Detecting artifact anomalies in business process specifications with a formal model

Ching-Huey Wang *, Feng-Jian Wang

Institute of Computer Science and Engineering, College of Computer Science, National Chiao Tung University, Room 510, EC Building, 1001 Ta-Hsueh Road, Hsinchu City, Taiwan, ROC

1. Introduction

A workflow is seemingly a set of interrelated tasks systematized to achieve certain business goals by accomplishing each task in a particular order under automatic control (The Workflow Management Coalition, 1995). Workflow implementation requires resources for supporting process execution. Resource allocation and resource constraint analysis (Senkul and Toroslu, 2005; Li et al., 2004; Liu et al., 2003; Du and Shan, 1999; Muehlen, 1999) are popular workflow research topics. However, data flow within workflow is seldom addressed (Sadiq et al., 2004; Sun and Zhao, 2004; Sun et al., 2004, 2006).

An artifact is a data instance within a workflow. Introducing artifact usage analysis into (control-oriented) workflow designs can help maintain consistency between execution order and data transition (Sadiq et al., 2004; Sun and Zhao, 2004; Sun et al., 2004, 2006), as well as prevents exceptions due to contradiction between data flow and control flow. In contrast to structural correctness, accuracy in artifact manipulation helps determine whether the execution result of a workflow is meaningful and desirable. Our earlier work (Hsu, 2005; Wang et al., 2006; Hsu and Wang, 2007) introduced the artifact usage analysis into workflow design phase and identified preliminary improper artifact usages affecting workflow execution.

This paper proposes a process model for describing the input/output of business processes and addresses three types of artifact usage anomalies. The model is based on component-based design technique (Zhuge, 2003; Hitomi and Le, 1998) and is compatible with existing control-oriented workflow design models. It provides an easier way to extract knowledge of artifact usages in a workflow. With the model, an analysis procedure of artifact usage is applied before deploying the workflow schema. On the other hand, the checking between data flow and control flow and the information of manipulating artifacts can be applied stepwise along with the specification process. An example demonstrating the contribution of our work and a comparison among related works and ours is also presented.

The remainder of this paper is organized as follows. Section 2 presents the research background and related works. Section 3 presents our process model, including the control flow and artifact flow. There are thirteen cases of artifact usage anomalies identified and then categorized into three types in Section 4. In Section 5, we present a method for detecting each type of anomalies. A comparison between our approach and some related works is given in Section 6. Finally, a conclusion and some recommendations of future work are given in Section 7.

2. Related work and background

2.1. Analysis in workflow specification

A workflow can be deemed as a collection of cooperating and coordinated activities designed to carry out a well-defined complex process, such as a trip planning, conference registration procedure, or business process in an enterprise. In addition, the technology is adopted further in developing service-oriented architecture in the last decade (Yau et al., 2008, 2007). A workflow model describes a workflow in terms of various elements, such as roles and resources, tools and applications, activities, and data, which
represent different perspectives of a workflow (Curris et al., 1992; Jablonski and Bussler, 1996). Roles and resource elements represent organizational perspective that describes the performers of the tasks instantiated. Tools and application elements represent operational perspectives by specifying what tools and applications are used to execute a particular task. Activity elements are defined with two perspectives: (1) functional: what tasks a workflow performs; and (2) behavioral: when and how tasks are performed. Data elements represent the informational perspective, i.e., what information entities are produced or manipulated in corresponding workflow activities.

A well-defined workflow model leads to efficient development of an effective and reliable workflow application. Correctness issues in a workflow might be classified into three dimensions: control-flow, resource, and data-flow. Generally, the analyses in control-flow dimension are focused on correctness issues of control structure in a workflow. Common control-flow anomalies include deadlock, livelock (infinite loop), lack of synchronization, and dangling reference (Karamanolis et al., 2000a; van der Aalst, 1998a, 1997, 1998b; van der Aalst and ter Hofstede, 2000; van der Aalst et al., 2000a,b, 2003; van der Aalst and Basten, 2002; Verbeek and van der Aalst, 2000; Karamanolis et al., 2000b; Verbeek et al., 2001). For example, a deadlock anomaly occurs if it is no longer possible to make any progress for a workflow instance, e.g. synchronization on two mutually exclusive alternative paths.

Activities belonging to different workflows or parallel activities in the same workflow might access the same resources. A resource conflict occurs when these activities execute over the same time interval. Thus, the analyses in resource dimension include identification of resource conflicts under resource allocation constraints and/or under temporal and/or causality constraints (Senkul and Toroslu, 2005; Li et al., 2004; Liu et al., 2003; Du and Shan, 1999; Muehlen, 1999). On the other hand, missing, redundancy, and conflict use of data are common anomalies in data-flow dimension (Sadiq et al., 2004; Sun and Zhao, 2004; Sun et al., 2004, 2006). A missing data anomaly occurs when an artifact is accessed before it is initialized. A redundant data anomaly occurs when an activity produces an intermediate data output but this data is not required by any succeeding activity. A conflicting data anomaly represents different versions of the same artifact.

Current workflow modeling and analyzing paradigms mainly focus on soundness of control logic, i.e., in the control-flow dimension, including process model analysis (van der Aalst and ter Hofstede, 2000; van der Aalst et al., 2000a,b, 2003; van der Aalst, 1997, 1998b; van der Aalst and Basten, 2002; Verbeek and van der Aalst, 2000; Karamanolis et al., 2000b; Verbeek et al., 2001; Gong and Wang, 2004; Sadiq and Orlowska, 2000), workflow patterns (van der Aalst et al., 2000a,b, 2003; van der Aalst, 1997, 1998b; van der Aalst and Basten, 2002; Verbeek and van der Aalst, 2000; Karamanolis et al., 2000b; Verbeek et al., 2001; Gong and Wang, 2004; Sadiq and Orlowska, 2000), and automatic control of workflow process (Bae et al., 2004). Aalst and ter Hofstede van der Aalst and ter Hofstede (2000) proposed a Workflow net (WF-net), based on Petri nets, to model a workflow: transitions representing activities, places representing conditions, tokens representing cases, and directed arcs connecting transitions and places. Son and Kim (2005) defined a well-formed workflow based on closure and control block concepts. He claimed that a well-formed workflow is free from structural errors, and that complex control flows can be made with nested control blocks. Sadiq and Orlowska (2000) proposed a visual verification approach and algorithm with a set of graph reduction rules to discover structural conflicts in process models for given workflow modeling languages.

There are several research topics discussed in resource dimension, including resource allocation constraints (Senkul and Toroslu, 2005; Li et al., 2004), resource availability (Liu et al., 2003), resource management (Du and Shan, 1999) and resource modeling (Muehlen, 1999). Senkul Senkul and Toroslu (2005) developed an architecture to model and schedule workflow with resource allocation constraints and traditional temporal/causality constraints. Li et al. (2004) concluded that a correct workflow specification should have resource consistency and provided a series of detection algorithms. Both Pinar and Hongchen extended workflow specifications with constraint descriptions. Liu et al. (2003) proposed a three-level bottom-up workflow design method to effectively incorporate confirmation and compensation in case of failure.

There was little attention paid to the data-flow dimension, although related analysis in the data-flow dimension is very important since activities cannot be executed properly without sufficient data information. For example, Sadiq et al. (2004) identified and justified the importance of data modeling in the overall workflow design process. In addition, data-flow validation issues and essential requirements of data-flow modeling in workflow specifications are identified. They illustrated and defined seven basic data validation problems: redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data. However, Sadiq worked only on the conceptual level and thus, neither concrete data-flow model nor detecting algorithms are proposed. Furthermore, operations on data are only classified into read and write type.

Sun and Zhao (2004), Sun et al. (2004, 2006) formulate the data-flow perspective by means of dependency analysis. The data-flow matrix and an extension of the unified modeling language (UML) activity diagram are proposed to specify the data flow in a business process. Then, three basic types of data-flow anomalies, missing data, redundant data, and conflicting data, were defined. Based on the dependency analysis, algorithms to data-flow analysis for discovering the data-flow anomalies are presented and execute in $O(n^3)$ time. However, as in Sun’s work, there is no explicit model proposed to characterize data behaviors. Also, the operation types are considered only with read and initial write.

Our previous work (Hsu and Wang, 2007) was concerned with five different types of operations, read, write, specify, destroy and revise, but the behaviors of an artifact are not explicitly modeled by a finite state machine. The three-layer workflow model is proposed for constructing state transition diagrams of each flow structure to detect inaccurate artifact usage. Detection algorithms are designed based on critical paths for discover the inaccurate state transition of an artifact. However, the time complexity of the algorithms is $O(n^3)$. Table 2.1 summaries the comparisons.

<table>
<thead>
<tr>
<th>Table 2.1 Summary of comparisons.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sadiq et al.</td>
</tr>
<tr>
<td>Process model</td>
</tr>
<tr>
<td>Operations concerned</td>
</tr>
<tr>
<td>Detecting method</td>
</tr>
<tr>
<td>Concrete algorithm</td>
</tr>
<tr>
<td>Complexity</td>
</tr>
</tbody>
</table>
2.2. BPEL, BPMN and loop-simplification

BPEL is one of the most popular workflow languages. In BPEL (Alves et al., 2007), the control structures, proposed to indicate the order where the individual activities are executed, include "sequential processing" (Sequence), "repetitive execution" (While and RepeatUtil), "parallel processing" (Flow), "exclusive branches" (If and Pick) and "inclusive branches" (OR and Complex). All types of business processes can be defined by nesting and/or combining the five structures in arbitrary ways (White, 2008).

BPMN (White, 2008) is a solution to represent the business process in a graph. The elements defined in BPMN are classified into sequence flow and object. A sequence flow links two objects to show the execution order. An object can be an event, a task or a gateway. An event may signal the start (start event), the end (end event) of a process, a message that arrives or a specific time-date being reached during a process (intermediate message/timer event). A task is an atomic activity and stands for the work to be performed within a process. A gateway is a routing construct used to control the divergence and convergence of sequence flow. For a set of parallel sequences, a parallel fork/join gateway creates/synchronizes concurrent sequence flows. For a set of exclusive sequences, a data/event-based XOR decision/merge gateway selects one from/joins a set of mutually exclusive sequence flows. The selection is based on either the process data (data-based) or external event (event-based). Besides, there are two pairwise OR and complex gateways: OR decision gateway, OR merge gateway, complex decision gateway and complex merge gateway, their functions can be replaced by combining and/or nesting two or more gateways indicated above.

Ouyang et al. (2006) asserts that the control structures can be represented with BPEL or BPMN in patterns; moreover, the well-structured components representing these patterns with BPMN are proposed. Fig. 2.1 shows examples depicted with BPMN in a well form. In the exclusive branch structures (c) and (d), no matter which succeeding sequence flow is selected (due to the data propagated from its preceding activities in (c) or an event signaled for a notification in (d)), the data propagation mechanisms of both cases are the same in an implicit data flow model (Sadiq et al., 2004). A data exchange between activities is done through global variable(s) stored in a common database. However, the analysis in an iteration is not well concerned.

The set of gateways defined in BPMN is not intended to be minimal (White, 2008). The semantics of one structure might be implemented with other structure(s). To simplify the discussion, this paper minimizes the set of gateways by excluding the ones whose functions are replaceable. In addition, the elements defined in a general model, e.g. BPMN or BPEL, which do not support the analysis of artifact usage, are omitted in our model. For example, the event object of BPMN determining the succeeding flow does not affect the design of data propagation mechanism, it is not concerned here.

![Fig. 2.1. Examples of well-structured control flows.](image-url)
3. Process modeling

3.1. Basic concept of loop reduction

Our major concern is to find artifact usage anomalies and reduce the computation time. The basic concept of loop reduction is to transform a loop into an XOR structure. The condition(s) of a RepeatUntil structure is to transform a loop into an XOR structure. The condition(s) of a While structure W is evaluated after each iteration but that of a RepeatUntil structure R is evaluated after each iteration. Obviously, the least time of a While structure W is not concerned. An artifact can be associated with a state representing the latest activity on the artifact (Hsu and Wang, 2007). For iteration times \( k \geq 2 \), all the possible state variations of the artifacts operated are the same as those for iterations \( k = 2 \). Therefore, the state variations in a loop can be grouped according to \( k \). For R, there are two groups of state variations. The first group is for \( k = 1 \) and the second one is for \( k \geq 2 \). For W, besides above two groups, there is a group for \( k = 0 \).

Each above group of operation(s) can be translated into a sequence flow (of operations). The R and W can be represented with corresponding XOR gateways and the flows in order to analyze the abnormal behaviors due to these operations. The XOR structures for the loops shown in Fig. 2.1e and f are presented in Fig. 3.1a and b, correspondingly.

3.2. Process specifications

A process consists of a network of activities designed to produce a product or service for a particular customer or market. A process specification is a formalized view of a business process, defines a set of linked (parallel and/or sequential) activities associated with clear defined inputs and outputs respectively. Each activity takes a subset of the process input(s) or the output(s) of its previous activity(ies) and transforms them into data for later use or as process outputs. These data are called artifacts. Thus, a process specification contains not only the control flow but also the artifact flow inside the business process. Here, we give a formal definition of process with these two parts in Definition 3.1.

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

- \( G = (V, E) \), representing the control flow, is a directed and acyclic graph, where \( V \) is a set of vertices of which each represents an activity and \( E \subseteq V \times V \) is a set of directed edges indicating the precedence relation between two activities.
- \( VT : V \to T \) is a type function that maps each activity into one of the activity types in \( T \), where
  
  \[ T = \{ \text{Task}, \text{SubProcess}, \text{ProcessStart}, \text{ProcessEnd}, \text{AndSplit}, \text{AndJoin}, \text{XorSplit}, \text{XorJoin} \} \]

The activities whose types are Task are called task activities while the others are called control activities.
- \( D \) is a set of artifacts used in the process.
- \( I_W \subseteq D \), a subset of \( D \), denotes the set of process inputs.
- \( O_W \subseteq D \), a subset of \( D \), denotes the set of process outputs.

3.3. Control flow specification

3.3.1. Activities and control blocks

An activity in a business process might be atomic or non-atomic (compound). An atomic activity is an indecomposable unit of work that is scheduled by a workflow engine. A sub-process activity within a process represents a compound activity. The function-based classification of atomic activities divides these activities into two groups: Task and control. A task activity is defined as making some function progress provided from its associated business process. Three pairs of control activities can be defined to bound a group of activities: (1) ProcessStart and ProcessEnd, (2) AndSplit and AndJoin and (3) XorSplit and XorJoin. Each pair and the activities bounded by them are named as a control block. Notations for activities in BPMN (White, 2008) are partially adopted and shown in Fig. 3.2.

Three control structures “sequential”, “parallel branch” and “conditional branch” are constructed from typed activities and their precedence relations, as the figures shown in Fig. 3.3. These structures are used in analyzing the artifact usages. In this paper, the three structures are implemented with Sequential, AND Control and XOR Control Block, respectively. Statements of the four implementations are given below:

- **Sequential block**: Activities within this structure are executed sequentially. Serial activities are fired while completing their preceding activities. Succeeding activities are triggered after their execution.

![Fig. 3.1. The XOR structures of the loops shown in Fig. 2.1e and f.](image)

![Fig. 3.2. The adopted notations.](image)
• **AND control block**: An AndSplit activity connects with more than one outflow. All outflows of the AndSplit activity execute currently. These outflow executions merge synchronously into one on AndJoin activity.

• **XOR (exclusive OR) control block**: An XorSplit activity connects with more than one outflow, called branches. The evaluation result of the XorSplit activity decides one of the outflows to continue. After execution, these branches converge on an XorJoin activity. No synchronization is required since there is only one thread chosen for execution.

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

- $G$ has a unique vertex $v$ of type ProcessStart, which has no incoming edge and one outgoing edge.
- $\exists v : V T(v) = \text{ProcessStart} \rightarrow \text{InDegree}(v) = 0 \land \text{OutDegree}(v) = 1$.
- $G$ has a unique ProcessEnd vertex $v$ of type ProcessEnd, which has one incoming edge and no outgoing edge.
- $\exists v : V T(v) = \text{ProcessEnd} \rightarrow \text{InDegree}(v) = 1 \land \text{OutDegree}(v) = 0$.
- Vertices of type AndSplit and XorSplit have one incoming edge and more than one outgoing edge.
- $\forall v : (V T(v) = \text{AndSplit} \lor \text{XorSplit}) \rightarrow \text{InDegree}(v) = 1 \land \text{OutDegree}(v) > 1$.
- Vertices of type AndJoin and XorJoin have more than one incoming edge and one outgoing edge.
- $\forall v \in (V T(v) = \text{AndJoin} \lor \text{XorJoin}) \rightarrow \text{InDegree}(v) > 1 \land \text{OutDegree}(v) = 1$.
- Any two control blocks can be nested but not overlapped.
- A typed control block $b$ is denoted with a startVertex and an endVertex. The two vertices correspond to a pair of control activities defined in Section 3.3.1.

$\forall b_1 = [v_i, v_j], \ b_2 = [v_k, v_l], \ b_1 \neq b_2 \rightarrow b_1 \subset b_2 \lor b_2 \subset b_1 \lor b_1 \cap b_2 = \emptyset$.

3.3.2. Relations among activities and control blocks

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

**Definition 3.2 (Paths)**. A path from $v_i$ to $v_k$ is a sequence of vertices $(v_1, \ldots, v_k)$ in a control graph $G = (V,E)$ such that each node is connected to the next vertex in the sequence i.e., the edges $(v_i, v_{i+1})$ for $i = 1, 2, \ldots, k - 1$ are in the edge set $E$. The path from $v_i$ to $v_k$ is denoted by $\text{Path}(v_i,v_k)$.

**Definition 3.3 (Reachability)**. Given two vertices $u$ and $v$, $\text{IsReachable}(u,v)$ is a Boolean function that indicates whether there is a path from $u$ to $v$.

$\forall u, v \in V, \ \text{IsReachable}(u,v) = \text{true} \leftrightarrow \exists \text{Path}(u,v) \lor v = u$.

**Definition 3.4 (Predecessors and successors)**.

$\text{V}_p^{\text{Predecessor}} = \{u \in V | (u,v) \in E\}$,

$\text{V}_p^{\text{Predecessor}} = \{t \in V | t \in \text{V}_p^{\text{Predecessor}} \lor \exists u \in \text{V}_p^{\text{Predecessor}} : t \in \text{V}_u^{\text{Predecessor}}\}$,

$\text{V}_p^{\text{Successor}} = \{u \in V | (v,u) \in E\}$,

$\text{V}_p^{\text{Successor}} = \{t \in V | t \in \text{V}_p^{\text{Successor}} \lor \exists u \in \text{V}_p^{\text{Successor}} : t \in \text{V}_u^{\text{Successor}}\}$.

$\text{V}_p^{\text{Predecessor}}$ comprises the set of vertices which are the source of an edge with destination vertex $v \in V$. Each element $u$ in $\text{V}_p^{\text{Predecessor}}$ is called a direct predecessor of the vertex and is denoted by $u \rightarrow v$. $\text{V}_p^{\text{Predecessor}}$ denotes the transitive closure of $\text{V}_p^{\text{Predecessor}}$. All $u \in \text{V}_p^{\text{Predecessor}}$, $v$ is reachable from $u$. Each element $u$ in $\text{V}_p^{\text{Predecessor}}$ is called a predecessor of $v$ and is denoted by $uv$. $\text{V}_p^{\text{Successor}}$ and its transitive closure $\text{V}_p^{\text{Successor}}$ are defined similarly.

**Definition 3.5 (Ancestor blocks and level of an activity)**. $\forall v \in V$, let $v.PB$ denote the parent control block containing $v$. $\text{AncestorBlock}(v)$ comprises the set of all control blocks that contain $v$.

$\text{AncestorBlock}(v) = \{b | b = v.PB \lor (b \in \text{AncestorBlock}(v.PB, \text{startVertex}))\}$.

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

$\text{Level}(v) = \begin{cases} \text{AncestorBlock}(v) & \text{if } v \in V, \\ \text{AncestorBlock}(v, \text{startVertex}) & \text{if } v \text{ represents a control block} \end{cases}$.
Definition 3.6 (Common ancestor blocks and nearest common ancestor blocks). Given a set of vertices, \( v_1, \ldots, v_n \), \( B_i \) is a common ancestor block of \( v_1, \ldots, v_n \) if and only if the following holds:

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

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

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

Definition 3.7 (Parallel activities). Given two vertices, \( u \) and \( v \), \( \text{IsParallel}(u, v) \) is a Boolean function to represent if \( u \) and \( v \) might be executed in parallel within a workflow instance.

\[
\text{IsParallel}(u, v) = \text{true} \iff \text{NCAB}(u, v).\text{Type} = \text{"AND"} \land \text{IsReachable}(u, v) \land \text{IsReachable}(v, u).
\]

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

Definition 3.8 (Exclusive activities). Given two vertices, \( u \) and \( v \), \( \text{IsExclusive}(u, v) \) is a Boolean function to represent some XOR characteristics of \( u \) and \( v \). Within a workflow instance, if \( u \) will not be selected for execution, \( v \) is selected for execution and vice versa

\[
\text{IsExclusive}(u, v) = \text{true} \iff \text{NCAB}(u, v).\text{Type} = \text{"XOR"} \land \text{IsReachable}(u, v) \land \text{IsReachable}(v, u).
\]

\[
\text{IsExclusive}(u, v) = \text{true}, \text{ denoted as } u \odot v, \text{ indicates that at most one of } u \text{ and } v \text{ can be selected for execution and } v \text{ is called an exclusive activity of } u.
\]

Definition 3.9 (Companion activities). Given two vertices, \( u \) and \( v \), \( \text{IsCompanion}(u, v) \) is a Boolean function which indicates whether both \( u \) and \( v \) are selected for computation

\[
\text{IsCompanion}(u, v) = \text{true} \land \text{Level}(u) \neq \text{Level}(v) \iff \forall b \in \text{AncestorBlock}(u) \cup \text{AncestorBlock}(v) \land \text{CAB}(u, v) : b.\text{type} = \text{"AND"},
\]

\[
\text{IsCompanion}(u, v) = \text{true} \land \text{Level}(u) = \text{Level}(v) \iff \forall b \in \{\text{NCAB}(u, v)\} : b.\text{type} = \text{"AND"},
\]

\[
\text{IsCompanion}(u, v) = \text{true}, \text{ denoted as } u \odot v, \text{ indicates that neither or both of them are selected for execution. } v \text{ is called a companion activity of } u.
\]

3.4. Artifact flow specification

Currently, as identified in Sadiq et al. (2004), there are three major implementation models for artifact flow: explicit data flow, implicit data flow through control flow, and implicit data flow through a process data store. This paper adopts implicit data flow model through a common process data store. Artifact exchanges between tasks are done through global variables stored in a common database. In a workflow, some activities store their output artifacts in the database, and their following activities may access these artifacts later. The activities in our model are regarded as black boxes, i.e., neither the internal computations nor intermediate execution states are visible for each activity. Thus, the artifact usages of an activity are identified as the inputs/outputs of the activity.

3.4.1. Artifacts and artifact operations

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

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

In general, an **Initialize** operation is used to create an artifact instance in a process. **Read** and **Update** operations are then used to access the instance. Finally, a **Destroy** operation is used to delete the artifact instance. **Destroy** operations are applied for temporary artifacts created during the workflow execution, but may not be strict for all artifacts (see Fig. 3.4).

Fig. 3.4 shows the state diagram of an artifact with the above four kinds of operations, “Uninitialized”, “Initialized”, “Updated”, and “Read”. “Uninitialized” represents the initial state of an artifact. “Initialized”, “Updated”, and “Read” represent states after an **Initialize**, **Update**, and **Read** operation is performed respectively. In addition, the artifact state is set to “Uninitialized” after a **Destroy** operation.

3.4.2. Artifact flow and artifact usages

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

**Definition 3.10** (Task/control activities). An task/control activity is a tuple \( \nu = (AT_{\nu}, SC_{\nu}, EC_{\nu}, RC_{\nu}, L_{\nu}, O_{\nu}, AS_{\nu}) \), where

- \( AT_{\nu} \) represents the type of the activity.
- \( SC_{\nu} \), \( EC_{\nu} \), and \( RC_{\nu} \) are the sets of logical expressions which are evaluated by a workflow engine.

![Fig. 3.4. The state diagram of an artifact.](image-url)
SC is the set of pre-conditions of which each is evaluated to
decide whether an activity within a process instance can be
started (only used by task activities).
EC is the set of post-conditions of which each is evaluated to
decide whether an activity within a process instance is com-
pleted (only used by task activities).
RC is the set of routing conditions of which each is evaluated
to decide the sequence of activity execution within a process
(only used by control activities).

- \( I \), the input set, identifies all the artifacts required to be
  accessed by the activity.
  - For a task activity, \( I \) contains all the artifacts required for its
    computation.
  - For a control activity, \( I \) contains all the artifacts used to eval-
    uate the routing conditions.
- \( O \), the output set, identifies all the artifacts produced, updated,
or destroyed after executing the activity \( v \). \( O \) is divided into two
  disjoint subsets, \( O^R \) and \( O^- \), where \( O^R \) represents the set of
  the artifacts initialized or updated by \( v \) and \( O^- \) represents the set
  of the artifacts destroyed by \( v \).
- AS, the activity specification (only used by task activities).

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

**Definition 3.11 (Consumer, producer, updator, and destroyer activ-
ities of an artifact).** For a given artifact \( a \), the memberships between
artifact \( a \) and \( I \), \( O^R \), and \( O^- \) can be applied for identifying the usage of
artifact \( a \) at activity \( v \). All the possible usages are categorized as
follows:

- if \( d \in I \) and \( \{ d \notin O^R \land d \notin O^- \} \), \( v \) is called a Reader (Activity) of artifact \( d \).
- if \( d \in I \) and \( d \in O^R \), \( v \) is called an Updater (Activity) of artifact \( d \).
- if \( d \in I \) and \( d \in O^- \), \( v \) is called a Destroyer (Activity) of artifact \( d \).
- if \( d \notin I \) and \( d \in O^R \), \( v \) is called an Illegal Destroyer\(^{1} \) (Activity) of artifact \( d \).
- if \( d \notin I \) and \( d \in O^- \), \( v \) is called a Producer (Activity) of artifact \( d \).
- if \( d \notin I \) and \( \{ d \notin O^R \land d \notin O^- \} \), \( v \) is called an Irrelevantor (Activity) of artifact
  \( d \).

In addition, if \( d \in I \) \( v \) is generally called a Consumer (Activity)
of artifact \( d \) and if \( d \in O^R \), \( v \) is generally called a Writer (Activity)
of artifact \( d \).

**Definition 3.12 (Consumer, updator, destroyer and producer activity
sets of an artifact).**

- \( \text{Consumer}_d \) = \( \{ v \in V | d \in I \} \) is called the Consumer Activity Set of
  artifact \( d \).
- \( \text{Updater}_d \) = \( \{ v \in V | d \in I \land d \in O^R \} \) is called the Updater Activity
  Set of artifact \( d \).
- \( \text{Destroyer}_d \) = \( \{ v \in V | d \in I \land d \in O^- \} \) is called the Destroyer Activity
  Set of artifact \( d \).
- \( \text{Producer}_d \) = \( \{ v \in V | d \notin I \land d \in O^R \} \) is called the Producer Activity
  Set of artifact \( d \).
- \( \text{Irrelevantor}_d \) = \( \{ v \in V | d \notin I \land d \notin O^R \land d \notin O^- \} \) is called the Reader Activity
  Set of artifact \( d \).

\(^{1}\) The illegal destroyer is not concerned in our model because the activity destroy
artifact arbitrarily. Any useful artifact could be destroyed by the activity during the
workflow execution.

### 4. Artifact usage anomalies

#### 4.1. Artifact usage anomalies

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

#### 4.2. Missing production anomalies

A missing production anomaly occurs when an artifact is con-
sumed before it is produced or after it is destroyed. In order to for-
mulate this type of anomaly, the propagation of an artifact is
introduced in Definition 4.1.

**Definition 4.1 (Propagation of artifacts to an activity).** Given an
activity \( v \), let a preceding execution order to \( v \) denote an execution
order leading to \( v \) without any parallel activities of \( v \), i.e., only
consisting of the predecessors of \( v \). Given an artifact \( a \), if there is at
least one preceding execution order to \( v \) such that \( d \) is produced
but not destroyed yet (i.e., \( d \) is not in Uninitialized state), \( d \) can be
propagated to \( v \). The propagations of artifact \( d \) regarding only the
preceding execution orders to \( v \) are called preceding propagations of
\( d \) to \( v \) and can be classified into three cases: (1) no preceding
propagation, (2) conditional preceding propagation, and (3) uncondi-
tional preceding propagation:

Case (1) indicates that \( d \) is always Uninitialized for all preceding
execution orders to \( v \).

Case (2) indicates whether \( d \) is Uninitialized depends on the pre-
ceding execution orders to \( v \) taken.

Case (3) indicates that \( d \) is Initialized for all preceding execution
orders to \( v \).

Let \( AA_d \) be the set composed of all artifacts which can be prop-
agated from the predecessors of \( v \). \( AA \) can be divided into two dis-
joint subsets, \( AA^c_d \) and \( AA^u_d \), where \( AA^c_d \) contains artifacts propagated
from the predecessors of \( v \) unconditionally and \( AA^u_d \) contains those
propagated from the predecessors of \( v \) conditionally.

The causes of missing production anomalies can be discussed as
follows. Intuitively, if \( v \in V\text{Consumer}_d \) and \( d \notin AA^c_d \) hold, a missing
production anomaly might occur due to No Preceding Propagation
of \( d \) to \( v \). Similarly, if \( v \in V\text{Consumer}_d \) and \( d \in AA^c_d \), a holding, a missing
production anomaly might occur by Conditional Preceding Propagation
of \( d \) to \( v \). Furthermore, considering the parallel activities of \( v \), even

---

**Table 4.1**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_d )</td>
<td>a writer ((d \in O^R))</td>
</tr>
<tr>
<td>( C_d )</td>
<td>a consumer ((d \in I))</td>
</tr>
<tr>
<td>( U_d )</td>
<td>an updator ((d \in I \land d \in O^R))</td>
</tr>
<tr>
<td>( P_d )</td>
<td>a producer ((d \notin I \land d \in O^R))</td>
</tr>
<tr>
<td>( R_d )</td>
<td>a reader ((d \in I \land d \notin O^R \lor d \notin O^-))</td>
</tr>
<tr>
<td>( D_d )</td>
<td>a destroyer ((d \in I \land d \in O^-))</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>reachable link</td>
</tr>
<tr>
<td>( \lor )</td>
<td>XOR control block</td>
</tr>
<tr>
<td>( \land )</td>
<td>AND Control block</td>
</tr>
</tbody>
</table>
though \( v \in V^\text{inConsumer}_d \) and \( d \in AA^u_v \) hold, a missing production anomaly might occur when there is a destroy activity of \( d \) in the parallel activities. The execution order of the parallel activities decides the production of \( d \). This kind of anomaly is named Uncertain Preceding Propagation.

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

1. **No preceding propagation:** \( v \in V^\text{inConsumer}_d \land d \notin AA^u_v \).

   **Usage Pattern 1:** \( R^\text{CD} \rightarrow C_d \rightarrow R^\text{P} \)
   - **Name:** No Production.
   - **Description:** Artifact \( d \) has at least one consumer activity \( v \); however, no producer activity of \( d \) exists in the process.
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land V^\text{Producer}_d = \emptyset \).

   **Usage Pattern 2:** \( R^\text{CD} \rightarrow C_d \rightarrow P_d \rightarrow \)
   - **Name:** Delayed Production.
   - **Description:** Artifact \( d \) has a consumer activity \( v \) which precedes every producer activity of \( d \).
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land (V^\text{Producer}_d \cap V^\text{Producer}_d) = \emptyset \land (V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \).

   **Usage Pattern 3:** \( \rightarrow P_d \rightarrow D_d \rightarrow C_d \rightarrow \)
   - **Name:** Early Destruction.
   - **Description:** Artifact \( d \) is produced and then destroyed before it is consumed.
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \notin AA^u_v \land (\emptyset \land V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \land (V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \).

   **Usage Pattern 4:** \( R^\text{CD} \rightarrow (C_d \otimes P_d) \rightarrow \)
   - **Name:** Exclusive Production.
   - **Description:** Given two exclusive activities \( v \) and \( u \) such that \( v \) is a consumer of artifact \( d \) and \( u \) is a producer of \( d \). Due to the characteristic of exclusive activities, only one of \( v \) and \( u \) might be selected for execution. Although \( u \) is a producer of \( d \), it makes no contribution to the propagation of \( d \) to \( v \) and thus a missing production anomaly occurs.
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \notin AA^u_v \land (\emptyset \land V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \land (V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \).

   **Usage Pattern 5:** \( R^\text{CD} \rightarrow (C_d \otimes P_d) \rightarrow \)
   - **Name:** Uncertain Production.
   - **Description:** Given two parallel activities \( v \) and \( u \) such that \( v \) is a consumer of artifact \( d \) and \( u \) is a producer of \( d \). Due to the race hazard of parallel activities, \( v \) might be executed before \( u \). Therefore, \( u \) may not make contribution to the propagation of \( d \) for \( v \). Consequently, a missing production anomaly occurs if artifact \( d \) will not be propagated from the predecessors of \( v \).
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \notin AA^u_v \land (\emptyset \land V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \land (V^\text{Producer}_d \cap V^\text{Producer}_d) \neq \emptyset \).

2. **Conditional Preceding Propagation:** \( v \in V^\text{inConsumer}_d \land d \in AA^u_v \).

   **Usage Pattern 6:** \( R^\text{CD} \rightarrow P_d \rightarrow R^\text{P} \rightarrow -d \rightarrow \)
   - **Name:** Conditional Production.
   - **Description:** Artifact \( d \) is produced conditionally before a consumer activity of \( d \).
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \in AA^u_v \).

   **Usage Pattern 7:** \( P_d \rightarrow (D_d \otimes R^\text{P}) \rightarrow -d \rightarrow \)
   - **Name:** Conditional Destruction.
   - **Description:** Artifact \( d \) is destroyed conditionally before a consumer activity of \( d \).
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \in AA^u_v \).

3. **Uncertain Preceding Propagation:** \( v \in V^\text{inConsumer}_d \land d \in AA^u_v \).

   **Usage Pattern 8:** \( P_d \rightarrow (D_d \otimes -d) \rightarrow \)
   - **Name:** Uncertain Destruction.
   - **Description:** Given two parallel activities \( v \) and \( u \) such that \( v \) is a consumer of artifact \( d \) and \( u \) is a destroyer of \( d \). Due to the race hazard of parallel activities, \( v \) might be executed before \( u \). \( d \) is unconditionally propagated from the predecessors of \( v \), but \( d \) might be destroyed by \( u \) before \( v \) is executed. A missing production anomaly occurs.
   - **Conditions:** \( \exists v \in V^\text{inConsumer}_d \land d \in AA^u_v \land (V^\text{Parallel}_d \cap V^\text{Destroyer}_d) \neq \emptyset \).

**Theorem 1** (Missing production verification). A process \( BP \) is free from missing production anomalies if the following condition holds: \( \forall \forall v \in V, \forall d \in I_d: d \in AA^u_v \land (V^\text{Parallel}_d \cap V^\text{Destroyer}_d) \neq \emptyset \).

**Proof.** This theorem is proven by contradiction. Firstly, we assume a missing production anomaly in \( BP \). The assumption indicates an activity \( v \in V, \) an artifact \( d \in I_d, \) and an execution order \( I' \) such that \( v \in I' \) and \( d \) is Uninitialized when \( v \) is selected for execution. However, \( d \in AA^u_v \) implies that \( d \) is always propagated from the predecessors of \( v \). Furthermore, \( (V^\text{Parallel}_d \cap V^\text{Destroyer}_d) = \emptyset \) implies that none of parallel activities of \( v \) affects the propagation of \( d \) to \( v \). Thus, no matter what preceding execution order of \( v \) is taken, \( d \) is always propagated to \( v \). It is obvious that \( I' \) does not exist. The fact contradicts the assumption given in the beginning and thus Theorem 1 holds.

4.3. Redundant write anomalies

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

**Definition 4.2.** (The set of artifacts unused before an activity). Given an activity \( v \) and an artifact \( d \), if there is one preceding execution path to \( v \), where \( d \) is written but not consumed, \( d \) is unused before \( v \). If artifact \( d \) is unused for the predecessors of the Process End vertex and is not a member of the set of process outputs, a redundant write anomaly occurs. The anomalous can be divided into two classes: complete and conditional. Artifact \( d \) is completely unused indicates that \( d \) is unused for all preceding paths of \( v \). Artifact \( d \) is conditionally unused indicates that \( d \) is unused in certain preceding path(s) of \( v \), i.e., that an anomaly occurs depending on which preceding path of \( v \) is taken.
Let $NC_v$ be the set composed of all unused artifacts for the predecessors of $v$. $NC_v$ can be divided into two disjoint subsets $NC_u_v$ and $NC_c_v$, where $NC_u_v$ contains completely unused artifacts and $NC_c_v$ contains conditionally unused artifacts.

The causes of redundant write anomalies can be discussed as follows. Intuitively, for every artifact $d \in NC_u_{\text{ProcessEnd}}$ and $d \neq O_w$, an Explicit Redundant Write anomaly always occurs for artifact $d$ of the process. For every artifact $d \in NC_c_{\text{ProcessEnd}}$ and $d \neq O_w$, a potential redundant write anomaly might occur for artifact $d$.

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

1. **Explicit redundant write**
   **Usage Pattern 9**:
   
   - $W_d \rightarrow R$
   - $W_d \rightarrow P$
   - $(W_u \otimes W_d) \rightarrow R$
   - $(W_u \otimes W_d) \rightarrow D$

   - **Name**: No Consumption After Last Write.
   - **Description**: For an artifact $d$, $d$ does not belong to the process outputs, when $d$ is written by an activity $v$ and when the artifact is unused for all succeeding activities of $v$, a redundant write occurs for the artifact.
   - **Conditions**: $\exists d \in NC_{\text{ProcessEnd}} : d \neq O_w$.

2. **Potential redundant write**
   **Usage Pattern 10**:

   - **Name**: No Consumption After Last Write.
   - **Description**: For an artifact $d$, $d$ does not belong to the process outputs, when $d$ is written by an activity $v$ and when the artifact is unused for all succeeding activities of $v$, a redundant write occurs for the artifact.
   - **Conditions**: $\exists d \in NC_{\text{ProcessEnd}} : d \neq O_w$.

![Flowchart](image)

**Fig. 5.1.** E-mail voting process.

**Table 5.1**
Artifacts in the e-mail voting process.

<table>
<thead>
<tr>
<th>Artifacts</th>
<th>Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$ Issue list</td>
<td>$R$ Reader</td>
</tr>
<tr>
<td>$d_2$ Applicant data</td>
<td>$P$ Producer</td>
</tr>
<tr>
<td>$d_3$ Discussion participant data</td>
<td>$D$ Destroyer</td>
</tr>
<tr>
<td>$d_4$ Calendar</td>
<td>$O_w$</td>
</tr>
<tr>
<td>$d_5$ Signature of manager</td>
<td>$W_u$ Updator</td>
</tr>
<tr>
<td>$d_6$ Announce issues</td>
<td>$W_d$</td>
</tr>
<tr>
<td>$d_7$ Comment on announce issues</td>
<td>$W_u \otimes W_d$</td>
</tr>
<tr>
<td>$d_8$ Discussion result</td>
<td>$W_u \otimes W_d$</td>
</tr>
<tr>
<td>$d_9$ Integrated results</td>
<td>$d_9$</td>
</tr>
<tr>
<td>$d_{10}$ Data shared</td>
<td>$d_{10}$</td>
</tr>
<tr>
<td>$d_{11}$ Deadline warning message</td>
<td>$d_{11}$</td>
</tr>
<tr>
<td>$d_{12}$ Conference time</td>
<td>$d_{12}$</td>
</tr>
<tr>
<td>$d_{13}$ Vote participant data</td>
<td>$d_{13}$</td>
</tr>
<tr>
<td>$d_{14}$ Vote result</td>
<td>$d_{14}$</td>
</tr>
<tr>
<td>$d_{15}$ Increment tally</td>
<td>$d_{15}$</td>
</tr>
<tr>
<td>$d_{16}$ Vote final result</td>
<td>$d_{16}$</td>
</tr>
</tbody>
</table>
Name: Conditional Consumption After Last Write.
Description: For an artifact \(d\), \(d\) does not belong to the process outputs. When \(d\) is written by an activity \(v\) and the artifact is unused for some succeeding activities of \(v\) conditionally, a redundant write might occur for the artifact.
Conditions: \(\exists d \in NC_{\text{ProcessEnd}} : d \notin O_v\).

**Theorem 2** (Redundant write verification). A process \(BP\) is free from the redundant write anomalies if \(NC_{\text{ProcessEnd}} \cap O_v = \emptyset\) holds.

**Proof.** \(NC_{\text{ProcessEnd}} \cap O_v = \emptyset\) indicates that artifacts produced in the process either contribute to the process output directly after its last write \((d \in O_v)\) or is read/destroyed after its last write on all possible (preceding) execution orders leading to \(\text{Process End}\) \((d \notin NC_{\text{ProcessEnd}})\). Therefore, no redundant write anomaly exists if \(NC_{\text{ProcessEnd}} \cap O_v = \emptyset\) holds. \(\square\)

4.4. Conflict writes anomalies

The conflict writes anomalies can be divided into three classes: (1) multiple parallel productions, (2) multiple parallel updates and (3) parallel read and update. An anomaly of multiple parallel productions occurs when more than one activity tries to initialize the same artifact in parallel. An anomaly of multiple parallel updates occurs when more than one activity updates the same artifact in parallel. An anomaly of parallel read and update anomaly occurs when two activities perform read and update on the same artifact concurrently. Each of these anomalies corresponds to the execution order. The anomalies make the artifact version hard to control.

**Usage Pattern 11:** \(\neg (P_d \oplus P_d)\)
- Name: Multiple Parallel Productions.
- Description: More than one activity initializes the same artifact in parallel.
- Conditions: \(\exists v \in V_{\text{Producer}} \land (V_{\text{Producer}} \cap V_{\text{Parallel}}) \neq \emptyset\).

**Usage Pattern 12:** \(\neg (U_d \oplus U_d)\)
- Name: Multiple Parallel Updates.
- Description: More than one activity updates the same artifact in parallel.
- Conditions: \(\exists v \in V_{\text{Producer}} \land (V_{\text{Producer}} \cap V_{\text{Parallel}}) \neq \emptyset\).

**Usage Pattern 13:** \(\neg (P_d \oplus U_d)\)
- Name: Parallel Read and Update.
- Description: Two activities perform read and update respectively on the same artifact concurrently.
- Conditions: \(\exists v \in V_{\text{Producer}} \land (V_{\text{Producer}} \cap V_{\text{Parallel}}) \neq \emptyset\).

**Theorem 3** (Conflict writes verification). A process \(BP\) is free from the anomalies of conflict writes if for any two parallel activities \(v\) and \(u\), \((O^+_v \setminus I_v) \cap (O^+_u \setminus I_u) = \emptyset\), \((O^+_v \cap I_v) \cap (O^+_u \cap I_u) = \emptyset\), \(I_v \cap (O^-_u \cap I_u) = \emptyset\), and \(I_u \cap (O^-_v \cap I_v) = \emptyset\) hold.
Table 5.2
Artifacts usages in the e-mail voting process.

<table>
<thead>
<tr>
<th></th>
<th>d1</th>
<th>d2</th>
<th>d3</th>
<th>d4</th>
<th>d5</th>
<th>d6</th>
<th>d7</th>
<th>d8</th>
<th>d9</th>
<th>d10</th>
<th>d11</th>
<th>d12</th>
<th>d13</th>
<th>d14</th>
<th>d15</th>
<th>d16</th>
</tr>
</thead>
<tbody>
<tr>
<td>p5</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td>d12</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td></td>
<td></td>
<td>x13</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>R</td>
<td>R</td>
<td>U</td>
<td>P</td>
<td>P</td>
<td>d14</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td></td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td>d15</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d1</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d16</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d2</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d3</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d4</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d5</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d6</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d7</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d8</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d9</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d10</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d11</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d12</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d13</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d14</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d15</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d16</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.3
Steps to detect missing production anomalies.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analyze data</td>
</tr>
<tr>
<td>2</td>
<td>Identify anomalies</td>
</tr>
<tr>
<td>3</td>
<td>Apply corrections</td>
</tr>
<tr>
<td>4</td>
<td>Verify results</td>
</tr>
</tbody>
</table>

5. The methods to detect artifact usage anomalies

The methods for detecting the artifact usage anomalies in a process specification are presented in this section. The goal in our study is to search the artifact usage anomaly only, and it is not necessary to construct the possible artifact activities in each process. Instead, loop can be replaced with a corresponding XOR structure. From the top-level view, a well-formed control flow can be deemed as one or a sequence of task(s) and/or top-level control block(s). Thus, an entire flow can be deemed as a sequence of nodes in which each node represents a task or a control block which represents a group of sequential flows. The same perspective can also be applied to the branches of the control block(s). In our approach, a business process is transformed into a sequence of nodes before applying the detection methods. Each method in Section 5.2 is aimed at detecting a type of artifact usage anomalies identified in Section 4.

5.1. Process transformation

Let control flow $G = (V, E)$ of a business process be transformed into a sequence of nodes $S$. The data structure of $S$ and the nodes within $S$ are defined in Definition 5.1.

Definition 5.1 (The data structure of a sequence and a node). $S$: a structure containing a sequence of nodes (a node could represent a task or a control block) node: a node is a structure denoted with type, startVertex, endVertex and subSequences

<table>
<thead>
<tr>
<th>node.type</th>
<th>the type of a task or control block node.startVertex</th>
<th>the start vertex of a control block node.endVertex</th>
<th>the end vertex of a control block node.subSequences</th>
<th>a set of sequences attached to the node</th>
</tr>
</thead>
</table>

The transformation is designed to convert a control flow enclosed by ProcessStart and ProcessEnd vertices. The initial value of the level attribute for each vertex in $V$ is 0. An empty sequence is declared for the control flow in the beginning. During the transformation, a node is created for each task and control block visited. The task node is appended to the sequence directly. Once a split activity $v$ reached, its corresponding control block is identified by function SetLevel which traverses a path with the vertexes enclosed by $v$ and its corresponding join activity $j$ in $G$. During the traverse, the level attributes of the vertexes traversed are updated. The transformation is applied to the block identified. Such a recursive operation continues until all activities in the block are processed. The corresponding sequence of node(s) generated for each branch of a control block is attached to the node created for their own control blocks when the transformation completes at the end of the branch. The details of transforming a task and a control block are shown in PseudoCode1.
PseudoCode 1 TransformControlBlock(G, v, level, controlNodes)

// Input: G=(V,E): a directed connected graph
// v: the traverse is started from vertex v.
// level: the level of start vertex v
// controlNodes: a stack containing a set of pairwise control activities bounded control blocks
// Output: S: a structure containing a sequence of nodes

Stack currentControlNodes = new Stack();
S.startVertex = currentVertex = v;

while (currentVertex != null) {
    if (currentVertex.type != "XorJoin") {
        currentControlNodes.push(currentVertex.level, currentVertex);
        currentVertex = currentVertex.next;
    } else if (currentVertex.type != "AndJoin") {
        currentControlNodes.pop(currentVertex.level);
        currentVertex = currentVertex.next;
        break;
    } else if (currentVertex.type != "EndProcess") {
        newNode = new Node();
        newNode.startVertex = currentVertex;
        newNode.type = currentVertex.type;
        newNode.subSequences.append(newNode);
        currentVertex = currentVertex.next;
        break;
    }
}

return controlNodes;

5.2. Anomaly detection methods

The methods for detecting the artifact usage anomalies in a process specification are presented in this section, named DetectMissingProduction, DetectRedundantWrite and DetectConflictWrites, respectively. Each is aimed at detecting a type of artifact usage anomalies identified in Section 4. The details of these methods are shown in the following subsections. An example described in Section 5.2.1.3 is adopted to demonstrate how they work correspondingly.

5.2.1. Method for detecting missing production anomalies

5.2.1.1. Calculation of propagated artifacts from predecessors. Given a sequence S of the process derived from transformation, let SAA_v denote the set of artifacts propagated from the predecessors of activity v and SAA_v^0 be the set of artifacts of which each can be propagated to the direct successors of v after the execution of v. At the top level, SAA_{startVertex} = I_o for the starting node of S. During the traverse of sequence S, when a node n is reached, SAA is calculated as follows:

- If n represents a task activity \( v \) with only one direct successor \( x \), SAA_v SAA_x and SAA_v are calculated as follows:
  - For each destroyed artifact \( d \in I_v \cap Q^+_v \), remove \( d \) from SAA_v^0 or SAA_x^0, where \( d \) is included.
  - For each produced artifact \( d \in (O_v \setminus I_v) \), add \( d \) to SAA_v^0 and remove \( d \) from SAA_v if \( d \) is included.
$S.AA_n = S.AA_n^i = \begin{cases} S.AA_n^u = (S.AA_u \setminus (I_n \cap O_n)) \cup (O_n \setminus I_n) \\ S.AA_n^e = (S.AA_e \setminus (I_n \cup O_n)) \setminus (O_n \setminus I_n) \end{cases}$

- If $n$ represents a control block with subsequences $SS = (SS_1, SS_2, \ldots, SS_k)$ and $k \geq 2$, each vertex within the block will be recursively traversed as follows:
  - The traverse starts from $n.startVertex$ which is the start vertex of the control block.
    - $S.AA_{n.startVertex} = S.AA_{n.startVertex}$, since $n.startVertex$ is the start vertex of the control block.
  - For each subsequence $SS_i$, apply method DetectMissingProduction recursively to calculate each $SS_i.AA$.
  - The traverse terminates on $n.endVertex$ which is the end vertex of the control block. All $SS_i.AA$ are merged according to the type of the control block.
    - If $n$ is an XOR control block,
      \[
      S.AA_{n.endVertex}^i = S.AA_{n.endVertex}^u = \bigcup_{i=1}^{k} SS_i.AA_{SS_i.endVertex} \setminus S.AA_{n.endVertex}^u \]
    - If $n$ is an AND control block,
      \[
      S.AA_{n.endVertex}^i = S.AA_{n.endVertex}^u = \bigcup_{i=1}^{k} SS_i.AA_{SS_i.endVertex} \setminus S.AA_{n.endVertex}^u \]

5.2.1.3. An example: a process of resolving issues through e-mail votes. To demonstrate the three analysis methods proposed in Section 5, this subsection introduces a process for resolving issues through e-mail votes (White, 2008) as an example. The methods presented are applied on this example to illustrate the steps to detect the artifact usage anomalies. Fig. 5.1 shows the control flow graph of the e-mail voting process where the artifacts are stated with details in Table 5.1. The result of representing the example with our process model is shown in Fig. 5.2. The artifact usages of activities are listed in Table 5.2.

5.2.1.4. Detection of missing production anomalies. Table 5.3 shows the calculation of propagated artifacts according to the activity taken order of the e-mail voting process introduced in Section 5.2.1.3. During detection, the method processes each activity once only and generates two sets of artifacts on each activity. The rules defined are applied on the calculated results of the operated activity recursively. The anomaly detection is detected as follows:

- If $n$ represents a task activity $\nu$, $S.NC^\nu_n = \bigcup_{i=1}^{k} SS_n.NC^\nu_{SS_i.endVertex}$.
  - For each read or destroyed artifact $d \in I_n \cup I_n \cap O_n$, remove $d$ from $NC^\nu_n$ and $NC^\nu_{SS_i}$.
  - For each produced or updated artifact $a \in O_n$, add $a$ to $NC^\nu_n$ and remove $d$ from $NC^\nu_n$.

- If $n$ represents a control block with subsequences $SS = (SS_1, SS_2, \ldots, SS_k)$ and $k \geq 2$, method DetectRedundantWrite is recursively applied to calculate $SS_i.NC$ of branches and then merge $SS_i.NC$ to $SS_NC$ according to the type of the control block.
  - If $n$ is an XOR control block,
    \[
    S.NC^u_{n.endVertex} = S.NC_{n.endVertex} = \bigcup_{i=1}^{k} SS_i.NC^u_{SS_i.endVertex} \setminus S.NC^u_{n.endVertex} \]
    - the unconditional set $S.NC^u_{n.endVertex}$ is constituted with two parts:
      - the intersection $\bigcap_{i=1}^{k} SS_i.NC^u_{SS_i.endVertex}$ of the sets of artifacts unused before the end vertexes of subsequence $SS_i, 1 \leq i \leq k$, and
      - the artifacts in $\bigcup_{i=1}^{k} SS_i.NC^u_{SS_i.endVertex} \setminus S.NC^u_{n.startVertex}$ which contains the ones produced in subsequences without using by the parallel activities before the end vertex of $n$. The case of an artifact reproduced after the last use in branch $SS_i$ is concerned.

- If $n$ is an AND control block,
  - the unconditional set $S.NC^u_{n.endVertex}$ is constituted with two parts:
    - the intersection $\bigcap_{i=1}^{k} SS_i.NC^u_{SS_i.endVertex}$ of the sets of artifacts unused before the end vertexes of subsequence $SS_i, 1 \leq i \leq k$, and
    - the artifacts in $\bigcup_{i=1}^{k} SS_i.NC^u_{SS_i.endVertex} \setminus S.NC^u_{n.startVertex}$ which contains the ones produced in subsequences without using by the parallel activities before the end vertex of $n$. The case of an artifact reproduced after the last use in branch $SS_i$ is concerned.
5.2.2.2. Rules for detecting redundant write anomalies. After visiting the endVertex of the top level sequence S, i.e. the end vertex of the process, the redundant write anomalies defined in Section 4.3 can be detected with the following rules:

- **Explicit redundant write**
  - If \( EC = NC^C_{endVertex} \setminus O_w \neq \emptyset \), a redundant write anomaly occurs for every artifact \( d \in EC \) due to No Consumption After Last Write.

- **Potential redundant write**
  - If \( CC = NC^C_{endVertex} \setminus O_w \neq \emptyset \), a redundant write anomaly occurs for every artifact \( d \in CC \) due to Conditional Consumption After Last Write.

5.2.2.3. Detection of redundant write anomalies. Table 5.4 shows the steps to calculate the set of unused artifacts for each activity in the e-mail voting process introduced in Section 5.2.1.3. During calculation, the method processes each activity once only and generates two sets of artifacts on each activity.

In this case, the redundant write anomalies detected belong to one of the following classes:

- **Explicit redundant write** \( EC = NC^C \setminus O_w = \{d_5, d_7, d_{11}, d_{14}\} \setminus \{d_{16}\} \) is not empty and thus, a redundant write anomaly occurs for every artifact \( d \in EC \) due to Completely Unused for the Process.

- **Potential redundant write** \( CC = NC^C \setminus O_w = \{d_{12}\} \setminus \{d_{12}\} \) is not empty and thus, a redundant write anomaly occurs for \( d_{12} \) due to Conditional unused for the process.

5.2.3. Method for detecting conflict write anomalies

5.2.3.1. Calculation of conflict writes. Given a sequence S of the process derived from transformation, let S.PA, S.UA and S.RA be the sets of artifacts produced, updated and read respectively within sequence S. Initially, the three sets are empty. During the traversal of sequence S, when a node n is reached, S.PA, S.UA and S.RA are calculated as follows:

- If \( n \) represents a task activity \( v \), \[
\begin{align*}
S.PA &= S.PA \cup (O^n_v \setminus I^n_v), \\
S.UA &= S.UA \cup (O^n_v \setminus I^n_v), \\
S.RA &= S.RA \cup (I^n_v \setminus O^n_v).
\end{align*}
\]
  - For every artifact d produced by \( v \), i.e. \( d \in (O^n_v \setminus I^n_v) \), add d to S.PA.
  - For every artifact d updated by \( v \), i.e. \( d \in (O^n_v \setminus I^n_v) \), add d to S.UA.
  - For every artifact d read by \( v \), i.e. \( d \in (I^n_v \setminus O^n_v) \), add d to S.RA.

- If \( n \) represents a control block with subsequences \( SS = \{SS_1, SS_2, \ldots, SS_k\} \) and \( k \geq 2 \), the method is recursively applied to each subsequence. Then, S.PA, S.UA and S.RA belong to the subsequence and are merged according to the following rules:

\[
\begin{align*}
S.PA &= S.PA \cup \bigcup_{i=1}^{k} SS_i \cdot PA, \\
S.UA &= S.UA \cup \bigcup_{i=1}^{k} SS_i \cdot UA, \\
S.RA &= S.RA \cup \bigcup_{i=1}^{k} SS_i \cdot RA.
\end{align*}
\]

5.2.3.2. Rules for detecting conflict write anomalies. For an And control block with subsequences \( SS = \{SS_1, SS_2, \ldots, SS_k\} \) and \( k \geq 2 \), the conflict write anomalies defined in Section 4.4 can be detected with the following rules:

- **Multiple parallel productions**
  - Before merging S.PA, PA if \( \exists i, j \leq 1 \leq i \leq k \land i \neq j \land (MPA_{SS_i, SS_j} = SS_i \cdot PA \cap SS_j \cdot PA \neq \emptyset) \) a conflict writes anomaly may occur for every artifact \( d \in MPA_{SS_i, SS_j} \) due to multiple parallel productions.

- **Multiple parallel updates**
  - Before merging S.UA, UA if \( \exists i, j \leq 1 \leq i \leq k \land i \neq j \land (MUA_{SS_i, SS_j} = SS_i \cdot UA \cap SS_j \cdot UA \neq \emptyset) \) a conflict writes anomaly may occur for every artifact \( d \in MUA_{SS_i, SS_j} \) due to multiple parallel updates.

- **Parallel read and update**
  - Before merging S.UA and S.RA, RA if \( \exists i, j \leq 1 \leq i \leq k \land i \neq j \land (SS_i \cdot RA \cap SS_j \cdot UA \neq \emptyset) \), then a conflict writes anomaly may occur for every artifact \( d \in SS_i \cdot RA \cap SS_j \cdot UA \) due to parallel read and update.

5.2.3.3. Detection of conflict write anomalies. Table 5.5 shows the steps to calculate the sets of artifacts produced, updated and read by activities in the e-mail voting process introduced in Section 5.2.1.3. During calculation, the method processes each activity once only and generates three sets of artifacts on each activity.

In this case, the conflict write anomalies detected belong to one of the following classes:

- **Multiple parallel productions**
  - MPA_{SS_i, SS_j} = (PA_{SS_i} \cap PA_{SS_j}) \cap (PA_{SS_i} \cap PA_{SS_j}) = \emptyset \) and MPA_{SS_i, SS_j} = \emptyset \) are empty and thus, no conflict write anomaly occurs because of multiple parallel productions.

- **Multiple parallel updates**
  - MUA_{SS_i, SS_j} = (UA_{SS_i} \cap UA_{SS_j}) \cap (UA_{SS_i} \cap UA_{SS_j}) = \emptyset \) and MUA_{SS_i, SS_j} = \emptyset \) are empty and thus, no conflict write anomaly occurs due to multiple parallel updates.

- **Parallel read and update**
  - SS_i \cdot UA \cap SS_j \cdot RA = (UA_{SS_i} \cap UA_{SS_j}) \cap (RA_{SS_i} \cap RA_{SS_j}) = \{d_8\} \) is not empty and thus, a conflict write anomaly occurs for \( d_8 \) due to parallel read and update.

6. Comparisons

6.1. Comparisons of anomalies identified in the approaches

The artifact usage anomalies identified in this paper are classified according to the classification proposed by Sun and Zhao (2004), Sun et al. (2004, 2006) who claimed that the definition of the types of artifact usage anomalies – missing data, redundant data, and conflicting data – is sufficient in analyzing the data flow at a conceptual level. Each anomaly, addressed by the two groups, Sadiq et al. (2004) and Sun and Zhao (2004), Sun et al. (2004, 2006), working on the analysis in a data-flow dimension, can be found a correspondence in our model. The mapping relations are shown in Table 6.1. Besides, Theorems 1–3 can be applied to help avoid the anomalies.

However, Sadiq et al. (2004), Sun and Zhao (2004), Sun et al. (2004, 2006) are not concerned with destroy operations, each of which is essential for deleting the artifact instance which is not required by the succeeding activities or a member of the process outputs. In their approaches, a missing production anomaly caused by an activity destroying an artifact before it is consumed cannot be identified. A lack of recognizing the critical anomalies endanger the accuracy of analyzing the workflow execution result. The destroy operations are concerned in our previous work (Hsu and Wang, 2007), but the anomalies are presented conceptually. There are three distinct anomalies – early destruction, conditional destruction, and uncertain destruction – caused by destroying an artifact requested in a succeeding process, formulated here.
In addition, the anomalies happening within a control block are illustrated conceptually in the methods proposed in Sadiq et al. (2004), Hsu and Wang (2007). There is no formal formulation for systematically discovering the artifact usage anomalies in a workflow model. Furthermore, the cases of causing conflict write anomalies: more than one activity initializes or updates an identical artifact in parallel or two activities perform read and update on the same artifact concurrently, are partially addressed in Sadiq et al. (2004), Sun and Zhao (2004), Sun et al. (2004, 2006). These anomalies are identified and formulated in this paper also. Regarding the conflict write anomalies, the comparison between the anomalies formulated in our method and those identified in Sun et al. (2006), Sadiq et al. (2004) and Hsu and Wang (2007) are shown in Table 6.2.

### 6.2. Comparison of the detection methods

Our previous work creates a table for each artifact to display the artifact usage. In contrast with the table, we utilize another presentation, e.g. Table 5.2, to show the operation of each activity performing on each artifact. Comparing with the distributed tables, it is easier to identify how each artifact is processed in a business process with the integrated presentation.

Sun et al. (2006) proposed a process data diagram through extending the UML activity diagram. The extension enriches the presentation power of an activity diagram to support artifact usage analysis. However, the irrelevant information could be represented in a diagram to interfere in the artifact usage analysis. Comparing to the extension, the methods published in Son and Kim (2005), Chang et al. (2002), Hsu and Wang (2007) are defined with their own notations. These custom-made notations are constructed without explaining the reasons of redefining a new one which corresponds to a specific one specified in BPMN. Adopting the BPMN notation instead of using BPMN is another case of redefinition. Our process model is defined by ridding unrelated elements and keeping related ones of the BPMN for fitting the model to analyze artifact usage. Three advantages obtained from the adaptation are that: (1) using a standardized graphical notation BPMN keeps the understandability held by it, (2) simplifying the arrangement of structures in a model reduces the analysis complexity and (3) eliminating the elements irrelevant to artifact usage analysis diminishes the possibility of design errors.

In addition, our detection methods work on a sequence of nodes transformed from a business process. Let the process specification be $BP = (G, VT, D, IW, OW)$. The transformation mechanism goes through each vertex and edge of $G(V,E)$ twice only to...
transform the graph into a sequence of nodes. Thus, the complexity of the transformation is \(O(|V| + |E|)\). Existing approaches (Sun et al., 2006, Hsu and Wang, 2007) detect the anomalies defined upon critical paths, but the computing time required for calculating critical paths is \(O(V^2)\) more than the one of the transformations.

Our detection methods are assigned individually to detect a particular type of defined artifact usage anomalies over the sequence of nodes. During detection, the methods process each node once only and generate two or three sets of artifacts on each node. The maximum storage demand of the set is \(|D|\), a bit-array is utilized for representing each element of the sets. Based on the data structure, time consumption of the operations, such as union, intersection, and complement, working on the elements is seen as constant.

If the number of nodes in \(BP\) is \(n\), method DetectRedundantWrite can detect the anomaly in \(O(n)\) time. If the destroy operation is not considered in method DetectMissingProduction, the time required by detecting missing production anomalies is \(O(n)\) also. Otherwise, involving destroy operation, checking the missing production anomalies on each pair of branches is requisite for each AND control block. Thus, with destroy operations, the time required by executing method DetectMissingProduction is \(O(n^2)\). In addition, checking the conflict write anomalies on each pair of branches is requisite for each AND control block, thus the time required by executing method DetectConflictWrites is \(O(n^2)\) time in previous approaches.

7. Conclusion and future work

Introducing an artifact usage analysis technique into workflow design phase is the main contribution of this paper. This paper presents a business process model for describing a business process and analyzes the artifact usages on this model to achieve this goal. Artifact usage in our model is characterized by its state transition diagram.

This work identifies thirteen cases of improper artifact usage affecting workflow execution and categorizes these anomalies into three types. The anomalies are identified by considering the destroy operations which are lack of concern before. A set of methods for discovering these anomalies is also presented. The execution time required by the set of methods is less than the methods proposed before. An example is demonstrated the usability of the proposed methods.

We currently continue our research in several directions. First, we plan to implement the proposed model and methods on current workflow management systems, such as Agentflow Flowring Technology Corp. (2006), so that our research result can be tested in real-world applications. Second, we continue the analysis on composite artifacts with more complex usages using Revise operations. Thirdly, we integrate resource constrains analysis techniques with our work to build a practical workflow design methodology.

References

Alves, A. et al., 2007. Web Services Business Process Execution Language 2.0, OASIS WSBPEL TC.


Ching-Huey Wang received the MS degree in computer science from Tung Hai University, Taiwan, in 2002. She is a PhD student of computer science at Chiao Tung University, Taiwan. Her research interests include service-oriented application modeling and analysis, social network service, and pervasive workflow application. She has published several papers in international journals and conferences. She is a member of the IEEE Computer Society.

Prof. Feng-Jian Wang completed his Ph.D. program in Dept. of E.E.C.S., Northwestern University, 1988. Since then, he worked in National Chiao-Tung University, Taiwan. During his Ph.D program, he worked on incremental analysis of data flow. Thereafter, he worked on the development, reuse, and data analysis based no object-oriented programming language. Since 1995, he worked on the analysis and design of workflow programs and his laboratory constructed a workflow management system named Agentflow, where he studied a series of supporting analysis and tools associated with editing activities. Currently, he is focused on how to apply workflow and grid computing techniques on pervasive workflow systems, and developing the design patterns of FLASH programs.