# A Change Support Workflow Model for Cooperative Software Development

By Phan, Huyen Thi Thanh

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu

September, 2010

# A Change Support Workflow Model for Cooperative Software Development

By Phan, Huyen Thi Thanh (0810202)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Koichiro Ochimizu

and approved by
Professor Koichiro Ochimizu
Associate Professor Masato Suzuki
Associate Professor Toshiaki Aoki

August, 2010 (Submitted)

# Acknowledgment

First and foremost I offer my sincerest gratitude to my supervisor, Professor Koichiro Ochimizu, who has supported me throughout my study with his knowledge, guidance and encouragement. Without his consistent help this thesis would not have been completed or written.

In addition, I would like to thank the Japanese Government (Monbukagakusho) Scholarship Program for financial support during my stay in Japan.

Last but not least, I thank my family who endured this long process with me, always offering love, support and understanding. Thanks are also due to numerous friends, especially those at Ochimizu Laboratory for their willingness to participate in challenging discussion and give help to tackle the language barrier in my daily life.

**Abstract**

This thesis proposes an approach to synchronize changes on shared software artifacts in a change support environment for cooperative software development, by detecting and solving errors caused by uncontrolled access to shared data by Change Support Workflows (CSWs). CSW is a sequence of activities defined to carry out a change request. Activities in CSWs take care of creating new software artifacts or modifying exiting ones.

In this thesis, we abstract errors caused by uncontrolled access to shared data as Unintentional Change in In-use Data (UCID), a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers, due to non-deterministic access to shared data by different activities. We first solve UCID problem in a general workflow management system and then apply the solution to the change support environment.

In contrast to previous work, we identify UCID patterns caused by not only concurrent activities in the same workflow, intra-UCID, but also activities in different concurrent workflows, inter-UCID. We also propose a Time Data Workflow (TDW), an extension of the WF-Net by integrating data and time as attributes of WF-Net transitions. Algorithms which help detect intra-UCID and inter-UCID patterns at build time in a Concurrent TDW Management System are developed too. Then, algorithm evaluation and some solutions to resolve UCID problem are given. Finally, we present how to apply the UCID theory to the change support environment to detect and resolve errors concerning shared software artifacts among CSWs.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

"The only permanent thing is change", said the Greek philosopher Hippocrates, and software system is not an exception. It must be changed under various circumstances during development and after delivery, such as for new requirements, error correction, performance improvement, etc. However, software change is not an easy task, especially in a cooperative environment where software artifacts with very complex dependency relationships are created based on the cooperation of many people. Also, other problems such as concurrent changes and synchronization of changes on shared artifacts, etc. make this task more difficult. Therefore, a better change support environment is necessary. CSE (Change Support Environment) is our project to build a change support environment for cooperative software development. One unique feature of this project is to semi-automatically generate workflows which represent tasks needed to implement change requests. Change workers can perform change activities safely and efficiently in a cooperative environment based on the generated workflows. The project is divided in two phases: (1) building an information model and (2) building an operation model.

In the first phase of the project, the information model has been built. This model helps to generate automatically dependency relationships among UML model elements, among Java classes, and between a UML model element and a group of Java collaboration Java classes. A dependency relationship is defined as a relationship between two elements in which a change in the supplier element requires a change in the client element.

As part of the CSE project, we are responsible for building the operation model which we call Change Support Workflow Model (CSW Model). We define a Change Support Workflow (CSW) as a sequence of activities defined to carry out a change request. A CSW Model is responsible for CSW construction and management. Activities in CSW take care of creating new software artifacts or modifying exiting ones. This means that data elements of CSW are software artifacts which need to be read, modified or created in the change implementation process. Inputs to this model are the dependency relationships generated by the information model. Tracing the generated dependency relationships helps to identify these data elements and the orders between them as well. In other words, the flow of control and data flow of CSW can be identified. However, in a large and cooperative system, there are many CSWs executed at the same time and they may share some software artifacts. Therefore, the CSW Model must consider not only the sequence

of activities of CSWs but also the synchronization of changes on shared artifacts, and access control problem. In the scope of this thesis, we concentrate on the synchronization control problems. Because workflow designers work independently, we need to detect and resolve errors caused by uncontrolled access to shared data by different CSWs in order to synchronize changes on shared data. In this thesis, we are going to abstract these problems as **Unintentional Change in In-use Data (UCID)**, a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers, due to non-deterministic access to shared data by different activities. We first solve this problem in a general Workflow Management System and then apply the solution to the CSW Model.

Differently from previous studies, we consider UCID in two different ways: between concurrent activities in a single workflow (intra-UCID) and between activities in different concurrent workflows (inter-UCID). We first investigate UCID situations in a workflow management system, and then we define a Time Data Workflow, an extension of WF-Net with time and data factors, with many attributes supporting UCID detection and correction. Based on these definitions, we develop an algorithm which helps to detect potential intra/inter-UCID at build time, along with algorithm evaluation and UCID resolution methods. Our approach in UCID detection is to observe behaviors of activities with data relations. For activities in the same workflow, their total orders can be decided based on control flow. However, control flow is useless in the case of activities in different workflows. Therefore, we must use activities' execution time attribute to identify their total orders. Regarding UCID resolution, we take advantage of composition features of Petri Net to create new workflows with UCIDs resolved.

In the remainder of this thesis, Chapter 2 covers the background of our research by introducing an overview of the CSE project, our research scope and approach. This chapter includes an introduction of Petri Net, the foundation of our workflow modeling language. Chapter 3 presents related work. Chapter 4 defines the Time Data Workflow (TDW), an extension of the Workflow Net with time and data factors. Chapter 5 identifies UCID patterns caused by concurrent activities in the same workflow (intra-UCID) or activities in different concurrent workflows (inter-UCID). Algorithms for detecting potential intra/inter-UCID at build time, along with algorithm evaluation, are given in Chapter 6. Chapter 7 describes UCID resolution methods. In Chapter 8, we give an instruction to apply UCID theory to the Change Support Workflow Model. Chapter 9 begins by some evaluation of the proposed method, followed by some improvement suggestions, and ends with future work. The last chapter concludes the thesis.

# Chapter 2

# Background

This chapter introduces the Change Support Environment Project in detail to help readers have a more comprehensive view of the purpose of our research. Our research objectives and approach are also clarified in this chapter. Finally we introduce Petri Net, a popular mathematical modeling language, which we use as a basis for modeling CSWs.

## 2.1 Change Support Environment - Flow of Control

The software change process can be divided into 4 phases: understanding the change, planning the change, implementing the change and validating the change. Most of previous works concentrated on supporting the first phase, understanding the change, in particular change impact analysis. Therefore, we want to build a change support environment for cooperative software development, CSE Project, which supports not only the first phase but also the second phase and the third phase of the software change process.

Figure 2.1 shows the control flow of the CSE system. There are two main tasks of the project: building an information model and building a Change Support Workflow Model (CSW Model).

The information model is in charge of generating the dependency relationships between UML model elements, and between UML model elements and Java collaboration classes.

1. Defining the Dependency Generation Model consisting of basic rules for identifying the dependency types between UML model elements.

2. Developing a dependency generator to generate the dependency relationships between UML model elements. The generator receives UML diagrams and process information, and returns the UML diagrams with the dependency relationships added.

3. Defining Meta patterns of popular Design Patterns.

4. Developing an algorithm for extracting the collaboration classes from Java source code based on the Meta patterns.

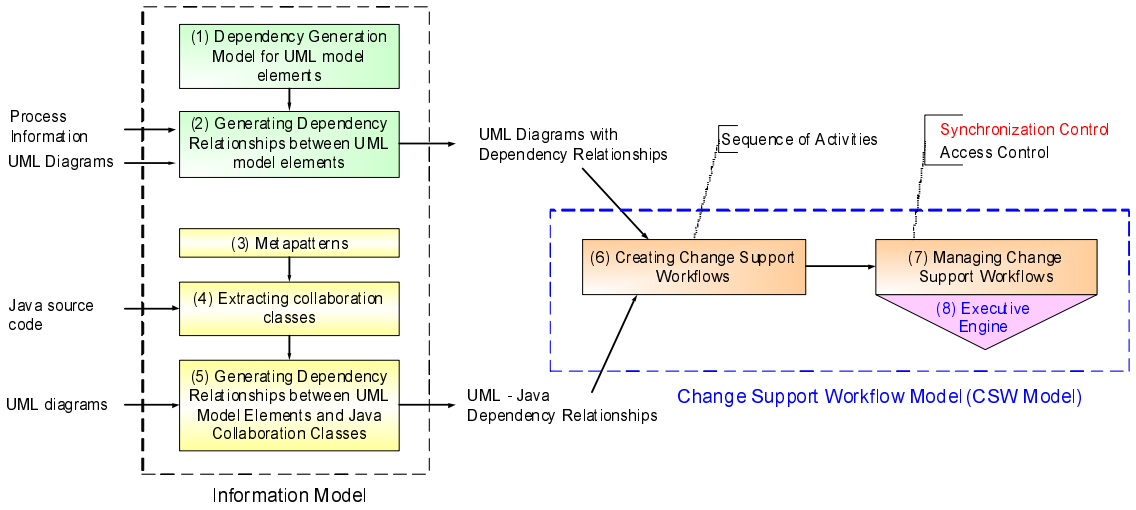



Figure 2.1: Flow of control of CSE

5. Developing an algorithm to generate the dependency relationships between UML model elements and Java collaboration classes.

The CSW Model is responsible for CSW construction and management. A Change Support Workflow (CSW) is a sequence of activities defined to carry out a change request.

6. The dependency relationships generated from the information model are input to the CSW Model. By tracing the generated dependency relationships, we can identify the data elements (software artifact impacted by a change request) and the control flow (change order of impacted software artifacts) of CSW.

7. In a large and cooperative project, there may be many workflows executing simultaneously and sharing data elements. Therefore, it is very important to have a mechanism to manage CSWs, especially synchronization of changes on shared artifacts and access control.

8. Finally, an execution engine will be developed.

### 2.1.1 Information Model

#### Generating Dependency Relationships between UML Model Elements

From the dependency concept in UML (a relationship between 2 elements in which a change to one element, the Target, may affect or supply information needed by another element, the Source), the authors [1] defined 4 types of basic dependency relationship (BDR) among UML model elements: Exist Together (Source will not exist without Target), Information Sharing (Information on the target is a part of information in the source), Copy (Information on the target and source is the same), Detail (Information in the source is made based on information on the target but at more detail levels).

To generate the dependency relationships automatically, they gave a Dependency Generation Model consisting of comparison rules, addition rules and selection rules. This model accepts as input a group of UML diagrams and information of phases to which they belong. Outputs will be these UML diagrams with newly added BDRs. Comparison rules look for pairs of UML model elements which may have some BDRs based on the similarity in names between UML model elements and inclusion relationships between a diagram and its components. Addition rules search types of BDR which may be set to a pair of UML model elements. First, UML model elements are classified into Generation Model Elements. Then, based on the diagram predefining BDRs which can be set between Generation Model Elements, we can know BDR candidates between them. Regarding information about phase, diagram, type and name of UML model elements, selection rules decide the attached type of BDR to the selected pair from candidates of BDR found by addition rules.

#### Generating UML-Java Dependency Relationships

In order to generate UML-Java dependency relationships, the authors [2] have performed two steps: extracting collaboration classes from Java source code and mapping UML class diagrams to collaboration classes. Proposed method is based on an assumption that design patterns are used to write Java source code and UML class diagram.

In the first step, collaboration classes are extracted based on Meta patterns given by William Pree: the unification Meta pattern, 1:1 Connection Meta pattern, 1:N Connection Meta pattern, 1:1 Recursive Connection, 1:1 Recursive Unification Meta patterns, 1:N Recursive Connection, and 1:N Recursive Unification Meta patterns. If Meta patterns are not enough, structure features (connection, instance, inheritance, multiplication), and behavior features (method definition, variable definition, data flow) of Java classes are used to extract collaboration classes.

In the second step, they built a diagram of implementation groups in which each element is a group of collaboration classes. Next, diagram of implementation groups and UML class diagram are transformed into graphs. Finally, sub-graph isomorphism algorithm of Jeffrey David Ullman is used to find connections between classes in UML class diagram and groups of collaboration classes. These connections are the UML-Java dependency relationships.

### 2.1.2 Change Support Workflow Model - Our research

Change Support Workflow Model (CSW Model) is responsible for CSW construction and management. Inputs to this model are the dependency relationships generated by the information model.

Starting from the changed elements, we can identify potentially impacted elements by tracing the dependency relationship generated among software artifacts. Intuitively, these relationships represent the order of works when we change model elements. However, impacted elements and the change order certainly are not the same for different change requirements and types of change, for instance, deletion, extension, and modification.

Therefore, these dependencies must be carefully analyzed according to each change type and requirement.

In addition, due to the importance of workflow data (software artifacts which need to be read, modified or created in the change implementation process), we must consider the correctness of data flow. In a large and cooperative environment, there may be many CSW running at the same time, and if some of these workflows modify software artifacts being used by others, this will affect the correctness of the whole work. Therefore, it is necessary to have a suitable method for verifying workflow data based on existing solutions which meets unique features of this problem.

Another fundamental issue in workflow's security is access control. Access control is a mechanism by which users are permitted access to various operations or data within a computer system, according to their identity (established by authentication) and associated privileges (established by authorization). In the context of workflow systems, it may operate at the level of: (a) log-on to the workflow service, and (b) access to undertake particular activities or work items according to functional role and/or data sensitivity [23]. To ensure that authorized subjects gain access on the required objects only during the execution of the specific task in a workflow application, granting and revoking of privileges need to be synchronized with the progression of the workflow from one task to another. This is not feasible unless there is a workflow access control mechanism that authorizes an individual in synchronization with the progression of workflow. Role Based Access Control (RBAC) is one of the most widely used access control models because of its flexibility and policy neutral. It separates users from permissions logically through the concept of role: users are assigned to roles, permissions are assigned to roles and users acquire permissions by being members of roles. However, RBAC does not consider workflow and does not support dynamic changes of access control in the software development process either. Therefore, we want to develop an access control model that overcomes the shortcomings of RBAC in workflow domain and satisfies synchronous authorization and dynamic access control requirement of this workflow model.

In summary, there are three major issues which the CSW Model will solve: CSW construction, synchronization of changes to shared data by different CSWs, and access control.

## 2.2 The Thesis's Objectives and the Change Support Environment

In the scope of this thesis, we concentrate on synchronization control problem by detecting and resolving errors caused by uncontrolled access to shared data by CSWs in a CSW Model.

### 2.2.1 Motivating Example

Let us take an example. We have two workflows $W_1$ and $W_2$, which are designed by different designers who work independently. Workflow activities are modeled by rectangles, and data modified by an activity are written inside the corresponding rectangle. A small arrow is attached to a rectangle to denote an activity which is currently being executed. Data of the system are stored in a central repository. For simplicity, we just show shared data elements and data elements are modified based on these shared data elements. $W_1$ has five activities $A_{11}$, $A_{12}$, $A_{13}$, $A_{14}$, and $A_{15}$. Data elements $B$ and $D$ of workflow $W_1$ are modified based on the value of data element $A$ created by $A_{12}$. $W_2$ also includes five activities which are $A_{21}$, $A_{22}$, $A_{23}$, $A_{24}$, and $A_{25}$. Both $A_{12}$ and $A_{22}$ will modify data element $A$, but designers of $W_1$ and $W_2$, who don't have a comprehensive view of the whole system, may not recognize this problem. This is a common problem, especially in a big system with many workflows.

Figure 2.2 shows some snapshots of the system at different times. In Snapshot 1, $A_{12}$ changes value of $A$ to $a1$. In Snapshot 2, $A_{13}$ modifies value of $B$ based on the value of A, $a1$. In the next snapshot, $A_{22}$ changes value of $A$ from $a1$ to $a2$. In the last snapshot, $A_{15}$ modifies value of $D$ based on the current value of $A$ which is $a2$. If $a1$ is different from $a2$, there are two problems in this scenario: a1 is lost and $D$ is assigned an unexpected value because $D$ is modified based on the value $a2$ instead of the value created by activity $A_{12}$, $a1$. This is different from the intentions of the designers of workflow $W_1$ and may cause an inconsistency between $B$ and $D$.

The first problem is similar to the lost update problem in database theory. Lost update problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect [21]. In this case, version control systems (VCSs) can be used if data of the system are individual artifacts like documents, source codes, etc.

Unfortunately, VCS cannot help to avoid the second problem. In this situation, if data of the system are stored in a central database, the database management system (DBMS) can provide some concurrency control techniques, which are used to ensure the noninterference or isolation property of concurrently executing transactions, such as locking techniques, timestamp ordering based techniques, etc. However, database transactions often require some critical data to be locked during the transactions which take only a few seconds. Therefore, treating them as a database transaction will be complex or completely impractical because of the long-running nature of workflows. In other words, an effective method to deal with these problems is necessary.

We have abstracted these problems as **Unintentional Change in In-use Data (UCID)**. We define UCID as a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers, due to non-deterministic access to shared data by different activities. In the scope of this thesis, we concentrate on solving UCID problem in a general Workflow Management System and then apply the solution to the CSW Model.

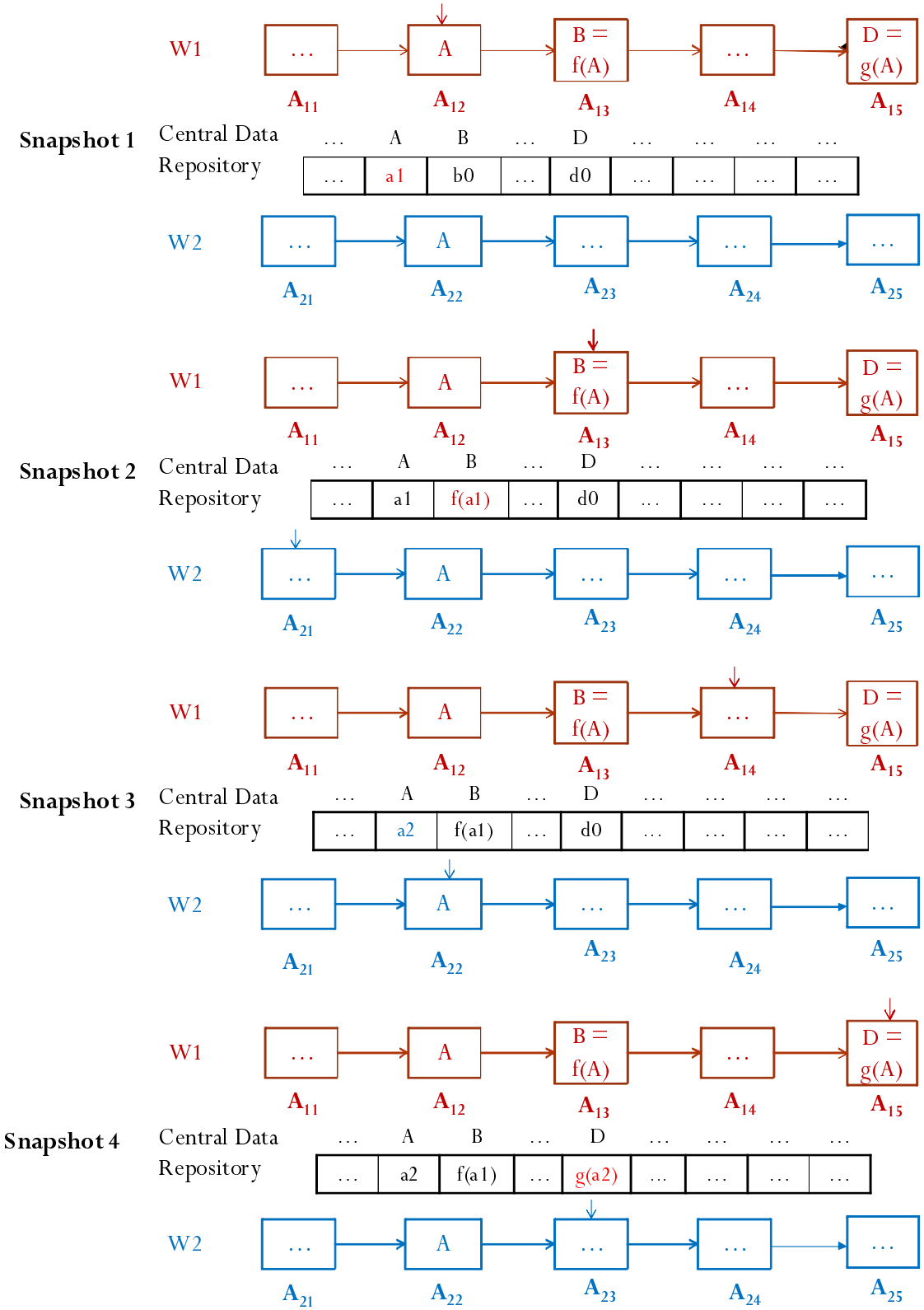



Figure 2.2: Motivating example

### 2.2.2 UCID Approach

In a workflow model, data flow can be implemented explicitly as a part of the workflow model by using a separate channel to pass data from one activity to another. Otherwise, it can also be implemented implicitly through a control flow or process data store [3]. The process data store is basically a central repository where all activities of workflows can access or update their data. We choose implicit data flow through the process data store as a basis for our approach. Also, we do not pay attention to the type of workflow data stored in the central repository of the system and the implementation of the central repository as well.

If UCID is discovered at runtime, a recovery mechanism must be performed to ensure the correctness of the whole system. However, recovery is a rather expensive work, especially in a cooperative environment with many concurrently executing workflows. Therefore, our approach is to detect potential UCID patterns at build time to reduce risk to the target process. In our approach, we assume that control flow (order of activities in workflow) and data flow can be given before workflow execution. CSE project satisfies these assumptions because the dependency relationships generated by the information model of this project help us to identify the structure (control flow) and data elements of CSW at build time.

First, we identify UCID patterns. Because different workflows are designed for different purposes by different designers and a designer may know nothing about the work of the others, detecting and resolving UCID relating to many workflows are more complicated and they should be considered separately. Therefore, we distinguish two types of UCID: intra-UCID and inter-UCID. The former is caused by concurrent activities in the same workflow while the latter is caused by concurrent activities in different workflows. In order to detect potential UCID patterns, we observe total orders of concurrent activities with data relations and map them to UCID patterns. For activities in the same workflow, their total orders can be decided based on control flow. However, control flow does not help in the case of activities in different workflows. Assuming that estimated execution time (earliest start time and latest finish time) of activities can be identified at build time, we can use this time attribute to identify total orders of activities in different workflows. In this case, early UCID detection will help workflow designers to have a more comprehensive view of the system, and make timely adjustments to the original workflows to avoid errors at runtime. With reference to workflows in which estimated execution time is not available at design time, UCID patterns and detection method can be used to detect UCID errors from workflow execution histories. However, this is outside the scope of this thesis. Regarding UCID resolution, we take advantage of fusion features of Petri Net to create new workflows with UCIDs resolved.

In order to use the UCID theory in the CSW Model, we need to model CSWs as TDWs and then use potential UCID detection algorithm to check the existences of potential UCID patterns. If some errors are reported, workflow designer should review original CSWs based on the suggested UCID resolutions of the system.

## 2.3 Modeling Language

There are many way to model a workflow, such as directed graphs, UML activity diagram, BPEL (Business Process Execution Language), BPMN (Business Process Modeling Notation), etc. In this thesis, we chose the WF-Net based approach to model the workflow process, because it has many useful features needed in the area of business process modeling, in addition to the mathematical nature of the underlying Petri Net (PN) formalism [19]. WF-Net is a subclass of Petri Net dedicated to process/workflow modeling and analysis.

Petri Net was devised in 1962 by Carl Adam Petri as a tool for modeling and analyzing processes. One of the strengths of this tool is that it enables processes to be described graphically. Although Petri Net is graphical, it has a strong mathematics basis and is entirely formalized. Therefore, it is often possible to make strong statements about the properties of modeled processes. There are also several analysis techniques and tools available to analyze a given Petri net. Over the years, the original model has been extended in many different ways to be able to model more complex processes. In this section, we will introduce some basic concepts of classical Petri Net and some popular extensions of classical Petri Net. The content of this section is based on reference [10].

### 2.3.1 Classical Petri Net

The classical Petri Net is a directed bipartite graph with two node types called places and transitions. The nodes are connected via directed arcs. Connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by rectangles.

**Petri Net Definition** A Petri Net is a triple $(P, T, F)$:

- $P$ is a finite set of places;
- $T$ is a finite set of transitions ($P \cap T = \emptyset$);
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation)

A place $p$ is called an input place of a transition $t$ if and only if there exists a directed arc from $p$ to $t$. Place p is called an output place of transition t if and only if there exists a directed arc from $t$ to $p$. We use $\cdot t$ to denote the set of input places for a transition $t$. The notations $t\cdot$, $\cdot p$ and $p\cdot$ have similar meanings, e.g., $p\cdot$ is the set of transitions sharing $p$ as an input place.

At any time a place contains zero or more tokens, drawn as black dots. State $M$, often referred to as marking, is the distribution of tokens over places, i.e., $M \in P \rightarrow IN$. To compare states we define a partial ordering. For any two states $M_1$ and $M_2$, $M_1 \leq M_2$ iff for all $p \in P : M_1(p) \leq M_2(p)$, where $M(p)$ denotes the number of tokens in place $p$ in state $M$. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following firing rule:

- A transition $t$ is said to be enabled iff each input place $p$ of $t$ contains at least one token.
- An enabled transition may fire. If transition $t$ fires, then $t$ consumes one token from each input place $p$ of $t$ and produces one token for each output place $p$ of $t$.

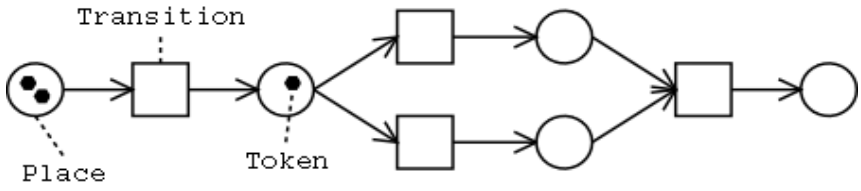



Figure 2.3: A classic Petri Net

### 2.3.2 High-level Petri Nets

#### The Color Extension

Token are used to model a whole range of things. However, in the classic Petri Net, it is impossible to distinguish between two tokens in the same place. In order to enable the coupling of an object's characteristics with the corresponding token, the classic PN is extended using "color". This extension ensures that each token is provided with a value or color. Because each token has a value, we can distinguish different tokens from one another.

A firing transition produces tokens which are based upon the values of those consumed during firing. The value of a produced token may therefore depend upon those of consumed ones. Unlike in the classic Petri Net, the number of tokens produced is also variable: the number of tokens produced is determined by the values of those consumed.

In a color-extended Petri Net, we can set conditions for the values of the tokens to be consumed. If this is the case, then a transition is only enabled once there is a token at each of the input points and the preconditions have been met. A transition's precondition is a logical requirement connected with the values of the tokens to be consumed.

The result of color extension is that, in contrast to the classic Petri Net, the graphic representation no longer contains all the information. For each transition, the following factors must be specified:

- Whether there is a precondition. If there is, then this must be defined precisely.
- The number of tokens produced per output point during each firing.
- The values of the tokens produced.

#### The Time Extension

Given a process modeled as a Petri Net, we often want to be able to make statements on its expected performance. In order to be able to answer these questions, it is necessary to include pertinent information about the timing of a process in the model. The classic Petri net does not, however, allow the modeling of 'time'. Even with color extension, it is still difficult to model the timing of a process. Therefore, this classic Petri Net is also extended with time.

Using the time extension, tokens receive a timestamp as well as a value. This indicates the time from which the token is available. A transition is only enabled at the moment when each of the tokens to be consumed has a timestamp equal or prior to the current time. In other words, the enabling time of a transition is the earliest moment at which its input places contain enough available tokens. Tokens are consumed on a FIFO (first-in, first-out) basis. The token with the earliest timestamp is thus the first to be consumed. Furthermore, it is the transition with the earliest enabling time which fires first. If there is more than one transition with the same enabling time, which fires first is not indicated. Moreover, the firing of one transition may affect the enabling time of another.

If a transition fires and tokens are produced, then each of these is given a timestamp equal to or later than the time of firing. The tokens produced are thus given a delay, which is determined by the firing transition. The timestamp of a produced token is equal to the time of firing plus this delay. The length of the delay may depend upon the value of the tokens consumed. However, it is also possible that the delay has a fixed value (for example, 0) or that the delay is decided at random. Firing itself is instant, and takes no time.

#### The Hierarchical Extension

Although we can already describe very complex processes using the color and time extensions, the resulting PN will still not usually provide a proper reflection of the process being modeled. Because the modeling of such a process results in a single, extensive network, any structure is lost. We do not observe the hierarchical structure in the process being modeled by the PN. The hierarchical extension therefore ensures that it does become possible to add structure to the PN model.

In order to structure a PN hierarchically, a new 'building block' is introduced into PN: a double-bordered square. This element is called a process. It represents a subnetwork comprising places, transitions, arcs and subprocesses. Because a process can be constructed from subprocesses, which in turn can also be constructed from (further) subprocesses, it is possible to structure a complex process hierarchically.

### 2.3.3 Workflow Net (WF-Net)

A Petri Net which models the control-flow dimension of a workflow is called a Workflow Net (WF-Net). It should be noted that a WF-Net specifies the dynamic behavior of a single case in isolation.

**WF-net Definition** A Petri Net $PN = (P, T, F)$ is a WF-Net (Workflow Net) if and only if:

- There is one source place $i \in P$ such that $\cdot i = \emptyset$;
- There is one sink place $o \in P$ such that $o\cdot = \emptyset$;
- Every node $x \in P \cup T$ is on a path from $i$ to $o$.

A WF-Net has one input place $i$ and one output place $o$ because any case handled by the procedure represented by the WF-Net is created when it enters the Workflow Management System (WFMS) and is deleted once it is completely handled by the WFMS, i.e., the WF-Net specifies the lifecycle of a case. The third requirement in this definition has been added to avoid 'dangling tasks and/or conditions', i.e., tasks and conditions which do not contribute to the processing of cases.

# Chapter 3

# Related Work

In this chapter, we present related work in workflow data verification and some existing approaches to deal with problems of lost data or data conflict mentioned in the motivating example: CVS and concurrency control techniques of Database Management System.

## 3.1 Previous Work on Data Flow Verification

Workflow is a collection of activities to carry out a well-defined business process [3] and a workflow model is a formal representation of the process. Workflow can be viewed from three aspects: 1) the control aspect, describing logical order of activities; 2) the data aspect, describing information exchanged between activities; 3) the resource aspect, describing entities that are capable of doing work [7].

Correctness of a workflow model is very important, because errors in workflow can lead to execution failure of the corresponding process. Therefore, workflow should be verified carefully before execution to reduce risks to the target process. Workflow verification has received a lot of attention since the birth of the workflow concept. However, researchers have only focused on structure verification, temporal verification and resource verification [4] [6] [9] [11]. Most verification techniques ignore the data aspect and there is little support for data flow verification. Previous work on the data flow aspect has concentrated on detecting common data flow errors such as missing data, redundant data, inconsistent data, garbage data, etc.

Reference [5] was one of the first studies to mention the importance of data-flow verification, and identified possible errors in the data-flow, like missing data, redundant data, conflict data, etc. Some discussions on data flow modeling, specifications and verifications have been given, but they have only been in abstraction. Authors in [14] used data flow matrix and UML activity diagram to specify data flow. Based on this specification, an algorithm for detection of some data anomalies in [5], such as missing data, redundant data, and potential data conflicts, was given. In [11], a new workflow model, named Dual Workflow Nets, was defined to explicitly describe both control flow and data flow. A graph traversal approach was used in [12] to build an algorithm for detecting lost data, missing data and redundant data. More data flow errors were recognized and conceptu-

alized as data flow anti-patterns and expressed in terms of temporal logic CTL* [7], [8]. By using temporal logic, available model checking techniques can be applied to discover these anti-patterns.

Nevertheless, all of these studies consider data flow errors in a single workflow only. In contrast to previous work, we address not only the interactions of concurrent activities inside a single workflow, but also the mutual influences between concurrent workflows, which are the sources of data flow errors. Regarding these two aspects, we focus on UCID errors and identify UCID situations. A new workflow model, extending the Petri-net with time and data factors, and using potential UCID detection algorithms, allows us to detect potential UCID at build time efficiently. Furthermore, some heuristics for making the algorithm more flexible and effective are discussed. UCID resolution methods are also proposed in this paper.

Concerning the mutual influences of the concurrent workflows approach, the research that is closest to ours is [9]. However [9] addressed the verification of workflow resource constraints, and in that work, by nature, handling the resource problem is simpler than the data problem. A Time Constraint Workflow Net was defined to model workflow. Then, they identified the problem of resource constraints in WFMS and proposed a pseudocode algorithm which checked the resource dependency between every two activities. Reference [6] used hybrid automata to model the influences between concurrent workflows, and adopted a model checking technique to detect resource conflict problems.

## 3.2 Version Control System

Version Control System can be used to manage changes to software artifacts in the CSW Model because of the ability to maintain different versions of a software artifacts. In this section, we introduce some basic concepts in version control based on reference [24].

Version control or revision control is the management of changes to documents, programs, and other information stored as computer files. Changes are usually identified by a number or letter code, termed the "revision number". Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged. Some popular version control systems (VCSs) are Concurrent Version System (CVS), Subversion (SVN), Revision Control System (RCS), etc.

Traditional VCSs use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

File locking is the simplest method in which only one developer at a time has *write* access to the central "repository" copies of those files. Once one developer "checks out" a file, others can *read* that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout). This technique has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a

user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in. Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. Therefore, the second developer checking in code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text based documents, unless a specific merge plugin is available for the file types.

## 3.3 Concurrency Control Techniques of Database Management Systems

As discussed in Section 2.2.1, Motivating Example, Database Management Systems (DBMSs) have already provided some concurrency control techniques to ensure the noninterference property of concurrently executing transactions. Most of the concurrency control techniques that are used most often in practice employ the technique of locking data items to prevent multiple transactions from accessing the items concurrently. Unfortunately, locking technique is perhaps unsuitable in the workflow management system because of the long-running nature of workflows which are different from the short-running nature of database transactions. However, because DBMSs are mature systems, it is useful to consider their concurrency control techniques and apply them to workflow area if possible.

A database transaction is a transaction which satisfies the ACID (atomicity, consistency, isolation and durability) properties. These properties should be enforced by the concurrency control and recovery methods of the DBMS.

Transactions submitted by various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database. This section introduces a number of popular concurrency control techniques that are used to enforce isolation (through mutual exclusion) among conflicting transactions, to preserve database consistency through consistency preserving execution of transactions and to resolve read-write and write-write conflicts [21].

### 3.3.1 Two-Phase Locking Techniques for Concurrency Control

A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for

each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Two major types of locks are utilized:

- Write-lock (exclusive lock) is associated with a database item by a transaction before writing this item.
- Read-lock (shared lock) is associated with a database item by a transaction before reading this item.

A transaction is said to follow the two-phase locking protocol if all locking operations (read-lock, write-lock) precede the first unlock operation in the transaction. According to the two-phase locking protocol, a transaction handles its locks in two distinct, consecutive phases during the transaction's execution:

- **Expanding phase:** new locks on items can be acquired but none are released.
- **Shrinking phase:** existing locks can be released but no new locks can be acquired

There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses before the transaction begins execution, by pre-declaring its read-set (the set of all items that the transaction reads), and the write-set (the set of all items that the transaction writes). If any of the pre-declared item needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. This protocol is deadlock-free protocol. However, it is difficult to use in practice because of the need to pre-declare the read-set, write-set, which is not possible in most situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules. In this variation, a transaction T does not release any of its exclusive (write) locks until after it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (shared or exclusive) until after it commits or aborts, and so it is easier to implement than strict 2PL.

### 3.3.2 Concurrency Control based on Timestamp Ordering

Timestamp is a monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation. Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

**Basic Timestamp Ordering**

1. Transaction T issues a write_item(X) operation:
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then a younger transaction has already read the data item so abort and roll-back T and reject the operation.
    - If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
2. Transaction T issues a read_item(X) operation:
    - If write_TS(X) > TS(T), then a younger transaction has already written to the data item so abort and roll-back T and reject the operation.
    - If write_TS(X) $\leq$ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

**Strict Timestamp Ordering**

Strict TO is a variation of basic TO which ensures that the schedules are both strict (for easy recoverability) and (conflict) serializable.

1. Transaction T issues a write_item(X) operation:
    - If TS(T) > read_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
2. Transaction T issues a read_item(X) operation:
    - If TS(T) > write_TS(X), then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

**Thomas's Write Rule**

A modification of the basic TO algorithm does not enforce conflict serializability; but it rejects fewer write operations.

1. If read_TS(X) > TS(T),then abort and roll-back T and reject the operation.
2. If write_TS(X) > TS(T),then just ignore the write operation and continue execution.
3. If the conditions given in 1 and 2 above do not occur, then execute write_item(X) of T and set write_TS(X) to TS(T).

### 3.3.3 Multiversion Concurrency Control Techniques

Multiversion concurrency control techniques maintain a number of versions of a data item and allocate the right version to a read operation of a transaction. Thus unlike other mechanisms, a read operation in this mechanism is never rejected.

An obvious drawback of this technique is that more storage is needed to maintain multiple versions of the database items. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.

Two popular multiversion concurrency control schemes are multiversion techniques based on timestamp ordering and multiversion techniques based on 2PL.

# Chapter 4

# Time Data Workflow

There are many ways to model a workflow, such as directed graphs, UML activity diagram, PERT, etc. In this thesis, we chose the WF-Net based approach to model the workflow process. WF-Net is a subclass of Petri Net dedicated to process/workflow modeling and analysis. Therefore, it has many useful features needed in the area of business process modeling, in addition to the mathematical nature of the underlying Petri Net formalism [19] .

Our Time Data Workflow (TDW) is an extension of WF-Net with time and data factors. Time and data are represented as attributes of transitions in a TDW. In this thesis, we consider two types of relationships between an activity and a data element. First, an activity may *read* a particular data element as its input data. Second, an activity may *write* a particular data element as its output data. This means that this data element can be assigned a new value. Inside an activity, read operation on a data element always happens before the write operation in the same data element. Assuming that durations of activities can be estimated at build time, we augment each activity $A$ with two time values $min(A), max(A)$ which describe the minimum and maximum execution durations of $A$, respectively. The time unit is selected depending on specific workflow applications. Based on reference point $P$, which is the start time of its corresponding workflow, we can infer the *Earliest Start Time*, $EST(A)$, and the *Latest Finish Time*, $LFT(A)$ of $A$. If $S(A)$ and $F(A)$ are the *Start Time* and *Finish Time* of this activity at run time respectively, we can conclude that the *Active Interval* of $A$, $[S(A), F(A)]$, is within its *Estimated Active Interval*, $[EST(A), LFT(A)]$, that is, $[S(A), F(A)] \subseteq [EST(A), LFT(A)]$.

In a TDW, activities are modeled by transitions, and causal dependencies are modeled by places and arcs, as shown in Figure 4.1. Building blocks such as the AND-Split, AND-Join, OR-Split and OR-Join are used to model sequential, conditional, parallel and iterative control structures of workflows. AND-Split and OR-Split transitions correspond to transitions with two or more output places, while AND-Join and OR-Join transitions correspond to transitions with multiple incoming arcs. Different symbols are attached to original rectangles to distinguish normal transitions from transitions containing branching conditions. Figure 4.1a illustrates a typical transition in a TDW, with execution duration ranging from $d_1$ to $d_2$; data elements $a$, $b$ are inputs and $c, d, e$ are outputs. The other parts of Figure 4.1 show how basic constructions of a workflow are represented by

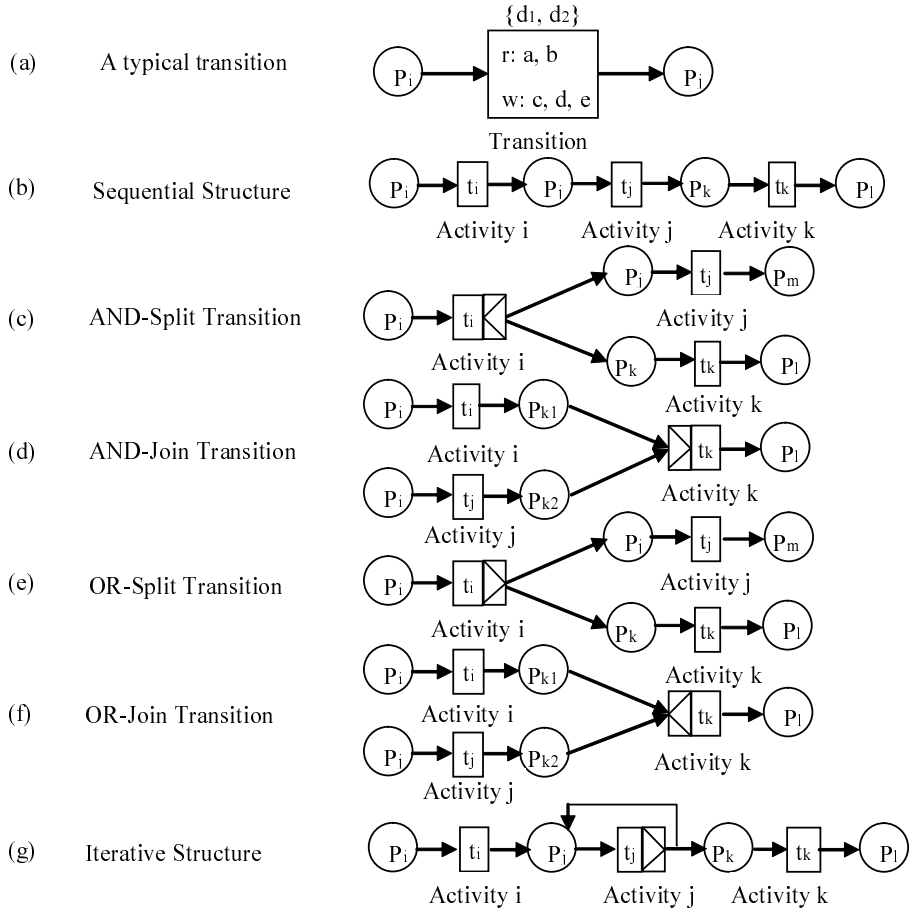



Figure 4.1: Workflow primitives specified by TDW

TDW notations. For the sake of simplicity, each activity is represented by a transition. Therefore, the terms 'activity' and 'transition' are interchangeably used in this thesis.

As an extension of WF-Net, TDW specifies the time and data properties of a single case in isolation, assuming that different cases are completely independent from each other. Therefore, UCIDs are caused by activities in a single TDW instance or activities belonging to workflow instances of different TDWs. Without the loss of generality, we assume that each TDW has one instance only.

**Definition 1 (Time Data Workflow - TDW)** A TDW, $w$, is a tuple $< P, T, F, id, D, R, DE, TI >$ where:

- $< P, T, F >$ is a WF-Net with places $P$, transitions $T$ and arcs $F$.
- $id$ is the workflow identifier.
- $D$ is a set of data elements.
- $R = \{r, w, u\}$ is the set of possible access rights to data elements (r: ***r**ead*, w: ***w**rite*, u: ***u**se* (either read or write)).

- $DE : T \times R \rightarrow 2^D$ is a function that returns a set of data elements associated with a transition and an access right.
- $TI : T \rightarrow (R^+ \times R^+)$ is a time interval function that returns minimum and maximum execution durations of a transition.

**Definition 2 (Concurrent Time Data Workflow Model)** A Concurrent TDW Model $cwm = (W, T_{cwm})$ is a collection of TDWs which have overlapping execution times (concurrent TDWs):

- $W = \{w_1, w_2, \cdots, w_n\}$ is a set of concurrent TDWs, where $w_i =< P_i, T_i, F_i, id_i, D_i, R_i, DE_i, TI_i >$.
- $T_{cwm} = T_1 \cup T_2 \cup \cdots \cup T_n$ is the set of all transitions (activities) in $cwm$.

Given a TDW $w =< P, T, F, id, D, R, DE, TI >$ as in Definition 1, we have the following definitions:

**Definition 3 (Path)** A Path in a TDW is a sequence of consecutive arcs $p = (x_o, x_1, \cdots, x_k)$ such that $(x_i, x_{i+1}) \in F$, where $0 \leq i < k - 1, x_i \in (P \cup T)$

**Definition 4 (Transition Path)** A sequence $p = (t_0, p_1, t_1, \cdots, p_k, t_k)$ is a Transition Path iff $p$ is a path and $t_0, t_k \in T$.

**Definition 5 (Transition Reachability)** Transition $t_i$ is reachable from $t_j$ if there is a transition path $(t_i, \cdots, t_j)$ in TDW. It can be denoted as a Boolean function:

$$Reachable(t_i, t_j) = \begin{cases} true, & \exists \quad \text{transition path p} = (t_i, \cdots, t_j) \\ false, & \text{otherwise} \end{cases}$$

**Definition 6 (Transition Distance)** The distance between two transitions $t_i, t_j$ in a TDW can be computed as follows:

$$Distance(t_i, t_j) = \begin{cases} Min\{|p_s||p_s = (t_i, \cdots, t_j), s = 1, \cdots, m\}, & Reachable(t_i, t_j) = true \\ Min\{|p_s||p_s = (t_j, \cdots, t_i), s = 1, \cdots, m\}, & Reachable(t_j, t_i) = true \\ +\infty, & otherwise \end{cases}$$

where $p_s$ represents all paths between $t_i$ and $t_j$.

**Definition 7 (Nearest Common Transition)** Given two transitions $t_i, t_j$ where $Reachable(t_i, t_j) = false$ and where $Reachable(t_j, t_i) = false$, their Nearest Common Transition is the transition $t$ from which $Reachable(t, t_i) = true$ and $Reachable(t, t_j) = true$, and the distances are shortest, denoted as $t_{nct}$.

**Definition 8 (Closest Data Relation Transition)** Given two transitions $t_i, t_j$, where their Nearest Common Transition is not an OR-Split transition, $t_j$ is called the Closest Data Relation Transition of $t_i$ on data element $d$ if both $t_j$ and $t_i$ use (read/write) $d$, and $t_j$ just precedes $t_i$ in terms of time, denoted as $t_{p-cdrt}$, or $t_j$ just succeeds $t_i$ in terms of time, denoted as $t_{s-cdrt}$.

# Chapter 5

# UCID Patterns in a Concurrent TDW Management System

A Concurrent TDW Management System is a workflow management system which is responsible for TDW construction and management. A module for UCID detection and correction is also integrated into this system.

Given a Concurrent TDW Model *cwm* as in Definition 2, we have the following definitions:

**Definition 9 (Data Relation)** Two activities $a_i, a_j$ $(i \neq j)$ have a data relation if $DE(a_i, u) \cap DE(a_j, u) \neq \emptyset$.

**Definition 10 (Concurrent activities)** Two activities are called concurrent activities if and only if they belong to two parallel branches of a single TDW, or if they are in different TDWs and have overlapping Active Intervals.

**Definition 11 (Unintentional Change in In-use Data)** A situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers, due to non-deterministic access to shared data by different activities.

Here we distinguish two kinds of UCID: intra-UCID and inter-UCID. The former considers UCID situations concerning concurrent activities in the same workflow, while the latter is related to concurrent activities in different workflows. Definitions 12, 13 are based on definitions of read-write conflict and write-write conflict in [3] .

**Definition 12 (RW Intra-UCID)** A situation in which an activity $A$ tries to read data from a shared data element $x$ and an activity $B$ tries to write data to the same shared data element $x$ and vice versa, where $A$, $B$ are concurrent activities in the same workflow.

**Definition 13 (WW Intra-UCID)** A situation in which two concurrent activities in the same workflow, $A$ and $B$, try to write data to the same shared data element.

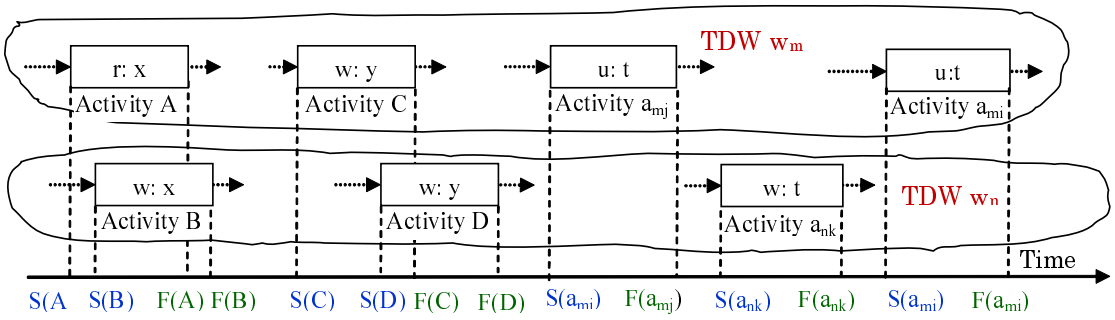



Figure 5.1: Inter-UCID patterns

**Definition 14 (RW Inter-UCID)** A situation in which an activity $A$ tries to read data from a shared data element $x$ and an activity $B$ tries to write data to the same shared data element $x$ and vice versa, where $A$ and $B$ are in different concurrent workflows and have overlapping Active Interval.

**Definition 15 (WW Inter-UCID)** A situation in which two activities $A$ and $B$ try to write data to the same shared data element, where $A$ and $B$ are in different concurrent workflows and have overlapping Active Interval.

**Definition 16 (UWU Inter-UCID)** A situation in which there are inconsistent views of shared data elements by two activities in the same workflow, because their shared data elements are written externally by an activity in a different concurrent workflow.

As depicted in Figure 5.1, two activities $a_{mi}$, $a_{mj}$ of TDW $wm$ ***u**se* (read or write) data element $t$, where $a_{mj}$ is the closest to $a_{mi}$ in terms of time and $F(a_{mj}) < S(a_{mi})$, which means $t_{p-cdrt}(a_{mi}, t) = a_{mj}$. A UWU Inter-UCID happens because activity $a_{nk}$ of a different workflow $w_n$ writes to $t$ within the time interval $[F(a_{mj}), S(a_{mi})]$. We also have RW Inter-UCID between activities $A$ and $B$, and WW Inter-UCID between activities $C$ and $D$.

# Chapter 6

# Detection of Potential UCID in a Concurrent TDW Management System

## 6.1 Potential Intra-UCID Detection Algorithm

This algorithm checks potential intra-UCIDs of a TDW $w =< P, T, F, id, D, R, DE, TI >$. The main idea is to select one activity in TDW and compare it with other activities. If two concurrent activities have data relation, we will check if this situation is a potential intra-UCID with respect to Definitions 12 and 13.

**Step 1** Initialization:
1.1 Let $S$ be a set of unchecked activities.
$S$ is initialized with all activities in $w$: $S = T$;
1.2 $flag = TRUE$ is a Boolean variable;

**Step 2** Repeat the following steps until $S = \emptyset$:
2.1 Remove an element, denoted as $a_i$, from $S$;
2.2 For each element $a_j$ in $S$, execute the following steps:
2.2.1 $DE(a_i, u) \cap DE(a_j, u) = U_{ij}$; /* *Check Data Relation* */
2.2.2 If $(U_{ij} = \emptyset)$ or $((DE(a_i, w) \cap U_{ij} = \emptyset) \wedge (DE(a_j, w) \cap U_{ij} = \emptyset))$, then jump to 2.2; /* *No Data Relation or both* $a_i$ *and* $a_j$ *just have read access on shared data* */
2.2.3 If $(Reachable(a_i, a_j) = true)$ or $(Reachable(a_j, a_i) = true)$, then jump to 2.2; /* *sequential activities* */
2.2.4 If $t_{nct}(a_i, a_j)$ is an OR-split block, jump to 2.2;
/* $a_i$, $a_j$ *are concurrent activities* $\rightarrow$ *check intra-UCID* */
2.2.5 Repeat the following steps until $U_{ij} = \emptyset$;
2.2.5.1 Remove a data element, denoted as $d_{ijk}$, from $U_{ij}$;
2.2.5.2 If $d_{ijk} \in DE(a_i, w)$ and $d_{ijk} \in DE(a_j, w)$, then $flag = FALSE$, printf ("Potential WW Intra-UCID between: ", $a_i$, $a_j$, "on data element ", $d_{ijk}$); jump to 2.2.5;

2.2.5.3 If $d_{ijk} \in DE(a_i, r)$ and $d_{ijk} \in DE(a_j, w)$, then $flag = FALSE$, printf ("Potential RW Intra-UCID between: ", $a_i$, $a_j$, "on data element ", $d_{ijk}$); jump to 2.2.5;

2.2.5.4 If $d_{ijk} \in DE(a_i, w)$ and $d_{ijk} \in DE(a_j, r)$, then $flag = FALSE$, printf ("Potential RW Intra-UCID between: ", $a_j$, $a_i$, "on data element ", $d_{ijk}$);

**Step 3** Return $flag$.

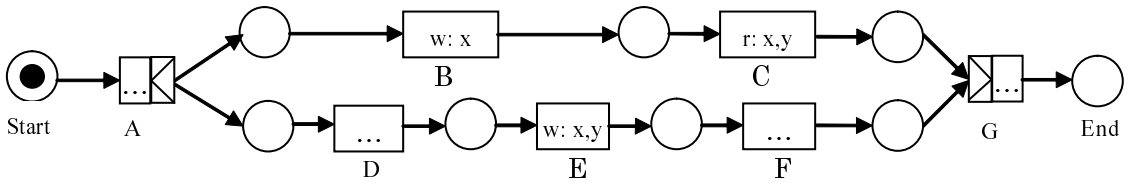



Figure 6.1: Potential intra-UCID example result

Figure 6.1 describes a simple example of a TDW. For the sake of simplicity, unnecessary information such as time and unrelated data are omitted. The main running steps and results when applying the above algorithm to this workflow are presented in Table 6.1. Potential WW Intra-UCID on data element $x$ between $B$ and $E$, and potential RW Intra-UCID on data elements $x$, $y$ between $C$ and $E$ are reported.

Table 6.1: Potential intra-UCID example result

| **S** | **Result** |
|---|---|
| $\{A, B, C, D, E, F, G\}$ | $< A, B >$, $< A, C >$, $< A, D >$, $< A, E >$, $< A, F >$, $< A, G >$: 2.2.2 (No data relation) |
| $\{B, C, D, E, F, G\}$ | $< B, C >$: 2.2.3 (Sequential Activities)<br>$< B, D >$, $< B, F >$, $< B, G >$: 2.2.2 (No data relation)<br>$< B, E >$: 2.2.5.2 (Potential WW Intra-UCID on x) |
| $\{C, D, E, F, G\}$ | $< C, D >$, $< C, F >$, $< C, G >$: 2.2.2 (No data relation)<br>$< C, E >$: 2.2.5.3 (Potential RW Intra-UCID on x, y) |
| $\{D, E, F, G\}$ | $< D, E >$, $< D, F >$, $< D, G >$ : 2.2.2 (No data relation) |
| $\{E, F, G\}$ | $< E, F >$, $< E, G >$: 2.2.2 (No data relation) |
| $\{F, G\}$ | $< F, G >$: 2.2.2 (No data relation) |

## 6.2 Potential Inter-UCID Detection Algorithm

Regarding UCID definitions, inter-UCIDs are identified based on the Active Interval of activities with data relation. However, Active Interval of an activity can only be determined at runtime when it has finished its execution. Therefore, we use Estimated Active

Interval instead of Active Interval to find potential UCID at build time, before a new TDW is put into the Concurrent TDW Management System to execute.

### 6.2.1 Calculation of Estimated Active Interval

Designating the start time of a TDW $w$ as a reference point, $P_w$, we can infer the Estimated Active Interval of an activity A, $[EST(A), LFT(A)]$, with respect to its minimum and maximum execution durations $\{min(A), max(A)\}$ and basic control structures.

Let us say that $A_s$ is the Start activity of a TDW $w$, then we have $EST(A_s) = P_w$ and $LFT(A_s) = P_w + max(A_s)$. For activity $A$ in progress, $EST(A) = S(A)$ and $LFT(A) = F(A)$ if $A$ has been completed.

**Sequential Connection** (Figure 4.1b)
$EST(A_j) = EST(A_i) + min(A_i); LFT(A_j) = LFT(A_i) + max(A_j)$

**AND-Split Connection** (Figure 4.1c)
$EST(A_j) = EST(A_i) + min(A_i); LFT(A_j) = LFT(A_i) + max(A_j)$
$EST(A_k) = EST(A_i) + min(A_i); LFT(A_k) = LFT(A_i) + max(A_k)$

**AND-Join Connection** (Figure 4.1d)
$EST(A_k) = MAX\{EST(A_i) + min(A_i); EST(A_j) + min(A_j)\}$
$LFT(A_k) = MAX\{LFT(A_i), LFT(A_j)\} + max(A_k)$

**OR-Split Connection** (Figure 4.1e)
$EST(A_j) = EST(A_i) + min(A_i); LFT(A_j) = LFT(A_i) + max(A_j)$
$EST(A_k) = EST(A_i) + min(A_i); LFT(A_k) = LFT(A_i) + max(A_k)$

**OR-Join Connection** (Figure 4.1f)
$EST(A_k) = MIN\{EST(A_i) + min(A_i); EST(A_j) + min(A_j)\}$
$LFT(A_k) = MAX\{LFT(A_i), LFT(A_j)\} + max(A_k)$

### 6.2.2 Algorithm

Given a Concurrent TDW Model $cwm = (W, T_{cwm})$, where $W = \{w_1, w_2, \cdots, w_k\}$ and $T_{cwm} = T_1 \cup T_2 \cup \cdots \cup T_k$, we assume that no errors are reported when applying the intra-UCID detection algorithm to all workflows in $cwm$. This algorithm is similar to the previous one, except that every two compared activities are in different workflows, and we must calculate the Estimated Active Intervals for all activities. If two activities have data relation and overlapping Estimated Time Intervals, there is a possibility of a RW/WW inter-UCID occurrence. If only the data relation exists and one activity occurs before the other, we will compare this situation with the pattern in Figure 5.1 to find out a potential UWU inter-UCID.

**Step 1** Initialization:
1.1 Let $S$ be a set of unchecked activities.

$S$ is initialized with all unfinished activities of $T_{cwm}$;
1.2 Calculate Estimated Active Interval for all activities in $S$;
1.3 $flag = TRUE$ is a Boolean variable;
1.4 $S'$ is the set of all unordered combination of $a_{mi}$, $a_{nk}$ where $(a_{mi}, a_{nk} \in S)$ $\wedge((a_{mi} \in T(w_m)) \wedge (w_m \in W)) \wedge ((a_{nk} \in T(w_n)) \wedge (w_n \in W)) \wedge (w_m \neq w_n)$;

**Step 2** Repeat the following steps until $S' = \emptyset$:
2.1 Remove a data element in $S'$, denoted as $< a_{mi}, a_{nk} >$;
2.2 Execute the following steps:
*/* Check Data Relation */*
2.2.1 $DE(a_{mi}, u) \cap DE(a_{nk}, u) = U_{mnik}$
2.2.2 If $(U_{mnik} = \emptyset)$ or $((DE(a_{mi}, w) \cap U_{mnik} = \emptyset) \wedge (DE(ank, w) \cap U_{mnik} = \emptyset))$, then jump to 2.1; */* No Data Relation or both $a_{mi}$ and $a_{nk}$ just have read access on shared data*/*

*/* check RW/WW inter-UCID */*
2.2.3 If $[EST(a_{mi}), LFT(a_{mi})] \wedge [EST(a_{nk}), LFT(a_{nk})] \neq \emptyset$, then repeat the following steps until $U_{mnik} = \emptyset$;
2.2.3.1 Remove a data element, denoted as $d_{mnikl}$, from $U_{mnik}$;
2.2.3.2 If $d_{mnikl} \in DE(a_{mi}, w)$ and $d_{mnikl} \in DE(a_{nk}, w)$, then $flag = FALSE$, printf("Potential WW Inter-UCID between: ", $a_{mi}$, $a_{nk}$, "of workflows ", $w_m$, $w_n$,"respectively on data element ", $d_{mnikl}$); jump to 2.2.3;
2.2.3.3 If $d_{mnikl} \in DE(a_{mi}, r)$ and $d_{mnikl} \in DE(a_{nk}, w)$, then $flag = FALSE$, printf("Potential RW Inter-UCID between: ", $a_{mi}$, $a_{nk}$, "of workflows ", $w_m$, $w_n$, "respectively on data element ", $d_{mnikl}$); jump to 2.2.3;
2.2.3.4 If $d_{mnikl} \in DE(a_{mi}, w)$ and $d_{mnikl} \in DE(a_{nk}, r)$, then $flag = FALSE$, printf("Potential RW Inter-UCID between: ", $a_{nk}$, $a_{mi}$, "of workflows ", $w_n$, $w_m$, "respectively on data element ", $d_{mnikl}$); jump to 2.2.3;

*/* check UWU inter-UCID based on Figure 5.1*/*
2.2.4 If $LFT(a_{nk}) < EST(a_{mi})$, then repeat the following steps until $U_{mnik} = \emptyset$;
2.2.4.1 Remove a data element,denoted as $d_{mnikl}$, from $U_{mnik}$;
2.2.4.2 If $d_{mnikl} \in DE(a_{nk}, w)$, then find the Preceding Closest Data Relation Transition of $a_{mi}$ on $d_{mnikl}$ denoted as $a_{mj}$: $a_{mj} = t_{p-cdrt}(a_{mi}, d_{mnikl})$;
2.2.4.3 If $((a_{mj} \neq null) \wedge ([EST(a_{nk}), LFT(a_{nk})] \subset [LFT(a_{mj}), EST(a_{mi})]))$, then $flag = FALSE$, printf("Potential UWU Inter-UCID among: ", $a_{mi}$, $a_{mj}$, $a_{nk}$, "of workflows ", $w_m$, $w_m$, $w_n$, "respectively on data element ", $d_{mnikl}$);
2.2.4.4 If $d_{mnikl} \in DE(a_{mi}, w)$, then find the Succeeding Closest Data Relation Transition of $a_{nk}$ on $d_{mnikl}$ denoted as $a_{nq}$: $a_{nq} = t_{s-cdrt}(a_{nk}, d_{mnikl})$;
2.2.4.5 If $((a_{nq} \neq null) \wedge ([EST(a_{mi}), LFT(a_{mi})] \subset [LFT(a_{nk}), EST(a_{nq})]))$, then $flag = FALSE$, printf("Potential UWU Inter-UCID among: ", $a_{nq}$, $a_{nk}$, $a_{mi}$, "of workflows ", $w_n$, $w_n$, $w_m$, "respectively on data element ", $d_{mnikl}$);

2.2.5 If $LFT(a_{nk}) > EST(a_{mi})$, then repeat the following steps until $U_{mnik} = \emptyset$;
2.2.5.1 Remove a data element,denoted as $d_{mnikl}$, from $U_{mnik}$;
2.2.5.2 If $d_{mnikl} \in DE(a_{nk}, w)$, then find the Succeeding Closest Data Relation Transition of $a_{mi}$ on $d_{mnikl}$ denoted as $a_{mo}$: $a_{mo} = t_{s-cdrt}(a_{mi}, d_{mnikl})$;
2.2.5.3 If $((a_{mo} \neq null) \wedge ([EST(a_{nk}), LFT(a_{nk})] \subset [LFT(a_{mi}), EST(a_{mo})]))$, then $flag = FALSE$, printf("Potential UWU Inter-UCID among: ", $a_{mi}$, $a_{mo}$, $a_{nk}$, "of workflows ", $w_m$, $w_m$, $w_n$, "respectively on data element ", $d_{mnikl}$);
2.2.5.4 If $d_{mnikl} \in DE(a_{mi}, w)$, then find the Preceding Closest Data Relation Transition of $a_{nk}$ on $d_{mnikl}$ denoted as $a_{np}$: $a_{np} = t_{p-cdrt}(a_{nk}, d_{mnikl})$;
2.2.5.5 If $((a_{np} \neq null) \wedge ([EST(a_{mi}), LFT(a_{mi})] \subset [LFT(a_{np}), EST(a_{nk})]))$, then $flag = FALSE$, printf("Potential UWU Inter-UCID among: ", $a_{nk}$, $a_{np}$, $a_{mi}$, "of workflows ", $w_n$, $w_n$, $w_m$, "respectively on data element ", $d_{mnikl}$);

**Step 3** Return flag.

We now show an example to demonstrate the efficiency of potential inter-UCID detection algorithm. Referring to Figure 6.2, we have a Concurrent TDW Model, $cwm$, with two TDWs $w_m$ and $w_n$. $w_n$ has been started, and activity $M$ has already completed with the active interval $[S(M), F(M)] = [2, 3]$. TDW $w_m$ intends to execute at $P_w = 3.5$. As described in Section 6.2.1, we can calculate the Estimated Active Interval of activities in $cwm$ as follows: $[EST(M), LFT(M)] = [2, 3]$, $[EST(N), LFT(N)] = [3, 5.5]$, $[EST(P), LFT(P)] = [5, 7.5]$, $[EST(A), LFT(A)] = [3.5, 4.5]$, $[EST(B), LFT(B)] = [3.5, 11.5]$, $[EST(C), LFT(C)] = [8.5, 14.5]$, $[EST(D), LFT(D)] = [3.5, 7.5]$, $[EST(E), LFT(E)] = [5.5, 9.5]$, $[EST(F), LFT(F)] = [10.5, 16.5]$.

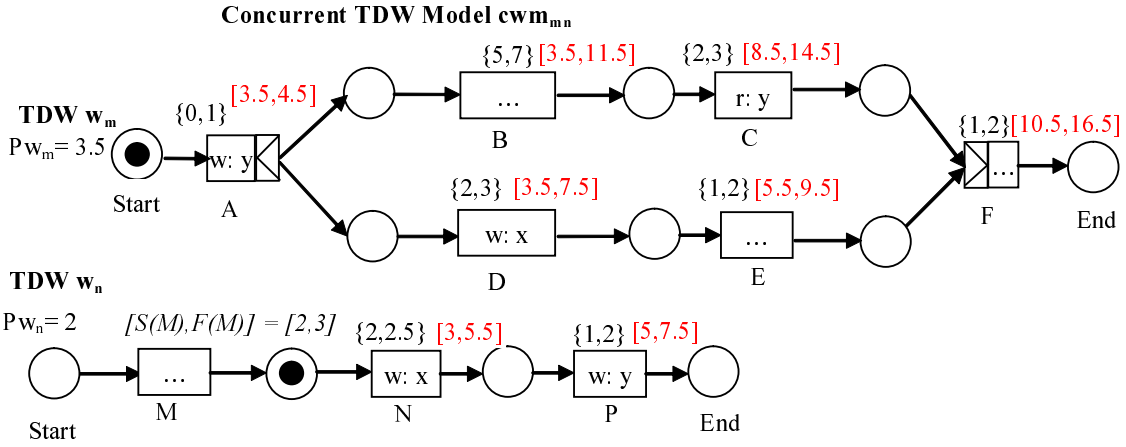



Figure 6.2: A Concurrent TDW Model example

Table 6.2 describes the main steps and the result when executing this algorithm on $cwm$. The following errors are detected: potential WW Inter-UCID on data element $x$

between $N$ and $D$, potential UWU Inter-UCID on data element $y$ among $C$, $P$, $A$.

Table 6.2: Potential inter-UCID example result

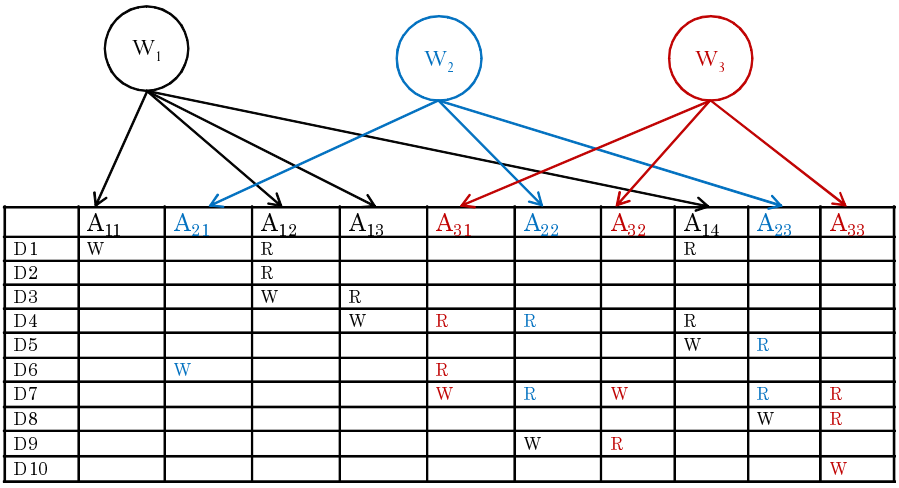
| **S'** | **Result** |
|---|---|
| $\{< N, A >, < N, B >,$ $< N, C >, < N, D >,$ $< N, E >, < N, F >,$ $< P, A >, < P, B >,$ $< P, C >, < P, D >,$ $< P, E >, < P, F >, \}$ | $< N, A >, < N, B >, < N, C >, < N, E >, < N, F >$ : 2.2.2 (No data relation)<br>$< N, D >$: 2.2.3.2 (Potential WW Inter-UCID on x)<br>$< P, B >, < P, D >, < P, E >, < P, F >$ : 2.2.2 (No data relation)<br>$< P, A >$: 2.2.4.5 (Potential UWU Inter-UCID on y among C, A, P) where $C = t_{s-cdrt}(A, y)$<br>$< P, C >$: 2.2.5.5 (Potential UWU Inter-UCID on y among $C, A, P$) where $A = t_{p-cdrt}(C, y)$ |

## 6.3 Algorithm Evaluation

Let's say $n$ is the number of unfinished activities in a Concurrent TDW Model *cwm*. In general, we must inspect $C_n^2$ combinations of any two unfinished activities to find some potential UCIDs. This algorithm allows us to detect not only potential UCID at build time of pre-execution TDWs, but also potential UCID at run time of running TDWs by recalculating the Estimated Active Intervals of their unfinished activities more accurately, based on the Active Intervals of finished activities. However, depending on application, we can reduce the number of checking steps by considering some of the following heuristics:

- A two-dimensional table can be used to record the access rights to data elements of activities in a Concurrent TDW Model *cwm*. Figure 6.3 shows an example of data flow matrix of a Concurrent TDW Model with three TDWs $W_1$, $W_2$ and $W_3$. $\{D_1, \cdots, D_{10}\}$ is the data set of the Concurrent TDW Model. Parallelization can be applied here to reduce execution time. For each element in the data set of *cwm*, there is a thread responsible for checking potential UCIDs caused by activities using this data element.

- After designing a new TDW, UCID check is conducted to find potential UCIDs before this TDW is put into the Concurrent TDW Management System for execution. Let's say $m$ is the number of activities in the pre-execution TDW under consideration, $k$ is the total number of activities in the other pre-execution TDWs and $l$ is the total number of unfinished activities in running TDWs, we have $n = m + k + l$. Because the other pre-execution TDWs have been checked previously, we can skip combinations of two activities in these TDWs to reduce the number of combination inspected to $C_n^2 - C_k^2$. If we just want to detect UCIDs caused by activities of the TDW under consideration, we will verify $m \times n$ activity combinations only. A parallel solution in this case is to create $m$ threads. Each thread will be responsible for




| | $A_{11}$ | $A_{21}$ | $A_{12}$ | $A_{13}$ | $A_{31}$ | $A_{22}$ | $A_{32}$ | $A_{14}$ | $A_{23}$ | $A_{33}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| D1 | W | | R | | | | | R | | |
| D2 | | | R | | | | | | | |
| D3 | | | W | R | | | | | | |
| D4 | | | | W | R | R | | R | | |
| D5 | | | | | | | | W | R | |
| D6 | | W | | | R | | | | | |
| D7 | | | | | W | R | W | | R | R |
| D8 | | | | | | | | | W | R |
| D9 | | | | | | W | R | | | |
| D10 | | | | | | | | | | W |

Figure 6.3: Data flow matrix example

one activity in this TDW and will verify potential UCIDs on combinations created by this activity with other activities in different TDWs.

- Because potential UCIDs just occur in activities that have shared data, we will only verify activities having shared a data element only. Each data element will store IDs of unfinished activities using it. Therefore, the set of checked activities can be limited to unfinished activities with data relation in the Concurrent TDW Model. If the number of data elements is small, we can start from data elements of the pre-execution TDW under consideration to select unfinished activities in the Concurrent TDW Model with data relations, and use UCID patterns to find potential errors.

# Chapter 7

# Potential UCID Resolution

In general, if potential UCIDs are detected, a review of designed workflows should be conducted to make sure that these situations were not created on purpose. Our given solutions (some of which will change the workflow structure) are simply reference models. The final decision will depend on workflow designers to perform modifications that actually lead to a resolution of the model.

## 7.1 Potential Intra-UCID Resolution

Potential Intra-UCID may be caused by a mistake by workflow designers in designing parallel branches of a workflow. Therefore, our solution for Intra-UCID is to change the workflow structure by sequentializing or combining UCID-related activities. Two activities causing potential WW Intra-UCID are merged into one by place/transition fusion (Figure 7.1a). Regarding RW Intra-UCID, sequentialization is applied to the related activities. One solution is to schedule the *read* activity before the *write* activity, and another is to schedule the *write* activity to occur before the *read* activity (Figure 7.1b). Resolution priority will proceed from WW Intra-UCID cases to RW Intra-UCID cases. Regarding potential UCIDs belonging to the same group, the priority is the order of occurrence.

## 7.2 Potential Inter-UCID Resolution

Resolving potential inter-UCID is more complex because workflows are designed for different purposes by different designers, and a designer may know nothing about the work of the others. To resolve inter-UCID, the cooperation from different designers is necessary, and the result will highly depend on the willingness of designers to communicate with each other.

One solution is to adjust the workflow schedule by modifying the workflow start time and the maximum and minimum execution durations of activities in workflows, so that inter-UCID patterns do not occur. However, rescheduling algorithm is outside the scope of this thesis.

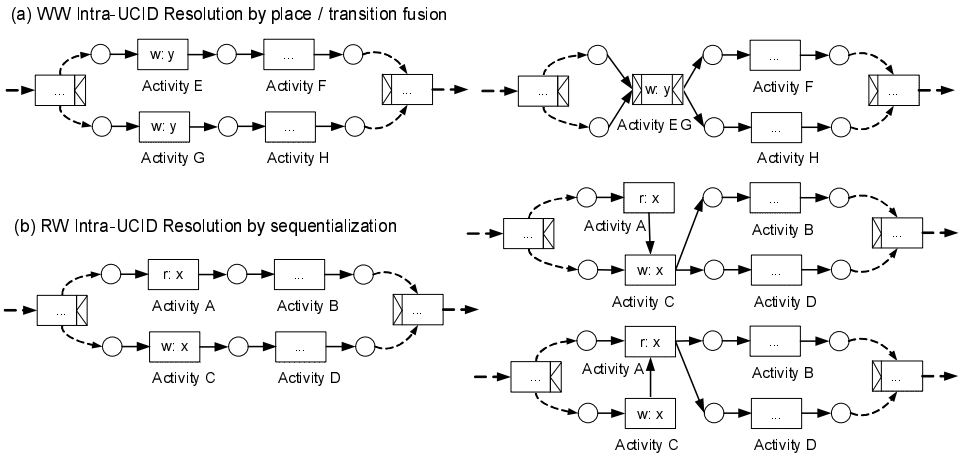



Figure 7.1: Potential intra-UCID resolution

Here, we suggest creating a **Global UCID-related Workflow (GUW)**. GUW is a synthesis of UCID-related workflows in which global constraints, constraints among UCID-related activities in different workflows, are added to resolve UCID patterns. Change workers can still perform change activities by following their original workflows with respect to the GUW. The GUW is operated by the Concurrent TDW workflow management system. System will observe the behavior of sub-workflows to update the states of activities in GUW. When the UCID-related activities prepare to be performed, system will notify related workers risky points and force them to follow the global constraints specified in the GUW. GUW is created by combining UCID related workflows into one with the help of AND-Split, AND-Join transitions and Time Start Transitions. In this workflow, fusion technique of Petri Net and sequentialization are used to reorder UCID-related transitions.

First, we will combine related TDWs into one workflow by using AND-Split and AND-Join transitions. In order to preserve the structure of the original TDWs, the source place of GUW connects to the AND-Split transition and its sink place is connected to the AND-Join transition. Each merged TDW corresponds to a subnet starting from the AND-Split transition and ending at the AND-Join transition. Because the merged TDWs start at different times, we insert a Time Start transition between the Start place of each merged TDW and the AND-Split transition so that we can control when sub-workflows start. Time activities are just null activities with some duration and they help to combine TDWs without modifying the start time of merged TDWs. The AND-Split transition, AND-Join transition, Time Start transitions, places and arcs connecting the related workflows together represent the global constraints between UCID-related activities in different workflows. They will not be used to identify the total order of activities in detecting potential intra-UCID in the synthesis workflow. In the case of a running TDW, we can create a new TDW from the original workflow by removing its finished activities, and this new TDW will be combined with other TDWs in a normal way.

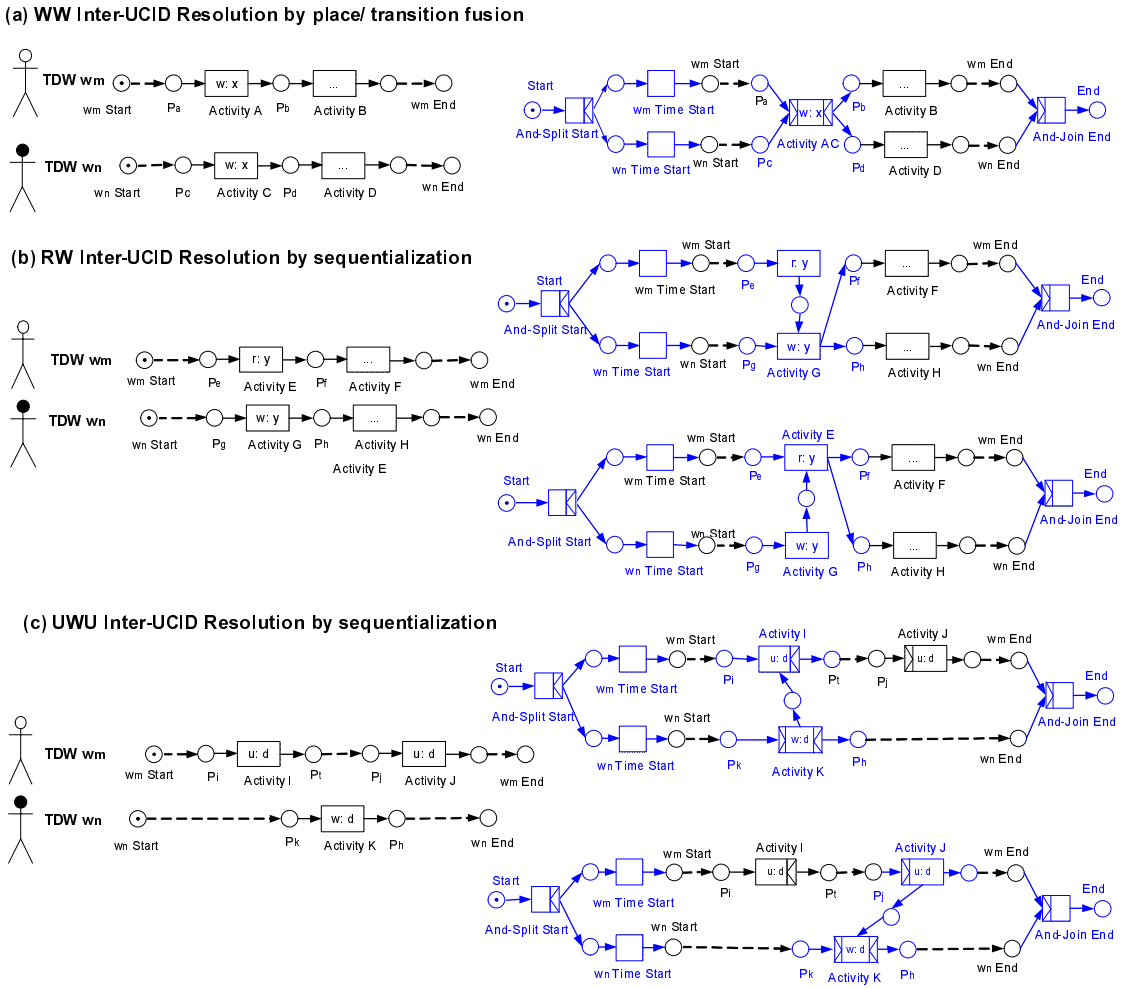



Figure 7.2: Potential inter-UCID resolution

Next, we will deal with activities causing potential Inter-UCID. The mechanism to handle potential WW/RW Intra-UCID is applied to potential WW/RW Inter-UCID cases. Two activities causing WW Inter-UCID are merged in to one using the transition fusion (Figure 7.2a). In the case of RW Inter-UCID, the *read* activity can be scheduled before or after the *write* activity (Figure 7.2b). Regarding potential UWU-UCID, the *write* activity causing the inconsistent view is rescheduled to occur before or after the two *use* activities, as shown in (Figure 7.2c). If there are many potential Inter-UCIDs between the same two TDWs, the priority is first by Inter-UCID types (WW > RW > UWU) and then by time of activities.

Although GUW offers a more comprehensive view of UCID-related workflows, the proposed solution is still simple and not effective in all cases. We will try to improve it in future work.

# Chapter 8

# UCID Theory and Change Support Workflow Model

In this chapter, we present how to apply the UCID Theory to the Change Support Workflow Model and give an example of this process.

## 8.1 Applying the UCID Theory to the Change Support Workflow Model

In order to apply the UCID theory to the CSW Model, we need to model CSWs as TDWs. To implement this task, we will create a data flow skeleton of CSW and use the algorithm given in Section 8.2 to generate a CSW based on the given data flow skeleton.

This draft of CSW will help workflow designers in developing the schedule of the change process. The other steps in developing change schedule, such as estimating activity resources and activity durations will be performed by workflow designers. From Activity Duration estimated by designers, minimum and maximum execution durations of transitions in this CSW can be derived. Because all necessary information for UCID check is enough, potential UCID detection algorithms can be executed on the system with the newly designed CSW. If some potential UCIDs are reported, UCID-related CSWs, especially the new CSW, should be reviewed based on suggested UCID resolutions.

## 8.2 Generating a CSW based on Data Flow Skeleton

Data flow skeleton of a CSW is generated based on dependency relationships among software artifacts supplied by the information model. In a naive way, this skeleton is constructed by tracing these relationships starting from the elements receiving the change request directly. However, construction of this skeleton is affected by many factors besides dependency relationships such as change requirements, resource, etc. Therefore, implementation of this task will be left as future work. In this section, we focus on how to generate a CSW from data flow skeleton by using TDW modeling language.

Assuming that we have already generated a data flow skeleton of the CSW which is a directed graph $G$ where:

- **Nodes** are software artifacts that will be created or modified to fulfill a change request.
- An **arc** $e = (x, y)$ is considered to be directed from $x$ to $y$, $y$ is called the head and $x$ is called the tail of the arc, if there is a dependency relationship with the Supplier $x$ and the Client $y$.
- **Direct Predecessor** of a node $y$, $P(y)$ is a set of nodes $x$ so that there is a dependency relationship with the Supplier $x$ and the Client $y$.
- **Indegree** of a node $v$ is the number of head endpoints adjacent to it.
- **Outdegree** of a node $v$ is the number of tail endpoints adjacent to it.
- **Source** is a node with $indegree(v) = 0$ (no input).
- **Sink** is a node with $outdegree(v) = 0$ (no output).

From $G$, we can generate CSW using TDW as follows (Figure 8.1). Each node $d$ in $G$ is modeled by a transition $t$ in TDW. Write data set of $t$ is the collection of output node of $d$ and read data set is the collection of input node and output node of $d$. The arc between two transitions is identified based on the arc connecting corresponding nodes in $G$. Transitions with no input are connected to the source place by an AND-split transition and transitions with no output are connected to the sink place by an AND-Join transition.

1. Initialization
   (a) Create a source place and a sink place
   (b) For each node $d$ of $G$, create a transition $t$ with one input place $p_i$ and one output place $p_o$: $DE(t, w) = d$, $DE(t, r) = d \cup P(d)$;
2. For each arc $(d_1, d_2)$ in $G$ with $d_1$ represented by transition $t_1$ and $d_2$ represented by transition $t_2$, execute the following steps:
   (a) If $outdegree(d_1) = indegree(d_2) = 1$, then merge output place of $t_1$ with the input place of $t_2$;
   (b) If $outdegree(d_1) > 1$, $indegree(d_2) = 1$, then:
      - If $t_1$ is not an AND-Split transition, then merge output place of $t_1$ with the input place of $t_2$ and add the AND-Split property to $t_1$;
      - If $t_1$ is an AND-Split transition, then add an arc connecting $t_1$ to the input place of $t_2$;
   (c) If $outdegree(d_1) = 1$, $indegree(d_2) > 1$, then:

- If $t_2$ is not an AND-Join transition, then merge output place of $t_1$ with the input place of $t_2$ and add the AND-Join property to $t_2$;
- If $t_2$ is an AND-Join transition, then add an arc connecting the output place of $t_1$ to $t_2$;

(d) If $outdegree(d_1) > 1$, $indegree(d_2) > 1$, then:

- If $t_1$ is not AND-Split transitions and $t_2$ is not an AND-Join transition, then merge output place of $t_1$ with the input place of $t_2$, add the AND-split property to $t_1$ and add the AND-Join property to $t_2$;
- If $t_1$ is an AND-split transition and $t_2$ is not an AND-Join transition, then add an arc connecting $t_1$ to the input place of $t_2$ and add the AND-Join property to $t_2$;
- If $t_1$ is not an AND-split transition and $t_2$ is an AND-Join transition, then add an arc connecting the output place of $t_1$ to $t_2$ and add the AND-Split property to $t_1$;
- If $t_1$ is an AND-split transition and $t_2$ is an AND-Join transition, then add a new place and connect $t_1$ to $t_2$ via new place;

3. Source place and sink place

(a) If there are many source nodes ($indegree = 0$), then connect the input places of the corresponding transitions to the source place by an AND-split transition;

(b) If there is only one source node ($indegree = 0$), then merge the input place of this transition with the source place;

(c) If there are many sink nodes ($outdegree = 0$), then connect the output places of the corresponding transitions to the sink place by an AND-Join transition;

(d) If there is one sink node ($outdegree = 0$), then merge the output place of this transition with the sink place;

## 8.3 Example

Let us take an example.

Figure 8.2 shows an example of dependency relationships between software artifacts created in different phases of a software development process. If we change Artifact 1, we need to change Artifacts 4, 5, 8 and 9 because of the relationships between them. Similarly, if we change Artifact 2, we need to change Artifacts 5, 6, 9, 10 and 11.

Using algorithm in Section 8.2, we can create two CSWs to respond to change requirements on Artifact 1 and Artifact 2 (Figure 8.3).

Based on the generated workflows, the project manager can conduct other steps in project time management such as estimating activity resources, estimating activity durations, etc. Information about activity duration is used to detect potential UCIDs. In

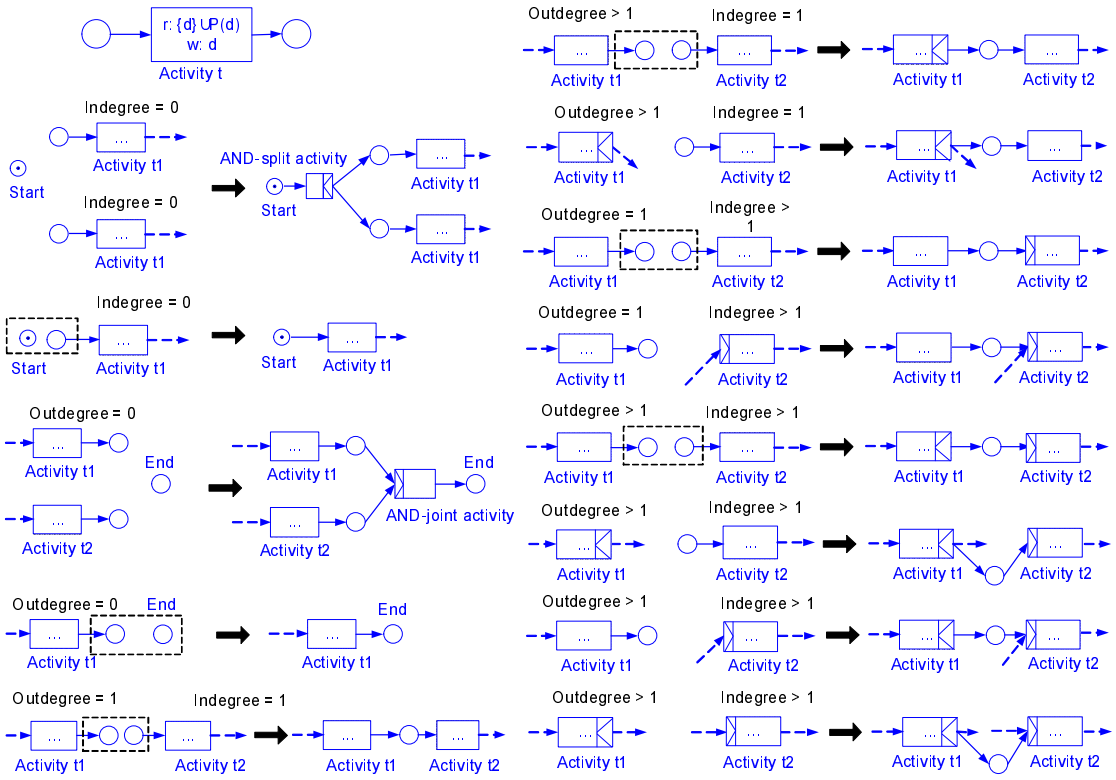



Figure 8.1: Generating CSW from data flow skeleton

Table 3, the minimum and maximum execution durations of each activity in CSWs described in Figure 8.3 are calculated from the Activity Duration Estimate which is the quantitative assessment of the likely number of work periods that will be required to complete an activity [20] , of the corresponding activity in the project time management. Based on these values and the start time of the corresponding workflow, we can calculate the Estimated Active Intervals using the formulas given in Section 6.2.1.

After executing the Inter-UCID detection algorithms, the following potential Inter-UCIDs are reported: WW Inter-UCID between Activity 3 and Activity B on Artifact 5, WW Inter-UCID between Activity 5 and Activity D on Artifact 9, RW Inter-UCID between Activity 3 and Activity A on Artifact 2, RW Inter-UCID between Activity 5 and Activity B on Artifact 5.

By applying the second Inter-UCID resolution method, we receive the Global UCID-related Workflow shown in Figure 8.4.

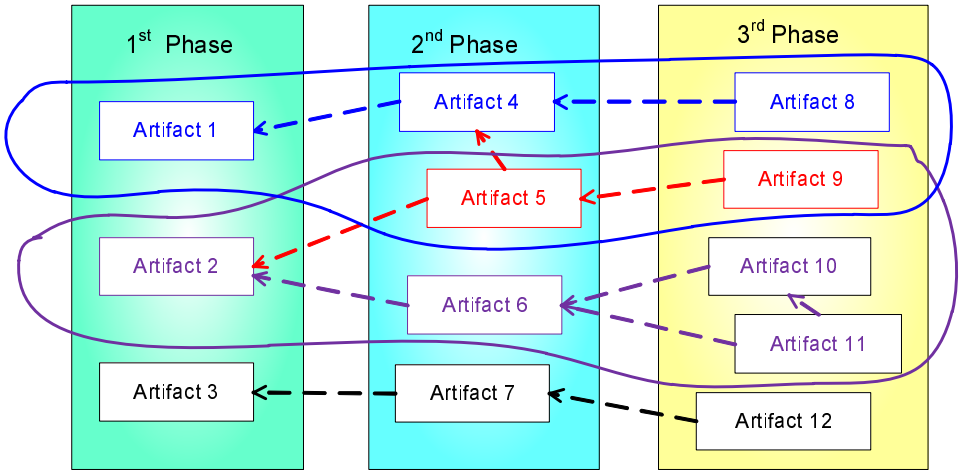



Figure 8.2: Example of dependency relationships between software artifacts created during a software development process

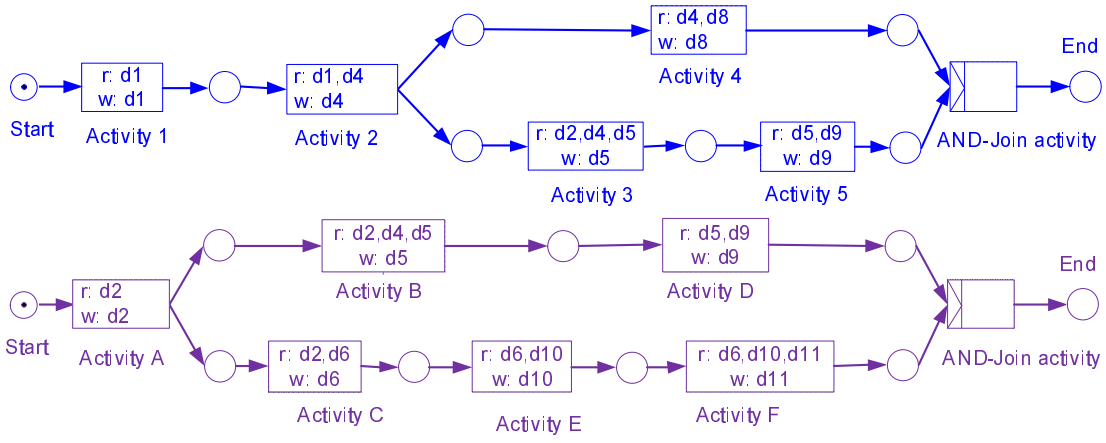



Figure 8.3: Examples of CSWs created based on the dependency relationships between software artifacts

Table 8.1: Time aspect of activities in CSWs described in Figure 8.3

| CSW ID | Start Time $P_w$ | Activity Name | Activity Duration Estimates (days) | Minimum and Maximum Execution Duration | Estimated Active Interval |
|---|---|---|---|---|---|
| W1 | 5 | Activity 1 | 7.5 ± 0.5 | {7,8} | [5,13] |
| | | Activity 2 | 5.5 ± 0.5 | {5,6} | [12,19] |
| | | Activity 3 | 11 ± 1 | {10,12} | [17,31] |
| | | Activity 4 | 6 ± 1 | {5,7} | [17,26] |
| | | Activity 5 | 7 ± 1 | {6,8} | [27,39] |
| | | And-Join | 0 | {0,0} | [33,39] |
| W2 | 15 | Activity A | 6 ± 1 | {5,7} | [15,22] |
| | | Activity B | 5 ± 1 | {4,6} | [20,28] |
| | | Activity C | 5 ± 1 | {4,6} | [20,28] |
| | | Activity D | 10 ± 1 | {9,11} | [24,39] |
| | | Activity E | 5.5 ± 0.5 | {5,6} | [24,34] |
| | | Activity F | 6 ± 1 | {5,7} | [29,41] |
| | | And-Join | 0 | {0,0} | [34,41] |

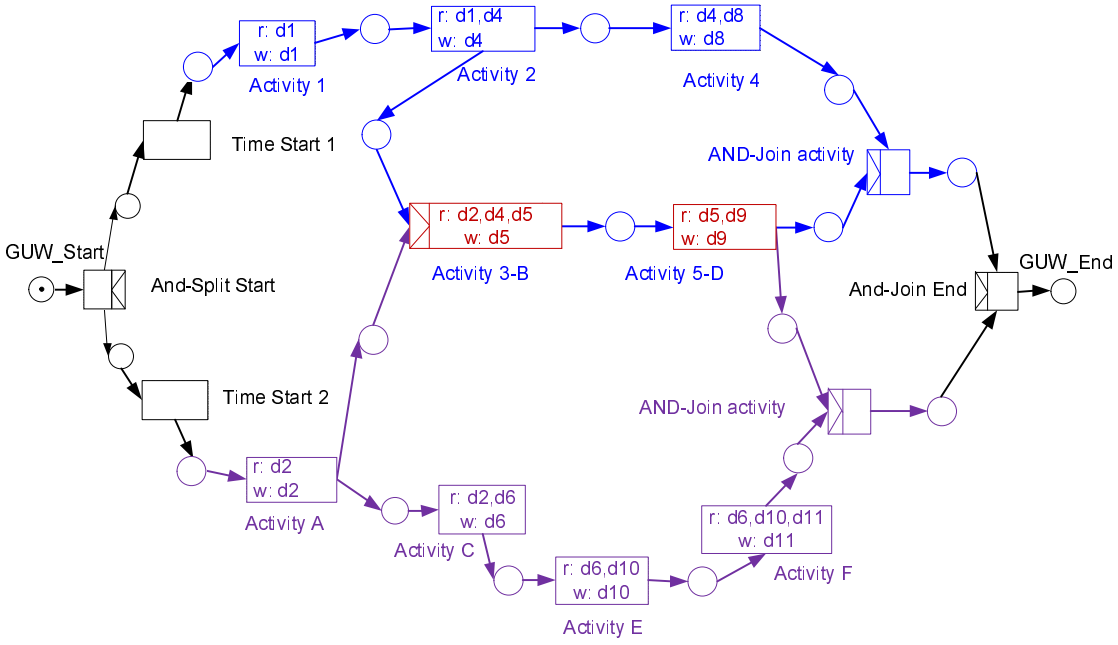



Figure 8.4: Global UCID-related Workflow of CSWs described in Figure 8.3

# Chapter 9

# Discussion

## 9.1 Evaluation of the Proposed Method

In this thesis, we have presented UCID patterns and methods to detect and resolve potential UCID errors at build time. Although there are some previous works on data flow verification, they only considered detecting errors in a single workflow. Therefore, by detecting and resolving UCID patterns caused by concurrent activities not only in a single workflow but also in different workflows, our approach is more complete. When applying the UCID solution to a Workflow Management System (WFMS), workflow designers could achieve a more comprehensive view of data related workflows in the system. In addition, detecting potential UCID at build time also helps workflow designers make timely adjustments to the original workflows to avoid errors at runtime. Also, Global UCID-related Workflow will help change workers to have a better cooperation in solving problems concerning shared data. It can also be used in the recovery process in the case of workflow failure.

However, there are still some limitations in our approach. Because we use the time metric to detect potential inter-UCIDs at build time, our problem domain is limited to static systems where execution time can be estimated before execution. However, this disadvantage could be overcome if we use this approach to detect UCID errors from workflow execution histories. Another disadvantage is that UCID detection algorithm is a best-effort method. We must check all transitions in a Concurrent TDW Model to detect potential UCIDs because data elements are integrated as attributes of transitions instead of being modeled explicitly. Also, the potential Inter-UCID resolution using Global UCID-Related Workflow demands considerable investments of time and effort to make it become an effective method.

## 9.2 Consideration of a New Method

To deal with the limitations of the proposed method, we are considering modeling workflow data as independent elements of workflow. In this way, we can manage data relations of all activities in a WFMS. Similar to the Global UCID-Related Workflow, we can build

a **Global Data-related Net (GDN)** which is a comprehensive view of data-related activities in the WFMS. When a new workflow is designed, it can be integrated into the existing GDN easily by merging shared data element. Data-related activities can be identified at a very early stage to have timely adjustment to reduce design errors as much as possible. Global constraints can be added to the GDN to control order of activities with data relation in different workflows. If UCID detection algorithm is applied here, its checking domain should be greatly reduced. GDN can also be used to manage workflows at run time by informing change workers of data-related activities when an activity prepares to use the shared data, and by supplying a dynamic channel to these change workers to help them reach an agreement in the case of potential UCID. The model versioning system AMOR [22] offers some methods to resolve collaborative conflict in model versioning. In AMOR approach, all of the people who perform changes are involved in eliminating the conflicts to obtain one consistent model version. We will consider applying this approach in our environment to increase flexibility of the system.

In the new method, we will still choose Petri Net as the foundation for modeling workflow. Existing Petri Net tools such as WoPeD [19] , CPN Tools [25] , Yasper [26], can also be leveraged to edit, simulate the execution of generated workflows or analyze properties of these workflows such as boundedness, liveness, Free-choice violence, etc. However, when data elements are modeled explicitly, the complexity of the generated workflows will increase. How to balance these two factors is one of the problems we must solve in future work. Therefore, we need to spend more time and efforts to answer the question about the effectiveness of this method.

Figure 9.1a show an example of this method. We transform workflows described in the motivating example into the new model using WoPeD tool [19] as follows:

- Because WoPeD does not support data modeling, we model each data element as a composite transition with two input places to receive Read Request and Write Request, and two output places to return Read Response and Write Response.
- Each activity of workflows is modeled by a transition.
- Activity which reads data will have an extra transition to prepare its input data: pre-transition. The pre-transition is connected to the input place which receives the Read Request of the corresponding data element. The main transition is also connected from the output place which returns the Write Response of this data element.
- Main transition of activity which writes data is connected to the input place receiving the Write Request of the corresponding data element. The output place which returns the Write Response of this data element will be connected to the transitions succeeding this transition. In the case of no succeeding transition, an extra-transition is added: post-transition to receive the Write Response.

Figure 9.1b is another example of the new method using Yasper tool [26]. We use "data store" concept of the Yasper tool to represent data element. A store is like a place in

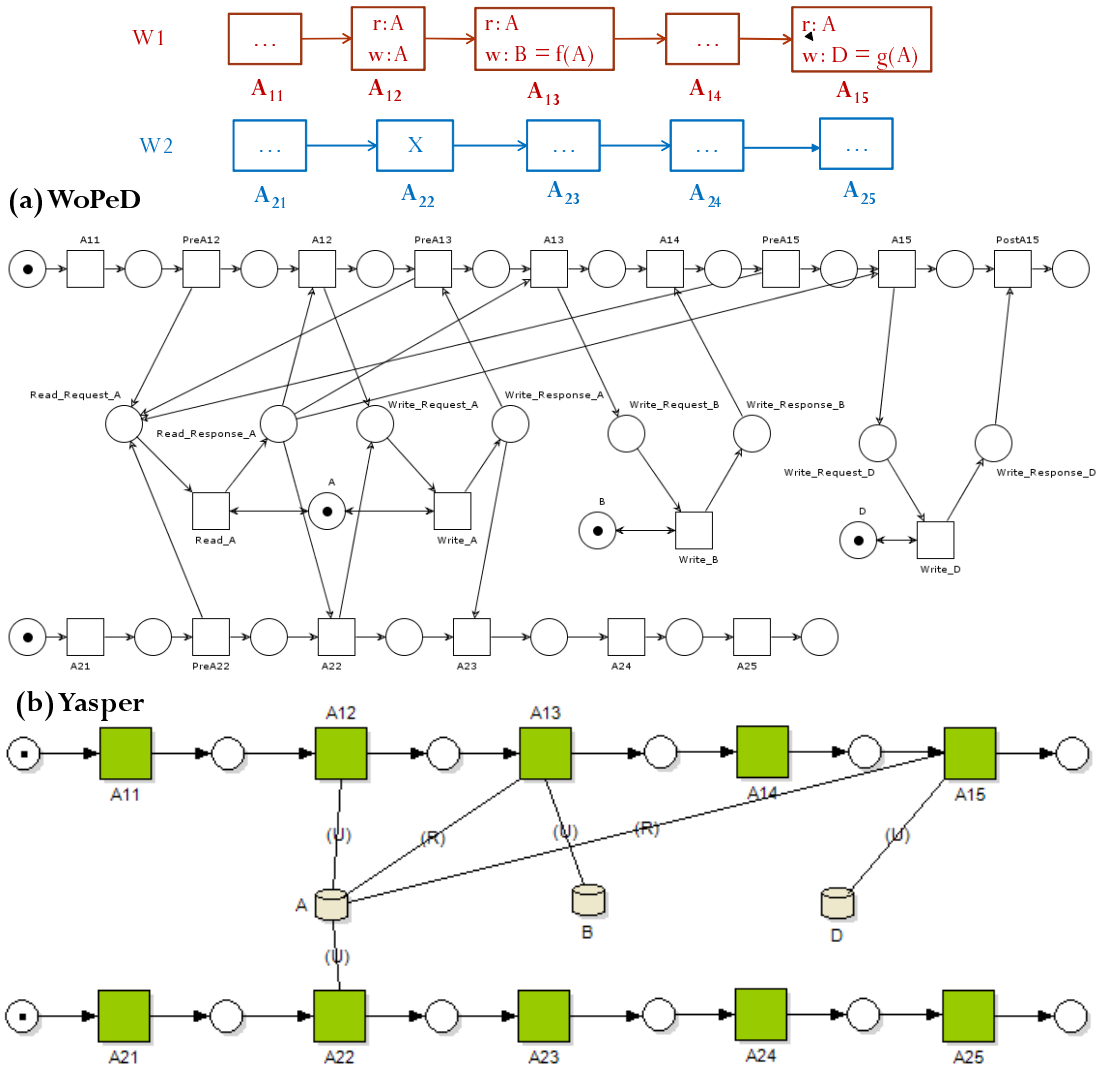



Figure 9.1: Example of workflows with visualized data elements

that it is connected with transition by a special arc type. This arc type is different the normal arc types since tokens cannot be added to or removed from a store. Instead, it can be specified whether a transition can create (C), read (R), update (U) and/or delete (D) values in the store. In this figure, each activity is modeled by a transition and each data element is modeled by a data store. Therefore, compared with modeling by WoPeD tool, modeling by Yasper is simpler. However, Yarper does not use the specifications of stores and store arcs in any way; simulations ignore them. Therefore, we must handle data analysis by ourselves.

## 9.3 Future Work

As mentioned in Chapter 1 and Section 2.1.2, the aim of our research is to develop the CSW Model, a part in the project on building a change support environment for cooperative software development. Therefore, in future, we will continue the remaining works in building the CSW Model:

1. Synchronization of changes on shared software artifacts by different CSWs

   In the scope of this thesis, we have tried to detect and solve error caused by uncontrolled access to shared data, UCID, in a general WFMS and have applied it to the CSW Model. There are still some drawbacks in this approach as we have discussed in Section 9.1. Therefore, we will improve and expand the proposed method with regard to the new method mentioned in Section 9.2:

   (a) Improving the Inter-UCID correction method.

   (b) Solving UCID detection and correction at run time.

2. Access control

   (a) An access control model is proposed by adapting RBAC model to specific requirements of software development process and change supporting workflow domain. RBAC consistent rules considered in this model are: cardinality constraints, role hierarchy constraints and separation of duties constraints. In a cooperative environment, this access control model must allow simultaneous access of the same data by multiple users under the software concurrent permission and policies of data access control should be adjusted dynamically as changes of projects can happen at any stages in the whole cycle. By assigning a local access control matrix to each transition in a workflow to grant rights to subjects (that will execute the transition) only to data being consumed or produced by the transition, subjects will have only the access rights (specified by this matrix) when the transition is active. This method makes sure that authorized subjects gain access on the required objects only during the execution of the specific task.

   (b) After that, we will study a formal technique to model and analyze our access control model using CPN and CPN Tool for editing, simulating and analyzing CPN. CPN Tools provides a graphical representation and an analysis framework that can be used easily by security administrators to understand why some permissions are granted or not and to detect whether security constraints are violated

3. CSW Construction

   To implement this task, we will create the data flow of CSW and develop an algorithm that can generate a CSW based on the data flow skeleton.

(a) Generating data flow skeleton

i. First, we will develop an algorithm that can generate a draft of data flow skeleton of CSW from UML model elements with dependency relationships described in [1]. In developing the algorithm, we need to consider how to remove redundant relationships because the result described in [1] may include a lot of redundant relationships.

ii. Next, we must define some selection rules to identify UML model elements which need to be changed and their change orders as exactly as possible according to the change requirement. These rules can be constructed by analyzing influences to each Basic Dependency Relationship defined in [1] of different kinds of change: deletion, extension (the substitution of an entity for another one that preserves the information, behavior and structure of the initial entity), modification (the substitution of an entity for another one that (partially) destroys the initial information, behavior and structure).

iii. As selection rules cannot cover every change situation, we can integrate knowledge of workers involved in the software change process by allowing them to revise the generated data flow.

(b) Generating CSW

CSW can be built directly from the data flow skeleton generated in the preceding step by using the algorithm presented in Section 8.2. However, because there are many aspects besides data such as human resource, access control, the control flow of the generated CSW can be different from this data flow skeleton.

4. Finally, the workflow model is realized as an Eclipse plug-in to increase its usability and applicability. Another possibility is to integrate our theory into open source systems, for example, WoPeD [19].

# Chapter 10

# Conclusion

This thesis is our first efforts in building the Change Support Workflow Model which is responsible for CSW construction and management in the CSE project. In this thesis, we have proposed an approach to synchronize changes on shared data by different CSWs in a cooperative software development environment by detecting and resolving errors caused by non-deterministic access to shared data by different CSWs. We have abstracted this kind of errors as Unintentional Change in In-use Data, a situation in which some data values are lost or some data elements are assigned values different from the intentions of workflow designers, due to non-deterministic access to shared data by different activities. In contrast to previous work, we have considered UCID patterns caused by not only the interactions of concurrent activities inside a single workflow, intra-UCID, but also the mutual influences between concurrent workflows, inter-UCID. We have also proposed a Time Data Workflow based on the WF-Net with many attributes supporting UCID estimation. Algorithms which help detect intra-UCID and inter-UCID patterns in a Concurrent TDW Management System have been developed too. Then, algorithm evaluation and some solutions to resolve UCID problem have been given. Finally, we have presented how to apply UCID theory to the Change Support Workflow Model to detect and resolve errors concerning shared data among CSWs at build time.

As future work, we will improve the algorithms and inter-UCID resolutions with paying attention to unique features of the CSW Model. Detecting and resolving this problem at runtime are our next targets. We will continue with the unsolved problems of the CSW Model: access control, CSW construction and finally a plugin as a realization of the CSW Model.

# Publication

1. Phan Thi Thanh Huyen and Koichiro Ochimizu: Detection of Unintentional Change on In-use Data for Concurrent Workflows. In: Proceedings of the 2010 International Conference on Software Engineering Research and Practice (SERP 10), pp 277-283. Las Vegas, Nevada, USA (2010)

2. Thi Thanh Huyen Phan and Koichiro Ochimizu: Detecting and Repairing Unintentional Change inn In-use Data in Concurrent Workflow Management System. In: Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'10), pp 89-110. Braga, Portugal (2010)

# Bibliography


[1] Masayuki Kotani and Koichiro Ochimizu: Automatic Generation of Dependency Relationships between UML Elements for Change Impact Analysis. Journal of Information Processing Society of Japan, vol. 49, no.7, pp 2265-2291 (2008)

[2] Nguyen Van Tuan and Koichiro Ochimizu: Extracting Collaboration Classes from Java Source Code. Master Thesis, Japan Advanced Institute of Science and Technology (2010)

[3] Lee, M., Han, D., Shim, J.: Set-based access conflicts analysis of concurrent workflow definition. In: Proceedings of Third International Symposium on Cooperative Database Systems and Applications, pp. 189–196. Beijing, China (2001)

[4] Li, H., Yang, Y., and Chen, T. Y.: Resource constraints analysis of workflow specifications. J. Syst. Softw. 73, 2, pp. 271–285 (2004)

[5] 5. Sadiq, S., Orlowska M., Sadiq W. and Foulger C.: Data flow and validation in workflow modeling. In: Proceedings of 15th Australasian Database Conference. LI, H. pp. 207–214 (2004)

[6] Kikuchi S., Tsuchiya S., Adachi M., and Katsuyama T.: Constraint Verification for Concurrent System Management Workflows Sharing Resources. In: Third International Conference on Autonomic and Autonomous Systems (2007)

[7] Trcka N., van der Aalst W.M.P., and Sidorova N.: Analyzing Control-Flow and Data-Flow in Workfow Processes in a Unified Way. Technical Report CS 08/31, Eindhoven University of Technology (2008)

[8] Trcka N., van der Aalst W.M.P., and Sidorova N.: Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows. In: 21st International Conference on Advanced Information Systems (CAiSE'09). LNCS, vol. 5565, pp. 425–439. Springer-Verlag Berlin Heidelberg (2009)

[9] Zhong, J. and Song, B.: Verification of resource constraints for concurrent workflows. In: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 253–261 (2005)

[10] Wil van der Aalst, Kees Max van Hee: Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, MA (2004)

[11] Zeng, Q., Wang, H. and Xu, D: Conflict detection and resolution for workflows constrained by resources and non-determined duration. Journal of Systems and Software 81(9), pp 1491–1504 (2008)

[12] Sundari M.H., Sen A.K., and Bagchi A.: Detecting Data Flow Errors in Workflows: A Systematic Graph Traversal Approach. In: 17th Workshop on Information Technology & Systems (WITS-2007). Montreal (2007)

[13] Fan S., Dou W.C., and Chen J.: Dual Workflow Nets: Mixed Control/Data-Flow Representation for Workflow Modeling and Verification. In: Advances in Web and Network Technologies, and Information Management (APWeb/WAIM 2007Workshops), LNCS, vol. 4537, pp 433–444. Springer-Verlag, Berlin (2007)

[14] Sun S.X., Zhao J.L., Nunamaker J.F., and Liu Sheng O.R.: Formulating the Data Flow Perspective for Business Process Management. Information Systems Research, 17(4), pp 374–391 (2006)

[15] Heinlein, C.: Workflow and process synchronization with interaction expressions and graphs. In: Proceedings of the 17th International Conference on Data Engineering (ICDE '01), pp. 243-252 (2001)

[16] Workflow Patterns, http://www.workflowpatterns.com

[17] Russell N., van der Aalst W.M.P., and ter Hofstede A.H.M.: Designing a Workflow System Using Coloured Petri Nets. Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) III, 5800, pp 1–24 (2009)

[18] Awad, A., Decker, G. and Lohmann, N.: Diagnosing and Repairing Data Anomalies in Process Models. In: 5th International Workshop on Business Process Design. LNBIP, pp 1–24. Springer, Heidelberg (2009)

[19] Workflow Petri Net Designer, http://193.196.7.195:8080/woped

[20] PMBOK Guide Fourth Edition. Project Management Institute (2008)

[21] Elmasri, R. and Navathe, S. B.: Fundamentals of database systems, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA (1989)

[22] Adaptable Model Versioning, http://modelversioning.org/

[23] Workflow Management Coalition, Workflow Security Considerations - White Paper Document Number WFMC-TC-1019, Document Status - Issue 1.0 (1998)

[24] Revision Control, http://en.wikipedia.org/wiki/Revision_control

[25] Computer Tool for Colored Petri Nets, http://wiki.daimi.au.dk/cpntools/_home.wiki

[26] Yet Another Smart Process Editor, http://www.yasper.org/