Analysing inaccurate artifact usages in workflow specifications

C.-L. Hsu and F.-J. Wang

Abstract: Although many workflow models have been proposed, analyses on artifacts are seldom discussed. A workflow application with well structured and adequate resources may still fail or yield unexpected results in execution due to inaccurate artifact manipulation, for example, inconsistency between data flow and control flow, or contradictions between artifact operations. Thus, artifact analysis is very important since activities cannot be executed properly without accurate information. This paper presents a three-layer workflow model for designing a workflow and characterises the behaviour of an artifact by its state transition diagram. By abstracting common usages of artifacts, six types of inaccurate artifact usage affecting workflow execution are identified and a set of algorithms to detect these inaccurate usages in workflow specifications is presented. An example is demonstrated and then related works are compared.

1 Introduction

Workflow can be viewed as a set of tasks that is systematised to achieve certain business goals by completing each task in a particular order under automatic control [1]. Resources are required for workflow implementation and to support process execution. Resource allocation and resource constraint analysis [2–6] are popular workflow research topics. However, data flow within workflow is seldom addressed [7, 8].

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

This work proposes an editing model for designing a workflow and addresses six types of artifact inaccuracy. An artifact usage analysis procedure associated with the model is applied before deploying the workflow schema. Reports of consistency checking between data flow and control flow and information of manipulating artifacts are automatically provided to designers when they edit or adjust workflow specification. The model is based on a component-based design technique [9, 10] and is compatible with existing control-oriented workflow design models. It provides an easier way to extract knowledge of artifact operations in a workflow. In our earlier work [11, 12], we have introduced the artifact usage analysis into workflow design phase and six types of inaccurate artifact usages affecting workflow execution have been identified. In this paper, the artifact usages are formularised and the concrete algorithms to discover the inaccurate usages in workflow specifications are proposed. In addition, an example to demonstrate the contribution of our work and a comparison among related works are presented.

The remainder of this paper is organised as follows. Section 2 presents the research background and related work. Section 3 presents our system architecture, including the workflow design methods and design criteria. Section 4 then defines certain properties of artifacts, introduces a technique of artifact state diagram to describe artifact state transition, and discusses six types of artifact inaccuracy. Section 5 presents algorithms for constructing artifact state diagrams. Section 6 proposes a set of algorithms to detect artifact accuracy in a workflow schema. Section 7 demonstrates the algorithms through an example. Section 8 compares our approach with related works. Conclusions are finally drawn in Section 9, along with recommendations for future work.

2 Related work

A workflow can be deemed as a collection of cooperating and coordinated activities designed to carry out a well-defined complex process, such as a trip planning, conference registration procedure, or business process in an enterprise. A workflow model is used to describe a workflow in terms of various elements, such as roles and resources, tools and applications, activities and data, which represent different perspectives of a workflow [13, 14]. Roles and resource elements represent organisational perspective that describes where and by whom tasks are performed and available resource tasks can be utilised in the organisation. 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) behavioural, when and how tasks are performed. Data elements represent the informational perspective, that is, what information entities are produced or manipulated in the corresponding activities in a workflow.

doi:10.1049/iet-sen:20070002

Paper first received 4th January and in revised form 28th June 2007

The authors are with the Department of Computer Science, National Chiao Tung University, Room 510, EC Building, 1001 Ta-Hsueh Road, Hsinchu City, Taiwan, People's Republic of China

E-mail: fjwang@cs.nctu.edu.tw

A well-defined workflow model leads to the efficient development of an effective and reliable workflow application. The correctness issues in a workflow might be classified into three dimensions: control flow, resource and data flow. Generally, the analyses in control-flow dimension are focused on correctness issues of control structure in a workflow. The common control-flow anomalies include deadlock, livelock (infinite loop), lack of synchronisation and dangling reference [15–19]. A deadlock anomaly occurs if it is no longer possible to make any progress for a workflow instance, for example, synchronisation on two mutually exclusive alternative paths. A livelock anomaly indicates an infinite loop, such as an iteration without a possible exit condition, which causes a workflow to make continuous progress, however, without progressing towards successful completion. A lack of synchronisation anomaly represents the case of more than one incoming node merging into an `OR-JOIN` node. Activities without termination or without activation are two common cases of dangling reference anomaly.

Activities belonging to different workflows or parallel activities in the same workflow might access the same resources. A resource conflict occurs when these activities execute over the same time interval. Thus, the analyses in resource dimension include the identification of resource conflicts under resource allocation constraints and/or under the temporal and/or causality constraints [2–6]. On the other hand, missing, redundancy and conflict use of data are common anomalies in data-flow dimension [7, 8]. A missing data anomaly occurs when an artifact is accessed before it is initialised. A redundant data anomaly occurs when an activity produces an intermediate data output that is not required by any succeeding activity. A conflicting data anomaly represents the existence of different versions of the same artifact.

Current workflow modelling and paradigm analysis are mainly focused on the soundness of control logic: in the control-flow dimension, including process model analysis [17–20], workflow patterns [21–23] and automatic control of workflow process [24]. Aalst and ter Hofstede [17] proposed a WorkFlow net (WF-net), based on Petri nets, to model a workflow: transitions representing activities, places representing conditions, tokens representing cases and directed arcs connecting transitions and places. Furthermore, control-flow anomalies, such as deadlock, livelock and dangling reference (activities without termination or activation) have been identified through Petri net modelling and analysis. Son [25] defined a well-formed workflow based on the concepts of closure and control block. He claimed that a well-formed workflow is free from structural errors and that complex control flows can be made with nested control blocks. Son [25] and Chang [26] identified and extracted the workflow critical path from the context of the workflow schema. They proposed extraction procedures from various non-sequential control structures to sequential paths, thus obtaining appropriate sub-critical paths in non-sequential control structures. Sadiq and Orlowska [20] proposed a visual verification approach and algorithm with a set of graph reduction rules to discover structural conflicts in process models for given workflow modelling languages.

There are several research topics discussed in resource dimension, including resource allocation constraints [2, 3], resource availability [4], resource management [5] and resource modelling [6]. Senkul [2] developed an architecture to model and schedule workflow with resource allocation constraints and traditional temporal/causality constraints. Li [3] concluded that a correct workflow specification should have resource consistence. His algorithms can verify resource consistency and detect the potential resource conflicts for workflow specifications. Both Pinar and Hongchen extended workflow specifications with constraint descriptions. Liu [4] proposed a three-level bottom-up workflow design method to effectively incorporate confirmation and compensation in case of failure. In Liu's model, data resources are modelled as resource classes, and the only interface to a data resource is via a set of operations.

The above approaches do not put emphasis on the data-flow dimension. The analysis in data-flow dimension is very important since activities cannot be executed properly without sufficient information. However, the researches related to analysis in data-flow dimension of a workflow are seldom. In the literature, there are two significant works found in data-flow dimension. Sadiq *et al.* [7] presented data-flow validation issues in workflow modelling, including identifying requirements of data modeling and seven basic data validation problems: redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data and insufficient data. However, no concrete verification procedures have been suggested. Sun *et al.* [8] presented a data-flow analysis framework for detecting data-flow anomalies such as missing, redundant data and potential conflicts of data. In addition, concrete analysis algorithms are also provided in their works. However, only read and first-initial-write operations of an artifact are considered.

3 Proposed workflow model: TLWM

The proposed model for workflow design processes is named the Three-Layer Workflow Model (TLWM) and is illustrated in Fig. 1. A workflow development, excluding the role and other software, can generally be distinctly interpreted in TLWM with the following three layers.

1. Workflow Design Layer describes the logistical control or execution order between activities, such as the sequence, choice, synchronisation and iteration.
2. Activity Design Layer arranges the artifact operations to be performed in an activity and the conditions to be tested at the initialisation and completion of the activity.
3. Artifact Design Layer defines the classes of artifacts and all valid methods/operations of each artifact class.

3.1 Workflow design layer

The Workflow Design Layer adopts a model to describe a schema based on the concept of well-formed workflows [25]. The product of this layer is a specification that is designed to describe the dependency between activities, that is, the execution order of activities in a workflow, which is determined by Control structures. The four primitive control structures defined in [1] are 'sequential', 'parallel', 'conditional branch' and 'iterative structure'.

This work describes a workflow specification based on these structures. The basic unit of work is an activity, which has pre-conditions (entry criteria), post-conditions (exit criteria) and actions with artifact operations and resource manipulation. The Control structures have paired nodes. This work defines eight primitive control nodes (four pairs). (1) `START-NODE` (SN) and `END-NODE` (EN) are special control nodes that represent the start and the end of a workflow, respectively. (2) `AND-SPLIT`

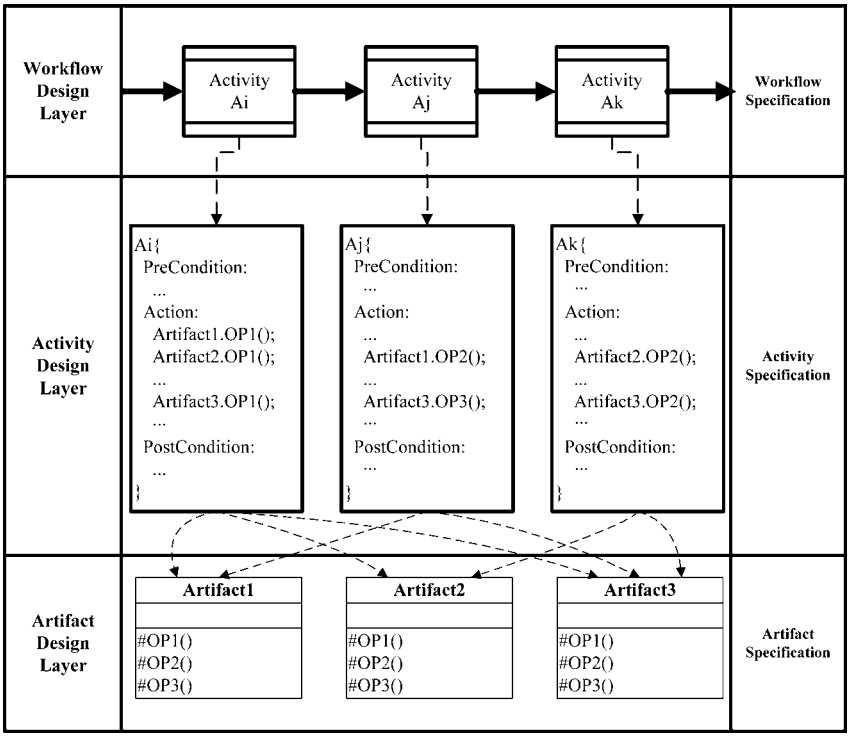


Fig. 1 *Three layer workflow model*

(AS) and AND-JOIN (AJ) are control nodes that construct a parallel structure. (3) XOR-SPLIT (XS) and XOR-JOIN (XJ) are control nodes that construct a branch structure. (4) LOOP-START (LS) and LOOP-END (LE) are control nodes that represent an iteration structure. The latter three control structures are called control blocks. Control blocks are self-contained and no two control blocks overlap with each other. Fig. 2 shows the corresponding notations of control nodes, activity node and flow.

According to the definitions mentioned above, the proposed primitive workflow schema defines four types of workflow constructions as follows.

1. Sequential Block: The activities in this block are executed in sequence under a single thread. The main characteristic is that the target activity executes after its preceding activity is completed. In other words, the completion of a target activity triggers the execution of its succeeding activity.
2. Iteration Control Block: The activities within the control block grouped by loop control nodes are executed repetitively until certain conditions are met.
3. AND Control Block: All outflows of an AND-SPLIT node are executed in parallel and converge synchronously into an AND-JOIN node. An AND-JOIN node synchronises all threads from parallel inflows into one thread.
4. XOR (eXclusive OR) Control Block: An XOR-SPLIT node decides which branches to take from multiple alternative outflows (workflow branches). These branches converge to a single XOR-JOIN node. No synchronisation is required since only one thread is chosen for execution.

Fig. 3 shows four types of workflow construction that can contribute to a primitive workflow schema. Additionally, the following sections use the notation '(leading node, ending node)' to indicate a block, starting from the leading node and terminating at the ending node.

3.2 Activity design layer

Activity specifications are anticipative products of this layer. Each activity has its own specification, which describes the operations associated with artifacts required to be performed in order to achieve the goal. All operations must be included in the corresponding specifications. Additionally, each activity may be designed with pre- and post-conditions if required. Artifact(s) acted on and/or concerned in pre-/post-condition in this activity are retrieved to contribute to artifact usage analysis.

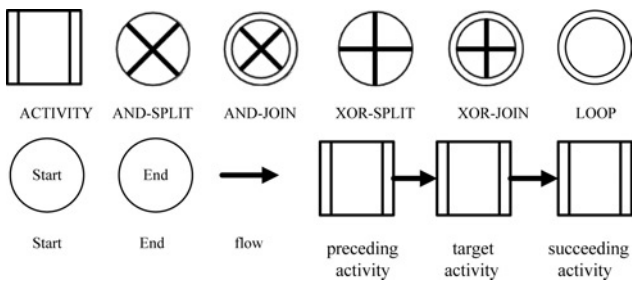


Fig. 2 *Notations of primitive workflow schema*

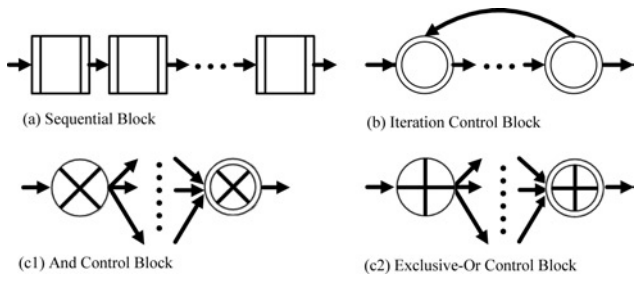


Fig. 3 *Four types of workflow construction in a primitive workflow schema*

3.3 Artifact design layer

The artifact design layer is the bottom layer and its contents are artifact specifications. All artifacts participating in a workflow execution must be pre-defined. The specification of each artifact contains a set of legal operations for its internal data. An activity designed to manipulate a certain artifact can work only with that artifact's legal operations.

The main objective of the artifact design layer is to make the internal design of artifacts independent of the workflow application. It has the following advantages:

1. An artifact has a set of interfaces to these operations. The upper layer design need not be altered so long as any modification to these operations does not affect these interfaces.
2. The artifact manipulations in activities or workflows are bound in these (designed) interfaces. Illegal or invalid operations on artifacts are not permitted.
3. An independent design of artifact operations facilitates the classification and grouping of artifact operations.

4 Artifact usage in TLWM

4.1 Artifact usage

An artifact is an abstract concept describing a type of data participating in workflow execution. It can be extended by combining some processing logic, such as e-form in AgentFlow [27]. In general, an artifact may refer to any data item participating in workflow execution, such as workflow control data, workflow relevant data and application data [28]. Owing to the complexity of artifact operations, this work focuses on the study of artifact usage in workflow specification of TLWM only.

Each artifact is defined as owning at least one of the following five types of usages:

1. Reference (Ri), such as a static variable or a constant in programming language;
2. Manipulation (Mi): processed data, such as an input, output, or processed results of an activity;
3. Deterministic (Di): the key condition result evaluated for the control;
4. Composite (Ci): a composition of sub-artifacts;
5. Property (Pi): utilised to control process or workflow execution, and not accessible during application design. For example, state information about each workflow instance, dynamic state of workflow system, execution duration or resource constraints.

An artifact may be utilised in more than one type of usage, thus the classification of artifacts with usage types is neither absolute nor exclusive. Since our analysis works on design specification, we are concerned with the first four artifact-usage types.

An artifact in TLWM is characterised by its attributes, states, operations and state transition rules. Artifact attributes define the data model of the workflow application handled in a process. An artifact has its own life cycle characterised by states. An artifact state contains a state condition that is a logic expression defined by one or more attributes. An artifact operation manipulates the artifact by changing its state. A state transition rule of an artifact defines the transition from a permissible state to the next. Consequently, the state transition rules characterise the allowable sequences of operations in the lifetime of the artifact. The operations with identical effects on the state transitions are categorised into an operation type.

Thus, a finite state machine of an artifact is proposed to characterise its behaviour over its lifecycle and is defined as a 6-tuple $\langle O, T, S, I, R, F\rangle$ as follows:

- O is a finite and non-empty set of operations.
- T is a finite and non-empty set of operation types. Operation types are mutual exclusive, finite, and non-empty set of operations. The union of elements (operation types) in T equals O.
- S is a finite and non-empty set of states.
- I is a set of initial states, $I \subseteq S$.
- R is a transition rule function $S \times T \rightarrow S$.
- F is a set of final states, $F \subseteq S$.

In TLWM, common artifact operations are categorised into five primitive operation types: `Specify`, `Read`, `Write`, `Revise` and `Destroy`.

1. `Specify`: type of a definition operation, for example, '`fill in`', '`create`', and '`define`' operations.
2. `Read`: type of a reference operation, for example, '`use`', '`fetch`', '`select`', and '`retrieve`' operations.
3. `Write`: type of a modification operation, for example, '`write`', '`change`', and '`update`' operations.
4. `Revise`: type of addition or deletion of a sub-artifact, for example, '`merge`', '`combine`', and '`divide`' operations.
5. `Destroy`: type of deletion of an artifact, for example, '`remove`', '`erase`', '`cancel`', and '`discard`' operations.

In general, `Specify` is the default operation type when an artifact appears in workflow execution. `Read` and `Write` are then used to access the artifact. `Revise` is an operation for merging two artifacts into one, or splitting a smaller artifact out of an existing one. `Destroy` is a final operation to delete the artifact. `Destroy` is necessary for temporary artifacts created during in workflow execution, but not strict for all artifacts.

Based on the above observation, the common behaviour of an artifact in TLWM can be modelled as follows.

- O = {'`fill in`', '`create`', '`define`', '`use`', '`fetch`', '`select`', '`retrieve`', '`write`', '`change`', '`update`', '`merge`', '`combine`', '`divide`', '`remove`', '`erase`', '`cancel`', '`discard`'}
- T = {`Specify`, `Read`, `Write`, `Revise`, `Destroy`}
- S = {'`Start`', '`End`', '`Specified`', '`Written`', '`Revised`'}
- I = {'`Start`'}
- R = {'`Start`' × `Specify` → '`Specified`',
  {'`Specified`', '`Revised`'} × `Write` → '`Written`',
  {'`Specified`', '`Written`'} × `Revise` → '`Revised`',
  ('`Specified`','`Written`', '`Revised`') × `Read` → ('`Specified`', '`Written`', '`Revised`'),
  {'`Specified`','`Written`', '`Revised`'} × `Destroy` → '`End`'}
- F = {'`End`'}

The common artifact behaviour comprises five states: '`Start`', '`End`', '`Specified`', '`Written`' and '`Revised`'. '`Start`' represents the initial state and

'End' represents the final state. 'Specified', 'Written' and 'Revised' represent the states that result from Specify, Write and Revise operations, respectively. Specify is the only allowable type of operation immediately after the initial state and Destroy is the only final operation type allowed immediaely before the final state. Unlike other operation types, Read does not change the artifact's state. In addition, only parallel Read-type operations are allowed. Fig. 4 shows the state diagram, a graph representation corresponding to the artifact's finite state machine.

4.2 Artifact inaccuracy

Based on the common artifact behaviour identified in the previous section, six inaccurate usages are identified as follows.

1. No Producer: 'No Producer' usage, which is a warning, indicates that an artifact has a different operation before Specify. This workflow might fail due to retrieval error or an exception. The exception might occur when an artifact is constructed from an invoked application or outside the system. Fig. 5*a*–*d* are examples of the 'No Producer' problem while *e* is correct.
2. No Consumer: 'No Consumer' defect implies that no activity exists to request the artifact after its modification, that is, Specify, Write or Revise. This defect might occur when the artifact is prepared for access by the external system, or when it is redundant and has no succeeding activity to access it.
3. Redundant Specification: A Redundant Specification defect indicates that another 'specified' state exists after a 'specified' state. It leads to the confusion in maintaining artifacts and making exceptions in execution.
4. Contradiction: A Contradiction defect describes a situation where the current artifact state does not conform to the in-state specified in the pre-condition of a succeeding activity. In an activity specification, the pre- and post-conditions provide a mechanism to specify the in-state before, and the out-state after, the execution of an activity. Increasing the number of state constraints specified in the pre- and post-conditions improves the precision of state-matching. Fig. 6 illustrates a simple example of the Contradiction problem.
5. Parallel Hazard: A Parallel Hazard problem occurs due to conflict interleaving of concurrent artifact operations, and is recognised if multiple concurrent subflows operate on the same artifact. Two examples of state-mapping competition are multiple state choices of incoming flow to an AND control block and multiple produced states of an AND control block. Such a state-mapping competition is a prerequisite for a potential Parallel Hazard. Among the primitive operations, only the concurrent Read operation does not cause a Parallel Hazard. Fig. 7 shows an example of a Parallel Hazard.
6. Branch Hazard: A Branch Hazard may be produced from an XOR control block because the branch subflows that contain operations on artifacts have been selected. A Branch Hazard may also be caused by inconsistency between the condition testing in the XOR-SPLIT node and the branch subflows, that is a losing out-state or insufficient in-state occurs between the current XOR control block and outside it. Fig. 8*a* is a partial workflow schema, containing an XOR control block and two activities A7, A9. Fig. 8*b* is the Branch Hazard of the condition mismatch and Fig. 8*c* of insufficient in-state/losing out-state.

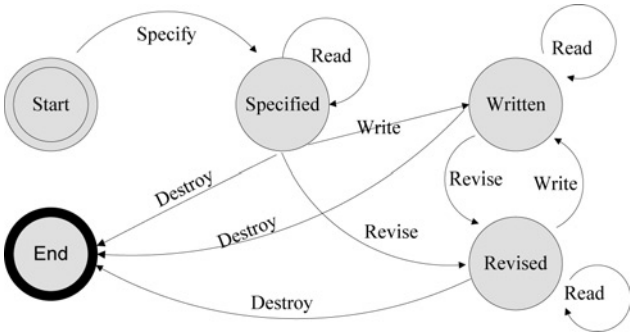


Fig. 4 *State diagram for the common artifact behaviour in TLWM*

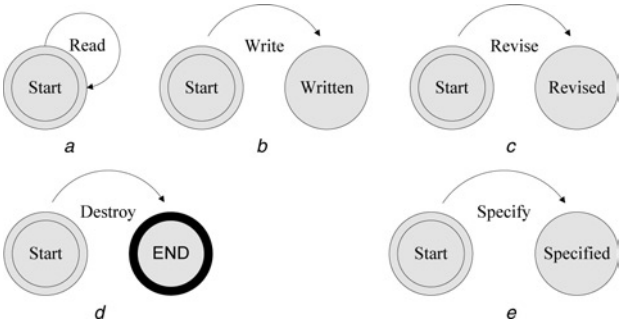


Fig. 5 *No producer problem (part a–d)*

Sun *et al.* [8] claim that seven types of data-flow anomalies proposed by Sadiq *et al.* [7] can be represented by their three basic data-flow anomalies, or are not a problem at the conceptual level. Thus, we discuss the mapping of the anomalies between Sadiq *et al.* and our work as follows.

- The Redundant data anomaly occurs when an activity produces an intermediate data output but this data is not

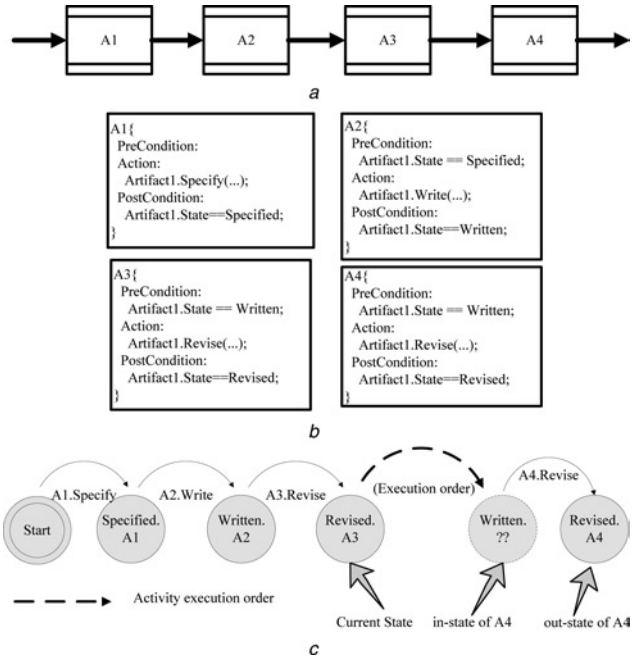
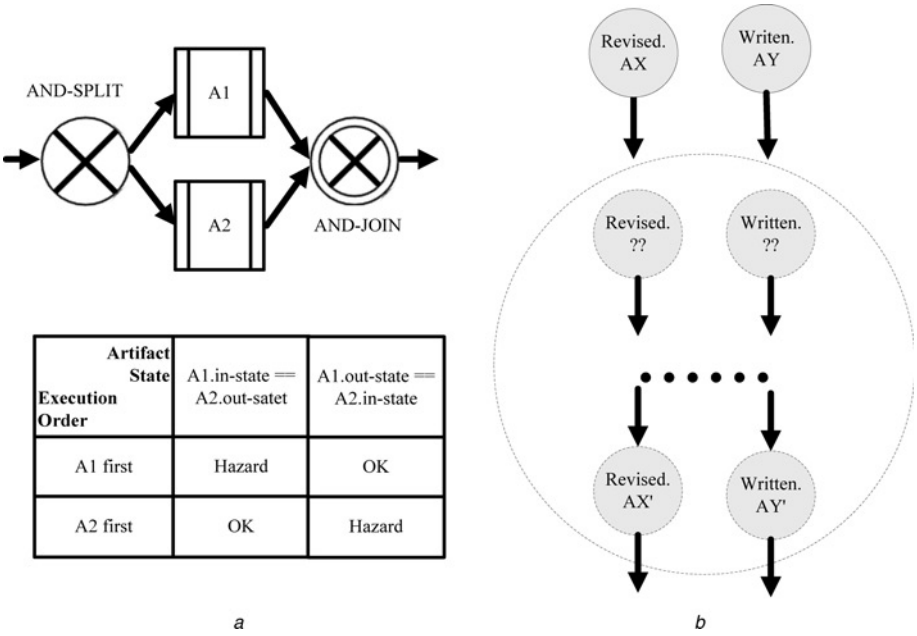


Fig. 6 *Contradiction problem*

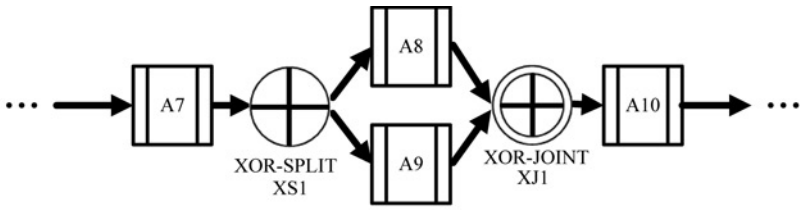


Artifact State / Execution Order	A1.in-state == A2.out-satet	A1.out-state == A2.in-state
A1 first	Hazard	OK
A2 first	OK	Hazard

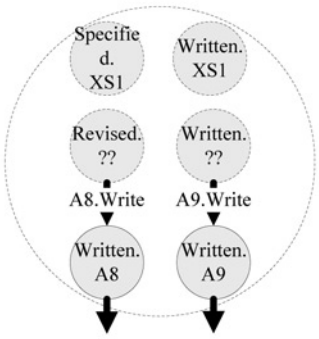
Fig. 7 *Parallel hazard*

required by any succeeding activity. This anomaly is classified as `No Consumer` in our approach.

- Lost data is an anomaly that occurs when parallel activities perform non-read operations on an artifact. This anomaly is classified as `Parallel Hazard` in our approach.
- The Missing data anomaly occurs when an artifact is accessed before it is initialised. This anomaly is classified as `No Producer` in our approach.
- Mismatched data is an anomaly that occurs when the structure of the data produced by the source is incompatible



Activity Specification	A8	A9
Pre-condition (In-state)	Revised	Written
ACTION	Write	Write
Post-condition (Out-state)	Written	Written



Activity Specification	A7	A8	A9	A10
Pre-condition (In-state)	Specified	Revised	Written	Revised
ACTION	Write	Write	Revise	Write
Post-condition (Out-state)	Written	Written	Revised	Written

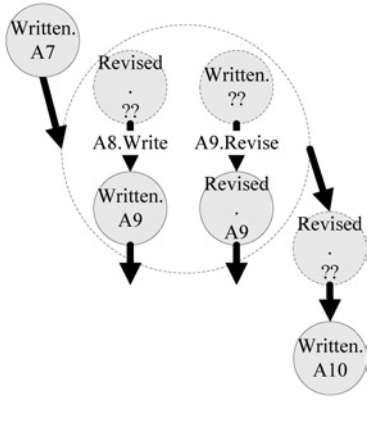


Fig. 8 *Branch hazard*

with the structure required by the activity that uses the artifact. This anomaly is classified as `Contradiction` in our approach.

- Inconsistent data is an anomaly that occurs when an initial input artifact of a workflow is updated externally during the execution time of the workflow. As stated by Sun *et al.*, this anomaly is not a problem at the conceptual level.
- Misdirected data anomaly occurs when a data-flow direction conflicts with the control flow in a workflow schema. This anomaly is classified as `Branch Hazard` in our approach.
- Insufficient data is an anomaly that occurs when data specified are not sufficient to complete an activity successfully. In our approach, this anomaly results from ill-designed activity and can be classified as `No Producer` at the semantic level.

Table 1 summarises the mapping of anomalies among Sadiq's *et al.*, Sun *et al.* and our work.

5 Constructing state transition diagrams for each flow structure and detecting inaccurate usage

The artifact state diagram and state transition criteria help in tracing and recording manipulation of artifacts in a workflow. This section presents approaches for detecting artifact inaccuracies in a TLWM specification. The proposed approach to analysing artifact usage is based on workflow construction reduction, state tracing on artifacts and error detection.

Since a well-formed workflow schema consists of a sequence of activity node and/or top-level control blocks, an entire workflow can be deemed as a sequential block if a control block is viewed as a composite activity. The same perspective can be applied to split flows of a control block. Thus, for an input workflow schema, the proposed reduction algorithm begins by sequentially traversing the main flow enclosed by the `Start Node` and `End Node`. A partial artifact state diagram is constructed for each node during traversal, which are then connected according to the relative positions of their corresponding nodes. The corresponding artifact state diagram of a single activity is constructed according to its specification by the reduction algorithm of a composite activity (control block) being applied to each split flow of the block. These state diagrams are then combined according to the characteristic of the block. Meanwhile, artifact state transitions are analysed to detect inaccuracies. The reduction algorithm is recursively applied until every activity and control block in the current level is processed.

To simplify the explanation, the proposed algorithm is divided into several sub-algorithms, each focused on one artifact at a time. This section presents the algorithms that are applied to construct an artifact state transition diagram for each well-formed workflow structures. These algorithms are integrated with the traverse algorithms presented in the next section to analyse artifact inaccuracies of a workflow schema.

Table 1: Mapping of anomalies

Sadiq *et al.*	Sun *et al.*	Our approach
redundant data	redundant data	no consumer
lost data	conflicting data	parallel hazard
missing data	missing data	no producer
mismatched data	redundant data and missing data	contradiction
inconsistent data	not a problem at conceptual level	not a problem at conceptual level
misdirected data	missing data	branch hazard
insufficient data	missing data	no producer

5.1 Representations for artifact state diagrams in a workflow

Fig. 9 displays the data representations utilised in the algorithms. `PartialFlowASDT` is a structure denoting the artifact state diagram *ASD* of artifact *A* corresponding to a partial workflow, a segment from node `HeadNode` to node `TailNode` in the input workflow schema. An artifact state diagram is mainly represented as a linked list of `StateContainer` structures. Two pointers, `InState` and `OutState`, are employed to point to the first and last `StateContainer`, respectively. `InState` contains state conditions required to enter the corresponding partial workflow, and `OutState` contains state conditions achieved for effects at the end of the partial workflow. For a partial workflow corresponding to an AND or XOR block, the state conditions to be required/achieved are accumulated from those of each subflow of the block. Additionally, two counters, `SpecifyCounter` and `NonReadCounter`, are introduced to count `Specify` operations and `non-Read` operations, respectively. `SpecifyCounter` helps detect redundant specification problems, and `NonReadCounter` helps detect `Parallel Hazard` errors.

A `StateContainer` is associated with a node in the workflow scheme and contains `Conditions`, which is a set of artifact state names required or achieved on that

```
Struct PartialFlowASDT {
  Artifact   *A;
  ArtifactStateDiagram   *ASD;
  Node   *HeadNode;   //Pointer to Head Node of Partial Flow
  Node   *TailNode;   //Pointer to Tail Node of Partial Flow
};
Struct ArtifactStateDiagram {
  StateContainer   *InState;      //Point to First StateContainer.
  StateContainer   *OutState;   //Point to Last StateContainer.
  Long   NonReadCounter; // Counter for NonRead operations
  Long   SpecifyCounter;   //Counter for Specify Operations
};
Struct StateContainer {    //
  Set Conditions//Set of Artifact State Name to be required/achieved
  Node    *Owner   //Pointer to Corresponding Workflow Node
  Set    Transitions   //Set of Pointer of Transitions
}
Struct Transition {
  String   Operation; //Specify, Read, Write, Revise, Destroy, Join
  StateContainer   *Next;   //Pointer to Next StateContainer
}
```

Fig. 9 *Data structures*

node. A StateContainer also includes a set of Transitions. A Transition links two StateContainers, and represents a change in artifact state resulting from an artifact operation performed on an activity node or on a joint action of a control block.

5.2 Algorithm to construct ASD for an activity

The algorithm ConstructActivityASD demonstrates the steps required to construct a partial state diagram of an artifact *Ai* on an activity node *Ni*. The main steps are described as follows:

1. Extract the set of state conditions of *Ai* to be required (InStateSet) according to the PreCondition of the activity.
2. Extract the set of state conditions of *Ai* to be achieved (OutStateSet) according to the PostCondition of the activity.
3. Extract the ordered set of operations being performed on the artifact *Ai* (OPSet) according to the Action of the activity.
4. If the three sets from these steps are all empty, then the activity does not contain any conditions or operations associated with the artifact. Return NULL in this case.
5. Create an InState containing Conditions (to be required) equivalent to InStateSet.
6. Create an OutState containing Conditions (to be achieved) equivalent to OutStateSet.
7. Starting from the InState in step 5, for each operation in OPSet, repeat the following:
   i. For a Read operation, create a Transition to link the current StateContainer with itself.
   ii. For a Specify operation, increment SpecifyCounter. If SpecifyCounter is greater than one, report 'Redundant Specification'.
   iii. For a non-Read operation, increment NonReadCounter. Then, create a new StateContainer according to the operation, and create a Transition to link the current StateContainer to the new StateContainer. Move to the new StateContainer.
8. If the Conditions set of OutState is not empty, and is not equivalent to the Conditions set of the last state node after the previous step, then report an internal specification inconsistency (Fig. 10).

```
Algorithm ArtifactStateDiagram ConstructActivityASD (Node Ni, Artifact Ai) {
  InStateSet= Set of required artifact Ai states in Ni->PreCondition;
  OutStateSet=Set of achieved artifact Ai states in Ni->PostCondition;
  OPSet=Set of operations performed on the artifact Ai in Ni->Action;
  IF ((InStateSet == OutStateSet== OPSet ==Ø)
    RETURN NULL; //Return a Null ASD
  ArtifactStateDiagram *AASD  New ArtifactStateDiagram();
  *InState=New StateContainer(InStateSet, Ni.previous, Ø);
  StateContainer *Current, *NextState, *OutState;
  Current  InState;
FOR each OP in OPSet {
  SWITCH (OP.Type) {
    CASE "Read" :
      NextState = Current; BREAK;
    CASE "Specify" :
      AASD->SpecifyCounter--;
      If (AASD->SpecifyCounter > 1) Report("Redundant Specification")
    Default :
      AASD->NonReadCounter ++;
      StateName= Passive_Verb_Form(OP.Type);
      NextState=New StateContainer({StateName}, Ni, Ø);
  }
  Transition *Tr=New Transition(OP.Type, NextState);
  Current->Transitions.Add(Tr); Current=NextState;
}
IF((OutStateSet!= Ø)&&(Current->Conditions!=OutStateSet)) {
  Report("Internal Specification Error!", Ni);
 //Create a State Node for OutState
  OutState=New StateContainer(OutStateSet, Ni, Ø);
  Transition *Tr=New Transition("InConsistency", OutState);
  Current->Transitions.Add(Tr);
} ELSE {
  OutState=Current;
}
AASD->InState = InState;
AASD->OutState=OutState;
RETURN AASD;
}
```

Fig. 10 *Algorithm to construct ASD for an activity*

5.3 Constructing ASD for control blocks

5.3.1 ASD for a sequential block: For a sequential block in the workflow scheme, the artifact state diagram is constructed by sequentially connecting artifact state diagrams corresponding to each node in the block. Algorithm ASD_SEQU_JOIN shows the steps for connecting two partial artifact state diagrams corresponding to two adjacent nodes. The main steps are as follows:

1. If one of the two *ASD*s is NULL, return the other as the result in this case.
2. Let PreOutStateSet denote the Conditions set of the OutState of the former ASD, and let NextInStateSet be the Conditions set of the InState of the latter ASD. Compare PreOutStateSet with NextInStateSet.
   i. If PreOutStateSet is not a subset of NextInStateSet, then report 'Contradiction'.
   ii. If the former ASD corresponds to an XOR block, and PreOutStateSet is not a subset of NextInStateSet, then report 'Branch Hazard (Losing in-state)'.
   iii. If the latter ASD corresponds to an XOR block, and NextInStateSet is not a subset of PreOutStateSet, then report 'Branch Hazard (Insufficient out-state)'.
3. Combine the OutState of the former ASD and InState of the latter ASD into one StateContainer (Fig. 11).

5.3.2 ASD for an iteration block: For an iteration control block, the artifact state diagram for the loop body is first constructed using the algorithm for a sequential block. Then, to simulate the effects of iteration, the artifact state diagram corresponding to the iteration block is constructed by using `ASD_SEQU_Join` to connect the ASD of the loop body with its copy.

5.3.3 ASD for an AND/XOR Block: Algorithm `ASD_BLOCK_JOIN` demonstrates how to construct an artifact state diagram corresponding to an AND/XOR block by combining artifact state diagrams for subflows of the block. The inputs are the starting and ending nodes of an AND/XOR control block, and a set of sub-artifact state diagrams of its subflows. The main construction steps are as follows:

1. Combine each `InState` of sub-ASDs into a new `InState` with its `Conditions` set equivalent to the union of `Conditions` set of each `InState`.
2. Link each `OutState` of sub-ASDs into a new `OutState` with its `Conditions` set equivalent to the union of `Conditions` set of each `OutState`.
3. Calculate `SpecifyCounter` and `NonReadCounter` according to the block type.
   i. For an AND block, the resulting value is the sum of the counters of sub-ASDs.
   ii. For a XOR block, the resulting value is the maximum among the counters of sub-ASDs.
4. For an AND block with more than one sub-ASD, if the new `InState` from step 1 contains multiple required conditions, the new `OutState` from step 2 contains multiple conditions to be achieved, or multiple sub-ASDs contains non-`Read` operations, then report '`Parallel Hazard`'.
5. For a XOR block, if the set of state test conditions on the XOR-SPLIT node does not match the `Conditions` of the new `InState` from step 1, then report '`Branch Hazard`' (Fig. 12).

```
Algorithm ASD_SEQU_JOIN(ASD1, ASD2) {
  IF (ASD1 == NULL) RETURN ASD2;
  IF (ASD2 == NULL) RETURN ASD1;
  ArtifactStateDiagram *NewASD=ASD1;
  NewASD->SpecifyCounter+=ASD2->SpecifyCounter;
  NewASD->NonReadCounter+=ASD2->NonReadCounter;

  //Merge ASD1->OutState with ASD2->InState
  NewASD->OutState->Transitions=ASD2->InState->Transitions;

  OC=ASD1->OutState.Conditions;
  IC= ASD2->InState.Conditions;
  IF ((OC!=IC) && (OC ⊄ IC) && (IC ⊄ OC))
    Alarm Contradiction;
  IF (ASD1->OutState->Owner->Type==”XOR-JOIN”) {
    IF (OC ⊄ IC))  Alarm “Branch Hazard (Losing in-state)”
  } ELSE IF (ASD2->InState->Owner->Type==”XOR-SPLIT”) {
    IF (IC ⊄ OC) Report(“Branch Hazard (insufficient out-state)”;
  }
  RETURN NewASD;
}
```

Fig. 11 *Algorithm to connect two sequential ASDs*

6 Integrated algorithms to traverse and analysis

6.1 Algorithm for analysing sequential blocks

Algorithm `AnalyzeSequence` is applied to construct and analyse the state diagram of artifact *Ai* corresponding to the sequential block originated from a starting control node (`START-NODE`, `LOOP-START`, `AND-SPLIT` or `OR-SPLIT`). The algorithm traverses the nodes of the block until it reaches the corresponding ending control node (`END-NODE`, `LOOP-END`, `AND-JOIN` or `XOR-JOIN`). The algorithm has the following steps:

1. Enter a loop to traverse the nodes in sequential order until the corresponding ending control node is reached. At each iteration, the following steps are performed.
   Construct the partial state diagram of artifact *Ai* according to the type of current traversed node.
   i. When an activity node is encountered, construct the corresponding artifact state diagram according to the specification of the activity (using algorithm `ConstructActivityASD` described in the previous section).
   ii. When a starting control node of a sub-block is encountered, construct the artifact state diagram corresponding to the sub-block (using algorithm `AnalyzeBlock` described in the latter sections).
   Apply algorithm `ASD_SEQU_JOIN` to concatenate the ASD constructed in the previous iteration with the ASD. Artifact inaccuracies of contradiction and branch hazard are also detected in Algorithm `ASD_SEQU_JOIN`.
   Continue the traversal by following the current activity node directly, or following the end of the current sub-block.
2. Pack the starting node, ending node and the corresponding ASD of the sequential block into a `PartialFlowASDT` structure and then return it (Fig. 13).

6.2 Algorithm for analysing control blocks

Algorithm `AnalyzeBlock` constructs the partial state diagram of artifact *Ai* corresponding to a control block and analyses the ASD based on the type of block.

6.2.1 Analysing an iteration control block: For an iteration control block, proceed as follows:

1. Apply `AnalyzeSequence` to traverse the loop body and construct the corresponding ASD of *Ai*.
2. Construct and analyse the ASD corresponding to the LOOP-block, simulating the effects of iteration by using `ASD_SEQU_Join` to connect and check the ASD of the loop body with its copy.
3. Pack the starting node, ending node and the corresponding ASD into a `PartialFlowASDT` structure, and then return it.

```
Algorithm ASD_BLOCK_JOIN(BlockStart, BlockEnd, Set SubASDs) {
  ArtifactStateDiagram *NewASD;
  StateContainer *NewInState=New StateContainer();
  StateContainer *NewOutState=New StateContainer();

  String BlockType=BlockStart->Type;

  NewInState->Owner=BlockStart;
  NewOutState->Owner=BlockEnd;

  FOR each SubASD in SubASDs {
    //Combine each InState of sub ASDs into one
    With (*NewInState) {
      Conditions=Conditions∪SubASD->InState->Conditions;
      Transitions= Transitions∪SubASD->InState->Transitions;
    }

    With (*NewOutState) {
      Conditions=Conditions∪SubASD->OutState->Conditions;
    }

    //Join each OutState of sub ASDs into a new OutState
    Transition *TrJoin=New Transition("", NewOutState);
    SubASD->OutState->Transitions.Add(TrJoin);

    SWITCH(BlockStart->Type) {
      CASE "AND-SPLIT"
        IF (SubASD->NonReadCounter>0)
          NonReadPathCounter++;

        NewASD->NonReadCounter+=SubASD->NonReadCounter;
        NewASD->SpecifyCounter +=SubASD->SpecifyCounter;
        BREAK;

      CASE "XOR-SPLIT"
        IF (NewASD->NonReadCounter<SubASD->NonReadCounter)
          NewASD->NonReadCounter=SubASD->NonReadCounter;

        IF (NewASD->SpecifyCounter < SubASD->SpecifyCounter)
          NewASD->SpecifyCounter=SubASD->SpecifyCounter;

        TestCondition=State Test Condition to Enter the SubFlow Corresponding to
        Current SubASD on XOR-SPLIT node;
        IF (TestCondition∉ SubASD->InState->Conditions)
          Report("Branch Hazard (Mismatch Condition)", BlockStart,
          SubASD->InState->Owner);
          BREAK;
      }
    }

    NewASD->InState=NewInState;
    NewASD->OutState=NewOutState;

    SWITCH(BlockType) {
      CASE "AND-SPLIT" :    // check Parallel Hazard
        IF ( SubASDs.Count > 1)
          IF ((NewInState->Conditions.Count>1) || (NewOutState->Conditions.Count>1) ||
              (NonReadPathCounter >0))
            Report("Parallel Hazard", BlockStart);
          BREAK;
        CASE "XOR-SPLIT" :    // check Branch Hazard
          TestSet=Set of State Test Conditions on XOR-SPLIT node;
          IF (TestSet!=NewInState->Conditions)
            Report("Branch Hazard", BlockStart);
      }
      RETURN NewASD;
    }
```

Fig. 12 *Algorithm to construct ASD for AND/XOR blocks*

6.2.2 Analysing an AND/XOR control block: The ASD of each subflow and the ASD resulting from the merger of these ASDs are analysed as follows:

1. Apply `AnalyzeSequence` to construct and analyse the ASD of each split flow.
2. Apply `ASD_BLOCK_JOIN` to merge the ASDs of each split flow.
3. For AND-block, detect Parallel Hazard by checking *NonReadCounter*.
4. For XOR-block, detect Branch Hazard by checking whether the condition testing on the XOR-SPLIT node is consistent with `InStates` of the block.
5. Pack the starting node, ending node and the corresponding ASD of the block into a `PartialFlowASDT` structure and then return it Fig. 14.

6.3 Main algorithm for analysing workflow

`AnalyzeWS` is the main algorithm for analysing the input workflow schema. The algorithm utilises `AnalyzeSequence` to construct an artifact state diagram for each artifact appearing in the input workflow. During the construction, detections for artifact inaccuracies of `Redundant Specification`, `Contradiction`, `Parallel Hazard` and `Branch Hazard` proceed. The constructed artifact state diagram is then utilised to detect artifact inaccuracies of `No Producer` and `No Consumer`.

The computational complexity of our analysis algorithms are now discussed. Our algorithms include three iterations: (1) the algorithm `AnalyzeWS` iterates through each artifact, (2) the algorithms `AnalyzeSequence` and `AnalyzeBlock`, are used to iterate through each node

```
Algorithm: PartialFlowASDT AnalyzeSequence(HeadNode, Ai) {
  PartialFlowASDT FlowASDT, SubASDT;
  FlowASDT.A=Ai;
  FlowASDT.HeadNode=HeadNode;
  Node LastNode=CurrentNode=HeadNode;
  WHILE (CurrentNode!=NULL) {
    SubASDT =NEW PartialFlowASDT();
    SWITCH (CurrentNode.Type) {
      CASE ACTIVITY:
        SubASDT->HeadNode=SubASDT->TailNode=CurrentNode;
        SubASDT->ASD=ConstructActivityASD(CurrentNode, Ai);
      CASE START-NODE, LOOP-START, AND-SPLIT, XOR-SPLIT:
        SubASDT =AnalyzeBlock(CurrentNode, Ai);
      CASE AND-JOIN, XOR-JOIN, LOOP-END, END-NODE:
        FlowASDT->TailNode=LastNode;
        Return FlowASDT;
    }
    FlowASDT=ASD_SEQU_JOIN(FlowASDT, SubASDT);
    LastNode=SubASDT.TailNode;
    CurrentNode=SubASDT.TailNode.next;
  }
}
```

Fig. 13 *Algorithm for analysing sequential blocks*

and (3) the algorithm `ConstructActivityASD` iterates through each operation performed on the artifact at an activity node. Suppose the input workflow schema has n nodes and m artifacts and the average number of conditions and operations for an artifact on each node is c. The complexity of our algorithms is O(nmc) (Fig. 15).

7 Illustrative example

This section presents an illustrative example of a primitive workflow schema and the procedure to detect artifact

```
Algorithm PartialFlowASDT AnalyzeBlock(Node BlockStart, Artifact Ai) {
  PartialFlowASDT BlockASDT, SubASDT;
  Set SubASDTs;
  BlockASDT.HeadNode=BlockStart;
  SWITCH (BlockStart.Type) {
    CASE START-NODE:
      BlockASDT.ASD=AnalyzeSequence(BlockStart.Next, Ai).ASD;
    CASE LOOP-START:
      SubASDT=AnalyzeSequence(BlockStart.Next, Ai);
      BlockASDT.ASD=ASD_SEQU_JOIN(SubASDT.ASD, SubASDT.ASD);
    CASE AND-SPLIT, XOR-SPLIT:
      FOR EACH outflow IN BlockStart {
        Node SubFlowStart=First node of current outflow;
        SubASDT=AnalyzeSequence(SubFlowStart, Ai);
        IF (SubASDT.ASD!=NULL)  SubASDTs.Add(SubASDT.ASD);
      }
      //JOIN ASD according to the type of block.
      BlockASDT.ASD=ASD_BLOCK_JOIN(BlockStart, SubASDT.TailNode.Next, SubASDTs);
  }
  BlockASDT->TailNode=SubASDT->TailNode->Next;
  BlockASDT->ASD->OutState->Owner=BlockASDT->TailNode;
  RETURN BlockASDT;
}
```

Fig. 14 *Algorithm for analysing control blocks*

```
Algorithm AnalyzeWS(Primitive_Workflow_Schema ws) {
  For Each Artifact Ai in ws {
    //Detect Redundant Specification, Contradiction, Parallel Hazard and Branch Hazard.
    PartialFlowASDT WSASDT = AnalyzeSequence(ws.StartNode, Ai);
    //Detect No Producer and No Consumer
    FirstState=WSASDT->ASD->InState;
    LastState= WSASDT->ASD->OutState;
    IF (FirstState->Transitions(0).Operation.Type != "Specify")
      Report("No Producer");
    IF (LastState->Transitions(0).Operation.Type != "Read")
      Report("No Consumer");
    IF (WSASDT->SpecifyCounter > 1)
      Report("Redundant Specify");
  }
}
```

Fig. 15 *Main algorithm for analysing workflow*

inaccuracies, and demonstrate the proposed analysis algorithms. Fig. 16 shows an example of primitive workflow schema.

Fig. 16 illustrates the three unblocked activity nodes {A1, A2 and A14} and two top-level control blocks {AND-1 (AS1, AJ1) and XOR-2 (XS2, XJ2)}. The first top-level control block AND-1 contains iteration control block LOOP1 (`LOOP1_Start, LOOP1_End`), control block AND-2 (AS2, AJ2) and XOR-1(XS1, XJ1) followed by an activity node A9. The second top-level control block XOR-2 contains control block AND-3 (AS3, AJ3) and XOR-3 (XS3, XJ3).

Table 2 displays the usage of an example artifact extracted from the activity specifications. The blank field indicates no conditions or no operations on the artifact. This workflow schema example is then analysed by the proposed artifact analysis algorithms as follows.

In the beginning of the analysis process, algorithm `AnalyzeSequence` is employed to sequentially traverse the main flow enclosed by the `Start Node` and `End Node`. In Fig. 17, first, activity nodes A1 and A2 are successively processed to construct their corresponding partial

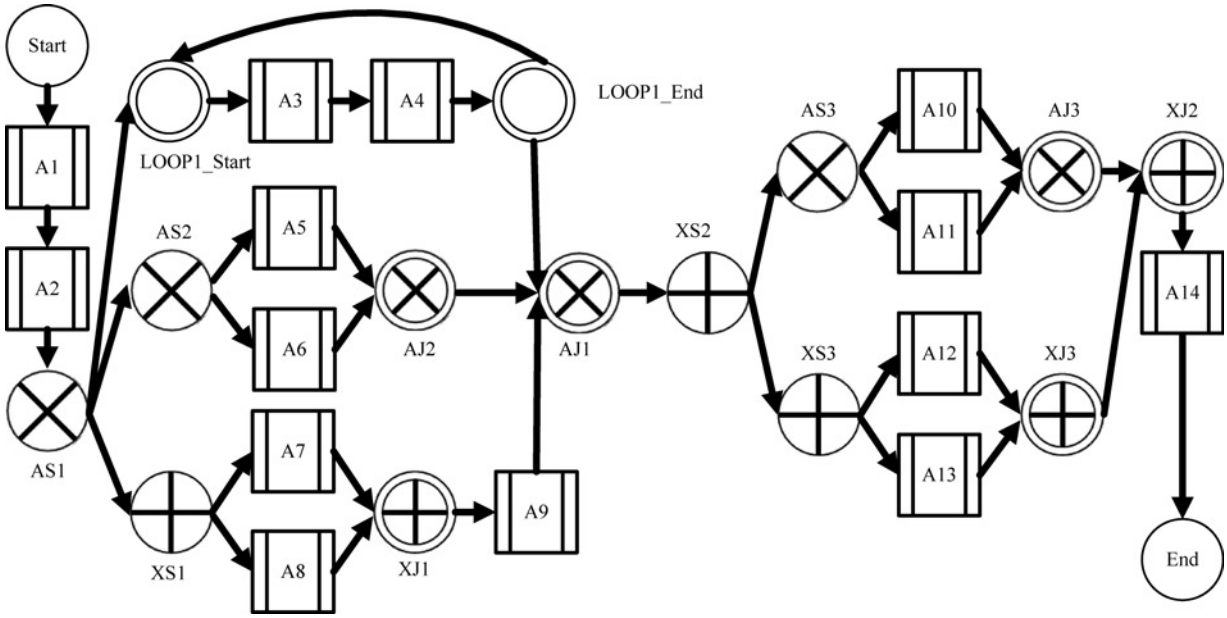


Fig. 16 *Example of primitive workflow schema*

artifact state diagrams using algorithm `ConstructActivityASD`. Then, these two partial state diagrams are concatenated by algorithm `ASD_SEQU_JOIN`.

In Fig. 18, algorithm `AnalyzeBlock` is applied on AND1 control block. First, `AnalyzeBlock` utilises algorithm `AnalyzeSequence` to construct corresponding partial artifact state diagram for each concurrent subflow of the block. Then algorithm `ASD_BLOCK_JOIN` is employed to merge these partial diagrams into one. The inner control blocks are processed by the same reduction procedures. In the third concurrent subflow, a Branch Hazard is detected because of the loss of the out-state.

In Fig. 19, a Parallel Hazard is detected during joining concurrent artifact state transitions. Fig. 20 shows the outcome after AND1 control block has been processed. In Fig. 21, the succeeding control block XOR2 is processed by `AnalyzeBlock`, and its branch subflows are processed using `AnalyzeSequence`.

In Fig. 22, the branch subflows of XOR2 are reduced to node AND3 and node XOR3. Control block XOR2 is reduced to node XOR2 in Fig. 23. In Fig. 24, a branch hazard is then detected because of the loss of out-state (revised) during sequential joining with activity node A14. Fig. 25 is the result of artifact usage analysis on this workflow schema. Additionally, the final artifact state diagram shows three instances of Artifact Inaccuracy.

The Branch Hazard, detected between control block XOR1 and activity node A9, exists because one of the out-states in XOR1 is lost. In the actual execution, the execution thread selecting the branch subflow (A8,A8) may halt by contradiction between the current artifact state and the Pre-condition of activity A9. The Branch Hazard, detected between the control block XOR2 and activity node A14, has the same cause.

The Parallel Hazard, detected inside control block AND1, exists because of a competition between these three concurrent subflows, (`LOOP1_Start`,`LOOP1_End`), (AS2,AJ2) and (XS1,A9). The interleaving between the concurrent subflows may result in Contradiction problems inside the control block.

Table 2: Usage of the example artifact

Activity node	In-state	Operation type	Out-state
A1		Specify	
A2	Specified	Write	Written
A3	Written	Revise	Revised
A4	Revised	Write	Written
A5	Written	Revise	Revised
A6			
A7	Written	Write	Written
A8	Written	Revise	Revised
A9	Written	Revise	Revised
A10			
A11	Revised	Write	Written
A12	Revised	Write	Written
A13	Revised	Read	
A14	Written	Revise	Revised

8 Comparisons of data-flow analysis approaches

As mentioned in the related work section, current workflow modelling and analysing paradigms mainly focus on control flow and resource dimension. The literature reports very little work in the data-flow dimension. In the literature, Sadiq *et al.* [7] and Sun *et al.* [8] are two significant works focused on the analysis in data-flow dimension. Thus, in this section, we compare our work with theirs as follows and summarises the main points in Table 3.

Sadiq *et al.* [7] identify and justify the importance of data modelling in overall workflow design process. In addition, data-flow validation issues and essential requirements of data-flow modelling in workflow specifications are identified. They illustrate and define seven potential data-flow anomalies in the above table. However, Sadiq's work is discussed only at 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.

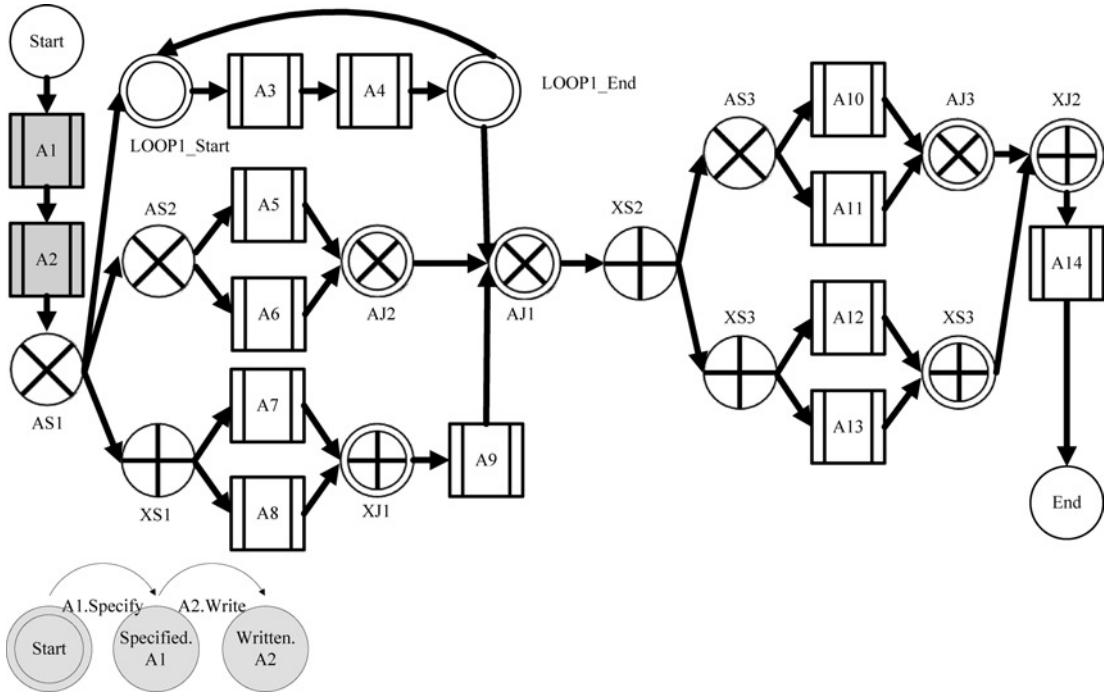


Fig. 17 *Phase 1*

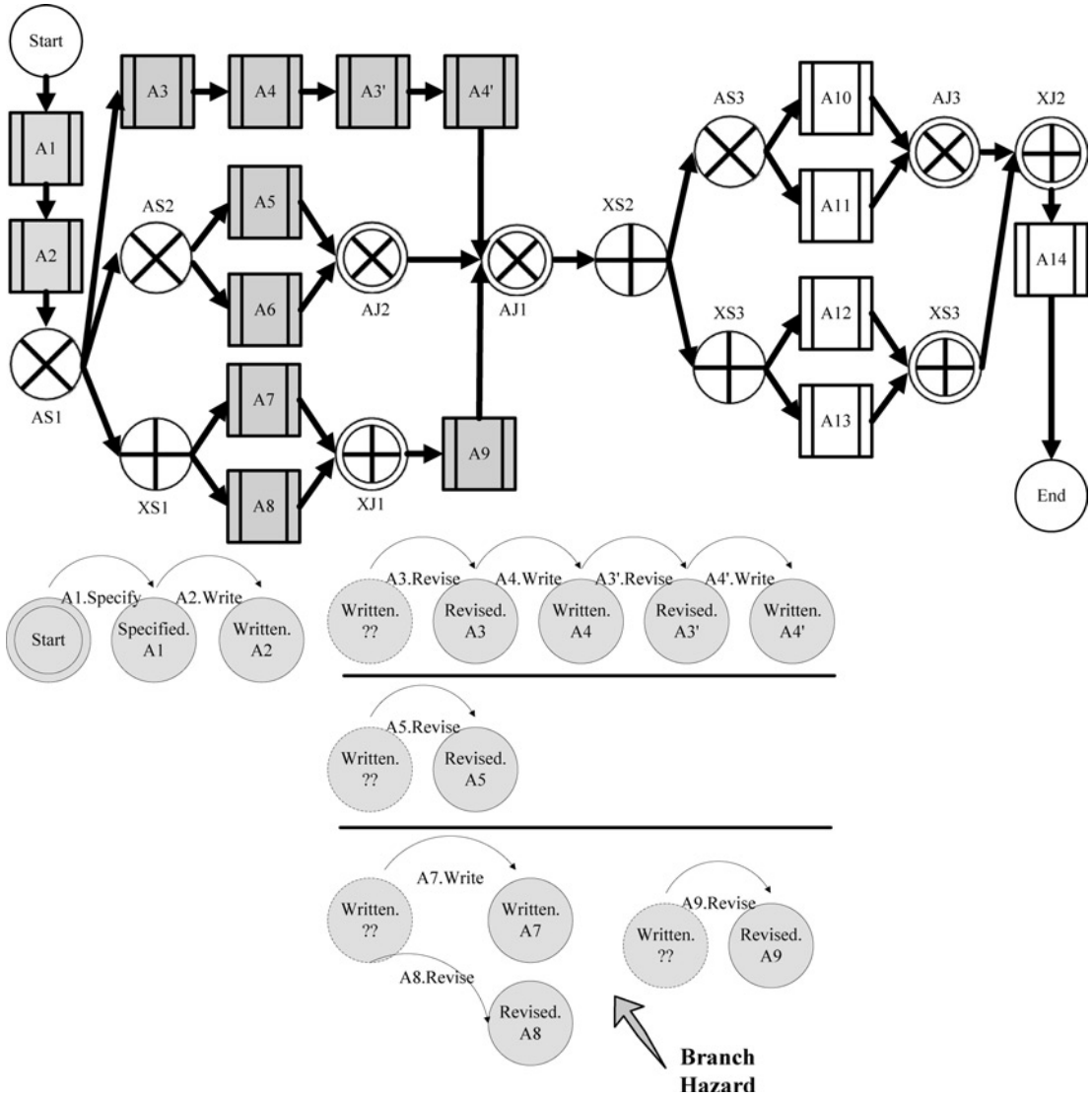


Fig. 18 *Phase 2*

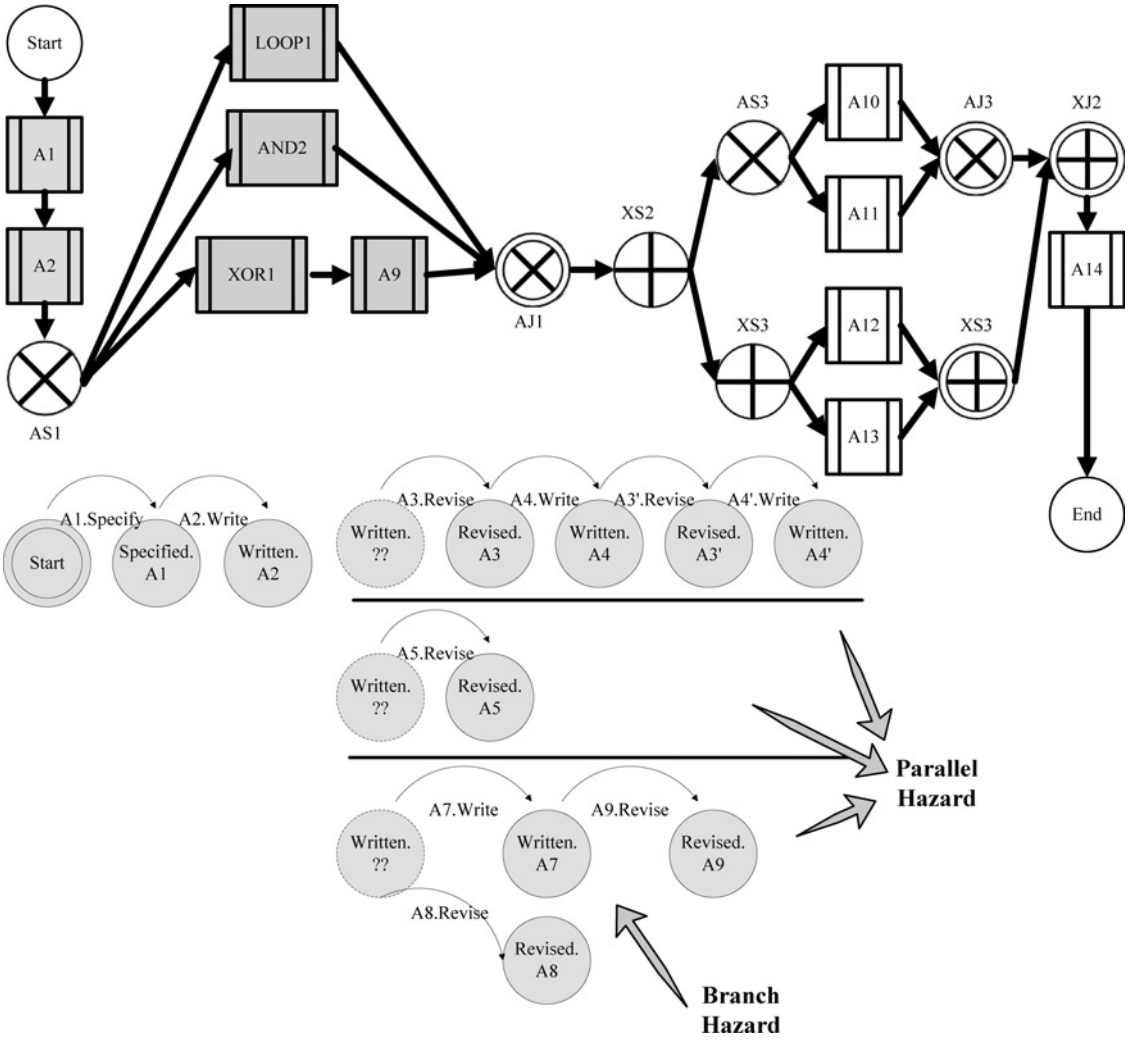


Fig. 19 *Phase 3*

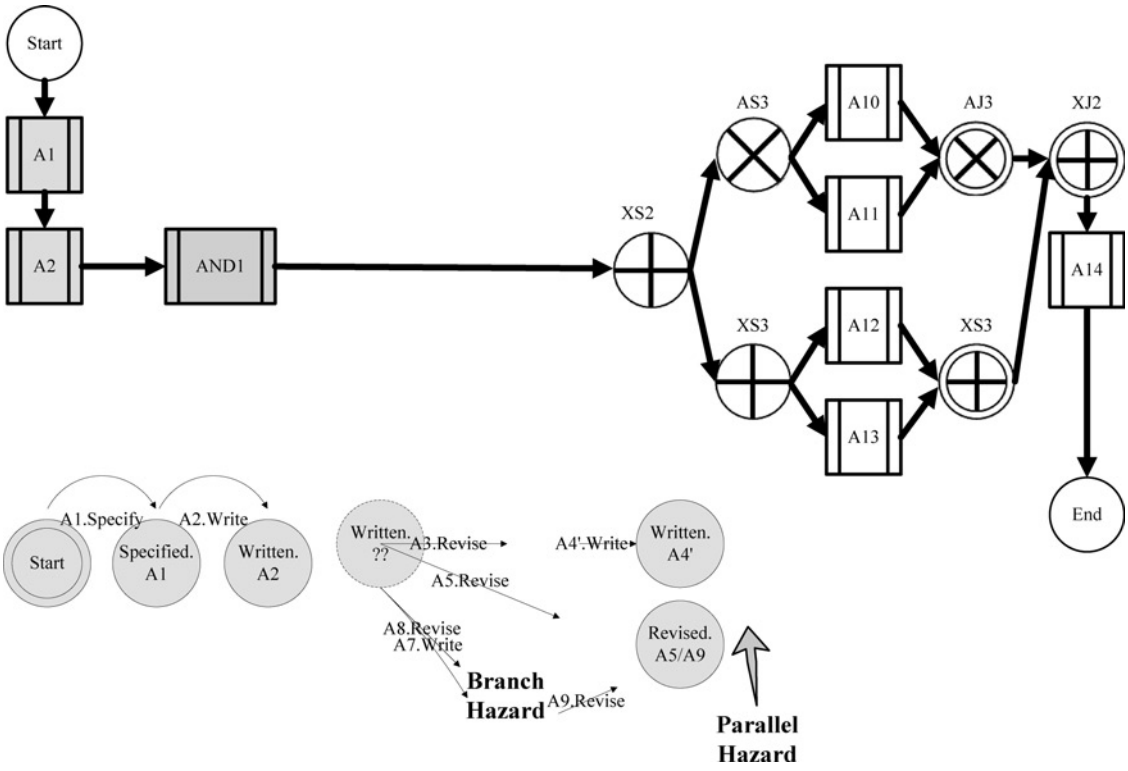


Fig. 20 *Phase 4*

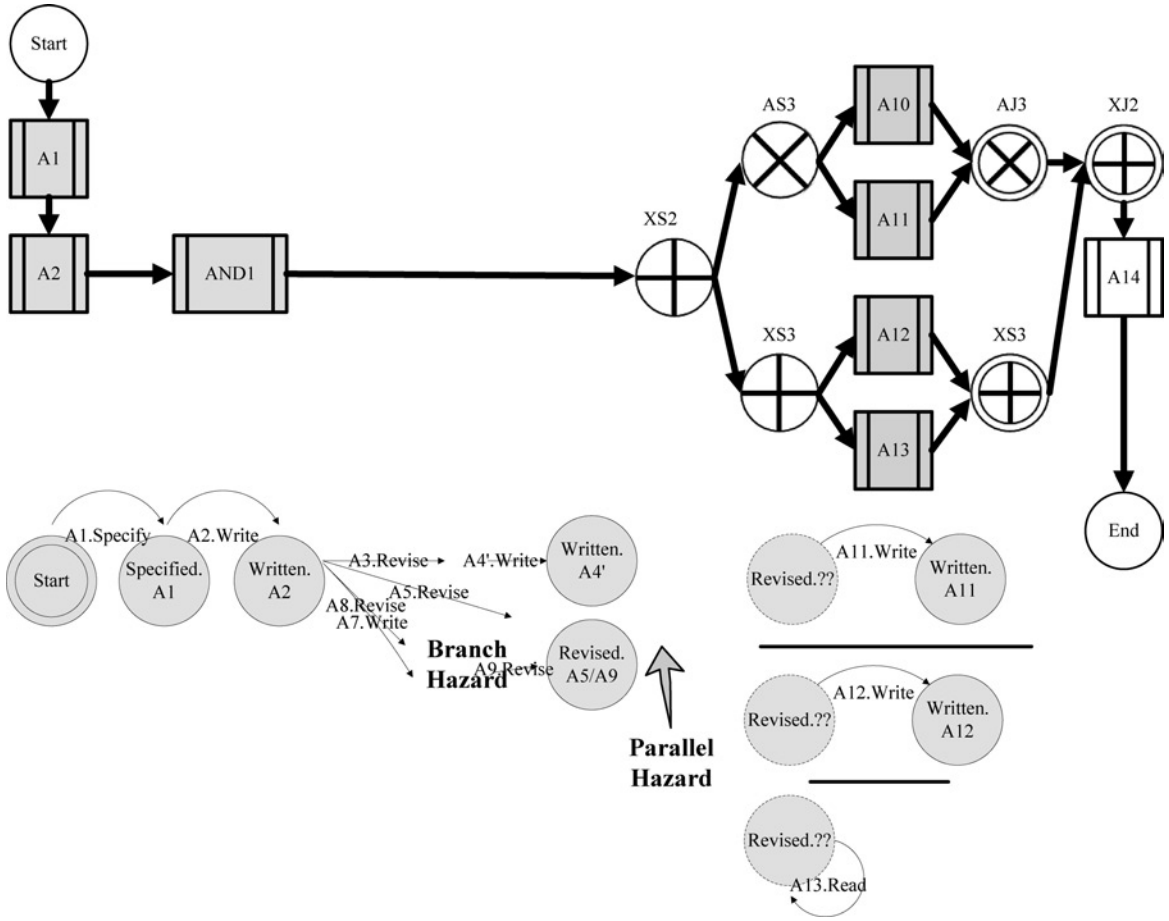


Fig. 21 *Phase 5*

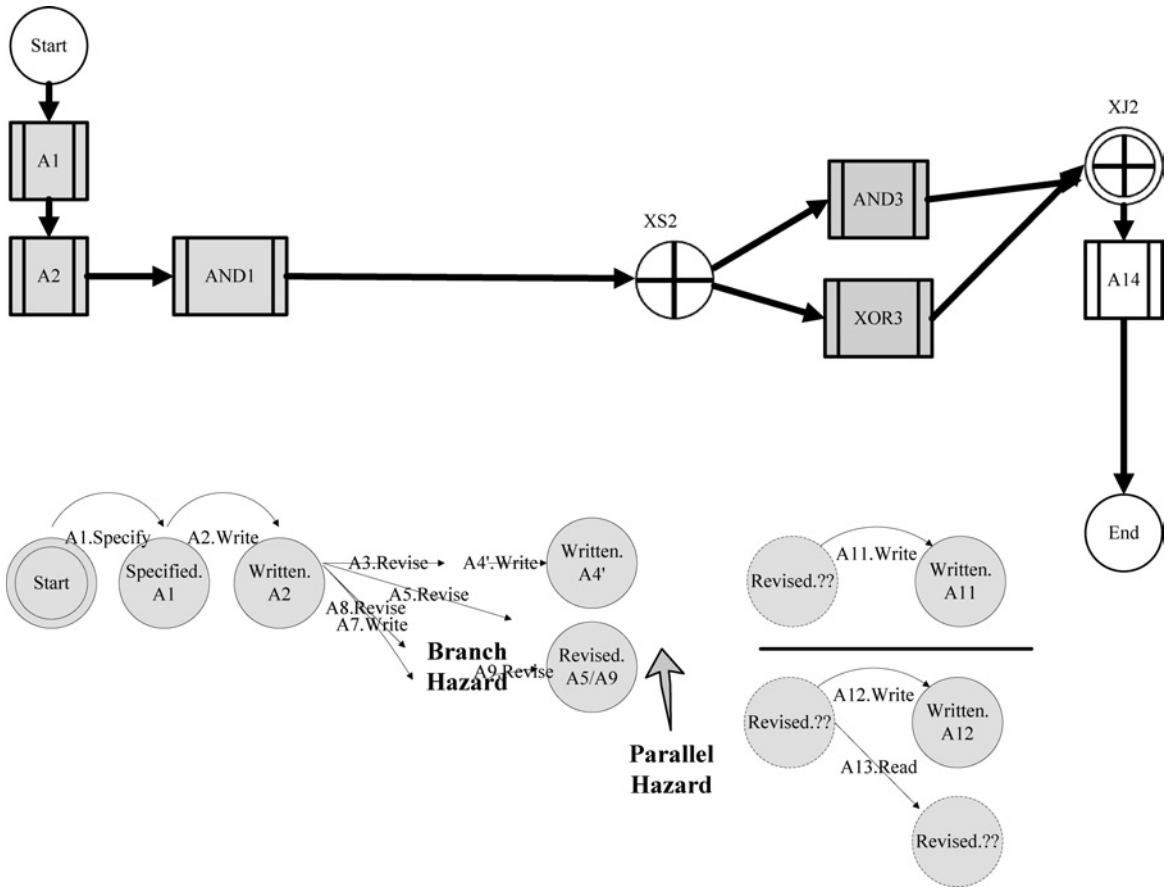


Fig. 22 *Phase 6*

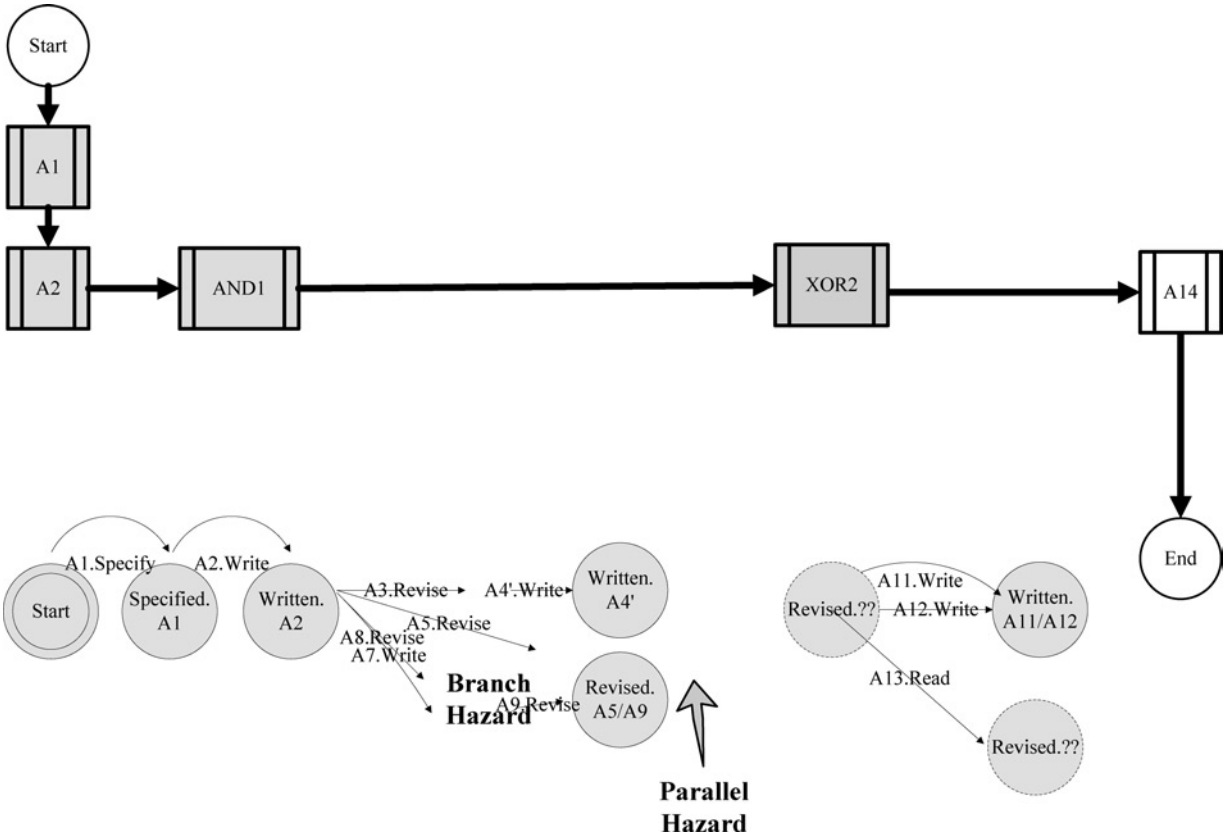


Fig. 23 *Phase 7*

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

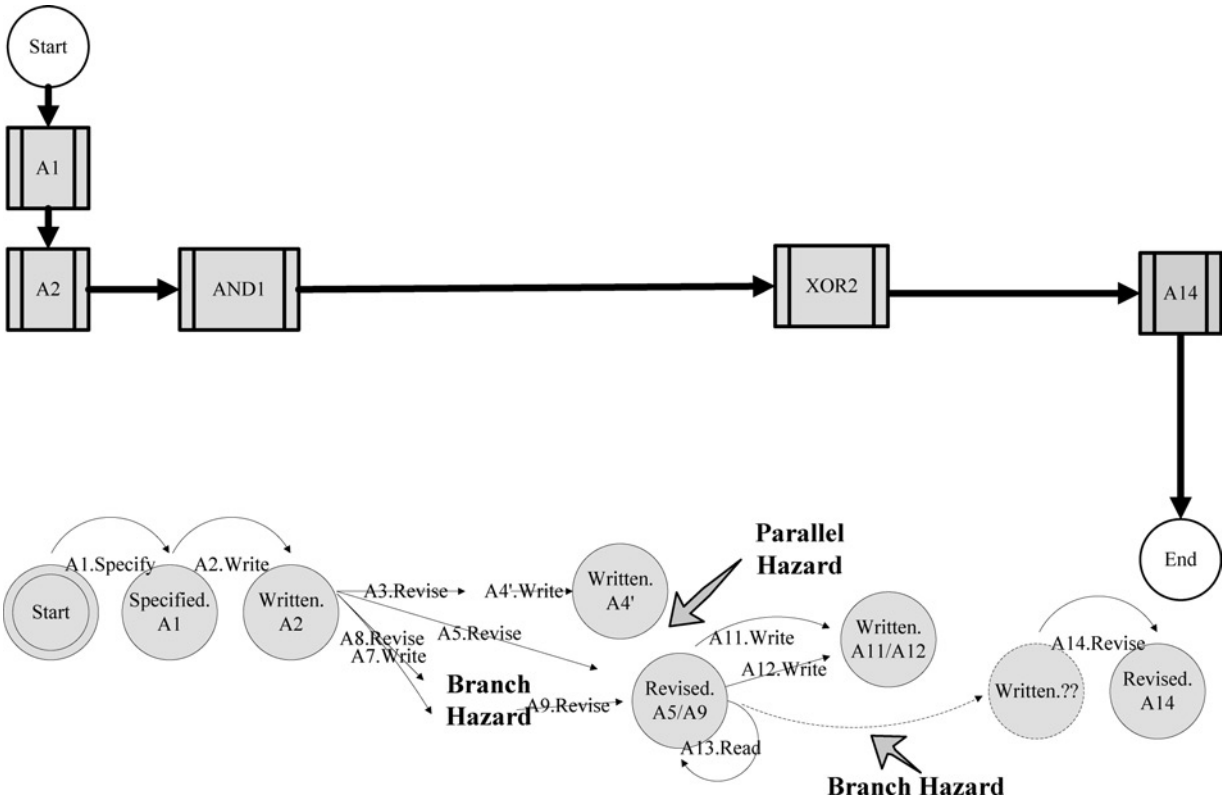


Fig. 24 *Phase 8*

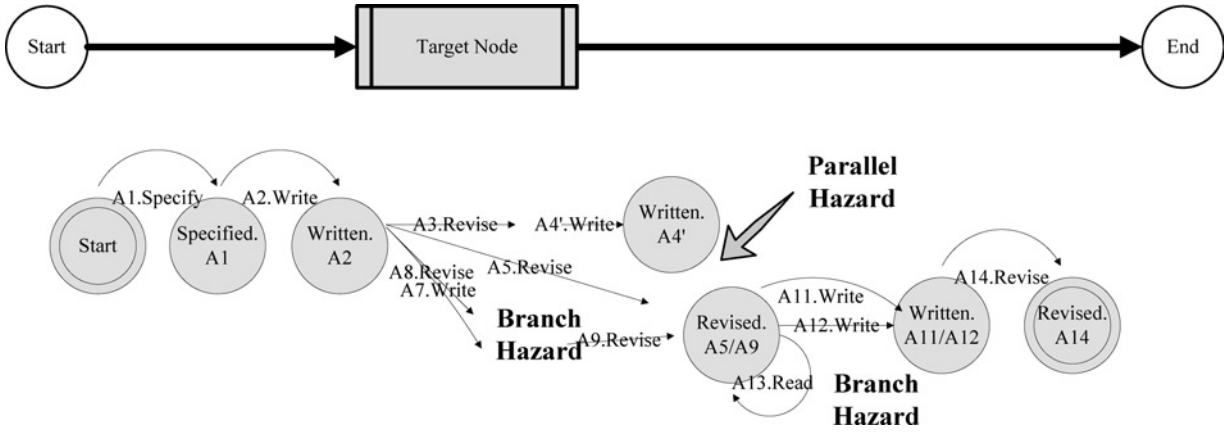


Fig. 25 *Phase 9*

Table 3: Summary of comparisons

	Sadiq *et al.*	Sun *et al.*	This work
workflow model	n/a (conceptual level)	data-flow matrices and process data diagram	three layer workflow model
anomalies discussed	missing data		`No producer`
	redundant data	missing data	`No consumer`
	lost data		`Redundant specification`
	mismatched data	redundant data	`Contradiction`
	inconsistent data		`Parallel hazard`
	misdirected data	conflicting data	`Branch hazard`
	insufficient data		
data behaviour	n/a	n/a	finite state machine
operations concerned	`Read, Write`	`Read, Initial Write`	Specify, Read, write, revise and destroy
detecting method	n/a	data dependency analysis	tracing on artifact state transition
concrete algorithm	n/a	yes	yes

of data. Also, read and initial write operation types are considered only.

Our approach uses a three-layer workflow model to describe workflow schemas. The behaviours of an artifact are explicitly modelled by a finite state machine. Based on this behaviour model, the classification on the operation types of an artifact is not limited to read and write. Designers are free of using any criteria for classification although the classification can be done based on the semantic meanings of operations on an artifact. Besides, the common operation types, including `Specify`, `Read`, `Write`, `Revise` and `Destroy`, are identified as examples in this paper.

9 Conclusion and future work

The main contribution of this work is to introduce an artifact usage analysis technique into workflow design phase. To achieve this goal, this work builds a Three-Layer Workflow Design Methodology. In the methodology, the behaviours of an artifact are characterised by its state transition diagram. Among the usages of artifacts, six types of inaccurate artifact usage affecting workflow execution are identified and a set of algorithms to discover these inaccuracies is presented. An example is demonstrated to validate the usability of the proposed algorithm.

We are currently continuing our research in several directions. First, we plan to implement the proposed model and algorithms on current workflow management systems, such as Agentflow [27], so that our research results can be tested in real-world applications. Second, we will continue the analysis on composite artifacts with more complex usages using `Revise` operations. The third is to integrate resource constraints analysis techniques with our work to build a practical workflow design methodology.

10 References

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

2 Senkul, P., and Toroslu, I.H.: 'An architecture for workflow scheduling under resource allocation constraints', *Inf. Syst.*, 2005, **30**, (5), pp. 399–422

3 Li, H., Yang, Y., and Chen, T.Y.: 'Resource constraints analysis of workflow specifications', *J. Syst. Softw.*, 2004, **73**, (2), pp. 271–285

4 Liu, C., Lin, X., Orlowska, M., and Zhou, X.: 'Confirmation: increasing resource availability for transactional workflows', *Inf. Sci.*, 2003, **153**, (1), pp. 37–53

5 Du, W., and Shan, M.C.: 'Enterprise workflow resource management'. Proc. Ninth International Workshop on Research Issues on Data Engineering: Inf. Technology for Virtual Enterprises, (IEEE Computer Society), March 1999, pp. 108–115

6 Muehlen, M.Z.: 'Resource modelling in workflow applications'. Workflow Management Conf., Münster, Germany, 1999

7 Sadiq, S., Orlowska, M., Sadiq, W., and Foulger, C.: 'Data flow and validation in workflow modelling'. Proc. Fifteenth Conf. on Australasian Database, Australian Computer Society 2004, vol. 27, pp. 207–214

8 Sun, S.X., Zhao, J.L., Nunamaker, J.F., and Sheng, O.R.L.: 'Formulating the data flow perspective for business process management', *Inf. Syst. Res.*, 2006, **17**, (4), pp. 374–391

9 Zhuge, H.: 'Component-based workflow systems development', *Decis. Support Syst.*, 2003, **35**, (4), pp. 517–536

10 Hitomi, A.S., and Le, D.: 'Endeavors and component reuse in web-driven process workflow'. Proc. California Software Symposium, Irvine, CA, USA, October 1998, pp. 15–20

11 Hsu, H.-J.: 'Using state diagrams to validate artifact specifications on primitive workflow schema'. MS thesis, National Chiao-Tung University, 2005

12 Wang, F.-J., Hsu, C.-L., and Hsu, H.-J.: 'Analyzing inaccurate artifact usages in a workflow schema', *COMPSAC*, 2006, **2**, pp. 109–114

13 Curtis, B., Kellner, M.I., and Over, J.: 'Process modelling', *CACM*, 1992, **35**, (9), pp. 75–90

14 Jablonski, S., and Bussler, C.: 'Workflow management: modelling concepts, architecture, and implementation' (International Thomson Computer Press, London, UK, 1996)

15 Karamanolis, C., Giannakopoulou, D., Magee, J., and Wheater, S.M.: 'Model checking of workflow schemas'. Fourth Int. Enterprise Distributed Object Computing Conf. (EDOC′00), IEEE Computer Society, 2000, pp. 170–179

16 van der Aalst, W.M.P.: 'The application of petri nets to workflow management', *J. Circuits Syst. Comput.*, 1998, **8**, (1), pp. 21–66

17 van der Aalst, W.M.P., and ter Hofstede, A.H.M.: 'Verification of workflow task structures: a petri-net-based approach', *Inf. Syst.*, 2000, **25**, (1), pp. 43–69

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

19 Verbeek, H.M.W., Basten, T., and van der Aalst, W.M.P.: 'Diagnosing workflow processes using woflan', *Comput. J.*, 2001, **44**, (4), pp. 246–279

20 Sadiq, W., and Orlowska, M.E.: 'Analyzing process models using graph reduction techniques', *Inf. Syst.*, 2000, **25**, (2), pp. 117–134

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

22 van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., and Barros, A.P.: 'Advanced workflow patterns'. 7th Int. Conf. on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2000, pp. 18–29

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

24 Bae, J., Bae, H., Kang, S.-H., and Kim, Y.: 'Automatic control of workflow processes using ECA rules', *IEEE Trans. Knowl. Date Eng.*, 2004, **14**, (8), pp. 1010–1023

25 Son, J.H., and Kim, M.H.: 'Extracting the workflow critical path from the extended well-formed workflow schema', *J. Comp. Syst. Sci.*, 2005, **70**, (1), pp. 86–106

26 Chang, D.-H., Son, J.H., and Kim, M.H.: 'Critical path identification in the context of a workflow', *Inf. Softw. Technol.*, 2002, **44**, (7), pp. 405–417

27 Flowring Technology Corp., available at: http://www.flowring.com, (accessed May 2006)

28 The Workflow Management Coalition: 'Terminology and glossary', Document Number WFMC-TC-1011', February 1999