrates an example of case (3) mentioned above. In Figure 11(a), the operations directly before *op* can be divided into two distinct groups {op_1, op_2, op_3} and {op_4, op_5, op_6}. The operations are concurrent to the ones within the same group and are exclusive to the ones belonging to different groups. Figure 11(b) illustrates an example of the case (4). op_1, op_2, and op_3 are mutually exclusive and should be separately considered when analysis. However, each of them is concurrent to op_4 and op_5. Therefore, the operations are organized with three groups, {op_1, op_4, op_5}, {op_2, op_4, op_5}, and {op_3, op_4, op_5}. The operations in the same group are concurrent to each other, and the exclusive operations are distributed among different groups.

Definition 16 (Set of Operation Sets derived from $DB4_{op}$)
$\forall\, op \in AOPL_a$,
$DB4OPS_{op}$ = {OPS | $OPS \subseteq DB4_{op}$, $\forall\, op', op'' \in OPS$, Concurrent($op'$, op'') == true, and $\forall\, op^3 \in DB4_{op} \backslash OPS$, $\exists\, op^4 \in OPS$ that Exclusive(op^3, op^4) == true }

Each of the groups, the operation sets, in which all the operations are mutually concurrent represents an execution case during run-time. With Definition 14, the set of the operation sets derived from $DB4_{op}$ is defined as above.

However, to retrieve all such operation sets from $DB4_{op}$ is equivalent to solve the well-known NP-hard problem "Maximal Clique Enumeration Problem" (Pardalos and Xue, 1994). Although many studies and efficient algorithms such as the approaches made by Tsukiyama et al. (1977) and Makino and Uno (2004) has been developed for this problem, to discuss the solution for maximal clique enumeration problem is beyond the scope of this project. To illustrate our methodology, we describe a polynomial algorithm to manufacture $DB4OPS_{op}$ satisfying the cases (1), (2), (3) completely and case (4) partially from $DB4_{op}$. The algorithm is described as following.

Algorithm 4 Collecting Directly Before Operation Sets - *CDBOPS*
Input: an operation *op*,
Pre-Condition: $DB4_{op}$ has been calculated by Algorithm 3
Set of Operation Sets CDBOPS {
01: $DB4OPS_{op} = \emptyset$;
02: duplicate $DB4_{op}$ to BaseSet;
03: while(BaseSet $\neq \emptyset$) {
04: CurrentOPS = Ø;
05: choose and remove arbitrary operation *op'* from BaseSet;
06: duplicate $DB4_{op} \backslash \{op'\}$ to CountSet;
07: add *op'* to CurrentOPS;
08: while(CountSet $\neq \emptyset$) {
09: choose and remove arbitrary *op"* from CountSet;
10: if($\forall op^3 \in$ CurrentOPS, Concurrent(*op"*, op^3) == true) {
11: add *op"* to CurrentOPS;
12: remove *op"* from BaseSet;
13: }
14: }
15: add CurrentOPS to ResultSet;
16: }
17: return $DB4OPS_{op}$;
}

First, the algorithm duplicates $DB4_{op}$ to BaseSet at line 2. The codes from line 3 to 16 form a loop. In the loop, an operation *op'* is arbitrarily chosen from BaseSet, and all the operations concurrent to *op'* and each other are gathered and put into CurrentOPS. CurrentOPS is added to the result set as one of the operation sets found by the algorithm at the end of the loop. The operations in CurrentOPS are removed from BaseSet, and the next loop starts if there is still operation remaining in BaseSet. Because any operations in $DB4_{op}$ are mutually concurrent or exclusive, the operation chosen in the next loop is exclusive to at least one of the operations gathered in this loop. Besides, the algorithm starts collecting an operation sets from different operations every loop, and thus none of the operation sets collected in Algorithm 4 are identical. After each operation in BaseSet is distributed into some operation set, the algorithm returns the calculated $DB4OPS_{op}$ at line 17.

To depict the correctness and the effectiveness of Algorithm 4, we show that the following lemmas hold.

Lemma 8
The result set returned by Algorithm 4 follows Definition 16.

Proof:

Let OPS be one of the operation set collected in CDBOPS(*op*). According to the pre-condition of Algorithm 4, $DB4_{op}$ has been calculated by Algorithm 3, and according to Lemma 1, Definition 9, and Definition 14, the operations in $DB4_{op}$ are either mutually concurrent or exclusive. From line 7 and line 11 of Algorithm 4, we know that all the operations collected in OPS are mutually concurrent. The operations gathered in OPS are removed from BaseSet at line 12. Therefore, for any operation remaining in BaseSet, there exists at least one operation exclusive to it in OPS. Because BaseSet is duplicated from DB4 at line 2, OPS follows Definition 16, and Lemma 8 is thus shown correct. □

Before Algorithm 4 is introduced, four possible cases of the set of operation sets derived from $DB4_{op}$ are described, and we claim the capability of Algorithm 4 based on the cases. Here, we show the claim holds with the following lemma.

Lemma 9
Algorithm 4 is able to find the operation sets for case (1), (2), and (3) completely, and for case (4) partially.

Proof:

The cases are separately discussed as following:

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

In this case, the algorithm collects all the operations in the first loop of the algorithm. Only one operation set is included in the result set of Algorithm 4.

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

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

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

In this case, $DB4_{op}$ can be divided into several distinct operation sets following Definition 16. However, the operations included in different sets are all mutually exclusive. According to Lemma 8, the operation sets collected by Algorithm 4 follow Definition 16. On the basis of the algorithm, each operation in $DB4_{op}$ is collected into some operation set in $DB4OPS_{op}$. Therefore, all the operation sets in this case can be found by Algorithm 4.

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

In Algorithm 4, at least one operation is removed from BaseSet in the loop

starting from line 3, and therefore the algorithm derives at most N operation sets from $DB4_{op}$. For case (4), the number of operation sets identified by Algorithm 4 is less than N, but the number of operation sets in this case might exceed N. The operation sets in case (4) follow Definition 16, and so is Algorithm 4. Since the number of operation sets in case (4) might exceed the maximal capability of Algorithm 4. Obviously, Algorithm 4 identifies the operation sets for case (4) only partially. □

4.3 Detecting Blank Branch

Besides the cases described above, analysis of blank branches, i.e. the branches in a decision structure where no process residing in the branch has operations effective on the same artifact, is still ignored. Figure 12 illustrates parts of an LRTS workflow that the definitions made by v_1 and v_2 are directly before the usage made by v_4. The definitions should be considered separately during analysis because they are exclusively executed during run-time. However, if the third branch is taken during execution, the usage made by v_4 is undefined because the definition of a is killed by v_0, and no further definition is made by activity processes on the third branch. The third branch is a blank branch which generates a blind spot in our methodology.

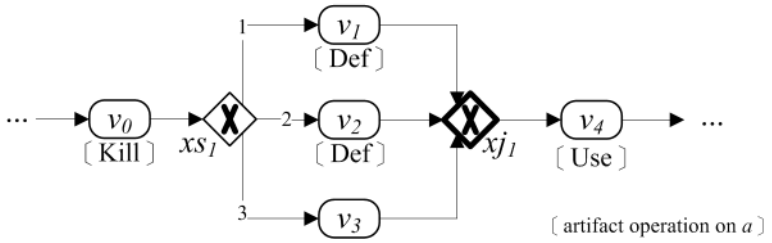

Figure 12 An Example of a Blank Branch

For any operation op, to eliminate the effect brought by blank branches when calculating its input states, all the operations reside in the decision structure with blank branches should be removed from OPL_{op}, and $DB4_{op}$ can then be recalculated for analysis. Algorithm 5 detects blank branches from the directly before operations of the input operation.

Algorithm 5 Detecting Blank Branch - *DBB*
Input: an operation *op*,
Pre-Condition: $DB4_{op}$ has been calculated by Algorithm 3
Branch Set DBB {
01: XSSet = Ø;
02: AllBranch = Ø;
03: OpBranch = Ø;
04: $\forall op' \in DB4_{op}$ {
05: $\forall si \in (op'.p.abstack \backslash op.p.abstack)$ where $si.p.type$ == XS {

```
06:        if( si.p ∉ XSSet ) {
07:            ∀ out-flow of si.p, f, add ( si.p, BM(f) ) to AllBranch;
08:            add si.p to XSSet;
09:        }
10:        add si to OpBranch;
11:    }
12: }
13: BlankBranch = AllBranch\OpBranch
14: return BlankBranch;
}
```

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

In Algorithm 5, all the branches of the decision structures with the operations directly before *op* are collected in AllBranch, and the individual branches resided by the operations are recorded in OpBranch. If all the branches collected in AllBranch are resided by the operations directly before *op*, no blank branch exists. Otherwise, the differences between AllBranch and OpBranch are the branches without operations effective on *op.a*, i.e. the blank branches.

4.4 Identifying Artifact Anomalies in an LRTS workflow

In this section, the algorithm integrating all the information gathered above to identify the artifact anomalies in an LRTS workflow is introduced. An operation transits the state of artifacts as Figure 10 illustrates, and artifact anomalies are produced when operations effective on artifacts with inappropriate state. For an artifact operation *op*, the artifact state produced by *op*, i.e. *op*'s output state, is recorded in *op*.OutState, and its input state is calculated from the output states of the operation(s) directly before it. Since only one of the mutually exclusive input operations is executed during run-time, the input states from these operations are discussed separately, and an operation might thus produce multiple output states accordingly. States of artifacts are recorded as state items, and are modeled as following.

Definition 17 (Records of Artifact States)
∀ state item *stItem*, *stItem* = (*st*, SRC),
stItem.*st* represents the output artifact state of *op*.*a*, and
stItem.SRC indicates the source operations producing the state.

With the definition above, Algorithm 6 describes the methodology to calculate the input state for each operation.

Algorithm 6 Gathering Input States of an Operation - *GIS*
Input: an LRTS workflow *w*,
an operation *op*
Pre-Condition: $DB4OPS_{op}$ has been calculated by Algorithm 4
Set of State Items GIS {
01: InStates = Ø;
02: if($DB4OPS_{op}$ == Ø)
03: add (UD, {*w*.*s*}}) to InStates;
04: else ∀ OPS ∈ $DB4OPS_{op}$ {
05: if(OPS is an RDS/RDK)
06: add (AB, {*op'* | *op'* ∈ OPS, *op*.*type* ∈ {Def, Kill}}) to InStates;
07: else if (OPS is an RDU)
08: add (DR, { *op'* | *op'* ∈ OPS, *op*.*type* == Def}) to InStates;
09: else if (OPS is an RKS/RKU)
10: add (UD, { *op'* | *op'* ∈ OPS, *op*.*type* == Kill}) to InStates;
11: else if (OPS is an RUS) {
12: if (∀ *op'* ∈ OPS, ∃ *si* ∈ *op'*.OutState that *si*.*st* == UD) {
13: UDSRC = Ø;
14: ∀ *op'* ∈ OPS and *si* ∈ *op'*.OutState,
15: if(*si*.*st* == UD) UDSRC = UDSRC ∪ *si*.SRC;
16: add(UD, UDSRC) to InStates;
17: }
18: if(∃ *op'* ∈ OPS and *si* ∈ *op'*.OutState that *si*.*st* ∈ (AB, DR)) {
19: ABSRC = Ø;
20: DRSRC = Ø;
21: ∀ *op'* ∈ OPS and *si* ∈ *op'*.OutState {
22: if(*si*.*st* == AB) ABSRC = ABSRC ∪ *si*.SRC;
23: else if(*si*.*st* == DR) DRSRC = DRSRC ∪ *si*.SRC;
24: }
25: if (ABSRC ≠ Ø) add(AB, ABSRC) to InStates;
26: if (DRSRC ≠ Ø) add(DR, DRSRC) to InStates;

```
27:         }
28:     } else  ∀ op' ∈ OPS, InStates = InStates ∪ op'.outStates;
29: return InStates;
}
```

The algorithm shows how to collect the input state of operation *op* from $DB4OPS_{op}$. An empty $DB4OPS_{op}$ indicates that no operation is operated before *op*. In this project, we assume that all the artifacts are initialized with state UD, and the state item (UD, {*w*.*s*}) is inserted to the result set in this circumstance. If $DB4OPS_{op}$ is not an empty set, the algorithm calculates the input state of *op* from each operation set in $DB4OPS_{op}$. An operation set containing multiple operations composes a racing operation, and the algorithm gives the input state of *op* generated from an RDS, RDK, RDU, RKS, and RKU from line 5 to 10 based on the description in section 3.1. For an RUS, if all the usages involved in the RUS propagate state UD in their output states, the artifact might be undefined after the RUS, and state UD is included in *op*'s input states accordingly. On the other hand, if there exists a usage involving in the RUS propagating state AB for the target artifact, the target artifact might be ambiguous in definition before *op* is operated. Similarly, if the target artifact is defined in one of the usages involved in the RUS, DR is recorded as one of the input states of *op*. The method to calculate the artifact states generated from an RUS is described from line 12 to 26 in the algorithm. Finally, if the operation set contains only one single operation. The input state of *op* is simply equivalent to the output state of the operation, and is handled at line 28. The input states of *op* are identified for each operation set collected by Algorithm 4. The completeness of the input states gathered by Algorithm 6 is restricted by the capability of Algorithm 4.

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

Definition 18 (Artifact Anomaly Table)
Let AAT_w be the artifact anomaly table for an LRTS workflow *w*
$\forall$ *aar* $\in AAT_w$, *aar* = (*op*, *type*, SRC),
aar.*op* indicates the abnormal artifact operation,
aar.*type* $\in$ {Useless Definition, Null Kill, Undefined Usage, Ambiguous Usage}
indicates the anomaly type, and
aar.*SRC* represents the set of operations leading to the anomaly.

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

of the output states for operations with different types.

Algorithm 7 Identifying Artifact Anomalies for No Operations - *IAAN*
Input: an LRTS workflow *w*,
an artifact operation *op*, and
a set of state items InState
Pre-Condition: *op.type* == Nop
IAAN {
01: $\forall$ *stItem* $\in$ InState,
02: if(*stItem.state* == DN)
03: add(*op'* | *op'* $\in$ *stItem*.SRC, Useless Definition, {*op*}) to AAT_w;
04: *op*.OutState = InState;
}

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

Algorithm 8 Identifying Artifact Anomalies for Definitions - *IAAD*
Input: an LRTS workflow *w*,
an artifact operation *op*, and
a set of state items InState
Pre-Condition: *op.type* == Def
IAAD {
01: $\forall$ *stItem* $\in$ InState,
02: if(*stItem.state* == DN)
03: add(*op'* | *op'* $\in$ *stItem*.SRC, Useless Definition, {*op*}) to AAT_w;
04: *op*.OutState = { (DN, {*op*}) };
}

Algorithm 8 identifies the artifact anomalies generated from a definition, and calculate its output state. For an artifact *a*, a definition which is not referenced by any usages before being defined again is a useless definition. Finally, a definition transits *a* to state DN, and the output state generated by the definition is recorded accordingly.

Algorithm 9 Identifying Artifact Anomalies for Kills - *IAAK*
Input: an LRTS workflow *w*,
an artifact operation *op*, and
a set of state items InState
Pre-Condition: *op.type* == Kill

```
IAAK {
01: ∀ stItem ∈ InState,
02:     if( stItem.state == DN )
03:         add( op' | op' ∈ stItem.SRC, Useless Definition, {op} ) to AAT_w;
04:     else if( stItem.state == UD ) add( op, Null Kill, stItem.SRC ) to AAT_w;
05: op_i.OutState = { ( UD, {op_i} ) };
}
```

Algorithm 9 identifies the artifact anomalies generated from a kill, and calculates its output state. A definition which is killed before being referenced is also useless, and the anomaly is detected at line 2 and 3. Besides, if an artifact remains undefined before a kill, the kill is redundant, and a Null Kill is raised accordingly. A kill transits an artifact to state UD, and the output state generated from the kill is recorded at line 5.

```
Algorithm 10 Identifying Artifact Anomalies for Usages - IAAU
Input: an LRTS workflow w,
       an artifact operation op, and
       a set of state items InState
Pre-Condition: op.type = Use
IAAU {
01: ∀ stItem ∈ InState {
02:     if( stItem.state == AB )
03:         add( op, Ambiguous Usage, stItem.SRC ∪
04:               ConcD_op ∪ ConcK_op ) to AAT_w;
05:     else if( stItem.state == UD ) {
06:         if(ConcD_op ≠ Ø )
07:             add (op, Ambiguous Usage, stItem.SRC ∪ ConcD_op) to AAT_w;
08:         else add(op, Undefined Usage, stItem.SRC) to AAT_w;
09:     }
10:     else if( stItem.state ∈ {DR, DN} )
11:         if( ConcD_op ∪ ConcK_op ≠ Ø )
12:             add(op, Ambiguous Usage, stItem.SRC ∪
13:                 ConcD_op ∪ ConcK_op) to AAT_w;
14:     if( stItem.state == DN ) add (DR, stItem.SRC) to op_i.OutState;
15:     else add stItem to op_i.OutState;
16: }
}
```

Algorithm 10 identifies whether a usage is abnormal, and calculates its output state. The input state AB indicates that the artifact is ambiguous in definition when the operation being

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

The expressions adopted in Algorithm 7 to Algorithm 10 are stated based on the description in section 4.1. With all the definitions and algorithms described in this chapter, the methodology detecting artifact anomalies in a TS workflow is introduced as following.

Algorithm 11 Identifying Artifact Anomalies - *IAA*
Input: an LRTS workflow *w*

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

At line 1, the algorithm first invokes Algorithm 1 to collect structural and temporal

information like EAIs, ABStacks, and artifact operation lists for the input LRTS workflow. For each artifact *a*, Algorithm 11 then identifies the concurrency between artifact operations with Algorithm 2 at line 3, and starts analysis of the each operation in $AOPL_a$ in order from line 4. Algorithm 3 is invoked at line 6 to collect the operations directly before op_i, and the operation sets directly before op_i is manufactured by Algorithm 4 from the previous result at line 7. At line 8, Algorithm 6 gathers the input state of op_i, and invokes corresponding algorithms from line 9 to 12 to detect artifact anomalies and calculate the output state of op_i. At line 13, Algorithm 5 is invoked to detect if there is any blank branch before op_i. If not, the anomaly detection work for op_i is accomplished. Otherwise, all the operations residing in the decision structure with blank branches are removed from OPL_{opi}, and Algorithm 11 repeats analysis of artifact anomalies for op_i until all the blank branches considered. The completeness of the artifact anomalies detected in our methodology is decided by the completeness of the operation sets identified by Algorithm 4. Developing an algorithm able to collecting more operation sets is helpful in enhancing our methodology, and is left as a future work of this study.

5. Case Study

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

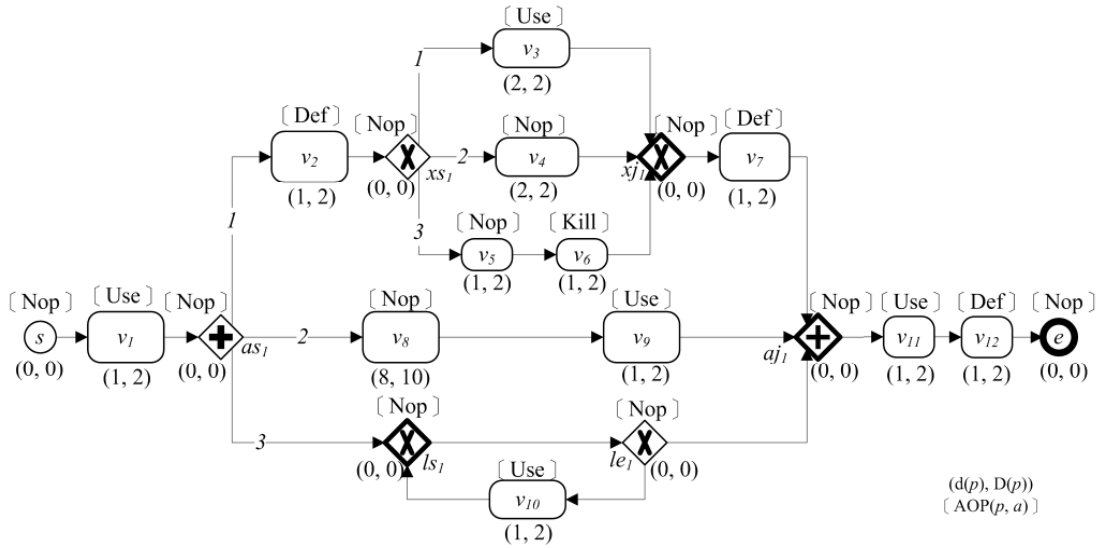

Figure 13 The Sample TS Workflow for the Case Study in Chapter 4

Figure 13 shows the sample TS workflow for our case study. The processes, flows, working durations, and the artifact operations made on artifact *a* are illustrated in the sample. To analyze the sample TS workflow with our methodology, the structured loops in the TS workflow should first be reduced. After loop reduction, the LRTS workflow generated from the sample TS workflow are illustrated as Figure 14.

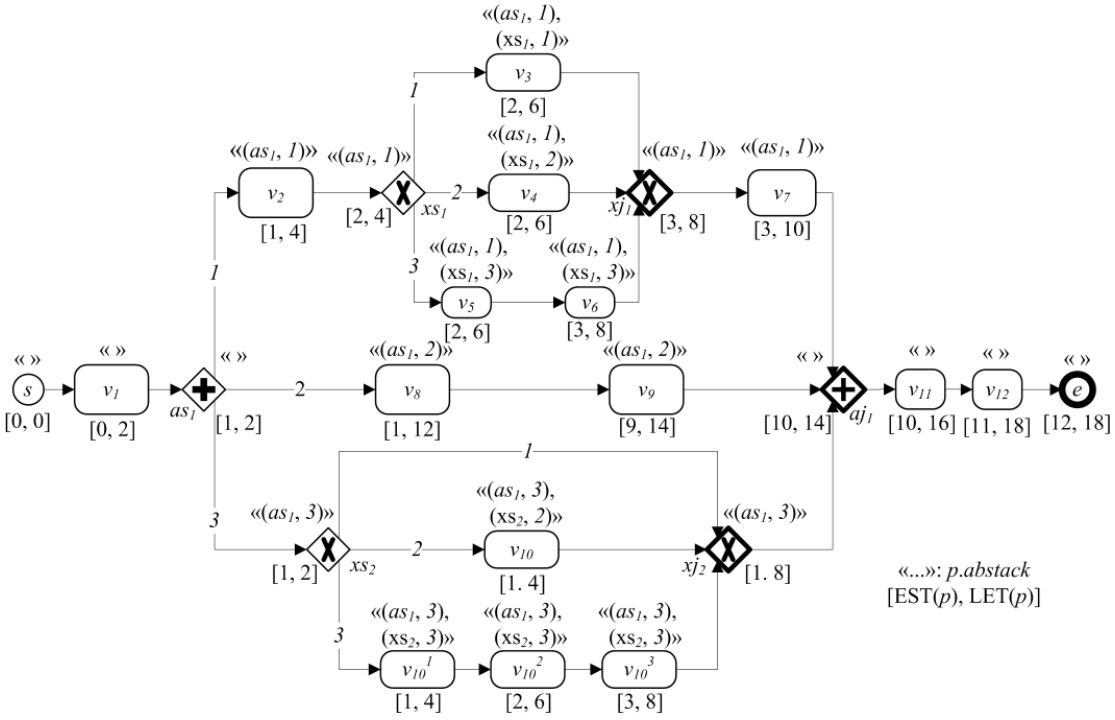

Figure 14 The Sample LRTS Workflow Derived from Figure 13 with Decoration of EAIs and ABStacks

After loop-reduction, Algorithm 1 is invoked to gather the temporal and structural information such as the EAI and the ABStack for each process, and the artifact operation list for each artifact. Table 1 illustrates the artifact operation list and the concurrent operations for artifact *a* generated by Algorithm 1 and Algorithm 2.

Table 1 Artifact Operation List for *a*, and the Corresponding Concurrent Operations

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

To be brief, we do not show all the details of detecting artifact anomalies in this case study, and focus on two representative examples, op_9 and op_{10}. Therefore, we assume that the operations before op_9 are calculated already, and Table 2 shows the output state of the operations with LETs smaller then op_9's.

Table 2 The Output State of the Operations before op_9 is Calculated

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

op_1 is an undefined usage because it is operated before any activity process gives definition to artifact *a*. op_3, op_4, op_6, and op_7 are ambiguous usages because there exist

definition concurrent to them. Before op_9 is calculated, the artifact anomaly table, AAT_w, records the following anomalies:

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

For op_9, Algorithm 11 retrieve all the operations with smaller LET from $AOPL_a$ as OPL_{op9}, {op_1, op_2, op_3, op_4, op_5, op_6, op_7, op_8}, and invokes Algorithm 3 to calculate $DB4_{op9}$, {op_5, op_7}. Since all the operations directly before op_9 are mutually exclusive, i.e. the case (2) described in section 4, the $DB4OPS_{op9}$ is calculated from Algorithm 4 as { {op_5}, {op_7} }. With $DB4OPS_{op9}$, Algorithm 6 gathers the input states of op_9 as the union of the output states of op_5 and op_7 as { (DR, {op_2}), (UD, {op_7}) }. op_9 is a definition, and Algorithm 8 is invoked for detection of artifact anomalies and generation of its output state. As a result, no artifact anomaly is found and the output state of op_9 is generated as { (DN, {op_9}) }. However, during the blank branch detection, (xs_1, 2) is found a blank branch, and the operation in the same decision structure should be removed to eliminate the effect of blank branch. op_5 and op_7 is removed from OPL_{op9}. $DB4_{op9}$, $DB4OPS_{op9}$, and the InState of op_9 are recalculated as {op_2}, {{op_2}}, and { (DN, {op_2}) }. After invoking Algorithm 8 once again, an artifact anomaly (op_2, Useless Definition, {op_9}) is raised because the definition made by op_2 is not used before redefinition when the blank branch is taken.

$DB4_{op10}$ is generated as {op_3, op_5, op_7, op_8}, and $DB4OPS_{op10}$ is generated as { {op_3, op_5}, {op_7, op_8} }. Since this case is relatively simple, we can easily identify that the operation sets {op_3, op_7} and {op_5, op_8} is neglected in our methodology. With $DB4OPS_{op10}$, the input states of op_{10} are generated. According to the definition of racing operations introduced in section 3.1, {op_3, op_5} is an RUS and {op_7, op_8} is an RKU, and { (DR, {op_2}) } and { (UD, {op_7}) } are generated as op_{10}'s input states correspondingly. Algorithm 10 is invoked to detect artifact anomalies and identify the output state of op_{10}. Two artifact anomalies, (op_{10}, Ambiguous Usage, {op_7, op_9}) and (op_{10}, Ambiguous Usage, {op_2, op_9}), are generated because op_9 makes a definition to a concurrently, and generates ambiguity to op_{10}. The output states of op_{10} is { (DR, {op_2}), (UD, {op_7}) }. Then the algorithm removes the blank branches for op_{10}, and finds no further anomalies.

Except for the artifact anomalies listed and described above, (op_{13}, Useless Definition, {e}) are detected and recorded to AAT_w when Algorithm 11 completes its work throughout w. The useless definition is detected at the end process of the LRTS workflow because the definition made by op_{13} is not used by any other activity process until the end of w.

6. Discussion

6.1 Related Works in Analysis of Artifact Anomalies

Sun et al. (2006) extend the Activity Diagram in UML for modeling data flow in a business process. Three classes of data-flow anomalies, missing data, redundant data, and conflicting data, are defined. With the routing information defined in a workflow specification, a detecting algorithm for the data-flow anomalies is constructed (Sun et al., 2006). However, Sun et al. (2006) do not build an explicit data model in characterizing the data behaviors, and consider only read and initial write in data operations.

Sadiq et al. (2004) reveal the importance about the validation of workflow data, and introduce seven basic data validation problems, Redundant Data, Lost Data, Missing Data, Mismatched Data, Inconsistent Data, Misdirected Data, and Insufficient Data in workflow models. Redundant Data occur when designers specify an activity to define a data item which is not required by any other succeeding activities. Lost Data occur when designers specify two activities that may be executed in parallel to define the same data item, and one of the definitions is lost when the data item is preempted by the process executed in advance. Missing Data occurs when designers specify an activity to consume a data item which is never defined by any preceding activities. Mismatched Data arise when the structure of data is incompatible between the definition and the usage of the data. Inconsistent data happen when the data required by a workflow are externally updated by other applications during the workflow execution, and the polluted data might cause errors of the workflow. Misdirected Data occur when the direction of the data flow is conflict with the direction of the control flow of the workflow. Insufficient Data happen when the data specified by designers is insufficient to successfully complete an activity.

Destruction of artifacts is not considered in both Sun and Sadiq's studies. Wang et al. (2006) and Hsu et al. (2007) consider the effect of destroying an artifact and re-model the inaccurate artifact manipulation by separating initialization and update as two different artifact operations, and define six inaccurate artifact usages: No Producer, No Consumer, Redundant Specification, Contradiction, Parallel Hazard, and Branch Hazard. No Producer is a warning indicating that a data item is operated before it is specified. No Consumer indicates that an artifact is not requested after its definition (initialization). Redundant Specification indicates that an artifact is repeatedly specified in a workflow. Contradiction implies the defect that the state of an artifact is not matched to the pre-condition or post-condition of the activity accessing it. Parallel Hazard occurs due to conflict interleaving of concurrent artifact operations, and is recognized if multiple concurrent activities operate on the same artifact. Branch Hazard occurs when branches in a decision structure contain operations on artifacts have been selected, or when there is inconsistency between the condition testing in the XOR-split process or the branches in the decision structure.

Wang et al. (2009) develop a systematic notation to describe artifact anomalies and simplify the description of artifact anomalies from the classification made by Hsu et al. (2007) into three categories, Missing Production, Redundant Write, and Conflict Write. Missing Production occurs when an artifact is consumed before it is produced or after it is destroyed. Redundant Write occurs when an artifact is written by an activity but the artifact is neither required by the succeeding activities nor a member of the process outputs. Conflict Write occurs when parallel processes race their access to the same artifact. According to different structural relationships between activities accessing some artifacts, thirteen abnormal usage patterns are described for the three categories to follow the previous models made by Sadiq et al.(2004), Hsu et al (2007), and Sun et al. (2006).

6.2 Comparison between Our Approach and the Related Works

Table 3 lists and compares the features between the related works and our approach. Artifact anomalies are appealed with different names in previous studies, but can still be mapped into the three basic categories made by Sun et al. (2006). By comparing the definition of the artifact anomalies defined in our approach and the related works, we conclude that Undefined Usage and Useless Definition can be directly mapped into Missing Data and Redundant Data descri intrduced by Sun et al. (2006). On the other hand, the Conflict Data defined by Sun et al. (2006) are anomalies generated when multiple definitions are made in parallel. In our approach, the concurrent definitions are considered being executed with undetermined order, and generate ambiguity in artifacts. They are not directly considered as an anomaly because (1) an anomaly actually occurs when a usage refers to the ambiguous definitions, and (2) similar anomaly may also occur when there exist kills or definitions concurrent to usages. Therefore, Ambiguous Usage is categorized in this project, and covers Conflict Data discussed in the previous works. Besides, Sadiq et al. additionally define Insufficient Data and Mismatched Data in (Sadiq et al., 2004) for conflicts about contents or format between definitions and usages. Since the studies made by Hsu et al. (2007), Wang et al, (2009), Sun et al. (2006) and this project do not discuss the contents of artifacts, Insufficient Data and Mismatched Data are ignored in these studies. Finally, although destruction of artifacts is considered in Hsu et al. (2007), Wang et al, (2009)'s studies, the redundancy generated by unnecessary destruction is not discussed in these works. In our studies, Null Kill is categorized and detected to eliminate such redundancies.

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

Table 3 Comparison between Our Approach and the Related Works

<table>
<tr><th>Our Approach</th><th colspan="2">Sun et al. (2006)</th><th>Sadiq et al. (2004)</th><th>Hsu et al. (2007)</th><th colspan="2">Wang et al. (2009)</th></tr>
<tr><td rowspan="4">Undefined Usage</td><td rowspan="4">Missing Data</td><td>Absence of Initialization</td><td>Missing Data</td><td rowspan="2">No Producer</td><td rowspan="4">Missing Production</td><td>No Production</td></tr>
<tr><td>Delayed Initialization</td><td rowspan="2">Misdirected Data</td><td>Delayed Production</td></tr>
<tr><td rowspan="2">Improper Routing</td><td rowspan="2">Branch Hazard</td><td>Conditional Production</td></tr>
<tr><td>N/A</td><td>Exclusive Production</td></tr>
<tr><td>Uncertain Availability</td><td>Misdirected Data</td><td>Parallel Hazard</td><td>Uncertain Production</td></tr>
<tr><td rowspan="2">Useless Definition</td><td rowspan="2">Redundant Data</td><td>Contingent Redundancy</td><td rowspan="2">Redundant Data Mismatched Data</td><td>Branch Hazard</td><td rowspan="2">Redundant Write</td><td>Conditional Consumption after Last Write</td></tr>
<tr><td>Inevitable Redundancy</td><td>No Consumer</td><td>No Consumption after Last Write</td></tr>
<tr><td>Ambiguous Usage</td><td>Conflict Data</td><td>Multiple Initialization</td><td>Lost Data</td><td>Contradiction</td><td>Conflict Write</td><td>Multiple Parallel Production</td></tr>
<tr><td>N/A</td><td>N/A</td><td>N/A</td><td>Insufficient Data Mismatched Data</td><td>N/A</td><td>N/A</td><td>N/A</td></tr>
<tr><td>Null Kill</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>Temporal Consideration</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
<tr><td>Anomaly Source Tracking</td><td colspan="2">N/A</td><td>N/A</td><td>N/A</td><td colspan="2">N/A</td></tr>
</table>

7. Conclusion and Future Works

In this project, the temporal factors in structured workflows are considered with TS workflow. The twisted temporal and structural relationships between processes in TS workflow are modeled, and the methodologies to analyze such relationships in TS workflow are presented. Three classes of artifact operations, Define, Use, and Kill, are discussed, and four kinds of artifact anomalies, Useless Definition, Undefined Usage, Ambiguous Usage, and Null Kill, are defined. The racing operations composed by concurrent operations are categorized into seven classes, the how artifact operations may affect the state transition of an artifact is depicted. With all the features modeled above, the methodology to detect artifact anomalies in a TS workflow is established. A case study and comparison between our approach and the related works are made to show the feasibility and contribution of our work.

Several advanced studies can still be made in this topic. First, a solution to group the operation sets from the operations directly before another operation can be further studied. The efficiency and completeness of the solution should be discussed. Second, the methodology described in this project traverses the whole workflow schema to detect artifact anomalies. In the future, on the basis of this work, we might construct an incremental algorithm to detect artifact anomalies generated or eliminated by each design operations made by the designers. The algorithm analyzes only partial workflow schema which is changed in last design operation and informs the designers immediately after a design operation is made. Finally, the artifact operations can be discussed in finer-grained, we may not only summarize the actions which an activity process made on an artifact into one single artifact operation, but consider all the operations made by activity processes and discuss the dependency between the operations in more detail.

Reference

(Adam et al., 1998) N. R. Adam, V. Atluri, and W.-K. Huang, 1998, Modeling and Analysis of Worfklows Using Petri Nets, Journal of Intelligent Information Systems, Vol. 10, Issue 2, pp. 131-158

(Allen, 1983) J. F. Allen, 1983, Maintaining knowledge about temporal intervals, Communication of the ACM, Vol. 26, Issue 11, pp. 832–843

(Chen and Yang, 2008) J. Chen, and Y. Yang, 2008, Temporal Dependency based Checkpoint Selection for Dynamic Verification of Fixed-time Constraints in Grid Workflow Systems, the Proceedings of the 30th International Conference on Software Engineering, pp. 141-150

(Eder et al., 1999a) J. Eder, E. Panagos, H. Pozewaunig, and M. Rabinovich, 1999, Time Management in Workflow Systems, the Proceedings of International Conference on Business Information Systems, pp. 266-280

(Eder et al., 1999b) J. Eder, E. Panagos, and M. Rabinovich, 1999, Time Constraints in Workflow Systems, Lecture Notes in Computer Science, Vol. 1626, pp. 286-300

(Hsu et al., 2005) H.-J. Hsu, D.-L. Yang, and F.-J. Wang, 2005, An Incremental Analysis to Workflow Specifications, the Proceedings of the 12th Asia-Pacific Software Engineering Conference, pp. 122-129

(Hsu et al., 2007) C.-L. Hsu, H.-J. Hsu, and F.-J. Wang, 2007, Analysing Inaccurate Artifact Usages in Workflow Specifications, IET Software, Vol. 1, Issue 4, pp. 188-205

(Hsu and Wang, 2008) H.-J. Hsu and F.-J. Wang, 2008, An Incremental Analysis for Resource Conflicts to Workflow Specifications, Journal of Systems and Software, Vol. 81, Issue 10, pp. 1770-1783

(Hsu et al. 2009) H.-J. Hsu, and F.-J. Wang, 2009, Using Artifact Flow Diagrams to Model Artifact Usage Anomalies, the Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference, Vol. 2, pp.275-280

(Kiepuszewski et al., 2000) B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler, 2000, On Structured Workflow Modelling, Lecture Notes in Computer Science, Vol. 1789, pp. 431-445

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

(Li et al., 2004a) J. Li, Y. Fan, and M. Zhou, 2004, Performance Modeling and Analysis of Workflow, IEEE Transaction on Systems, Man, and Cybernetics - Part A: Systems and Humans, Vol. 34, Issue 2, pp.229-242

(Li et al., 2004b) H. Li, Y. Yang, and T. Y. Chen, 2004, Resource Constraints Analysis of Workflow Specifications, Journal of Systems and Software, Vol. 73, Issue 2, pp. 271-285, 2004

(Li and Yang, 2005) H. Li, and Y. Yang, 2005, Dynamic Checking of Temporal Constraints for

Concurrent Workflows, Electronic Commerce Research and Applications Vol. 4, pp. 124-142

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

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

(Pardalos and Xue, 1994) P. M. Pardalos, and J. Xue, 1994, The Maximum Clique Problem, Journal of Global Optimization, Vol. 4, No. 3, pp. 301-328

(Sadiq et al., 2004) S. Sadiq, M. E. Orlowska, W. Sadiq, and C. Foulger, 2004, Data flow and validation in workflow modeling, the Proceedings of the 15th Conference on Australasian Database, Vol. 27, pp. 207-214

(Sun et al., 2006) Formulating the data flow perspective for business process management, Information Systems Research, Vol. 17, Issue 4, pp. 374-391

(Tsukiyama et al., 1977) (Makino and Uno, 2004)S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, 1977, A new algorithm for generating all the maximal independent sets, Society for Industrial and Applied Mathematics (SIAM) Journal on Computing, Vol. 6, pp. 505–517

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

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

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

(Wang et al., 2009) C.-H. Wang, and F.-J. Wang, 2009, Detecting Artifact Anomalies in Business Process Specification with a Formal Model, Journal of Systems and Software, Vol. 82, Issue 10, pp. 1064-1212

(WfMC, 1999) Workflow Management Coalition (WfMC), 1999, WFMC-TC-1011 Ver 3 Terminology and Glossary English, Workflow Management Coalition

(Zhuge et al., 2001) H. Zhuge, T.-Y. Cheung, and H.-K. Pung, 2001, A Timed Workflow Process Model, Journal of Systems and Software, Vol. 55, Issue 2, pp. 231-243