國 立 交 通 大 學

資 訊 工 程 學 系

碩 士 論 文

使用狀態圖在工作流程規格上

進行定義 Artifacts 確認

Using State Diagrams to Validate Artifact

Specifications on Primitive Workflow Schema

研究生：許薰任

指導教授：王豐堅 教授

中華民國九十四年八月

使用狀態圖在工作流程規格上進行定義 Artifacts 確認

Using State Diagrams to Validate Artifact Specifications on Primitive Workflow Schema

研究生：許薰任　　Student：Hsun-Jen Hsu

指導教授：王豐堅 博士　　Advisor：Dr. Feng-Jian Wang

國立交通大學

資訊工程學系

碩士論文

A Thesis
Submitted to Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
In Partial Fulfillment of the Requirements
For the Degree of Master
In

Computer Science and Information Engineering

August 2005

HsinChu, Taiwan, Republic of China

中華民國九十四年八月

使用狀態圖在工作流程規格上進行定義 Artifacts 確認

研究生：許薰任　　　　　指導教授：王豐堅 博士

國立交通大學

資訊工程研究所

新竹市大學路 1001 號

摘要

資料在工作流程設計中扮演一個重要角色。雖然主導流程設計是以控制為主，但對於資料操作的相對關係，仍可造成工作流程執行錯誤或遺漏資訊等等。對於工作流程中資料的相關研究，總是遠遠少於對於工作流程結構正確性的研究；更甚者，資料常被視為資源的一部份，只對著墨於如何分配於各工作上或透過介面化的存取等，對於操作的正確性與相對於控制流的一致性，卻鮮少被提及。本篇論文著眼於驗證資料的正確性，並提供一設計方法便於擷取資料操作資訊，同時此方法也適用於工作流程再用與再設計。針對資料的驗證，我們提出六大資料錯誤與偵測演算法。未來，將在把資源分析與權限認證研究導入本設計方法。

關鍵字： 工作流程、資料、驗證、狀態圖

Using State Diagrams to Validate Artifact Specifications on Primitive Workflow Schema

Student: Hsun-Jen Hsu Advisor: Dr. Feng-Jian Wang

Institute of Computer Science and Information Engineering

National Chiao Tung University

1001 Ta Hsueh Road, Hsinchu, Taiwan, ROC

Abstract

Structural correctness and resource allocation are two major topics of workflow researches, but few researchers are interested in artifact analysis. A workflow execution may fail because of incorrect structures or resource conflict. On the other hand, inaccurate artifact manipulation will bring some problems during workflow execution too, e.g., inconsistency between data flow and control flow, contradictions between artifact operations…, etc. The thesis studies a model, compatible with most models for specifications, and thus a simple methodology for validating the use of artifacts. In the model, we also present a bottom-up workflow design method based on artifact operations. The concept of state transitions for artifacts is adopted to construct six types of artifact inaccuracy impacting on workflow execution, and inaccuracy detection algorithms in order to validate artifact usages in workflow specifications.

Keywords: workflow, artifact, validation, state transition, state diagram.

誌謝

本篇論文的完成，首先要感謝我的指導教授王豐堅博士兩年來不斷的指導與鼓勵，讓我在軟體工程及工作流程的技術上，得到很多豐富的知識與實務經驗。另外，也非常感謝我的畢業口試評審委員留忠賢博士、葉義雄博士以及廖珗洲博士，提供許多寶貴的意見，補足我論文裡不足的部分。

其次，我要感謝實驗室的學長姊們，有博士班懷中學長參與討論，以及其他學長姐在研究生涯上的指導與照顧，讓我學得許多做研究的技巧，得以順利的撰寫論文。

最後，我要感謝我的家人，由於有你們的支持，讓我能心無旁騖地讀書、作研究然後到畢業，此外，謝謝好友育安這段時間的照顧，以及智宏、舜禹的打氣，在我遇到挫折時能互相勉勵。由衷地感謝你們大家一路下來陪著我走過這段研究生歲月。

Table of Contents

List of Figures

Chapter 1. Introduction

Workflow is the computerized facilitation or automation of a business process, in whole or part [1]. In other words, workflow is a set of works which are systematized to achieve certain business goal by completing each work in particular order under automatic control. And the execution order of these works, equivalent to control flow or control logic, is under some constrains and conditions according to workflow specification. On the other hand, resources are necessary for workflow implementation, and they support process execution. A resource might be any type of entity required in a work. Sample resources include participants, software components, hardware machines, documents, electronic data, and tables in database. Resource allocation [2] and resource constrain analysis [3] are popular topics of workflow research. But few works [4] deeply consider about data flow within workflow.

Artifact is an abstraction of all data instances within workflow. Introducing artifact validation into control-oriented workflow designs will help keeping consistency between execution order and date transition, and avoid the exceptions caused by contradiction between data flow and control flow. Whether a workflow schema is executable depends on its own structural correctness. In contrast to structural correctness, accuracy in artifact manipulation can help determine whether the execution result of workflow is meaningful and desirable. The majority of experiences in validation and verification indicate that the knowledge of a domain expert on business rules, process logic and environment constrains is necessary [4].

Misarrangement of artifacts has impact on correctness of workflow execution. In terms of data validation in workflows, there is very little reported in literature [4]. Designers usually drew a set of data flow diagrams ad hoc to examine consistency between control flow and date flow. Sahzia Sadiq identified three implementation models for process data requirements and data flow: explicit data flow, implicit data flow through control flow, and implicit data flow through process data store [4]. Those models are not much different from traditional method, and they need to be constructed by designers with experiences. In addition, workflow data patterns [21] were introduced to capture various ways in which data are represented and utilized in workflows. But no data validation/verification issues are introduced. Researches on verification of workflow schema [22] [23] focus on the modeling and analysis of workflow schema but not on artifact operations precisely.

In this thesis, we provide an editing model for designing a workflow and address six types of artifact inaccuracy. Associated with the model, an artifact validation technique is applied before deploying the workflow schema. This model is based on component-based design technique and control flow design. It facilitates workflow design reusing, has compatibility with other control-oriented workflow design models, and provides an easier way to extract knowledge of artifact operations in workflow. Our artifact validation algorithm brings lots of advantages during workflow design phase. When designers edit or adjust workflow specification, reports of consistency checking between data flow and control flow and information of manipulating artifacts are automatically provided to designers.

The rest of this thesis are organized as follows. Research background and literature review are presented in Chapter 2. Chapter 3 describes our system architecture including workflow design methods and design criteria. Chapter 4 defines certain properties of artifacts, introduces a technique of artifact state diagram to describe artifact-state transition, and discusses six types of artifact inaccuracy. In Chapter 5, a set of algorithms are presented to validate artifact accuracy for each workflow schema constructed. Chapter 6 demonstrates our artifact validation algorithm and reduction algorithms using an example workflow schema. Chapter 7 concludes our research in this thesis, and future works.

Chapter 2. Background

The tools can help designers to maintain correctness of the developed workflow. There are many topics about correctness of workflow. For example, structural correctness focuses on soundness of control logic [7], process model analysis, workflow patterns [8] [9], and automatic control of workflow process [10]. Resource management contains resource allocation constrains [2] [3], resource availability [11], resource management [12], resource modeling [13], etc.

2.1 Related Work

Shazia Sadiq [4] presented data flow validation issues in workflow modeling, including identifying requirements of data modeling and seven basic data validation problems. She suggested that type, source, and structure of data are essential requirements of data modeling. These requirements and data validation problems are sensible and reasonable. However, there was no discussion about any implementation or formal method to demonstrate how to apply their researches and which types of workflow model are compatible with their activity-based data model.

Chengfei Liu proposed a three level bottom-up workflow design method to effectively incorporate confirmation and compensation in case of failure [11]. This model is expected to incorporate both compensation and confirmation into a workflow management environment. In the model, data resources are modeled as resource classes, and the only interface to a data resource is via a set of operations together with their compensations and conformations. And the work facilitates wrapping legacy systems and developing compensations and conformations in some workflows with invoking legacy systems. Indeed, it may be extended with artifact constrains etc. for validation topics of artifact.

Pinar Senkul [2] presented an architecture to model and schedule workflow with resource allocation constrains and traditional temporal/causality constrains. In his architecture, the workflow specification language can model resource information and resource allocation constrain, and the scheduler model can incorporate a constrain solver to find proper resource assignment. On the other hand, Hongchen Li [3] claimed 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 certain constraint descriptions for their researches. In fact, workflow specifications can also be extended with data constrains for more precise validation and verification of data resource consistence.

Duk-Ho Chang [14] and Jin Hyun Son [15] identified and extracted the workflow critical path from context of the workflow schema. They presented extraction procedures from various non-sequential control structures to sequential paths so that appropriate sub-critical paths in non-sequential control structures are obtained. The concept of structure extraction can be utilized not only in resource or time management, but also in validating causal relationship between data operations in workflow. Specially, Jin Hyun Son [15] defined a well-formed workflow based on the concepts of closure and control block. He claimed the well-formed workflow is free from structural errors and complex control flows can be made by nested control blocks.

Wasim Sadiq and Maria E. Orlowska [16] presented a visual verification approach and algorithm with a set of graph reduction rules to identify structural conflicts in process models for the given workflow modeling languages. The structural conflicts in the complex workflow structures are easily identifiable by the incremental reduction. The concept of incremental reduction can be utilized in data validation, too.

John Thangarajah, Lin Padgham and Michael Winikoff [19] [20] proposed Iteration Tree to maintain summary information about definite/potential conditional requirement and resulting effects of goals and their associated plans for detecting and avoiding interference between goals in intelligent agents. The classification of definite and potential conditional requirement and summary calculations can be adapted for propagating state requirements of artifact operations in artifact validation.

2.2 Motivation and Goal to Achieve

A workflow application with well structure and sufficient resource still has possibility to fail or to get an unexpected execution result in execution. One of the factors to impact the workflow execution result is artifact manipulation. For example, artifact operations might be out of order, or inconsistency with control flow. However, research topics on artifacts are little concerned with verification and validation on workflow schema. Within their approach, artifacts are usually treated as one type of resource accessed by activities in workflow execution. Few researchers are interested in or working on artifact-relevant topics, and there is no validation algorithm provided to validate or check accuracy of artifact during workflow design.

It is possible to abstract artifact operations for analysis without considering semantic problems. Correspondingly, it might be valid to analyze artifact-state transition and provide useful information about artifact operations within workflow specification. A systematic artifact state diagram, generated automatically, seems helpful to keep track of artifact operations distributed in workflow specification. A multiple layer designed methodology is another topic we are interested in. Also, it is expected that workflow reuse and redesign are easier by abstracting artifact operations and separating workflow design and activity design.

In summary, it is interesting to construct a tool which can automatically provide critical analysis for artifact validation and check out artifact consistency to encourage designers during workflow design.

Chapter 3. System Architecture

3.1 Three Layer Workflow Model

To validate artifact accuracy on workflow specifications, we propose a simple type of workflow schema. This primitive workflow schema is reduced form well-formed workflow schema [15] and it can satisfy four primitive types of workflow structure defined in [1], which are "sequential", "parallel", "conditional branch", and "iterative structure". Besides, it is consider with: 1) the technique in [11] 2) representing artifacts the state transition diagrams with object-oriented techniques as in UML, 3) state joining for propagating artifact operations, 4) and state mapping for detecting artifact accuracy. Our model, on the view point of design, is named Three Layer Workflow Model (TLWM), as in Figure 3.1. TLWM represents workflow with three layers. A sample approach to designing methods applied in each layer is composed of workflow design, activity design and artifact design distinctly:

1. Workflow Design Layer: Describing the logistical control or execution order between activities, such as the sequence, choice, synchronization, and iteration… etc.
2. Activity Design Layer: Arranging artifact operations to be executed in an activity and conditions at the initialization and completion of the activity.
3. Artifact Design Layer: Defining the classes of artifacts, and all valid methods/operations of each artifact class.

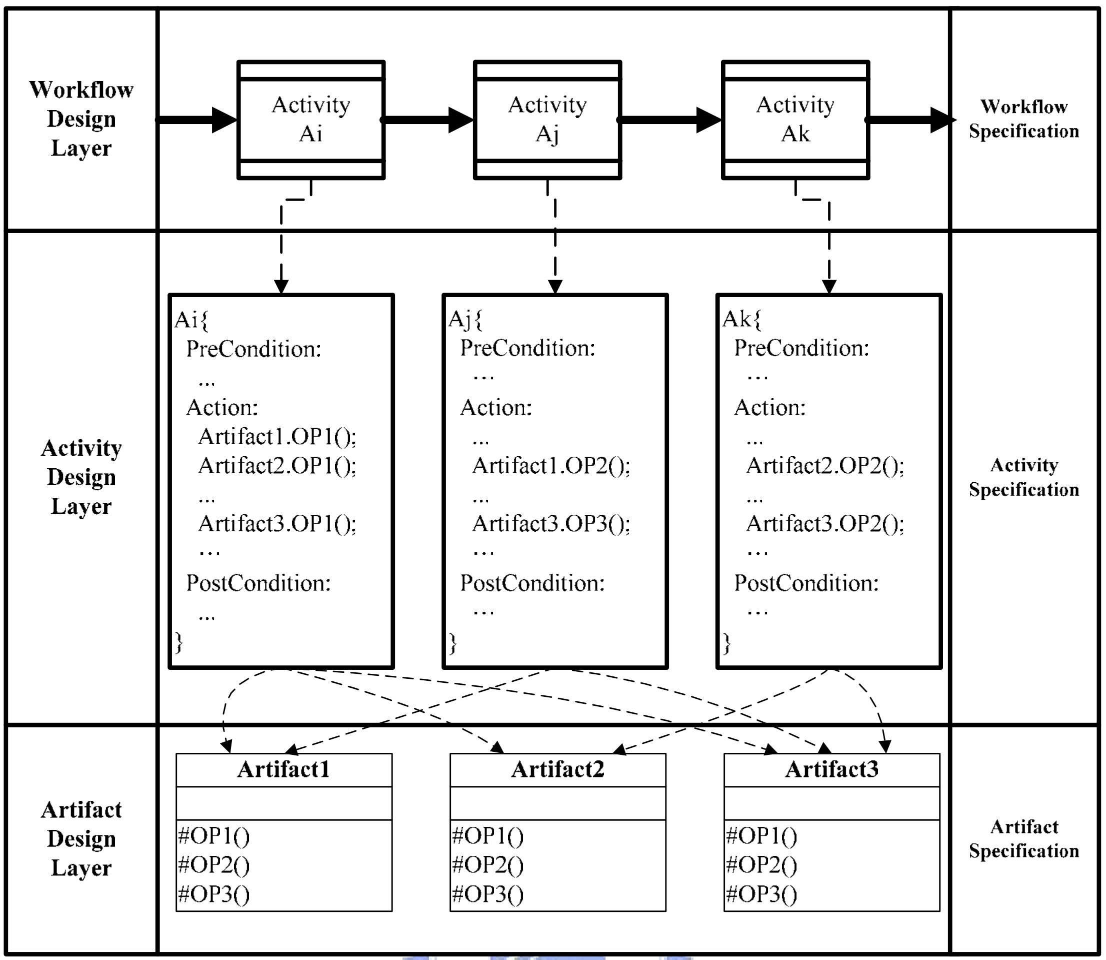

Workflow Design Layer
Activity Ai
Activity Aj
Activity Ak
Workflow Specification
Activity Design Layer
Ai{ PreCondition: … Action: Artifact1.OP1(); Artifact2.OP1(); … Artifact3.OP1(); … PostCondition: … }
Aj{ PreCondition: … Action: … Artifact1.OP2(); … Artifact3.OP3(); … PostCondition: … }
Ak{ PreCondition: … Action: … Artifact2.OP2(); … Artifact3.OP2(); … PostCondition: … }
Activity Specification
Artifact Design Layer
Artifact1 #OP1() #OP2() #OP3()
Artifact2 #OP1() #OP2() #OP3()
Artifact3 #OP1() #OP2() #OP3()
Artifact Specification

Figure 3.1 Three Layer Workflow Model

3.2 Workflow Design Layer

In Workflow Design Layer, a workflow model is used to describe a workflow schema as the product of workflow design layer, based on the concept of well-formed workflow [15] to describe workflow schemas. The product of this layer is a workflow specification, and it is designed to describe the dependence between activities, i.e. the execution order of activities in a workflow. Control structures of a workflow form the execution order of the activities in the workflow. The four primitive control structures defined in [1] are "sequential", "parallel", "conditional branch", and "iterative structure".

In this thesis, a workflow specification is specified in a primitive workflow schema in Definition 3.2, which is constructed with control and activity nodes according to certain syntax rules. The basic unit of works is an activity, and it has pre-condition (entry criteria), post-condition (exit criteria), and activities with artifact operations, resource manipulations…, etc. An activity is usually called an **activity node** in contrast to a control node. **Control nodes** are used for constructing the control structures, and there are four types of primitive control nodes in our thesis: AND-SPLIT (AS), AND-JOINT (AJ), XOR-SPLIT (XS), XOR-JOINT (XJ) and LOOP. The connection between two nodes is **flow,** which indicates a transition from one node to another.

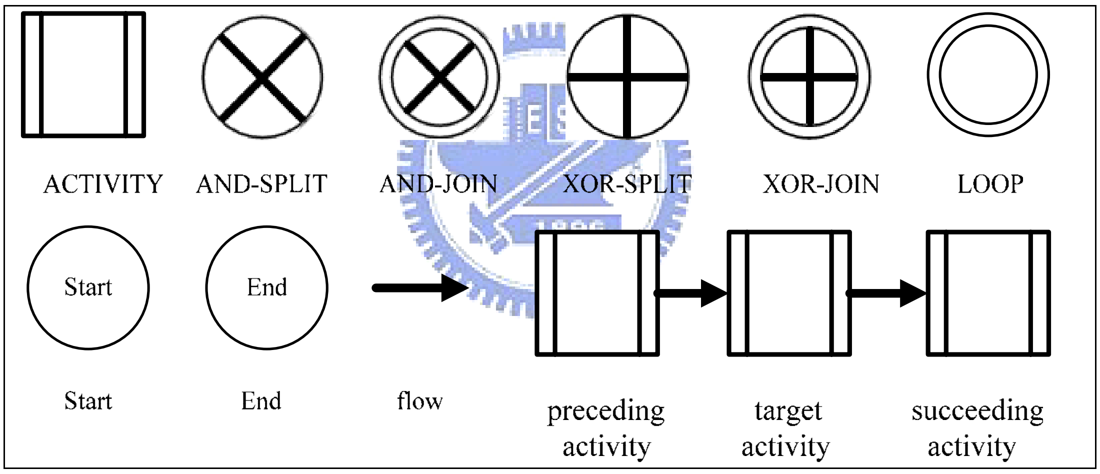


Figure 3.2 Notations in Workflow Diagram

Figure 3.2 shows the corresponding notations of control nodes, activity node, and flow. In addition, LOOP control node has two subtypes, LOOP-START and LOOP-END. The former indicates the start point of iteration, and the latter indicates the termination. The notation of a flow is a line with one arrow. The direction of an arrow implies the execution order. An activity node has only one inflow and one outflow, and a control node might have multiple inflows or outflows. Finally Start node and End node take responsibility for initializing and finalizing a whole workflow.

Our workflow schema is represented with a set of nodes and directed edges, standing for the flow between nodes. It begins from Start node and terminates with End node. The intermediary nodes are arranged with workflow control structures. We define a primitive workflow model for discussing in this thesis.

Definition 3.1 Control Block

Any subflow in a workflow schema that satisfies the following conditions is called a control block.

1. A subflow begins from one control node and terminates at another corresponding control node.
2. There are only one inflow (entrance) and outflow (exit) for this subflow.
3. A subflow is either **completely contained** in another subflow or not contained in any other subflow.

In Definition 3.1, condition 1 constrains that leading control node and ending control node of a control block are paired, such as AND-SPLIT and AND-JOINT, XOR-SPLIT and XOR-JOINT, LOOP-START and LOOP-END. Condition 2 localizes a control block as a closed subflow. Condition 3 eliminates the possibility of subflow interleavings. For example, there are two control blocks A and B. Control block A contains a set of nodes denoted by α, and control block B contains a set of nodes denoted by β. In our model, there are exactly four possible relations between α and β.

1. $\beta \subset \alpha$: A completely contains B, and A is **super-block** of B.
2. $\alpha \subset \beta$: B completely contains A, and A is **sub-block** of B.
3. $\alpha \cap \beta = \phi$: Control block A and B do not overlap with each other.

4. $\alpha = \beta$: Control block A and B are identical.

If a control block does not contain any other control block, it is called an **atomic control block**. Otherwise, it is called a **nested control block**. If an activity node is not contained in any control block, it is called an **un-blocked activity**. A control block is called a **top-level control block** if it is not contained in any other control block.

Definition 3.2 Primitive Workflow Schema

A workflow schema built by a set of un-blocked activities nodes and top-level control blocks is called a primitive workflow schema.

According to definitions mentioned above, we conclude four types of workflow construction in our primitive workflow schema as follows.

1. **Sequential Block**: The activities in this block are executed in sequence under a single thread of executions. There is no control node or conditional branch between these activities. The main characteristic is that the target activity will execute after its preceding activity completes. 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 will be executed repetitively until certain conditions are met.
3. **AND Control Block**: All outflows of an AND-SPLIT node are executed parallel and converge synchronously into an AND-JOIN node. An AND-JOIN node takes charge of synchronizing all threads from parallel inflows into the thread.
4. **XOR (eXclusive OR) Control Block**: An XOR-SPLIT node makes a decision upon which branches to take from multiple alternative outflows (workflow branches). And these branches converge to a single XOR-JOIN node. There is no

synchronization required because of no parallel thread executed.

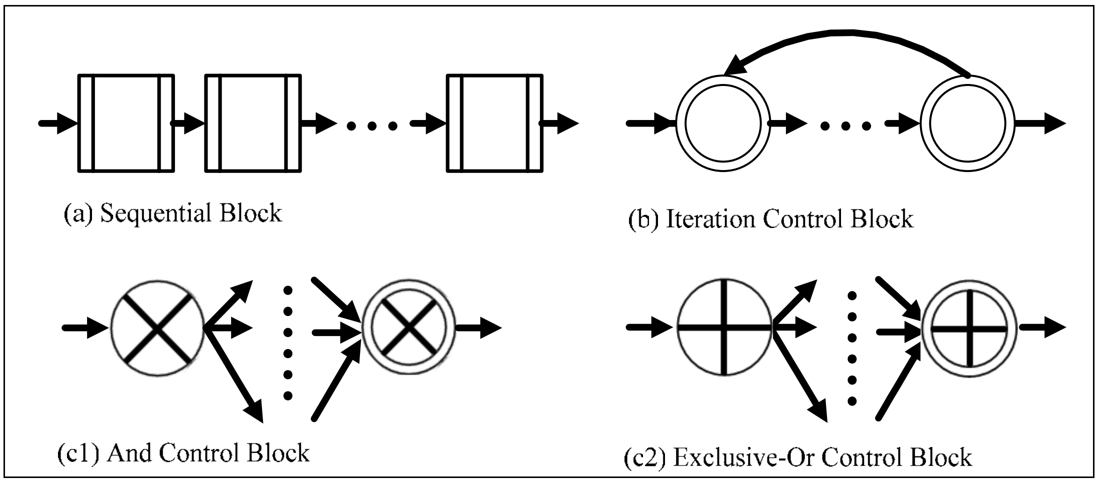


Figure 3.3 Four Types of Workflow Construction in a Primitive Workflow Schema

Figure 3.3 shows four types of workflow construction, and these workflow constructions can contribute to construct a primitive workflow schema. In addition, we use the notation *(leading node, ending node)* to indicate a block, starting from the leading node and terminating at ending node, in following sections.

3.3 Activity Design Layer

Activity specifications are anticipative products of this layer. Each activity has its own activity specification, which describes the operations associated with artifacts to perform to achieve the goal. Besides the manipulations of artifacts, there are also pre-condition and post-condition (optional) to be defined. The format of activity specifications are shown in Figure 3.4.

An activity is the smallest unit of work in workflow, and it performs a set of operations on certain artifacts to achieve its designed goal. All operations of an artifact which performs must be specified in the corresponding artifact specifications. Besides, each activity may be designed with pre-condition and post-condition if necessary. Artifact(s) acted on and/or concerned in pre-condition/post-condition in this activity will be retrieved to contribute to artifact analysis and validation.

```
ACTIVITY_NAME{
PRE-CONDITION:
        //A logical expression evaluated by a workflow engine
        //to decide if this activity instance may be started or not.
        //It may refer to certain artifact to check the current state of artifact.
ACTION:
        //A sequence of operations performed on artifacts to achieve a goal.
        ARTIFACT_NAME.OPERATION_NAME(PARAMETER_LIST);
POST_CONDITION:
        //A logical expression evaluated by a workflow engine
        //to decide if this activity instance is completed or terminated by failure.
        //It may refer to certain artifact or external events etc.
}
```

Figure 3.4 Script of Activity Specification

3.4 Artifact Design layer

Artifact design layer is the bottom layer, and its contents are artifact specifications. All artifacts participating in a workflow execution must be pre-defined in artifact specifications. For each artifact, its specification contains a set of operations for legally manipulating on its internal data. An activity designed to manipulate a certain artifact can only performs these operations to interact with this artifact.

The main concept of artifact design layer is to make the internal design of artifacts independent from the workflow application. That brings following profits:

- An artifact has a set of interfaces, representing the operations on the artifact. Modifications on implementing these operations will not impact the upper layer design if the format to invoke these operations keeps steady.

- The artifact manipulations in activities or workflows are bound in these (designed) specifications. Illegal or invalid operation on artifacts is not allowed. It can also provide an implementation platform for authentication, and that is our future work.
- Independent designing of artifact operations facilitates classifying and grouping types of artifact operations. And it also provides an easier way to extract useful information of artifact operations in workflow execution.

Figure 3.5 shows format of an artifact specification.

```
ARTIFACT    "ARTIFACT_NAME"
        INTERNAL:
        //Declare internal data, data source or destination
        OPERATION:
        RETURN_TYPE OPERATIPON_NAME (ARTUMENT_LIST) {
        //Code for artifact manipulations
        }
```

Figure 3.5 Script of Artifact Specification

Chapter 4. Artifacts

In this chapter, we will discuss the concept of artifact, artifact state diagram, state transition, and artifact inaccuracy and the corresponding outcomes.

4.1 Overview of Artifacts

An artifact describes any the data participating in workflow execution no matter its type is input, output, temporal or permanent. Moreover, it can be extended by combining some processing logic if necessary, such as e-form in AgentFlow [17]. An artifact can represent workflow control data, workflow relevant data, application data [18], electronic document, manual data, and every data item participating in workflow execution. In this thesis, we reduce the description power of artifacts, and focus on the artifacts visible in workflow specification (schema) by TLWM introduced in chapter 3.

Artifacts can be divided into five types:

1. **Reference (Ri)**: A reference artifact in workflow is like the static variable or constant in programming language. It is rarely modified after initializing, and acts as an index of a certain instance or case. For example, the serial number of an official document, the number of identification card, etc.
2. **Manipulation (Mi)**: A manipulation artifact acts as the processing target in/among activities. It can be an input, output, or processing results of an activity. For example, the name of last editor, the total amount of goods, the population statistic…, etc.
3. **Deterministic (Di)**: A deterministic artifact acts as the key condition result evaluated. It participates in the branch of control nodes (parallel, choice,

Iteration…, etc.), and conditional constrains in activity (pre-condition, post-condition, processing logic…, etc.). For example, the credit gained (if the credit gained is less then 24, the student is not allowed to apply for graduation.), department, or major…, etc.

4. **Composite (Ci)**: A composite artifact is a composition of sub-artifacts. For example, an application form, an official document, a business contract…, etc.
5. **Property (Pi)**: A property artifact is used to control process or the 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 constrains, etc.

The classification of artifacts in type is neither absolute nor exclusive. Some artifacts are Mi and also are Di. In this thesis, we would not emphasize the type of artifacts discussed. The behavior and effect of artifact operations is the key interested. Although the types of artifacts are not mentioned clearly here, the range of artifacts approximately contains Ri∪Mi∪Di∪Ci.

In general, the whole workflow is the scope of artifacts, including workflow engine, workflow instance, invoked application…, etc. However, in this thesis, we diminish the scope and only focus on artifact appearing in activities and workflow specification.

We group common operations of artifact into five types, which are **Specify**, **Read**, **Write**, **Revise** and **Destroy**.

1. **Specify**: an initialization activity of artifact, such as "fill in", "create", "define"…, etc.
2. **Read**: a reference to artifact, such as "use", "fetch", "select", "retrieve"…, etc.

3. **Write**: an artifact modification, such as "write", "change", "update"…, etc.
4. **Revise**: addition/deletion of a sub-artifact, such as merge, combine, divide…, etc.
5. **Destroy**: deletion of artifact, such as remove, erase, cancel, discard, etc.

4.2 Artifact State Diagrams

Generally, process design and flow design settle major part of workflow design, but the artifact design does not usually. Here we define a method based on state transition technique to model both behavior changes and state transitions of an artifact during workflow execution. There are five primitive operations, introduced in Section 4.1 for triggering artifact-state transition.

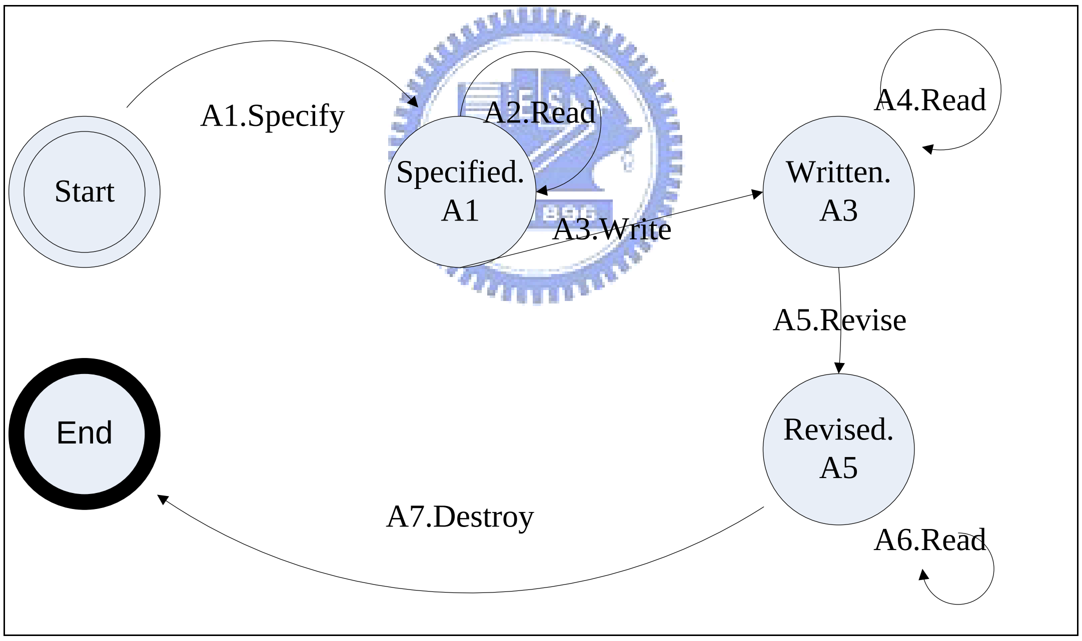


Figure 4.1 An Example of an Artifact State Diagram

There is an example demonstrated in Figure 4.1; it shows the life cycle of an artifact during workflow execution. During its life time, the artifact is specified by activity A1, read by A2, written by A3, read by A4, revised by A5, read by A6 and finally destroyed by A7. Furthermore, this artifact state diagram implies one reasonable execution order of these activities.

Definition 4.1 shows definition of artifact state diagram discussed in this thesis.

Definition 4.1 Artifact State Diagram

Artifact state diagram is 5-tuple ＜Q, O, T, q0, F＞

- Q: a finite set of states s1, s2… si, for i ≧1
- O: an input operation of artifact. O = [Specify | Read | Write | Revise | Destroy].
- T: transition function Q × O → Q
- q0: start state or initial state
- F: a set of final states, F⊆Q

The artifact state diagram is constructed by two components, state (circle) and transition (arc with arrow). The first transition starts from initial state q0 by default. There are some criteria for drawing artifact state diagram as follows.

1. $\forall s_i \in Q, \exists p_i \in O \text{ and } s_j \in Q, \text{ s.t. } s_j \xrightarrow{p_i} s_i$

2. $\exists s_i \in Q, \ p_i \in O \text{ and } p_i = \text{Read} \Rightarrow s_i \xrightarrow{p_i} s_i$

3. $\exists s_i, s_j, s_k \in Q, \ p_i, p_j \in O \text{ and } s_i \xrightarrow{p_i} s_j \xrightarrow{p_j} s_k \Rightarrow p_i \prec p_j$

Criterion 1 tells that every state has at least one input operation to trigger transition from itself to next state. Criterion 2 emphasizes that a transition transfers from one state to the same state if the type of input operation is Read. Criterion 3 indicates the input order of operations. There are two transitions among s_i, s_j, and s_k. And p_i and p_j are input operations for these transitions. We can conclude that operation p_i is performed before p_j in workflow execution order, and notation $\prec$ indicates a temporal relation between operations.

We will use artifact state diagram mentioned in this chapter to trace the artifact transition between activities within workflow schema and to validate artifact accuracy.

4.3 Artifact Inaccuracy

Inappropriate operations on artifact might lead the workflow (execution) into failure or cause illegal side effects. In this section, we point out the abnormal situations of artifact operations and discuss the potential problems they may bring.

We infer six types of artifact inaccuracy from cases observed. They are *No Producer*, *No Consumer*, *Redundant Specify*, *Contradiction*, *Parallel Hazard* and *Branch Hazard*.

4.3.1 No Producer

In general, the first operation on an artifact in workflow is Specify, acting as initialization. *No Producer* problem indicates that an artifact has a different operation earlier than Specify. It indicates that this workflow might fail due to retrieval error or an exception of missing target artifact. Figure 4.2 describes the five state transitions; four of them causing *No Producer* problem and the one reasonable. The exception case is that this artifact is created by invoked application / outer system or is an existing artifact before this workflow execute. *No Producer* problem is a warning for potential error but not absolute one.

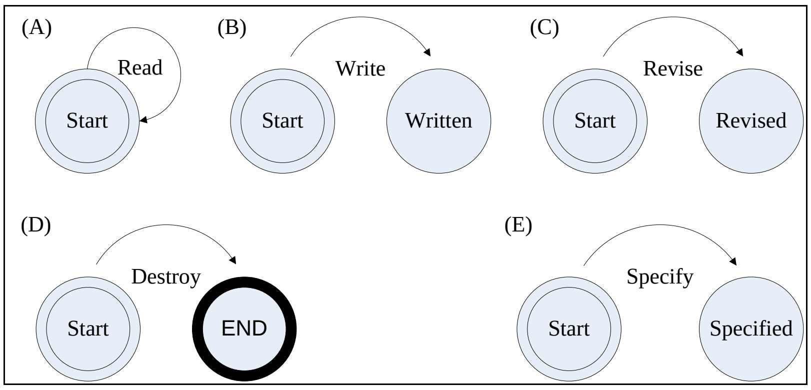


Figure 4.2 No Producer Problem (part A, B, C, D) and Expectant (part E)

4.3.2 No Consumer

No Consumer problem means that there is no activity requesting the artifact after last modification, which might be Specify, Write or Revise. There are two situations which this problem occurs in. First, this artifact is designed to be manipulated in this workflow and packed as a result artifact for access of external system. Second, this artifact is redundant and no succeeding activity (control node) for the access it. How to solve this problem depends on designers' desire or is according to system requirement.

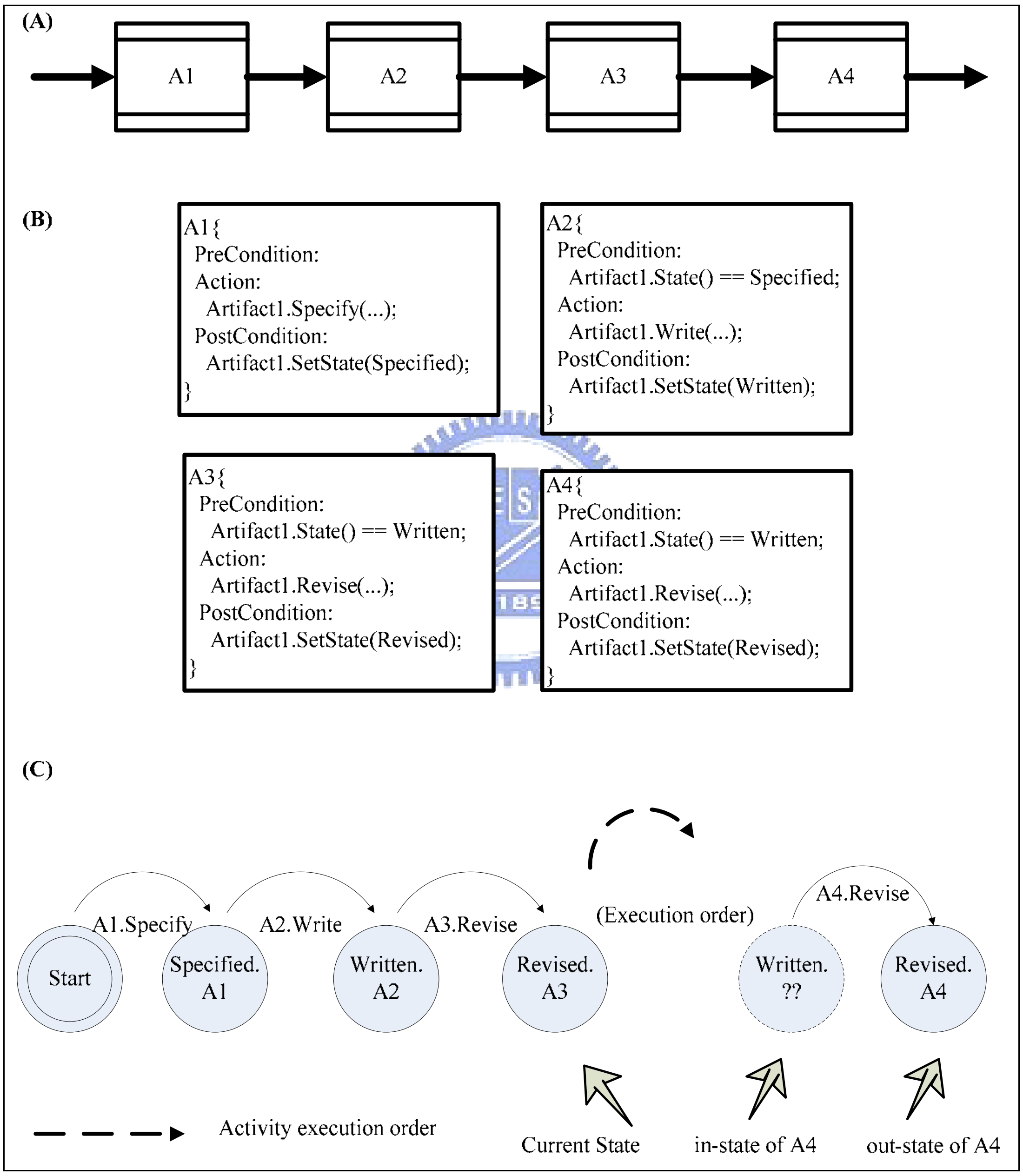
4.3.3 Redundant Specify

Redundant Specify problem indicates that there is another specified state following the first specified state. In other words, the artifact specified in current activity is specified by succeeding activity again. It will cause the confusion in maintaining artifacts and make exceptions in execution.

4.3.4 Contradiction

Contradiction problem describes a situation that current artifact state does not conform to the in-state specified in the pre-condition of a succeeding activity. Figure

4.3 shows a simple example of *Contradiction* problem. Figure 4.3(A) is a workflow schema shows the execution order of activities A1, A2, A3 and A4. Figure 4.3(B) is a set of activity specifications with pre-condition and post-condition. There is a *Contradiction* problem between A3 and A4.



```
A1{
  PreCondition:
  Action:
    Artifact1.Specify(...);
  PostCondition:
    Artifact1.SetState(Specified);
}
```

```
A2{
  PreCondition:
    Artifact1.State() == Specified;
  Action:
    Artifact1.Write(...);
  PostCondition:
    Artifact1.SetState(Written);
}
```

```
A3{
  PreCondition:
    Artifact1.State() == Written;
  Action:
    Artifact1.Revise(...);
  PostCondition:
    Artifact1.SetState(Revised);
}
```

```
A4{
  PreCondition:
    Artifact1.State() == Written;
  Action:
    Artifact1.Revise(...);
  PostCondition:
    Artifact1.SetState(Revised);
}
```

Figure 4.3 Contradiction Problem

In an activity specification, the pre-condition and post-condition provide a mechanism to specify the in-state before and out-state after the execution of an activity. More state constrains specified in pre-condition and in post-condition will make state-matching more precisely.

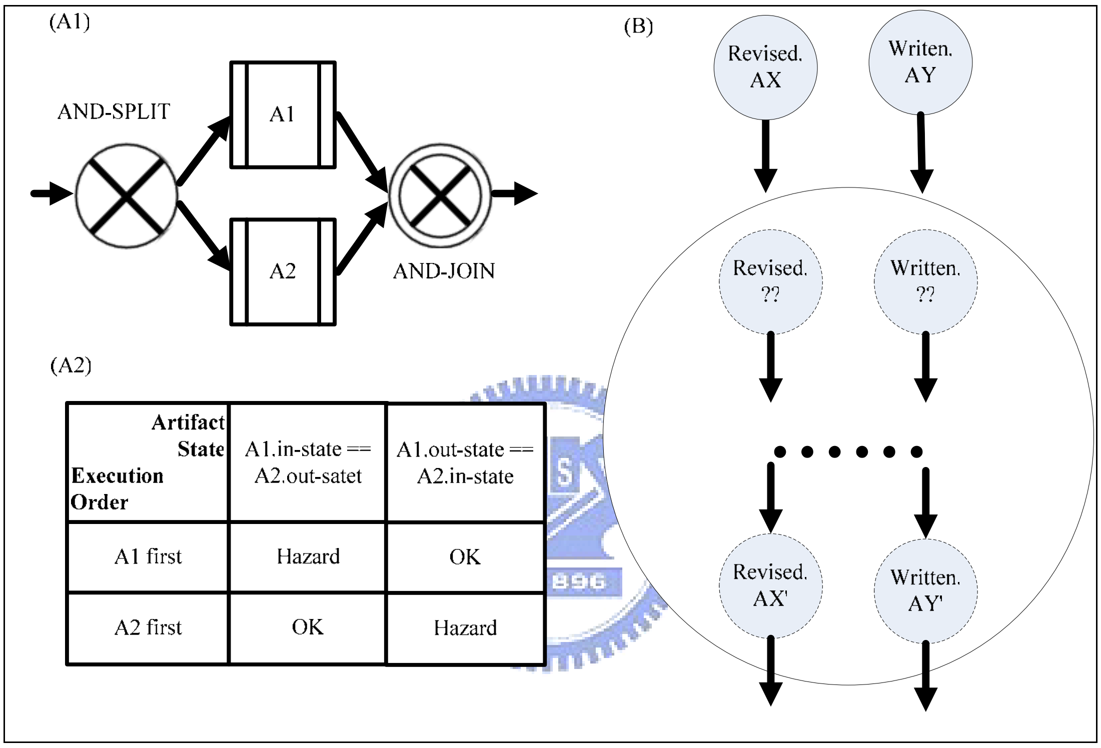
4.3.5 Parallel Hazard

Parallel Hazard problem happens because of conflict interleaving of concurrent artifact operations in activities. When more than one concurrent subflow manipulates an artifact in parallel, and the activities in different subflows on the artifact are not in a deterministic execution order expect the concurrency constrains are defined.

The main characteristic patterns of artifact-state transition are concurrent operations and competition of state-mapping. *Parallel Hazard* problem will be recognized if there are multiple concurrent subflows operating the same artifact. Besides, multiple state choices of incoming flow to an AND control block and multiple produced states of an AND control block are two symptoms of state-mapping competition. And state-mapping competition is prerequisite for potential *Parallel Hazard*. In Section 5.4, multiple produced states of out-states and multiple concurrent subflows operating the same artifact are two major patterns of Parallel Hazard detection.

Figure 4.4 illustrates two simple examples of Parallel Hazard problem. In Figure 4.4(A1), Activity A1 and A2 are concurrent activities which are not in a strict execution order. Figure 4.4(A2) shows four situations that A1 and A2 have dependence of artifact manipulation. If the execution order conforms to dependence of artifact operations, there is no problem. Otherwise, Parallel Hazard will occur. Figure 4.4(B) shows an example of state-mapping competition in a composite state extracted from a AND control block.

Regarding below five types of operation, only concurrent operations of Read will not cause *Parallel Hazard*. Other combinations of operations on the same artifact will bring *Parallel Hazard* potentially or absolutely. More explicit state and transition describing artifact (constrains of in-state or conditions of out-state) will make *Parallel Hazard* detection more precisely.



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

Figure 4.4 Parallel Hazard

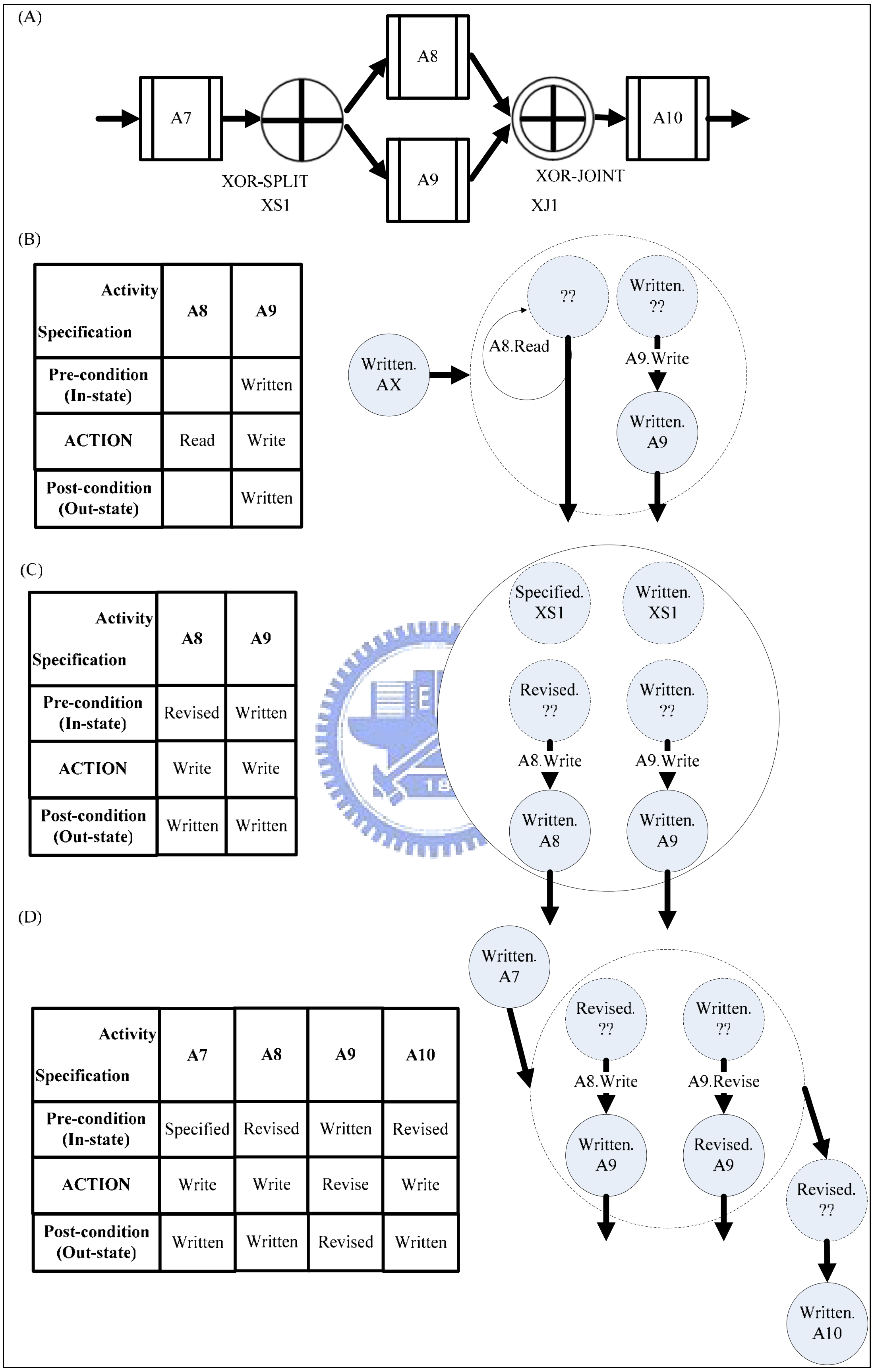
4.3.6 Branch Hazard

Branch Hazard may be produced from an XOR control block because of the possibility of selecting branch subflows, which contain operations on artifacts. For example, the result of artifact operations within a branch subflow, not selected for execution, can contribute to succeeding artifact operations outside of current XOR control block. *Branch Hazard* problem will occur at this case.

Another *Branch Hazard* is that there is no artifact state consistency between the condition testing in XOR-SPLIT node and branch subflows. If a condition testing in XOR-SPLIT node is relative to the state of some artifacts, the state-mapping will be performed to detect *Branch Hazard*. If artifact state constrains on XOR-SPLIT are not totally compatible to in-state sets of branch subflows, there is a *Branch Hazard* detected inside the control block. The last situation of Branch Hazard is that there is losing out-state or insufficient in-state occurring between this XOR control block and outside of it.

There are three types of *Branch Hazard* as shown in Figure 4.5, which are hidden effect, condition mismatch, and insufficient in-state/losing out-state. Figure 4.5(A) is a partial workflow schema, containing a XOR control block and two activities A7, A9. Figure 4.5(B) is *Branch Hazard* of hidden effect. Figure 4.5(C) is *Branch Hazard* of condition mismatch. And Figure 4.5(D) is *Branch Hazard* of insufficient in-state/losing out-state.

These Six types or artifact inaccuracy mentioned above might impact on workflow execution potentially or absolutely. Following chapter will introduce how to extract information of artifact manipulations and detect these artifact inaccuracies to achieve artifact validation on workflow schema.



(B)

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

(C)

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

(D)

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

Figure 4.5 Branch Hazard

Chapter 5. Artifact Validation for Primitive Workflow

Artifact state diagrams and state transition criteria assist our work in tracing and validating manipulation of artifacts in workflow. TLWM and characteristics of primitive workflow schema reduce the complexity of artifact validation. Artifact inaccuracy mentioned in Chapter 4 makes a set of cases that might impact on the correctness of artifact manipulation. Our approach to validate the use of artifacts is based on workflow reduction and the state tracing on artifacts.

There are three steps during validation. The first is to extract state transitions from artifact manipulation within activities. The second is to perform the state mapping of artifacts within sequential blocks. At the same time, an artifact state diagram is constructed and artifact inaccuracy detection is achieved. The last is to perform the state combinations of artifacts in each control block according to its characteristic. *Parallel Hazard* and *Branch Hazard* are two noticeable artifact inaccuracy problems with control block; their detections are performed during combination of sub artifact-state transitions.

Extracting artifact-state transition here is a bottom-up style. It finds artifact-state transitions of each local block, and combines these artifact-state transitions according to their relative positions during workflow execution. The corresponding algorithms are presented one bye one where each algorithm handles a distinct workflow structure and inside the algorithm one artifact is considered only to prevent reciprocal effect between multiple artifacts.

5.1 A General Reduction Algorithm

Our artifact extraction algorithm constructs an artifact state diagram for each artifact to search the artifact inaccuracy in a primitive workflow schema. Due to the characteristics of primitive workflow schema, the input workflow schema is a sequence combination of un-blocked activities and top-level control blocks. Each atomic control block and each sequential block are reduced to corresponding composite activity nodes, and the reduction procedure can repeat bottom-up until there is no block left. During the reduction, artifact state tracing and validations both proceed. The bottom-up approach is done in a primitive workflow schema.

Algorithm 1. Validate a Primitive Workflow Schema – validate a primitive workflow schema by reducing it to one composite activity node and extracting the artifact-state transition to validate artifact accuracy.

```
StartNode = starting node of the input workflow schema;
EndNode = ending node of the input workflow schema;
ArtifactStateDiagram ASD = initializing artifact-state diagram;
TargetNode = Reduce_SequentialBlock(StartNode.next, EndNode.previous);
ASD = ExtractAST(TargetNode);
Trace ASD to search for No Producer, No Consumer, and Redundant Specify;
Output the artifact state diagram and artifact validation result;
```

Algorithm 1 processes the input primitive workflow schema, starting from Start Node and ending at End Node defined in Section 3.2, and output the produced artifact-state diagram and validation result. The input workflow schema is considered as a sequential block and proceeded by sub routine **Reduce_SequentialBlock**, describing in Algorithm 2 of Section 5.2. The output artifact-state diagram is the record of one artifact manipulating life cycle and meaningful for artifact analysis.

5.2 Reduction Algorithm for Sequential Block

In a sequential block, all activity nodes in this block are in sequential order. If the input schema contains a control block, this control block will be reduced as a composite activity node before reduction of the outer sequential block. Algorithm 2 describes algorithm **Reduce_SequentialBlock**. **Reduce_SequentialBlock** extracts artifact-state transitions from the nodes in the input block, connects them, create a composite activity node with the artifact-state transition, and replace the input block by this new created node.

Algorithm 2. Reduce_SequentialBlock(Node *Ni*, Node *Nj*) – reduce a sequential block, starting from *Ni* and ending at *Nj*, into a composite activity node and return it.

```
CurrentNode = Ni;
ArtifactStateTransition CurrentAST = initializing artifact-state transition;
WHILE ( CurrentNode is in the block (Ni, Nj) ) {
   IF ( CurrentNode.type == ActivityNode ) THEN
      // connect artifact-state transitions by sequential order
      CurrentAST = Sequential_Join(CurrentAST, ExtractAST(CurrentNode));
      CurrentNode = CurrentNode.next();
   ELSE IF ( CurrentNode.type == LOOP-START ) THEN
      CurrentNode = Reduce_IterationControlBlock(CurrentNode);
   ELSE IF ( CurrentNode.type == AND-SPLIT ) THEN
      CurrentNode = Reduce_ANDControlBlock(CurrentNode);
   ELSE IF ( CurrentNode.type == XOR-SPLIT ) THEN
      CurrentNode = Reduce_XORControlBlock(CurrentNode);
}
Create a composite activity node Nij with CurrentAST;
Nij.previous = Ni.previous;
Nij.next = Nj.next;
RETURN Nij;
```

Extract_AST extracts artifact-state transition from an activity node and is defined in Algorithm 3. Sequential_Join called by Reduce_SequentialBlock is in charge of connecting two artifact-state transitions in sequential order, and its detail is described in Algorithm 4. Reduce_IterationControlBlock, Reduce_ANDControlBlock and Reduce_XORControlBlock reduce control blocks in the input block, and they will be introduced in following sections.

Algorithm 3. Extract_AST(ActivityNode *Ni*) – extract artifact-state transition from an activity node *Ni*

```
// a composite activity with an artifact-state diagram
IF (Ni has an artifact-state transition bound in it) THEN
    RETURN the artifact-state transition;
// no artifact operation
IF (Ni has no operation on the artifact) THEN
    RETURN a null artifact-state transition;
// Read artifact operation
IF (the artifact operation of Ni is Read type) THEN
    Create a pseudo in-state Si according to pre-condition of Ni;
    Create a transition arc from Si to itself labeled as Read;
    RETURN this artifact-state transition;
// other types of artifact operation
Create a pseudo in-state Si according to pre-condition of Ni;
Create an out-state Sj according to post-condition of Ni;
Connect Si and Sj with a transition arc labeled as the type of operation;
RETURE this artifact-state transition;
```

In Algorithm 3, there are four cases when Extract_AST is performed on the input activity node *Ni*:

1. *Ni* is a composite activity node reduced from a certain block and is associated with an artifact-state transition computed from reduction

algorithms. The artifact-state transition is returned directly.

2. *Ni* is an activity node with no operations on the artifact, and a null artifact-state transition is returned.
3. *Ni* is an activity node with Read operation on the artifact, and a transition of Read type is generated and returned.
4. *Ni* is an activity node with a not-Read operation, and an artifact-state transition is generated according to Pre-condition, Action, and Post-condition in the activity specification.

In case 1, the artifact-state transition might have multiple in-states (out-states) because it might be a combination of multiple artifact-state transitions of subflows inside the block. For an artifact-state transition with multiple in-states (out-states), each state of in-states (out-states) will be classified into four types:

1. *Definite*: this state is extracted from a composite activity node reduced from an XOR control block. Each *Definite* state might have ***Branch Hazard*** with other *Definite* state.
2. *Potential*: this state is extracted from a composite activity node reduced from an AND control block. Each *Potential* state might have ***Parallel Hazard*** with other *Potential* state.
3. Both of *Definite* and *Potential*: this state is propagated from AND and XOR control blocks. It might have ***Branch Hazard*** with other *Definite* state and ***Parallel Hazard*** with other *Potential* state.
4. *Transparent*: a transparent state is computed from an XOR control block to stand for a branch subflow which has no artifact state constrains or artifact operations.

These four classes of artifact state are useful in artifact-state joining and validation. More will be expressed in Algorithm 4 ,sections 5.4 and 5.5.

In Algorithm 4, Sequential_Join connects two artifact-state transitions in sequential order, and detects Artifact Inaccuracy. In this stage, Contradiction and Branch Hazard tests will be performed.

Algorithm 4. Sequential_Join(ArtifactStateTransition ***STi*,** ArtifactStateTransition ***STj*) –** join two state transitions *STi* and *STj*.

```
IF ( $STi$ == NULL ) THEN RETURN $STj$;
IF ( $STj$ == NULL ) THEN RETURN $STi$;
$Si$ := $STi$.out-state; $Sj$ := $STj$.in-state;
SWITCH ( (|$Si$|,|$Sj$|) )
   CASE (1,1):
      IF ( $si \in Si, sj \in Sj, si$ and $sj$ are compatible ) THEN
         Combine $si$ and $sj$;
      ELSE
         Alarm Contradiction between $STi$ and $STj$;
   CASE (>1,1):
      IF ( $sj \in Sj, \forall si \in Si, si$ and $sj$ are not compatible ) THEN
         Alarm Contradiction;
      IF ( $\exists si \in Si.Definite, sj \in Sj, si$ and $sj$ are not compatible ) THEN
         Alarm Branch Hazard; // losing out-state
      IF ( $sj \in Sj, \exists si \in Si, sj$ and $si$ are compatible ) THEN
         Combine $si$ and $sj$;
   CASE (1,>1):
      IF ( $si \in Si, \forall sj \in Sj, si$ and $sj$ are not compatible ) THEN
         Alarm Contradiction;
      IF ( $si \in Si, \exists sj \in Sj.Definite, sj$ and $si$ are not compatible ) THEN
         Alarm Branch Hazard; // insufficient in-state
      IF ( $si \in Si, \exists sj \in Sj, si$ and $sj$ are compatible ) THEN
         Combine $si$ and $sj$;
   CASE (>1,>1):
      IF ( $\forall si \in Si, \forall sj \in Sj, si$ and $sj$ are not compatible ) THEN
         Alarm Contradiction;
      IF ( $si$ and $sj$ are not compatible WHERE
          $\forall si \in Si.Definite, \forall sj \in Sj$ OR $\forall si \in Si, \forall sj \in Sj.Definite$ ) THEN
         Alarm Branch Hazard // losing in-state or insufficient out-state
      IF ( $\exists si \in Si, \exists sj \in Sj, si$ and $sj$ are compatible ) THEN
         Combine $si$ and $sj$;
END SWITCH
RETURN the joined artifact-state transition;
```

5.3 Reduction Algorithm for Iteration Control Block

An iteration control block is a block of nodes beginning from a Loop-Start and ending in a Loop-End. The procedure to transform an iteration control block into a composite activity node is listed in Algorithm 5. A new sequential block, created by unrolling the loop body twice, substitutes for the original iteration control block and then is processed by Reduce_SequentialBlock. The produced composite activity node is returned.

Algorithm 5. Reduce_IterationControlBlock(Node *Ni*) – reduce an iteration control block starting from *Ni* into a composite activity node and return it.

```
// locate the input iteration control block (Ni,Nj)
CurrentNode = Ni.next();
Stack CNstack; // a stack to contain control nodes
WHILE (1) {
   IF ( CurrentNode.type == LOOP-START ) THEN
      stack.push(CurrentNode); // nested iteration control block
   ELSE IF ( CurrentNode.type == LOOP-END ) THEN {
      IF ( stack.size() >= 1 ) THEN stack.pop();
      ELSE BREAK;
   }
   CurrentNode = CurrentNode.next();
}
Nj = CurrentNode;
// for simulating iteration behavior
Unroll loop body of (Ni,Nj) twice into a new sequential block (ni',nj');
ni'.previous = Ni.previous;
nj'.next = Nj.next;
// perform sequential block reduction on (ni',nj')
ActivityNode nij = Reduce_SequentialBlock(ni', nj')
RETURN nij;
```

5.4 Reduction Algorithm for AND Control Block

AND control block is a workflow construction which starts from an AND-SPLIT, ends on an AND-JOIN, and has multiple concurrent subflows (paths). The most difference between AND control block and Sequential block is concurrent contradiction, which causes Parallel Hazard defined in Section 4.3.5. To validate each concurrent subflow and to merge these subflows into a composite activity are the major work in this stage.

Algorithm 6. Reduce_ANDControlBlock(Node *Ni*) – reduce an AND control block starting from *Ni* into a composite activity node and return it

```
// locate the input AND control block (Ni,Nj)
CurrentNode = Ni.next();
WHILE (1) {
    IF ( CurrentNode.type == AND-SPLIT ) THEN
        stack.push(CurrentNode); // nested AND control block
    ELSE IF ( CurrentNode.type == AND-JOIN ) THEN {
        IF ( stack.size() >= 1 ) THEN stack.pop();
        ELSE BREAK;
    }
    CurrentNode = CurrentNode.next();
}
Nj = CurrentNode;
// reduce these concurrent subflows, and join these concurrent state transitions
ArtifactStateTransition CurrentAST = NULL;
FOR (each concurrent subflow (nk,nl) within block (Ni,Nj)) {
    ActivityNode SubNode = Reduce_SequentialBlock(nk, nl);
    ArtifactStateTransition SubAST = ExtractAST(SubNode);
    IF ( (SubAST != NULL ) THEN
        CurrentAST = AND_Join(CurrentAST, SubAST);
}
```

```
// Continue above …

// check Parallel Hazard
IF ( ( | CurrentAST.out-state.Potential| > 1 ) ||
        ( | CurrentAST.in-state.Potential| > 1 ) ||
        ( multiple concurrent artifact operations exist) ) THEN
    Alarm Parallel Hazard;
// transform this AND control block into a composite activity
Create a composite activity node Nij with CurrentAST;
Nij.previous = Ni.previous;
Nij.next = Nj.next;
RETURN Nij;
```

In Algorithm 6, there are four steps to reduce an AND control block: locating the input AND control block, reducing concurrent subflows, checking Parallel Hazard, and producing a composite activity node to substitute for the block. During reducing concurrent subflows, algorithm AND_Join is called for joining these concurrent artifact-state transitions.

```
Algorithm 7. AND_Join(STi,STj) – join two state transitions STi and STj,
where STj is not null

// NULL Join
IF (STi == NULL) THEN RETURN STj;
// indicate in-state set and out-state set
Si = STi.in-state; Ei = STi.out-state;
Sj = STj.in-state; Ej = STj.out-state;
// perform AND join calculation on STi and STj
CurrentAST.in-state = $Si \odot Sj$;
CurrentAST.out-state = $Ei \odot Ej$;
// return the new state transition
RETURN CurrentAST;
```

Algorithm 7 describes the AND_Join, which joins two input state transitions according to the characteristic of AND control block. The joining calculation function $\odot$ for AND_Join is defined in Definition 5.1.

For AND join calculation, the two input concurrent artifact-state transitions STi and STj are non-null. If there is only one state of in-state (out-state) of the input artifact-state transition, the state will be labeled as *Potential* state by default.

Definition 5.1 AND Join Calculation Function

For artifact-state transitions STi and STj, let α and β be the corresponding sets of in-states, and let γ and δ be sets the corresponding sets of out-states. To simplify the representation, we define AND join calculation with $\odot$ as follows.

$$\begin{aligned}\alpha \odot \beta &= <\alpha.D, \alpha.P> \odot <\beta.D, \beta.P>\\ &= <(\alpha.D \cup \beta.D), (\alpha.P \cup \beta.P \cup \alpha.D \cup \beta.D)>\\ \gamma \odot \delta &= <\gamma.D, \gamma.P> \odot <\delta.D, \delta.P>\\ &= <(\gamma.D \cup \delta.D), (\gamma.P \cup \delta.P \cup \gamma.D \cup \delta.D)>\end{aligned}$$

where D is sets of *Definite* states and P is sets of *Potential* states.

$$\mathrm{Si} \cup \mathrm{Sj} = \mathrm{Si} + \mathrm{Sj} - \mathrm{Si} \cap \mathrm{Sj}$$

where $\forall s \in Si \cap Sj, \exists si \in Si, sj \in Sj,$ such that *s*, *si* and *sj* are compatoble

One special case is that the types of all concurrent operations are Read, and this control block will be considered as a composite activity with a Read operation. On the other hand, if there are multiple concurrent artifact-state transitions distributed in concurrent subflows, a ***Parallel Hazard*** might occur between these subflows.

5.5 Reduction Algorithm for XOR Control Block

An XOR control block is a workflow construction which starts from an XOR-SPLIT, ends at an XOR-JOIN, and it has multiple branch subflows. The major difference between XOR control block and AND control block is that the former has only one subflow will be selected to execute according to conditions of branch. It might lead to Branch Hazard defined in section 4.3.6. How to detect ***Branch Hazard*** and how to figure out in-state(s) and out-state(s) of the new composite activity are the major work in this section.

In Algorithm 8, there are four steps to reduce an XOR control block: locating the input XOR control block, reducing branch subflows, checking Branch Hazard, and producing a composite activity node to substitute for the block. During reducing Branch subflows, there are two parts different from algorithm Reduce_ANDControlBlock: inserting *Transparent* state to the current artifact-state transition if there is a NULL artifact-state transition and calling procedure XOR_Join to join these branch artifact-state transitions. The reason to insert a *Transparent* state, which is universally compatible to any artifact states outer of the current block, is to represent a branch path which will not impact the artifact-state transition.

Algorithm 9 describes procedure XOR_Join, which joins two input state transitions according to the characteristic of XOR control block. The joining calculation function for XOR_Join is defined in Definition 5.2.

In addition, all activity nodes with the XOR control block have no operation of the artifact, and this control block will be considered as a composite activity with no artifact operation.

Algorithm 8. Reduce_XORControlBlock(Node *Ni*) – reduce an XOR control block starting from *Ni* into a composite activity node and return it

```
// locate the input XOR control block (Ni,Nj)
CurrentNode = Ni.next();
WHILE (1) {
   IF ( CurrentNode.type == XOR-SPLIT ) THEN
      stack.push(CurrentNode); // nested XOR control block
   ELSE IF ( CurrentNode.type == XOR-JOIN ) THEN {
      IF ( stack.size() >= 1 ) THEN stack.pop();
      ELSE BREAK;
   }
   CurrentNode = CurrentNode.next();
}
Nj = CurrentNode;
// reduce these branch subflows, and join these branch state transitions
ArtifactStateTransition CurrentAST = NULL;
FOR (each branch subflow (nk,nl) within block (Ni,Nj)) {
   ActivityNode SubNode = Reduce_SequentialBlock(nk, nl);
   ArtifactStateTransition SubAST = ExtractAST(SubNode);
   IF ( (SubAST == NULL) ) THEN
      CurrentAST.addState(TransparentState);
   ELSE
      CurrentAST = XOR_Join(CurrentAST, SubAST);
}
// check Branch Hazard
IF ( condition testing in the XOR-SPLIT node is not consistency with
     CurrentAST.in-states ) THEN
   Alarm Branch Hazard;
// transform this XOR control block into a composite activity
Create a composite activity node Nij with CurrentAST;
Nij.previous = Ni.previous;
Nij.next = Nj.next;
RETURN Nij;
```

Algorithm 9. XOR_Join(*STi*,*STj*) – join two state transitions *STi* and *STj*, where *STj* is non-null

```
// NULL Join
IF (STi == NULL) THEN RETURN STj;
// indicate in-state set and out-state set
Si = STi.in-state; Ei = STi.out-state;
Sj = STj.in-state; Ej = STj.out-state;
// perform XOR join calculation on STi and STj
CurrentAST.in-state = $Si \oplus Sj$;
CurrentAST.out-state = $Ei \oplus Ej$;
// return the new state transition
RETURN CurrentAST;
```

For XOR join calculation, the two input concurrent artifact-state transitions STi and STj are non-null. If there is only one state of in-state (out-state) of the input artifact-state transition, the state will be labeled as *Definite* state by default.

Definition 5.2 XOR Join Calculation Function

For artifact-state transitions STi and STj, let α and β be the corresponding sets of in-states, and let γ and δ be sets the corresponding sets of out-states. To simplify the representation, we define XOR join calculation with $\oplus$ as follows.

$$\alpha \oplus \beta = <\alpha.D, \alpha.P> \oplus <\beta.D, \beta.P>$$
$$= <(\alpha.D \cup \beta.D \cup \alpha.P \cup \beta.P), (\alpha.P \cup \beta.P)>$$
$$\gamma \oplus \delta = <\gamma.D, \gamma.P> \oplus <\delta.D, \delta.P>$$
$$= <(\gamma.D \cup \delta.D \cup \gamma.P \cup \delta.P), (\gamma.P \cup \delta.P)>$$

where D is sets of *Definite* states and P is sets of *Potential* states.

$$\text{Si} \cup \text{Sj} = \text{Si} + \text{Sj} - \text{Si} \cap \text{Sj}$$

where $\forall s \in Si \cap Sj, \exists si \in Si, sj \in Sj,$ such that *s*, *si* and *sj* are compatoble

Chapter 6. Examples for Artifact Validation on Primitive Workflow Schema

For illustrating and demonstrating our validation algorithms, an example of primitive workflow schema and the procedure of artifact validation are presented in this chapter. Figure 6.1 shows an example of primitive workflow schema.

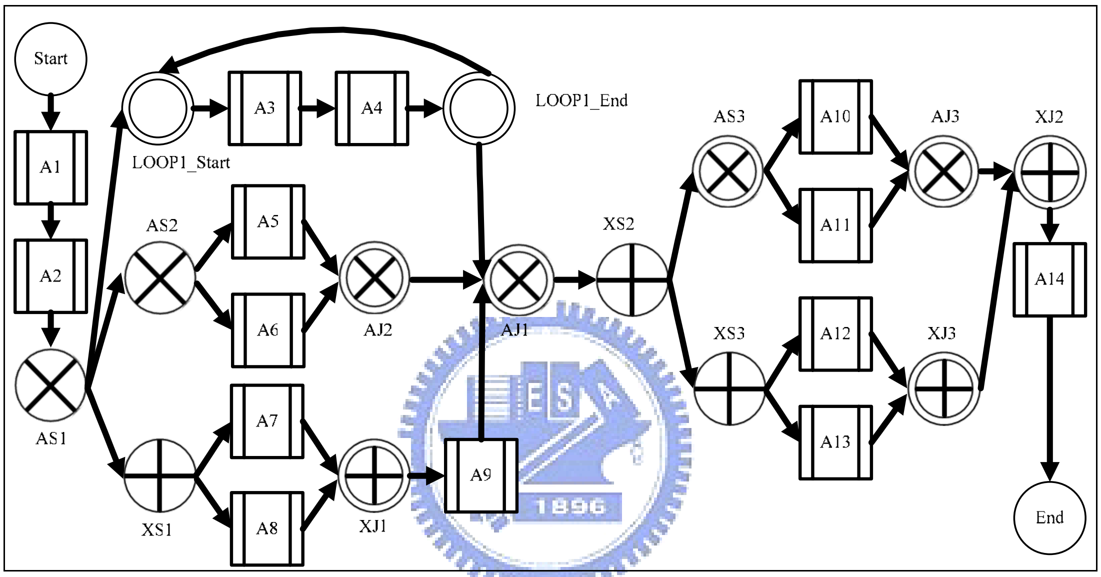


Figure 6.1 An example of Primitive Workflow Schema

In Figure 6.1, there are three un-blocked 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), XOR-1(XS1,XJ1), and activity node A9. The second top-level control block XOR-2 contains control block AND-3 (AS3,AJ3) and XOR-3 (XS3,XJ3). Figure 6.2 is a table of artifact information extracted from activity specifications of these activity nodes. The blank field means "don't care" or no artifact operation. Next, this workflow schema example is evaluated by our Artifact Validation algorithms as follows.

In Figure 6.3, activity nodes A1 and A2 are validated and constructed an artifact-state transition, but the next node is an AND control block needing procedure Reduce_ANDControlBlock to reduce to a composite activity node. In Figure 6.4, Reduce_ANDControlBlock is performed on AND1 control block, and each concurrent subflows are evaluated by Reduce_SequentialBlock. The inner control blocks are evaluated by corresponding reduction procedures. In the third concurrent subflow, a Branch Hazard is detected because of losing out-state.

In Figure 6.5, a Parallel Hazard is detected during joining concurrent artifact-state transitions. In Figure 6.6, the AND1 control block is reduced to a composite activity node, and the next node is another control block. In Figure 6.7, the succeeding control block XOR2 is proceeded by Reduce_XORControlBlock, and its branch subflows are proceed by Reduce_SequentialBlock.

In Figure 6.8, the branch subflows of XOR2 are reduced to node AND3 and node XOR3. Control block XOR2 is reduced to node XOR2 in Figure 6.9. And, in Figure 6.10, a Branch Hazard is detected because of losing out-state during sequential joining with activity node A14. Figure 6.11 is the result of artifact validation on this input workflow schema. In addition, three instances of Artifact Inaccuracy are addressed in the final artifact-state diagram.

The Branch Hazard, detected between control block XOR1 and activity node A9, indicates that one *Definite* state of 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 Pre-condition of activity A9. The Branch Hazard between control block XOR2 and activity node A14 is the same as above.

The Parallel Hazard, detected inside control block AND1, indicates a competition between these three concurrent subflows, (LOOP1_Start,LOOP1_End), (AS2,AJ2) and (XS1,A9). The interleaving between the concurrent subflows may produce contradiction problems inside the control block.

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

Figure 6.2 A Table of Activity Specifications

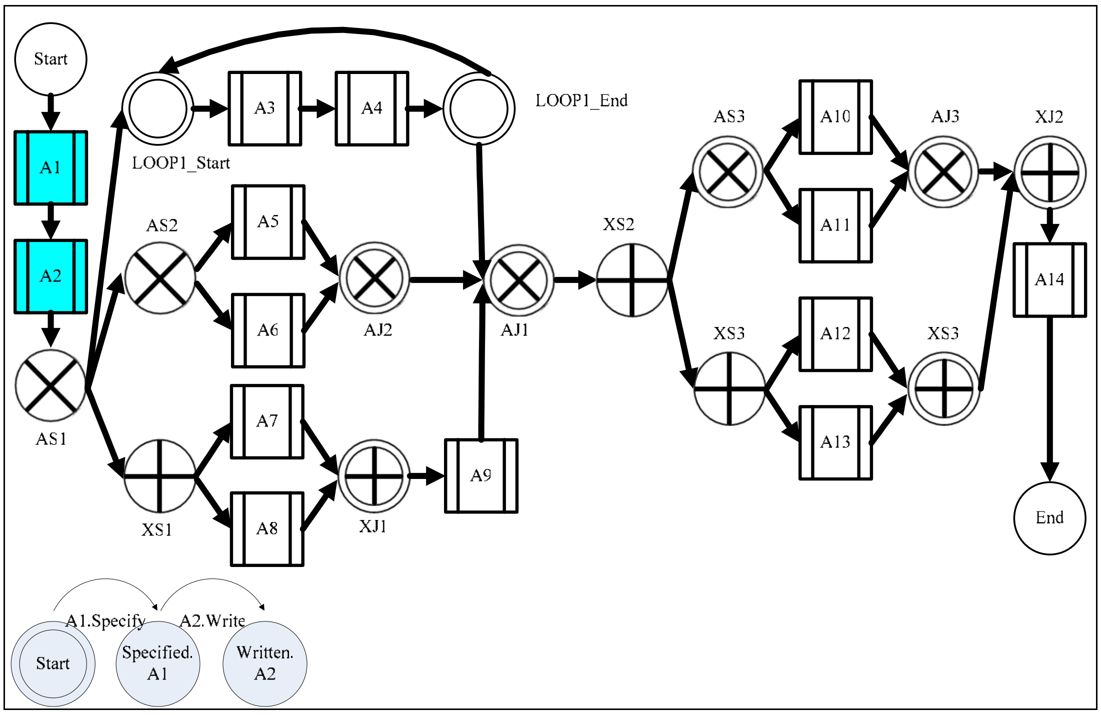


Figure 6.3 Phase 1

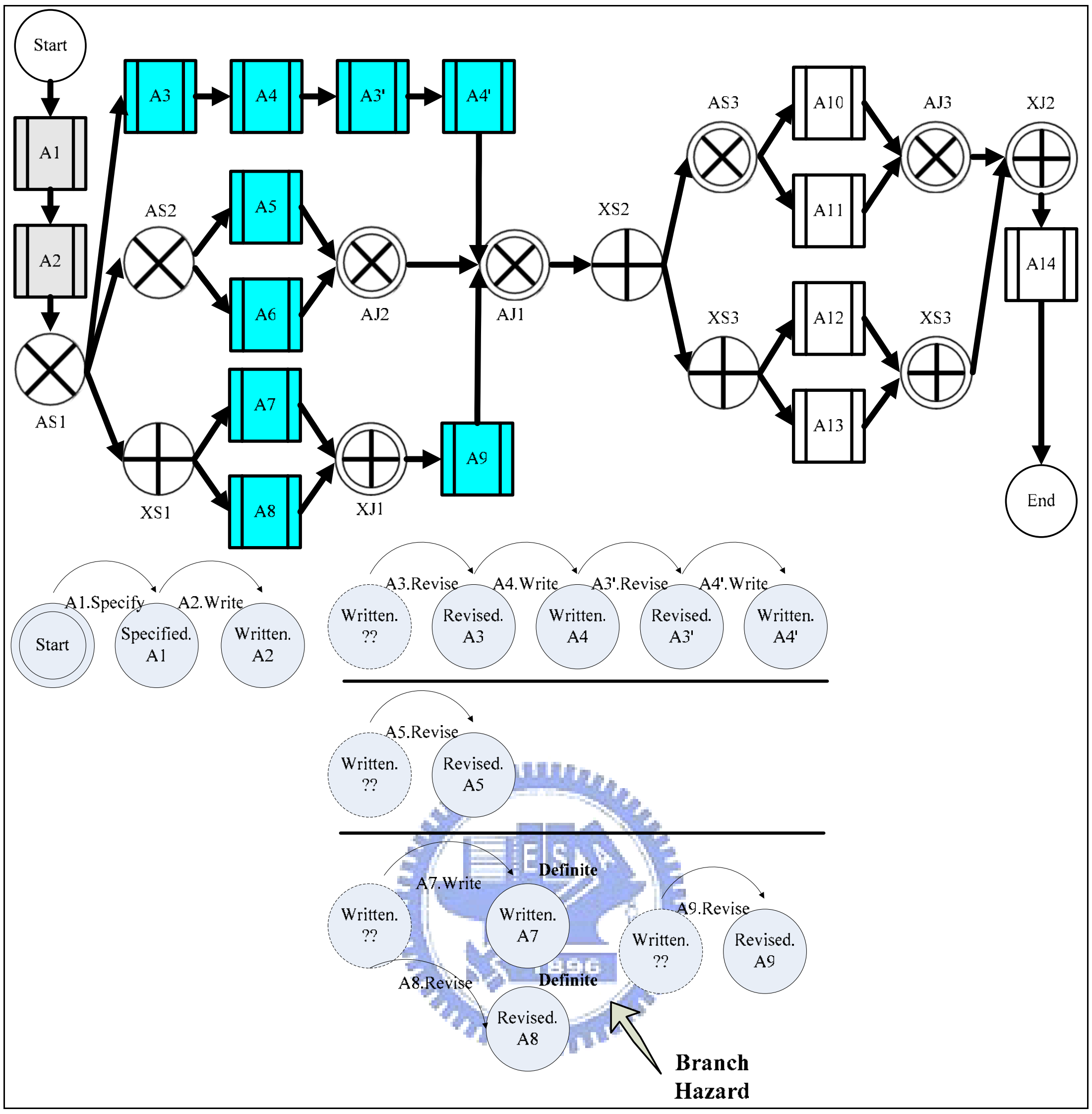


Figure 6.4 Phase 2

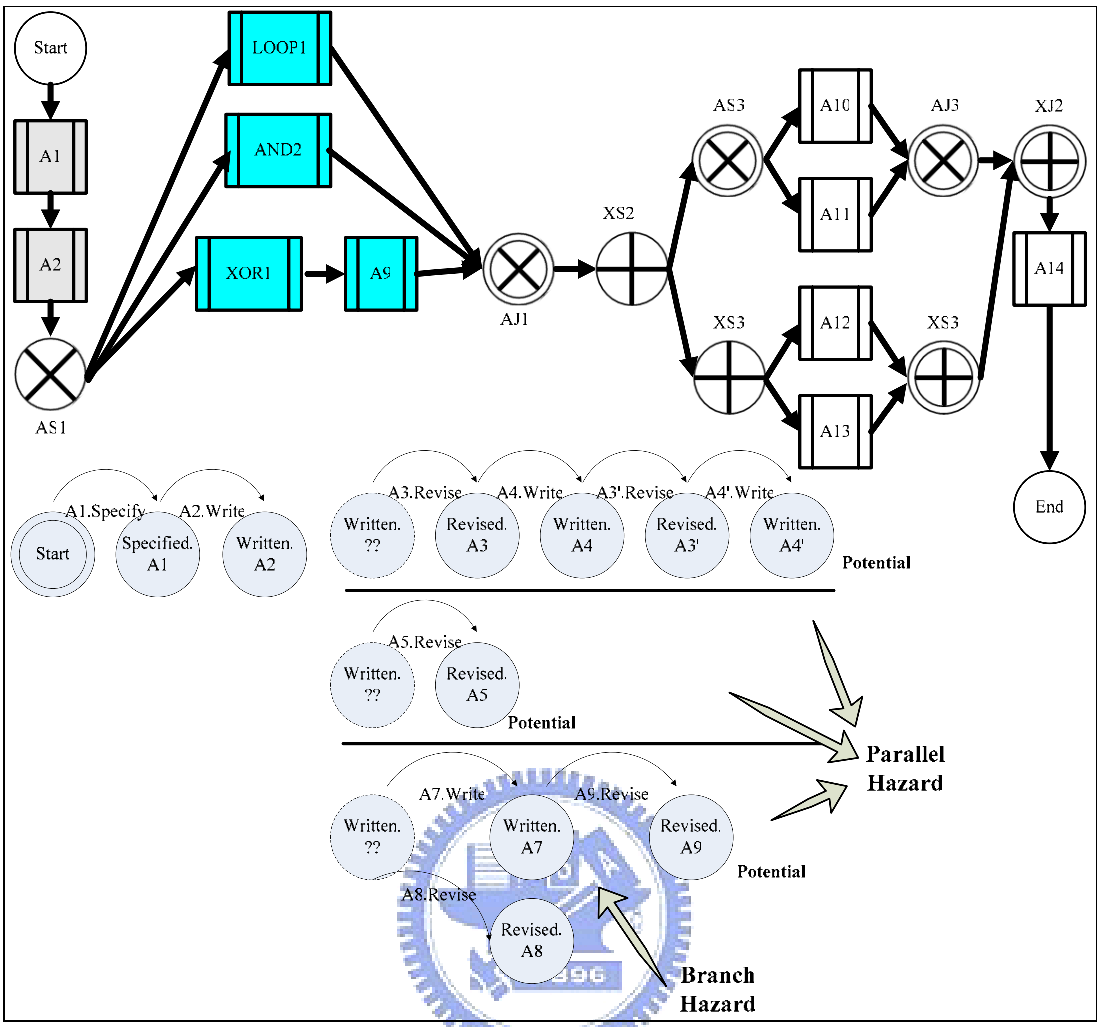


Figure 6.5 Phase 3

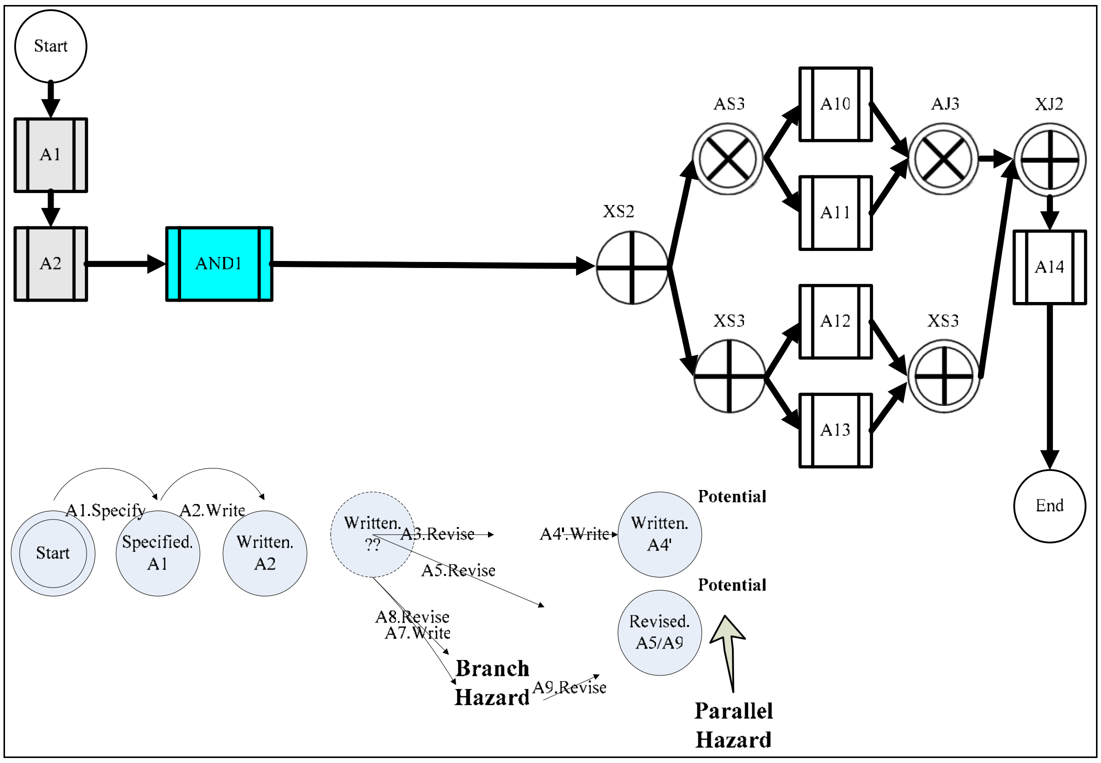


Figure 6.6 Phase 4

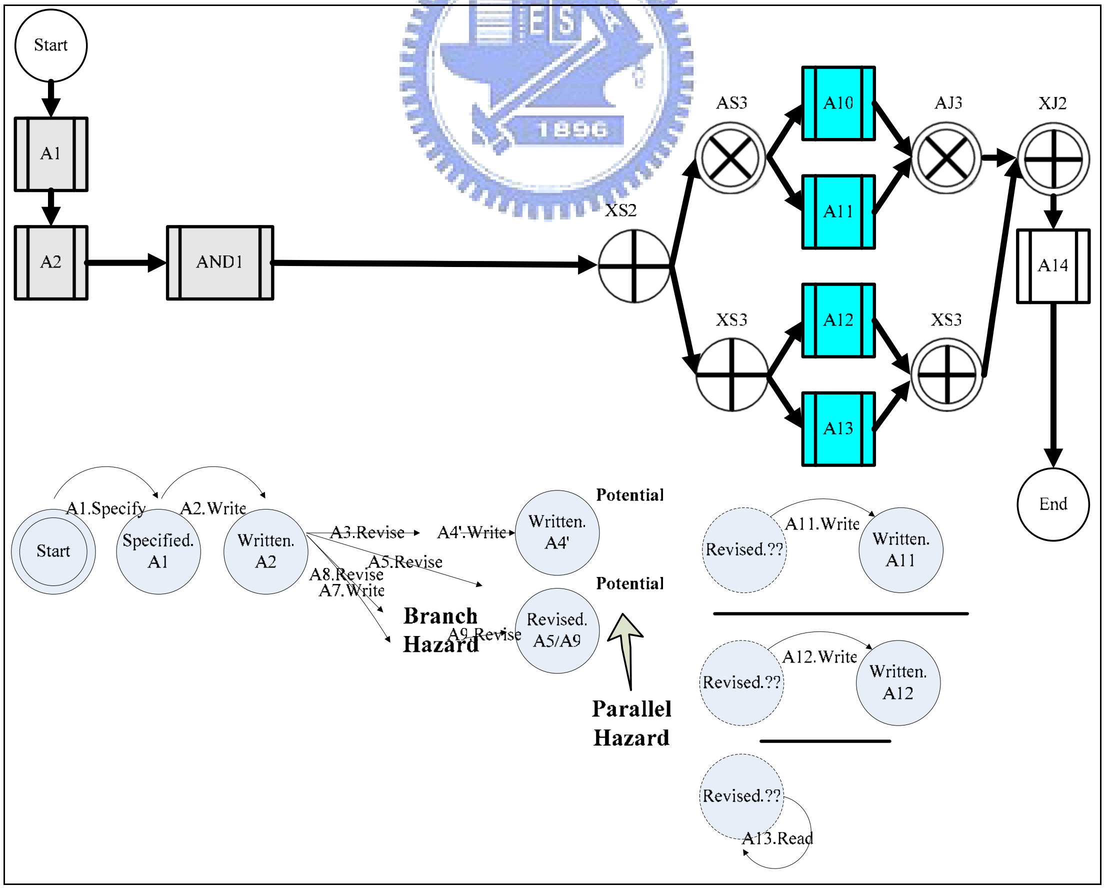


Figure 6.7 Phase 5

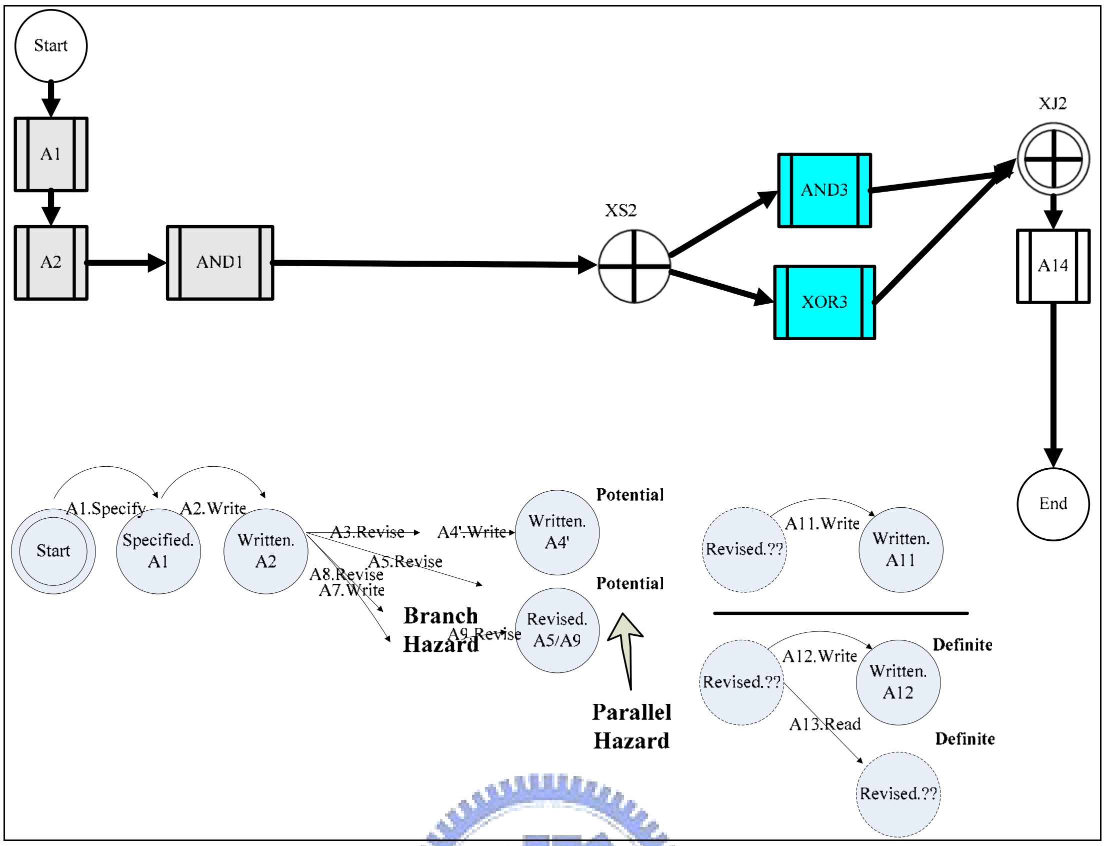


Figure 6.8 Phase 6

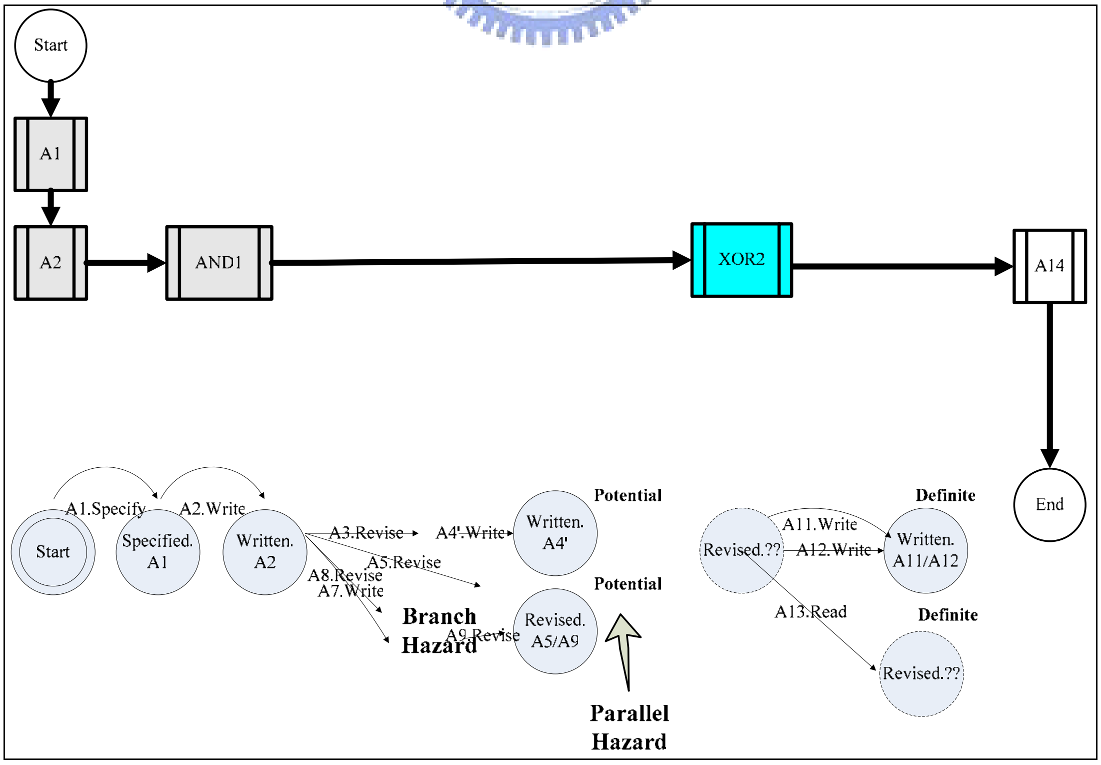


Figure 6.9 Phase 7

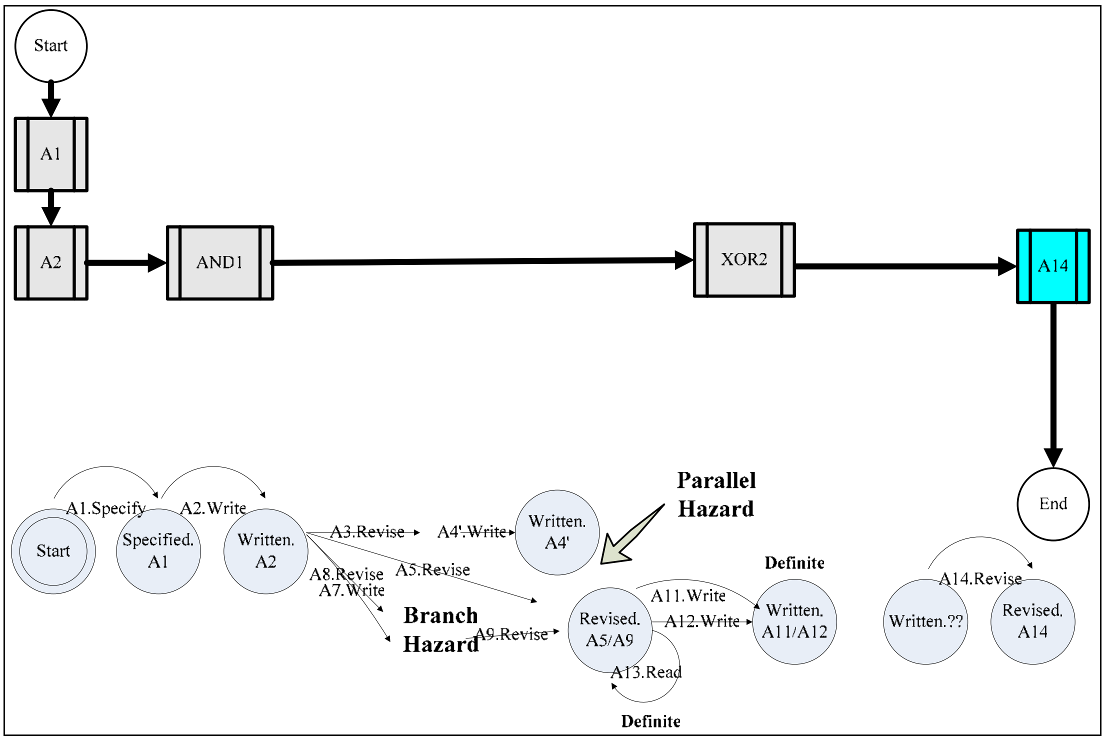


Figure 6.10 Phase 8

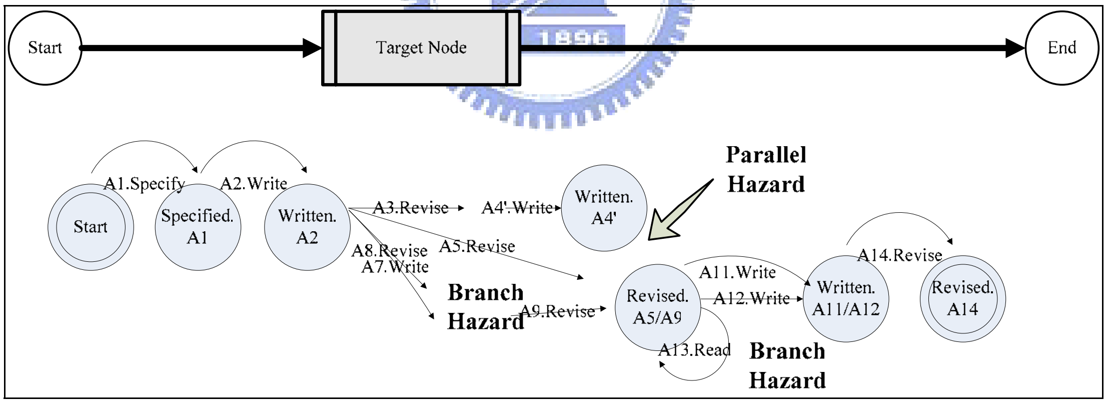


Figure 6.11 Phase 9

Chapter 7. Conclusion & Future Work

The main contribution of this thesis is to introduce a concept of automatic artifact validation during workflow design phase. To achieve this goal, Three Layer Workflow Model is introduced, where artifacts are defined with a set of operations and activities are define with artifact constrains. Then, a sequence of artifact validation algorithms is presented to validate a Primitive Workflow Schema. Besides, six types of Artifact Inaccuracy, which may impact workflow execution result, are addressed. The realistic implementation of our system architecture is beyond the scope of this thesis.

We sketched a control-oriented workflow system, which is based on our TLWM, with a visual artifact analysis tool binding to the workflow specification. Besides, the workflow application processes can be modeled as Figure 7.1.

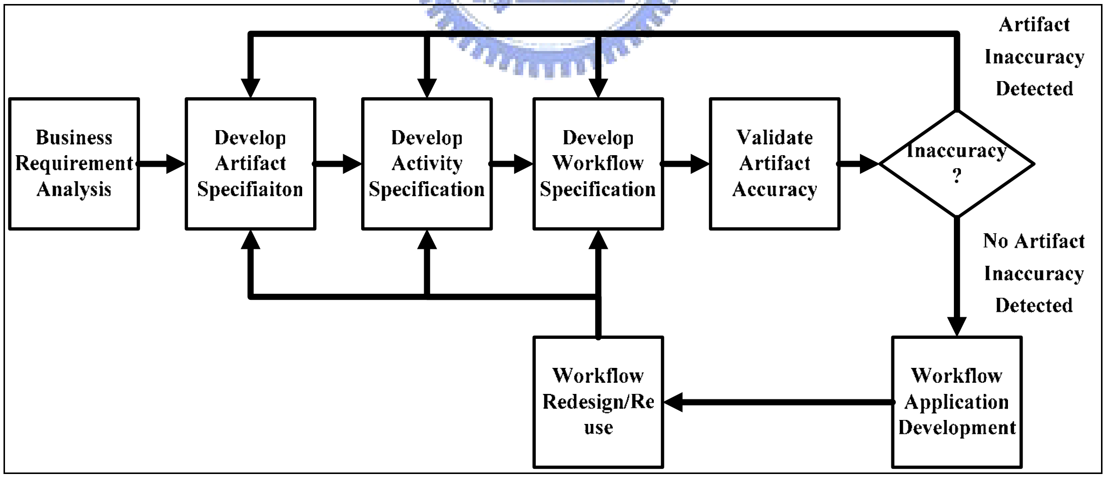

Business Requirement Analysis
Develop Artifact Specifiaiton
Develop Activity Specification
Develop Workflow Specification
Validate Artifact Accuracy
Inaccuracy ?
Artifact Inaccuracy Detected
No Artifact Inaccuracy Detected
Workflow Application Development
Workflow Redesign/Reuse

Figure 7.1 Procedure of Workflow System Development

In the future, an implementation of our algorithm is implemented first. The methodology based on Figure 7.1 will be constructed to put our research into practice. In addition, we will continue our research in composite artifact which has more complex behaviors from Revise operations, such as interaction and interference between artifacts, and introduce more state descriptions/constrains for more precise validation. Besides, resource constrains analysis and role authentication will be introduced in our workflow design method to integrate related research topic of our laboratory.

Reference

[1] The Workflow Management Coalition, "Workflow Management Coalition The Workflow Reference Model", Document Number TC00-1003, January 1995.

[2] Pinar Senkul and Ismail H. Toroslu, "An architecture for workflow scheduling under resource allocation constraints", Information Systems, Volume 30, Issue 5, pp.399-422, PERGAMON, July 2005.

[3] Hongchen Li, Yun Yang and T.Y. Chen, "Resource constraints analysis of workflow specifications", The Journal of Systems and Software, Volume 73, Number 2, pp.271–285, Elsevier Science, October 2004.

[4] Shazia Sadiq, Maria Orlowska, Wasim Sadiq and Cameron Foulger, "Data Flow and Validation in Workflow Modelling", Proceedings of the fifteenth conference on Australasian database, Volume 27, pp.207-214, Australian Computer Society, 2004.

[5] Hai Zhuge, "Component-based workflow systems development", Decision Support Systems, Volume 35, Issue 4, pp.517-536, Elsevier Science Publishers, July 2003.

[6] Arthur S. Hitomi and Dong Le, "Endeavors and Component Reuse in Web-Driven Process Workflow", Proceedings of the California Software Symposium, pp.15-20, Irvine, CA, USA, October 1998.

[7] Lei Gong and Hai-yang Wang, "A method to verify the soundness of workflow control logic", Computer Supported Cooperative Work in Design, Volume 1, pp.284-388, May 2004.

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

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

[10] Joonsoo Bae, Hyerim Bae, Suk-Ho Kang, and Yeongho Kim, "Automatic Control of Workflow Processes Using ECA Rules", IEEE Transaction on Knowledge and Date Engineering, Volume 14, Number 8, pp.1010-1023, IEEE Computer Society, August 2004.

[11] Chengfei Liu, Xuemin Lin, Maria Orlowska and Xiaofang Zhou, "Confirmation: increasing resource availability for transactional workflows", Information Sciences, Volume 153, Issue 1, pp.37-53, Elsevier Science Inc, July 2003.

[12] Weimin Du, and Ming-Chien Shan, "Enterprise Workflow Resource Management", Proceedings of the Ninth International Workshop on Research Issues on Data Engineering: Information Technology for Virtual Enterprises, pp.108-115, IEEE Computer Society, March 1999.

[13] Michael zur Muehlen, "Resource Modeling in Workflow Applications", Workflow Management Conference, Münster, Germany, 1999.

[14] Duk-Ho Chang, Jin Hyun Son, and Myoung-Ho Kim, "Critical path identification in the context of a workflow", Information & Software Technology, Volume 44, Number 7, pp.405-417, Elsevier Science Publishers, May 2002.

[15] J.H.Son and M.H.Kim, "Extracting the Workflow Critical Path from the Extended Well-Formed Workflow Schema", Journal of Computer and System Sciences, Volume 70, Issue 1, pp.86-106, Elsevier Science Publishers, February 2005.

[16] Wasim Sadiq and Maria E. Orlowska, "Analyzing Process Models Using Graph Reduction Techniques", Information Systems, Volume 25, Number 2, pp.117-134,

Elsevier Science Publishers, 2000.

[17] [URL] Flowring Technology Corp., http://www.flowring.org

[18] The Workflow Management Coalition, "Terminology & Glossary", Document Number WFMC-TC-1011, February 1999.

[19] John Thangarajah, Lin Padgham and Michael Winikoff, "Detecting & Avoiding Interference Between Goals in Intelligent Agents", 18th International Joint Conference on Artificial Intelligence, August 2003.

[20] John Thangarajah, Lin Padgham and Michael Winikoff, "Detecting & Exploiting Positive Goal Interaction in Intelligent Agents", The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, pp.401-408, ACM, July 2003.

[21] Nick Russell, Arthur H.M. ter Hofstede, David Edmond, and Wil M.P. van der Aalst, "Workflow Data Patterns", QUT Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, 2004

[22] C. Karamanolis, D. Giannakopoulou, J. Magee and S. M. Wheater, "Model Checking of Workflow Schemas", Fourth International Enterprise Distributed Object Computing Conference (EDOC'00), pp.170-179, IEEE Computer Society, September 2000.

[23] C. Karamanolis, D. Giannakopoulou, J. Magee and S. M. Wheater, "Formal Verification of Workflow Schemas", Technical Report, Control and Coordination of Complex Distributed Services, ESPRIT Long Term Research Project, 2000.