DATAFLOW ANALYSIS AND WORKFLOW DESIGN IN BUSINESS PROCESS MANAGEMENT

by

Sherry Xiaoyun Sun

A Dissertation Submitted to the Faculty of the
COMMITTEE ON BUSINESS ADMINISTRATION
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
WITH A MAJOR IN MANAGEMENT
In the Graduate College
THE UNIVERSITY OF ARIZONA

2007

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Sherry Xiaoyun Sun

entitled Dataflow Analysis and Workflow Design in Business Process Management

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

_____ Date: 4/19/2007
J. Leon Zhao

_____ Date: 4/19/2007
Jay F. Nunamaker, Jr.

_____ Date: 4/19/2007
Daniel Zeng

_____ Date: 4/19/2007
Martin Frické

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: 4/19/2007
Dissertation Directors: J. Leon Zhao

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ____Sherry Xiaoyun Sun_____

ACKNOWLEDGEMENT

I am greatly indebted to my dissertation advisor, Professor J. Leon Zhao, who has truly been a mentor to me. It is he who has firstly introduced me to the field of business process management and who has continually encouraged me to achieve the best I can in my academic life. Without his inspiration, guidance, support, and professional advice, I would be nowhere close to completing this work. He has been a role model for me in being a rigorous and dedicated scholar. I am truly grateful to his tremendous help with my academic growth and career development.

I would like to thank my dissertation committee members, Professor Jay F. Nunamaker, Professor Daniel Zeng, and Professor Martin Frické, for their invaluable suggestions, stimulating discussions, and insightful advices, which have been guiding me along my academic journey.

I would like to extend special thanks to Professor Olivia Sheng at the University of Utah, who never stops supporting, helping, and encouraging me ever since I chose this career path. Special thanks are also expressed to our department head, Professor Mohan Tanniru, for his continuous support during my doctoral study and to Dr Victoria Stefani from the Writing Skills Improvement Program for her helpful comments on revising this dissertation. I am also grateful to all the faculty members in the MIS department for providing such an open and resourceful research environment.

Last but not least, I thank my friends and fellow colleagues for the wonderful time we have spent together. In particular, I would like to thank Surendra Sarnikar, Limin Zhang, Yan An, Jason Li, Jennifer Xu, Fang Chen, Yiwen Zhang, Jian Ma, Xin Li, Manlu Liu, Saiwu Lin, Ling Zhu, Huihui Zhang, Mei Lu, Liangdong Huang, Bu Fang, Zhongxiang Xia, and Yong Jiang for their friendship and companionship since I joined the MIS department.

DEDICATION

This dissertation is dedicated

to my husband Yongdan Hu,

for his love, understanding, support, encouragement, and patience over all these years,

to my father, Li Sun, my mother Rulan Feng,

and my brothers, Xiaodong Sun and Xiaoguang Sun,

for their endless love and unconditional support.

TABLE OF CONTENTS

TABLE OF CONTENTS -- *Continued*

TABLE OF CONTENTS -- *Continued*

TABLE OF CONTENTS -- *Continued*

LIST OF FIGURES

LIST OF FIGURES -- *Continued*

LIST OF FIGURES -- *Continued*

LIST OF TABLES

LIST OF TABLES -- *Continued*

ABSTRACT

Workflow technology has become a standard solution for managing increasingly complex business processes. Successful business process management depends on effective workflow modeling, which has been limited mainly to modeling the control and coordination of activities, i.e. the control flow perspective. However, given a workflow specification that is flawless from the control flow perspective, errors can still occur due to incorrect dataflow specification, which is referred to as dataflow anomalies.

Currently, there are no sufficient formalisms for discovering and preventing dataflow anomalies in a workflow specification. Therefore, the goal of this dissertation is to develop formal methods for automatically detecting dataflow anomalies from a given workflow model and a rigorous approach for workflow design, which can help avoid dataflow anomalies during the design stage.

In this dissertation, we first propose a formal approach for dataflow verification, which can detect dataflow anomalies such as missing data, redundant data, and potential data conflicts. In addition, we propose to use the dataflow matrix, a two-dimension table showing the operations each activity has on each data item, as a way to specify dataflow in workflows. We believe that our dataflow verification framework has added more analytical rigor to business process management by enabling systematic elimination of dataflow errors.

We then propose a formal dependency-analysis-based approach for workflow design. A new concept called “activity relations” and a matrix-based analytical procedure are developed to enable the derivation of workflow models in a precise and rigorous manner.

Moreover, we decouple the correctness issue from the efficiency issue as a way to reduce the complexity of workflow design and apply the concept of inline blocks to further simplify the procedure. These novel techniques make it easier to handle complex and unstructured workflow models, including overlapping patterns.

In addition to proving the core theorems underlying the formal approaches and illustrating the validity of our approaches by applying them to real world cases, we provide detailed algorithms and system architectures as a roadmap for the implementation of dataflow verification and workflow design procedures.

Keywords: workflow modeling, dataflow specification, dataflow anomalies, dataflow verification, dependency analysis, process data diagram, workflow design, activity relations, business process automation

1 INTRODUCTION

Business processes are considered invaluable organizational assets, and the emerging "business process revolution" offers companies an opportunity to innovate in the way they do business (Smith and Fingar, 2003). As a result, corporations are confronted with the challenges of constantly increasing the productivity and efficiency of their business processes. A business process is defined as "the specific ordering of work activities across time and place, with a beginning, an end, and clearly identified input and output" (Davenport, 1993). Organizations implement business processes in order to produce value for customers (Earl, Sampler, and Short, 1995). Typically, a business process involves people from different functional units in the same organization and may go across organizational boundaries for reasons of business partnership, considerably increasing the complexity of managing the process (Stohr and Zhao, 2001).

As the information technology for business process automation, workflow systems have become a standard solution for managing complex processes in business domains such as supply chain management, customer relationship management, and knowledge management (Abecker et al., 2000; Stohr and Zhao, 2001; Kumar and Zhao, 2002; Panzarasa et al., 2002; Sarnikar, Zhao, and Kumar, 2004). Successful business process management depends on effective workflow design, modeling and analysis. Workflow models can be used to represent a business process from five perspectives: functional, behavioral, informational, operational, and organizational (Curtis, Kellner, and Over, 1992; Stohr and Zhao, 2001). The functional perspective describes what tasks a workflow

performs. The behavioral perspective specifies the conditions for tasks to be executed. The information perspective defines what data are consumed and produced with respect to each activity in a business process. The operational perspective specifies what tools and applications are used to execute a particular task. The organizational perspective describes the relationships among personnel that are qualified to perform various job functions

Current workflow modeling paradigms mainly focus on activity sequencing and coordination, including Petri nets (Aalst, 1998; Aalst and Hofstede, 2000) and activity-based workflow modeling (Bi and Zhao, 2004a and 2004b; Georgakopoulos, Hornick and Sheth, 1995). However, many business processes, such as insurance claims and loan applications, involve creation of intermediate data that is critical for proper process execution. The dataflow perspective is important in workflow management because relationships among data elements may drive the operational constraints that control activity sequencing (Kwan and Balasubramanian, 1997 and 1998). For example, in an "auto insurance claim" workflow, the estimated repair cost is required for claim authorization. Therefore, the activity *vehicle inspection,* which produces an output of "estimated repair cost", must precede the activity *claim authorization,* which uses "estimated repair cost" as input. If *claim authorization* occurs before *vehicle inspection*, a dataflow error would occur. Obviously, this type of error can only be detected and prevented by incorporating dataflow analysis into workflow modeling and design.

Presently, workflow management systems enable the discovery of dataflow errors only through simulation, which is inefficient and inaccurate. Moreover, the traditional

approach to workflow design, referred to as "participative approach" (Herrmann and Walter 1998), is pragmatic and sufficient for documenting the business requirements about the workflow but it does not offer any formalism for generating the workflow model in a rigorous manner. Due to the lack of the formalisms for dataflow analysis and workflow design, it is difficult to avoid dataflow error when workflow models are created. A workflow model containing dataflow errors can cause unexpected process interruptions, resulting in high costs to debug and fix at run time. In order to fill the void in dataflow analysis and workflow design, this dissertation aims to develop a formal method to enable automatic detection of dataflow anomalies from a given workflow model and a rigorous design approach, which can help to prevent dataflow anomalies during the design stage.

In order to achieve the first objective, developing a complete framework for determining dataflow errors in workflow management, we first formally define three basic types of dataflow errors: missing data, redundant data, and conflicting data. We then propose a method for specifying dataflow in a workflow model at a very detailed level. Third and most important, we provide an analytical approach for detecting and eliminating the three types of dataflow errors. Our new approach formally establishes the correctness criteria for dataflow modeling. As a theoretical foundation for dataflow verification, these criteria enable systematic and automatic elimination of dataflow errors.

In order to achieve the second objective, we propose an analytical method of workflow design based on data dependency analysis. The basic idea is to decide how activities should be sequenced in a workflow by examining the transformation from input

data to output data via a sequence of activities. Our approach is innovative in several respects: First, our workflow design approach is based on a new concept called "activity relations" to represent potential activity execution steps. Second, we develop an analytical procedure that enables the derivation of workflow models from data dependencies. Third, we simplify the procedure by decoupling the issue of model correctness and the issue of workflow efficiency. These novel techniques together make it possible to handle complex workflow models including unstructured workflow and overlapping patterns.

To the best of our knowledge, this dissertation is the first complete framework to analyze dataflow errors and to incorporate dependency analysis into workflow design through formal procedures, thus making the dataflow analysis and workflow design process more rigorous. This may have significant economic implications for companies that have hundreds of complex business processes by reducing the costs of fixing workflow errors at both design time and runtime.

This dissertation is structured as follows. In Chapter 2, we review the literature, which forms the foundation of this dissertation. In Chapter 3, we propose to use a dataflow matrix and a process data diagram to specify the details of dataflow in workflow management and present a classification of dataflow anomalies using a property loan approval process as an illustration. In Chapter 4, we propose a dependency based approach for dataflow verification, which can help detect dataflow anomalies under different scenarios. In Chapter 5, we present a dependency based approach to workflow design, which takes consideration of dataflow issues. In Chapter 6, we present the

methods, tools, algorithms, and system architectures for implanting the dependency based design approach. Chapter 7 concludes this dissertation by summarizing our contributions and outlining future research directions.

2 LITERATURE REVIEW

In this chapter, we review the related literature that forms the foundation of this work. We categorize the relevant literature into five areas: (1) workflow modeling and verification, (2) data usage analysis, (3) formal program verification, (4) workflow design, and (5) process mining.

2.1 Workflow Modeling and Verification

Workflow modeling and workflow verification are two closely related areas. Workflow modeling focuses on creating models that describe business processes. A workflow model often consists of elements such as roles, actors, tools/applications, activities and processes, rules, and data/documents (Kumar and Zhao, 1999; Stohr and Zhao, 2001). Workflow verification examines a workflow model to decide whether the model contains any syntactic errors such as such as deadlock, activities without termination or activation, and infinite cycles. Currently, most workflow modeling and verification paradigms mainly focus on activity sequencing and coordination, i.e., the control flow perspective, such as Petri nets (Aalst, 1998; Aalst, Hofstede, 2000), activity-based workflow modeling (Georgakopoulos, Hornick and Sheth, 1995; Bi and Zhao, 2003a, 2003b, 2004a, and 2004b), Object Coordination Nets (Wirtz, Weske, and Giese 2001), and rule-based process modeling (Lee, Kim, and Park, 1999).

A Petri net uses four components to represent a workflow model: transitions representing activities or tasks, places representing states, tokens representing cases, and directed arcs connecting transitions and places. Petri nets have previously been used for

modeling and verification of production systems (Agarwal and Tanniru 1992a and b). As a formalism for workflow modeling, Petri nets provide rigorous methods for workflow verification (Aalst, 1998; Aalst and Hofstede, 2000). Syntactic errors in control flow can be discovered through Petri net modeling and analysis (Aalst and Hofstede, 2000).

Activity-based modeling is another paradigm in workflow modeling. One of the mostly adopted activity-based modeling methods is the UML activity diagram (OMG 2003). However, the UML activity diagram is not built on any formalism and therefore provides little analytical capability. Recently, the theories of directed graph and propositional logic have been incorporated into activity-based modeling, which enables workflow verifications in activity-based modeling paradigm (Bi and Zhao, 2003a, 2003b, 2004a, and 2004b; Zhao and Bi, 2003).

The Object Coordination Nets extends Petri nets through an incorporation of UML structure diagram. As an enhancement to both Petri nets and object-oriented approach, the Object Coordination Nets help to bridge the gap between the design of a workflow model and the implementation of workflow software (Wirtz, Weske, and Giese 2001).

In addition, some other models focus on modeling business rules needed to control activity scheduling and role/actor mapping. For example, as a modeling method focusing on business rules, the Knowledge-based Workflow Model (Lee, Kim, and Park, 1999) emphasizes on representing a workflow model as a set of business rules. A change propagation mechanism based on dependency management is provided to accommodate frequent changes in organizational structures and business rules.

In order to verify complex workflow structures, such as overlapping patterns and

cyclic workflows, a matrix based approach has been introduced (Choi and Zhao 2002, 2003). It has been shown that the matrix based workflow verification can identify deadlocks and lack of synchronization in cyclic workflows. Moreover, inline blocks are applied to simplify the verification process in this approach.

The above models do not emphasize the dataflow perspective, suggesting this is an open area of research. Analyzing both data and activities in one single model adds an extra level of complexity, as opposed to only focusing on activities. Therefore, for the purpose of simplicity, these workflow models focus on control flow, and data requirements are either simplified through abstraction in Petri Net modeling (Aalst and Hofstede, 2000) or not incorporated at all in activity based modeling (Bi and Zhao, 2004a and 2004b; Zhao and Bi, 2003). However, the dataflow perspective, also known as data usage analysis, is important because activities cannot be executed properly without sufficient information (Basu and Kumar, 2002). As a critical component of business process management, the dataflow perspective complements the control flow perspective and the organizational perspective. Next, we discuss some research areas that are relevant to data usage analysis.

2.2 Data Usage Analysis

Data Flow Diagram (DFD) is widely used in system analysis and design (Yourdon and Constantine, 1979). However, DFD is not sufficient to support formal analysis of dataflow, mainly because it lacks an underlying theoretical foundation. As a well-known modeling methodology, Data Flow Diagram (DFD) is used to specify the flow of data from external entities, via various data processing steps, into logical data storages

(Yourdon and Constantine, 1979). However, a DFD only captures activities that are directly related to data processing and those activities that do not involve any data processing are omitted from the DFD. More importantly, the DFD literature offers no analytical tools for examining the correctness of dataflow models. Our dataflow verification framework focuses on dataflow analysis in the context of workflow management and the formal methodology we propose enables automatic detection of dataflow anomalies. Therefore, our work goes significantly beyond the DFD framework.

To support data requirement analysis in workflow management, several informal and formal modeling methods have been developed. For instance, as informal approaches, the data model proposed in the ConTracts project can deal with the long duration of transactions in workflow systems (Reuter and Schwenkreis, 1995), and the object-oriented database used by Kappel et al. (1995) is primarily oriented to coping with frequently changing requirements in an organization.

In contrast, some recent modeling methods provide more formal analysis techniques such as the State-Entity-Activity-Model (SEAM), which is a data model recently developed for workflow management. A unique feature of SEAM is that activities and data objects are integrated in a single view (Bajaj and Ram, 2002). SEAM enables construction of workflow applications using relational database management systems. However, SEAM focuses on database modeling rather than dataflow analysis.

Metagraphs have been proposed as a workflow-modeling formalism (Basu and Blanning, 2000) that combines the notation of directed graphs and hypergraphs (Basu and Blanning, 1994a and b, 1997, and 1998). Metagraphs can provide insight on how sets of

elements, such as activities, data, and resources, are interrelated to each other since metagraphs can be used to analyze the connectivity between sets of elements. As such, metagraphs enable analysis of the data transformation from initial input to final output through a sequence of activities and therefore metagraphs can be considered as a formal framework for data dependency analysis in workflow management. However, the literature on metagraphs has not emphasized the issue of detecting dataflow errors although as a language, metagraphs may be used to verify dataflow with appropriate extension.

At the conceptual level, a dataflow model is considered correct if it is free from a variety of important errors. Sadiq et al. (2004) have started investigating the different problems of dataflow validation and identified the essential requirements of dataflow modeling in workflow management. They define seven types of data anomalies: redundant data, lost data, missing data, mismatched data, inconsistent data, misdirected data, and insufficient data. However, no concrete solutions to the dataflow validation problems have been reported.

2.3 Formal Program Verification

In software engineering, a program is considered correct when it executes the functions required by its specification (Berg, Boebert, Franta, Moher, 1982; Mili, 1985; Guaspari, Marceau, and Polak, 1990). Software verification typically involves labor intensive testing and simulation (Wallace and Fujii, 1989; Krauskopf and Rash, 1990). Mathematical proof can be used to establish the consistency between a program and a particular specification, which is known as formal program verification (Berg, Boebert,

Franta, Moher, 1982; Mili, 1985). There are several general steps to perform formal program verification. First, mathematical meaning is defined for a particular programming language. In order to verify a particular program, the program is then annotated with formal statements of pre and post conditions for the program. The last step is to prove mathematically if the pre condition is true before the program body is executed, then the post condition must be true after the execution is complete.

Formal program verification can help to determine whether a program meets a specification. However, formal program verification differs significantly from dataflow verification. Instead of verifying consistency between a specification and the related execution results, as in formal program verification, dataflow verification focuses on identifying dataflow errors by means of a set of well-defined correctness criteria. In addition, dataflow analysis and control flow analysis are two intertwined aspects of workflow analysis, and dataflow analysis can be used to validate the correctness of activity sequencing (Sun and Zhao, 2004; Sun et al., 2006). This unique feature of dataflow analysis distinguishes it further from formal program verification.

2.4 Workflow Design

While a significant amount of research in the workflow area has focused on modeling, verification, and analysis issues (Aalst and Hee 2002; Bi and Zhao 2004a and 2004b), formal approach to workflow design is scant in the literature (Stohr and Zhao 2001).

The traditional workflow design approach, referred to as participative approach (Herrmann and Walter 1998), adopts the principle of joint application design (JAD), which describes a variety of methods for conducting workshops where users and

technical developers work together to define requirements of information systems (Davidson 1999). The participative approach is useful for collecting the needed information for workflow design, such as business documents, relevant business activities, business rules, and data elements needed to execute the activities. However, the participative approach does not provide any formalism for generating a workflow model with the information that has been collected. The challenge, then, is to generate correct workflow models more systematically based on the information collected.

The product-based workflow design proposed by Reijers, Limam and Aalst (2003) uses product specifications to establish the set of activities required in a workflow for producing a product or offering a service. Moreover, workflow models can be generated from the relationship among data elements derived from product specifications through breadth first search and depth first search. Further, cost and flow time are considered as criteria for selection of workflow models.

The policy-driven workflow mapping method (Wang and Zhao 2006) focuses on deriving a workflow model from documented business policies. In this method, information on the set of activities in a business process and their data input and output is extracted from narrative business policies and used to help create a workflow model.

The review of the workflow design literature shows a lack of formal workflow design methodologies with detailed design guidelines. As such, formal procedures are still needed on how to determine activity sequences and how to use parallelism and conditional routing on the basis of analyzing relationships among data.

Research in workflow design can also benefit from the stream of work in business

process redesign. Business process redesign deals with both technical issues and socio-cultural issues when restructuring a business process for improvement in cost, quality, speed, and service. Most work in this area focuses on providing guidelines for optimization along the dimensions of cost, quality, and time (Reijers and Mansar, 2005). For instance, the analytical model proposed by Aalst (2001) focuses on minimizing time and maximizing resource utilization through sequential and parallel routing of tasks in a particular type of processes where each activity can produce only two possible results. As suggested by Reijers et al. (2003), the multiple optimization criteria found in business process redesign can be used in workflow model evaluation.

2.5 Process Mining

Process mining, a recently developed research area which is also known as workflow mining, aims to apply data mining techniques to workflow management and to derive useful information for workflow diagnosis, model improvement, and performance analysis from event log data recorded in various information systems.

Most research in this area focus on deriving a workflow model representing the work practice observed from even log. Aalst et al. (2004) propose a Petri Net based algorithm for discovering workflow models. They define four types of log-based ordering relations to describe the causal dependency between activities. An algorithm is proposed for constructing workflow nets models, which are essentially a special of Petri Net models. This algorithm can also be used to reconstruct workflow models from timed event log (Aalst and Dongen 2002). In order to handle noise and incomplete logs, a heuristic based method is introduced (Aalst et al., 2003a), which uses dependency and frequency tables

to remove noise. Furthermore, heuristic rules are developed to determine log-based ordering relations in an event log out of dependency and frequency tables.

If the same activity can appear multiple times in a workflow model, then the workflow model can be constructed using an inductive approach with two steps. In the induction step, a stochastic task graph is created from event log data (Herbst, 1999; Herbst and Karagiannis, 2000). In the second step, the stochastic task graph is transformed to a workflow model in the ADONIS modeling language. This approach is further characterized by an ad hoc mechanism for identifying splits and joins.

Methods of process mining are also developed for performance analysis. Greco et al. (2005) propose algorithms for identifying the workflow structures executed frequently by systems, which can help system administrator to determine configurations leading to desirable results. Process mining has been shown also useful in analyzing audit trails to support detection of security violation (Aalst and Medeiros, 2005). However, the result is only preliminary.

Process mining and workflow design are closely related in that both provide methods to construct workflow models. Workflow design is significantly different from process mining in the following aspects. First, workflow does not assume the existence of a well-structured workflow model while this assumption holds for process mining. Moreover, in workflow design, information about a business process, which extracted from meeting, interviews, and existing documents, may not contain clear repetitive patterns. In process mining, repetitive patterns of activity execution in event log and trace file help to reconstruct the workflow models executed by the system. In addition, workflow design

usually occurs at an early stage of workflow development lifecycle when workflow developers start to collect the requirements and plan implementation. Process mining can occur at a later stage of workflow development lifecycle when workflow system is already in use and the system is diagnosed for performance improvement.

In this dissertation, we propose a new concept called "activity relations" as a foundation for a formal workflow design method. The concept of activity relations is inspired by the concept of "log-based ordering relations" used in process mining (Aalst et al. 2004). However, the concept of "activity relations" is different from "log-based ordering relations" in that the former describes the structures of a workflow model and the latter describes the event sequences found in an event log.

3. DATAFLOW SPECIFICATION AND DATAFLOW ANOMALIES

This chapter presents two methods, dataflow matrix and process data diagram, which enable detailed dataflow specification in workflow management. Dataflow matrix is a two dimension table showing the operations each activity has on every piece of data item in a workflow. Process data diagram is an extension of UML activity diagram, which shows not only the control flow of a process but also the input and output data for each activity involved in the process. Furthermore, this chapter defines and classifies dataflow errors, which is the foundation for dataflow verification. All the key concepts are illustrated with a real-world example, the property loan approval process.

3.1 A Business Process Example

This section introduces a property loan approval process shown as a UML activity diagram[1] in Figure 1. This example is used to illustrate the concepts in this chapter and the next chapter. Below, we examine the key steps of this process, involving business decisions, data processing and activity routing:

1. An application is received.
2. The completeness of the application is verified.
3. The process is routed based on whether the application is complete or not.

---

[1] For simplicity, we do not show the swimlanes in the UML activity diagram.

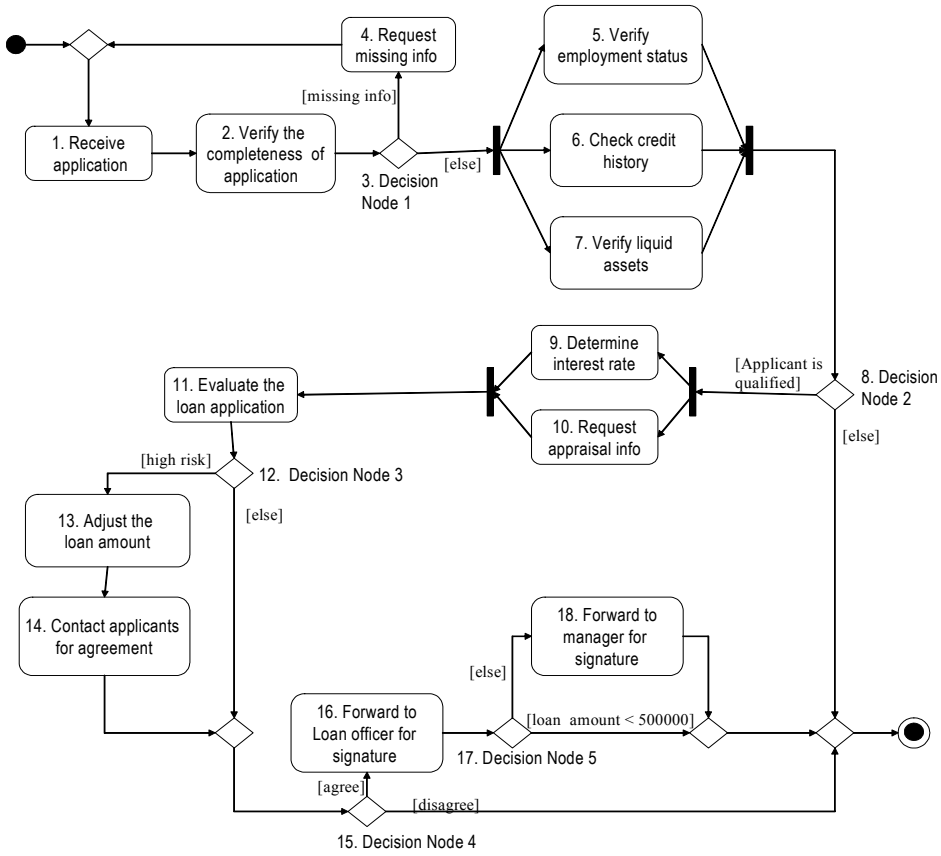


Figure 1. Property Loan Approval Process

4. If the application is not complete, the missing information is requested.
5. To determine the applicant's qualifications, the financial services company first verifies the applicant's employment status.
6. To qualify the applicant, the financial services company checks the applicant's credit history.
7. The financial services company also checks the applicant's liquid assets.
8. The process is routed according to the applicant's qualifications. If the applicant is not qualified, the process terminates.

Data items		
d_1: Applicant name	d_8: Account balance	d_{15}: Risk
d_2: Loan amount	d_9: Account balance verified	d_{16}: Amount adjusted
d_3: Annual incoming	d_{10}: Applicant qualified	d_{17}: Agreed by applicants
d_4: Application complete	d_{11}: Interest rate	d_{18}: Property insured
d_5: Application summary	d_{12}: Property address	d_{19}: Signed by applicants
d_6: Employment status verified	d_{13}: Current Owner of Property	d_{20}: Signed by loan officer
d_7: Credit score	d_{14}: Appraised value of the property	d_{21}: Signed by manager
Activities		
v_1: Receive application	v_7: Verify liquid assets	v_{15}: Decision node 4
v_2: Verify the completeness of application	v_8: Decision node 2	v_{16}: Forward to loan officer for signature
v_3: Decision node 1	v_9: Determine interest rate	v_{17}: Decision node 5
v_4: Request missing info	v_{10}: Request appraisal info	v_{18}: Forward to manager for signature
v_5: Verify employment status	v_{11}: Evaluate the loan application	s: Start node
v_6: Check credit history	v_{12}: Decision node 3	e: End node
	v_{13}: Adjust the loan amount	
	v_{14}: Contact applicants for agreement	
Operations		
r: Read	w: Write	

Table 1. Symbols Used in the Property Loan Approval Process

9. If the applicant is qualified, the current interest rate is locked in for a certain period of time, as requested by the applicant. The interest rate is determined based on the amount of money the applicant is eligible to borrow.
10. The financial services company requests the appraisal information.
11. The loan application is evaluated. Given the applicant's credit score, the appraised value of the property, the loan amount, and the level of risk associated with the loan are calculated.
12. The process is routed according to the risk level.
13. If the risk is higher than the applicable threshold, the loan amount must be adjusted.
14. The financial services company contacts the applicant to discuss the necessary adjustment and other conditions, such as property insurance options.
15. The process is routed according to whether the applicant agrees with the necessary

adjustment and other conditions. If the applicant disagrees with the conditions and adjustment, the process ends.

16. If the applicant agrees with everything, the application is forwarded to the loan officer for signature after the applicant signs the application.
17. The process is routed based on the loan amount.
18. When the loan amount is more than $500,000, the manager's signature is required.

Table 1 contains the symbols for the activities and data items in this process.

3.2 Dataflow Specification

In this section, we develop a method for specifying dataflow in a workflow system.

3.2.1 Dataflow Operations

In a workflow, each activity can perform different operations on a data item. We call these operations dataflow operations. Through these operations, data items are produced, accessed and modified. We classify the dataflow operations in the property loan approval process into *initializing*, *approval*, *updating*, *referral*, and *verification*, according to the semantic meanings of dataflow operations in the business process environment. From the database implementation point of view, all dataflow operations can be considered as either *read* or *write* operations, regardless of their semantic meaning.

When analyzing a dataflow, we need to know the input and output data for each activity. Categorizing dataflow operations into *read* / *write* operations helps identify the input and output data for each activity. It is intuitive that when activity *v* reads data item *d, d* is the input data for *v*, and when *v* performs a write operation on *d*, *d* is the output data

from *v*. Moreover, there are two types of *write* operations, creating the initial value, also called initialization, and overwriting the existing value. It is possible for a data item to be overwritten after it is initialized. However, in the interest of simplicity, the current focuses only on the initial *write* operation because initialization is critical for discovering dataflow anomalies.

3.2.2 Dataflow Matrices

To specify dataflow in workflow applications, we introduce the concept of dataflow matrix, a two-dimensional table that records the dataflow operations each activity performs on various data items in a workflow. More formally, dataflow matrix can be defined as follows:

Definition 1 (Dataflow Matrix) Given the total number of activities n and the total number of data items z in a workflow W, dataflow matrix M is a n by z table where the element (i, j) shows the operation that activity v_j performs on the data item d_i. Activity v_j is a logical step of work in a business process that can be scheduled by workflow engine during process enactment. Data d_j is an atomic data element that can be stored in a database.

Table 2 shows the dataflow matrix for the property loan approval process. Note that each cell contains no more than one operation for the purpose of dataflow analysis. It is also worth noting that Table 2 also includes a special type of control activity nodes called decision nodes. A decision node uses a set of data items to decide how to route a workflow instance by choosing an activity for execution from a set of optional activities. For instance, decision node 1 (i.e., activity v_3) uses d_4, *application complete*, as input to

Data Objects	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}
d_1: Applicant name	*w*	*r*			*r*	*r*	*r*		*r*	*r*	*r*		*r*	*r*		*r*		*r*
d_2: Loan amount	*w*	*r*							*r*		*r*			*r*		*r*	*r*	*r*
d_3: Annual incoming	*w*	*r*			*r*				*r*									
d_4: Application complete		*w*	*r*	*r*														
d_5: Application summary	*w*																	*r*
d_6: Employment status verified					*w*						*r*							
d_7: Credit score						*w*			*r*		*r*							
d_8: Account balance	*w*	*r*					*r*				*r*							
d_9: Account balance verified							*w*				*r*							
D_{10}: Applicant qualified					*w*	*w*	*w*	*r*			*r*			*r*		*r*		*r*
d_{11}: Interest rate									*w*		*r*		*r*			*r*		*r*
d_{12}: Property address	*w*	*r*								*r*								
d_{13}: Current owner of property										*w*								
d_{14}: Appraised value of the property									*r*	*w*	*r*		*r*					
d_{15}: Risk											*w*	*r*	*r*			*r*	*r*	*r*
d_{16}: Amount adjusted									*r*				*w*	*r*		*r*	*r*	*r*
d_{17}: Agreed by applicants														*w*	*r*	*r*		*r*
d_{18}: Property insured														*r*				
d_{19}: Signed by applicants														*w*	*r*	*r*		*r*
d_{20}: Signed by loan officer																*w*		
d_{21}: Signed by manager																		*w*

Table 2. Dataflow Matrix for the Property Loan Approval Process

decide the next activity to be executed. If d_4="*No*", activity v_4, *request missing info*, will be executed. Otherwise, activity v_5, *verify employment status*, activity v_6, *check credit history*, and activity v_7, *verify liquid asset*, will be executed. Decision nodes should be included in the dataflow matrix because they require input data. A dataflow matrix can contain activities that do not generate output data. If the primary function of an activity is

routing control, the activity may not generate an output data. With a dataflow matrix, we can easily find out how each data item is processed in a workflow.

3.2.3 Integration of Dataflow in the Workflow Model

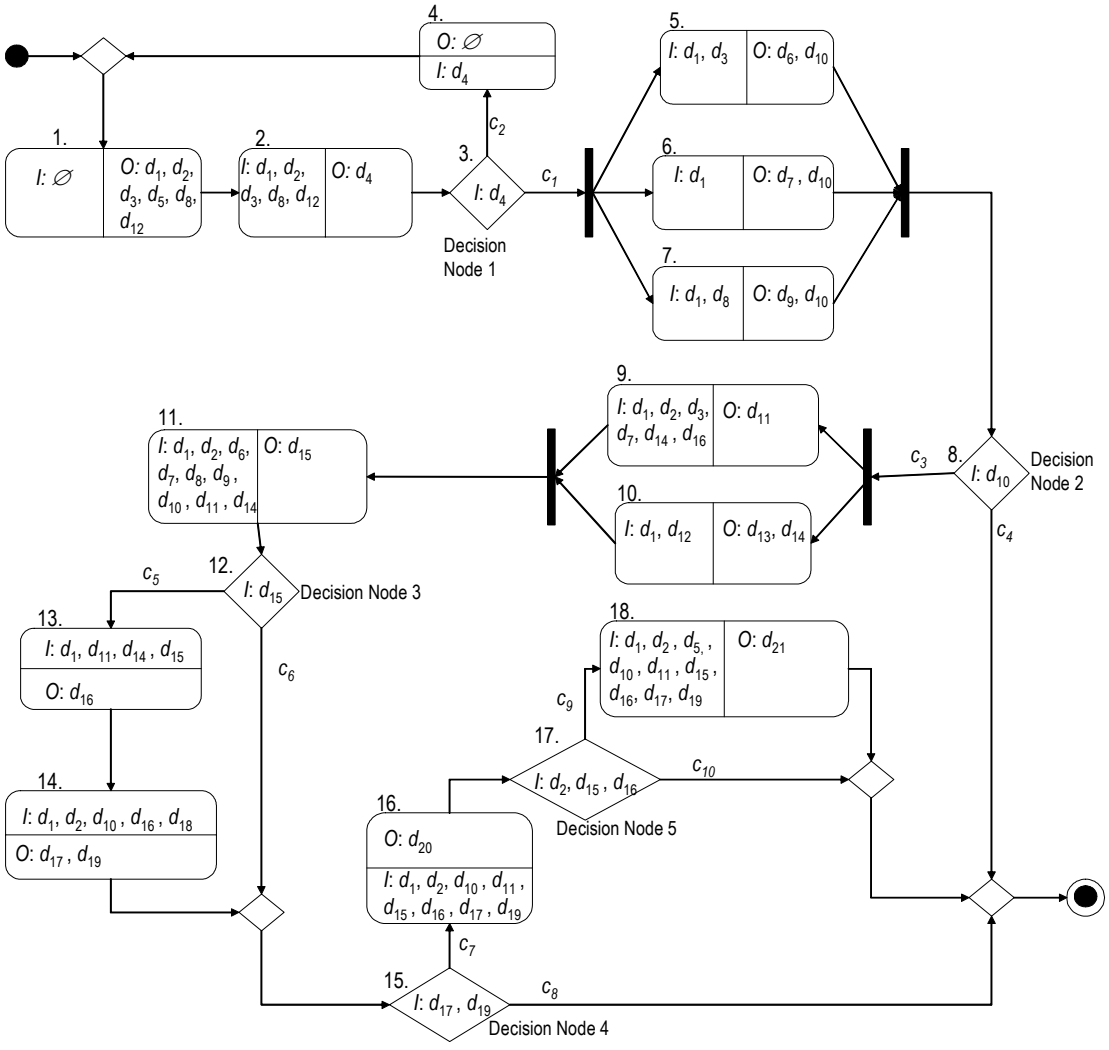


Figure 2. Process Data Diagram for the Property Loan Approval Process

Dataflow information can be integrated into the control flow model as shown in Figure 2, where the input and output data are shown in the activity diagram. In a UML activity

diagram, object flows can be used to describe the input and output for an activity. Each object is connected to one or more activities, indicating an action-object relationship. However, the object flow is insufficient for modeling dataflow in workflow management because it provides no details on how different data items associated with one object are processed differently in a workflow. Therefore, we extend the UML activity diagram to show how each individual data item is produced and used in a workflow.

Figure 2 provides a graphical representation for the dataflow in the loan approval process. In Figure 2, each regular activity is associated with two sets of data, the input data set following the symbol *I* and the output data set following the symbol *O*. Each decision node uses a set of input data to decide the route for a workflow instance. If an activity *v* performs a *read* operation on a data item *d* in the dataflow matrix, *d* is shown as an input data of *v* in Figure 2. If an activity *v* performs a *write* operation on a data item *d* in the dataflow matrix, *d* is shown as an output data of *v* in Figure 2. The detailed dataflow specification shown in Figure 2 is a prerequisite for analyzing the correctness of a dataflow. The symbols c_i indicate routing conditions that will be detailed in Chapter 4.

3.3 Dataflow Anomalies

In a workflow, each activity contributes to the production of final output data by generating either some intermediate output data or a subset of the final output data. Meanwhile, each activity may need a set of data as input, which may be produced by other activities in the same workflow or by sources external to the workflow. If the dataflow is not specified correctly in a workflow system, errors and conflicts can occur, referred to as dataflow anomalies (Sun, Zhao, Sheng, 2004). Dataflow anomalies can be

classified into three types: missing data, redundant data, and conflicting data, as discussed next.

3.3.1 Missing Data

When a data item is accessed before it is initialized, a missing data anomaly occurs. Each of the following scenarios can cause missing data anomalies.

Scenario 1. (Absence of Initialization) A data item is never assigned an initial value; however, some activities use it as input or it is required as the final output of the workflow.

EXAMPLE 1. As Table 2 shows, data item d_{18}, *property insured,* has never been initialized, but activity v_{14} reads it as input data. This is a missing data anomaly.

Scenario 2. (Delayed Initialization) A data item is used by activity v as input, but it is initialized only by another activity that is executed after v is performed.

EXAMPLE 2. In the property loan approval process, activity v_9, *determine interest rate*, reads data item d_{16}, *amount adjusted*, which is initialized by activity v_{13}, *adjust the loan amount*. However, as shown in Figure 2, v_{13} is executed after v_9. Therefore, when v_9 is executed, the data item d_{16} has not been initialized, leading to another missing data anomaly.

Scenario 3. (Uncertain Availability) Given two parallel activities, v and u, v needs an input data item initialized by u. When v is executed, the data item may not have been initialized.

EXAMPLE 3. In the property loan approval process, to calculate the interest rate, activity v_9, *determine interest rate*, reads data item d_{14}, *appraised value of the property*,

which is produced by activity v_{10}, *request appraisal info* (shown in Table 2 and Figure 2). According to the control flow in Figure 2, v_9 and v_{10} are executed in parallel; therefore, when d_{14} is read by v_9, there is a chance that v_{10} has not initialized d_{14}, leading to another missing data anomaly in this workflow.

Scenario 4. (Improper Routing) Under certain workflow routing conditions, a data item is not initialized, although it is used by some activities as input.

EXAMPLE 4. In the property loan approval process, the output data d_{17}, *agreed by applicants*, and d_{19}, *signed by applicants*, are produced by activity v_{14}, *contact applicants for agreement*, only when the risk is high, as shown in Figure 2. However, even when the risk is low, d_{17} and d_{19} are also read as input by activities v_{15}, *decision node 4*, v_{16}, *forward to loan officer for signature*, and v_{18}, *forward to manager for signature*. Therefore, another missing data anomaly occurs.

3.3.2 Redundant Data

If an activity produces data items that do not contribute to the production of the final output data, then there is a redundant data anomaly. Redundant data causes inefficiency. The two following scenarios can cause redundant data anomalies.

Scenario 5. (Inevitable Redundancy) A data item is produced as an intermediate data output; however, no other activities need it as input data, and it is not part of the final output data.

EXAMPLE 5. In Figure 2, data item d_{13}, *current owner of property*, produced by activity v_{10}, *request appraisal info*, is neither used by any other activities in the process nor required as the final data output of the process. Hence, d_{13} is redundant and can be

eliminated.

Scenario 6. (Contingent Redundancy) A data item is produced as an intermediate data output. However, it is used only under some routing conditions. Under other routing conditions, that data is not used by any activities as input.

EXAMPLE 6. As shown in Figure 2, data item d_5, *application summary*, is only required by activity v_{18}, *forward to manager for signature*, if the loan amount is greater than $500,000. When the loan amount is less than $500,000, the application summary is still produced but is not needed by any activities.

3.3.3 Conflicting data

In a workflow instance, if there exist different versions of the same data item, conflicting data anomalies occur. When conflicting data anomalies occur, it is impossible to decide which version of the data item should be taken unless we have appropriate rules that define the way to draw a final conclusion. The following scenario can cause conflicting data anomalies.

Scenario 7. (Multiple Initializations) More than one activity attempts to initialize the same data item in one workflow instance.

EXAMPLE 7. In Figure 2, activities v_5, *verify employment status*, v_6, *check credit history*, and v_7, *verify liquid assets*, all make decisions on whether the applicant is qualified for the loan, i.e., they all produce data item d_{10} as output. When one activity qualifies the applicant and one activity disqualifies the applicant, we have trouble deciding if the applicant is qualified or not.

3.3.4 Discussion

We take a minimalist approach and define three basic dataflow anomalies -- missing data, redundant data, and conflicting data -- because the definition of these three types of dataflow anomalies are sufficient in analyzing the dataflow requirements at the conceptual level. We can show that the seven types of dataflow anomalies proposed by Sadiq et al. (2004) can either be represented by these three basic dataflow anomalies or are not a problem at the conceptual level, as follows:

- *Missing data* and *redundant data* are defined similarly. Missing data occurs when a data item needed by one activity is not available at the time of use, and redundant data occurs when an unnecessary data item is produced.
- *Lost data* occurs when an initialization is overwritten by another activity. It is caused by what we refer to as conflicting data.
- *Mismatched data* arises when the structure of an output data item is incompatible with the structure required by the activity that uses the data item as input. Mismatched data can be regarded as the occurrence of both redundant data and missing data.
- *Inconsistent data* happens when the initial data input for a workflow is updated externally while the workflow is still being executed, resulting in an inconsistent view of the data. Inconsistent data is not a problem at the conceptual level and should be dealt with at the operational level.
- *Misdirected data* occurs when the dataflow direction is not consistent with the control flow in a workflow model. Misdirected data is classified as missing data in our

framework.

- *Insufficient data* happens when no sufficient data are specified for a successful completion of a workflow. It is an issue due to ill-designed activities and can be categorized into missing data at the semantic level.

In summary, using the three types of dataflow anomalies instead of seven makes dataflow analysis more manageable and covers the key issues of ensuring dataflow integrity at the conceptual level.

3.4 Conclusions

Despite the existence of dataflow modeling tools in literature and practice, there is a lack of methods for specifying dataflow at detailed level in workflow management. The dataflow matrix and process data diagram proposed in this chapter fill this gap by making it easy to identify how each data item is processed in a workflow. Moreover, we define three basic types of dataflow anomalies, i.e. missing data, redundant data, and conflicting data, and describe various scenarios under which dataflow anomalies can occur. We also show that our classification of dataflow anomalies can capture the basic dataflow problems occurring at the conceptual level. Other types of conceptual-level dataflow anomalies proposed by Sadiq et al. (2004) can be converted into these three types of problems. The clear definition and classification of dataflow anomalies are the foundation for examining the correctness of a dataflow model.

4 ACTIVITY DEPENDENCY ANALYSIS FOR DATAFLOW VERIFICATION

In this chapter, we propose a dependency-based approach for dataflow analysis. Table 3 shows all the symbols used in this section. The example of property loan approval process introduced in Chapter 3 is used as illustrations throughout this chapter.

• c, c_i: workflow routing constraint • s: start activity • e: end activity • x, y, u, v, v_i, v_j, v_{temp}: any activity • V, V_{temp}: a set of activities • W: control flow model • M: dataflow matrix • E: the overall set of external data to W, $E = \cup E_i$ • E_i: the set of external data at activity v_i • O_0: the set of data items as final output from W • d, d_i: data item • I_{vi}, I_i: the set of data items as input for activity v_i • I^m_v: the mandatory input data set for activity v • I^c_v: the conditional input data set for activity v • O_v: the set of data items as output of activity v • C: workflow routing constraint set	• C_w: the routing constraint set for W • C^u_v, C^d_v, C^i_v: upstream, downstream, or irrelevant routing constraint sets for activity v • λ^u_v: mandatory data dependency for activity v, and v is any activity including s and e • λ^c_v: conditional data dependency for activity v, , and v is any activity including s and e • $\Rightarrow$: activity dependency • Δ^u: mandatory requisite set • Δ^c: conditional requisite set • Γ: instance set • Γ_w: all the instance sets of W • $\bigcup_{i=1}^{n} I_i$: the union of the input data sets of all the activities in W • $\bigcup_{i=1}^{n} O_i$: the union of the output data from all the activities in W

Table 3. Symbols Used in Dataflow Verification

4.1 Basic Concepts

Next, we define the concept of workflow routing constraint that specifies how a workflow instance is routed. The concept of routing constraint is similar to that of routing rules found in (Kumar and Zhao 1999). However, we use a simple predicate logic format

rather than following the event-role-object-condition-action format since the simpler format is sufficient for the purpose of dataflow analysis in this chapter.

Definition 2 (Decision Variable) A routing decision can be made based on a set of data items inputted to the decision node. Each of such data items involved in a routing decision is called a decision variable.

In the property loan approval process, decision node 1 uses the data item d_4, *application complete*, to route a workflow instance (Table 2). Therefore d_4 is a decision variable. The set of possible values for d_4 is ["*Yes*", "*No*"].

Definition 3 (Routing Constraint): A workflow routing constraint c is defined as a logic formula used to route a workflow instance where a set of decision variables is quantified.

A decision node can use a set of routing conditionsto route a workflow instance, with each routing constraint corresponding to one arc leaving from a decision node. For example, in the property loan approval process, when d_{17}= "*Yes*" and d_{19}="*Yes*", i.e., the loan conditions are agreed upon and signed by the applicants, decision node 4 routes the application to the loan officer for signature. When d_{17}= "*No*" and d_{19}="*No*", decision node 4 routes the application to the end of the workflow. Therefore, the routing constraint set used by decision node 4 is C={c_7=(d_{17}= "*Yes*" and d_{19}="*Yes*"), c_8=(d_{17}= "*No*" and d_{19}="*No*")}. Table 4 shows all the routing conditionsused in the property loan approval process, each of which can be mapped to an arc as previously illustrated in Figure 2.

Definition 4 (Routing Constraint Set for Workflow): Given a workflow W, all its routing conditions(possibly represented in clausal form) constitute the routing constraint set for W, denoted as C_w.

c_1=(d_4="Yes")	c_6=(d_{15}="Low")
c_2=(d_4="No")	c_7 =(d_{17}= "Yes" AND d_{19} = "Yes")
c_3=(d_{10}="Yes")	c_8=(d_{17}= "No" AND d_{19} = "No")
c_4=(d_{10}="No")	c_9=((d_{15}="High" AND d_{16} > 500,000)OR (d_{15}="Low" AND d_2>500,000))
c_5=(d_{15}="High")	c_{10} =((d_{15}="High" AND $d_{16}\leq$ 500,000)OR (d_{15}="Low" AND $d_2\leq$ 500,000))

Table 4. Routing Constraints in the Property Loan Approval Process

There are a total of seven decision variables in the property loan approval process (d_2, d_4, d_{10}, d_{15}, d_{16}, d_{17}, and d_{19}), collectively configuring a workflow instance. The routing constraint set for this workflow can be written as C_w={$c_1 \vee c_2$, $c_3 \vee c_4$, $c_5 \vee c_6$, $c_7 \vee c_8$, $c_9 \vee c_{10}$}, where $c_i \vee c_j$ (i=1, 3, 5, 7, 9; j=2, 4, 6, 8, 10) indicates that c_i and c_j are mutually exclusive so that only one of them can be triggered.

Definition 5 (Upstream and Downstream Routing Constraint Sets for Activities): The upstream routing constraint set C^u_v for activity v is defined as the set of routing conditionsneeded for workflow W to execute v. The downstream routing constraint set C^d_v for activity v is defined as the set of routing conditionsfor W to execute the downstream activities of v, i.e., those activities to be executed after v. Routing conditionsnot included in C^u_v or C^d_v form the irrelevant routing constraint set of v, denoted as C^i_v. C^u_v, C^d_v, and C^i_v are mutually exclusive.

Table 5 shows the upstream (C^u_v), downstream (C^d_v), and irrelevant (C^i_v) routing constraint sets for the activities in the property loan approval process. Some activities, such as v_1, v_2, and v_3, have the same upstream (C^u_v), downstream (C^d_v), and irrelevant (C^i_v) routing constraint sets, and they are always executed in the same workflow instance. When a loop occurs, the routing constraint c that causes the execution of an activity v in the loop is considered as the upstream routing constraint for v only if c is needed in order

to execute v. For example, the routing constraint c_2 is always needed in order to execute activity v_4. Thus, c_2 is the upstream routing constraint for v_4. Furthermore, c_2 is not the upstream routing constraint for v_1 and v_2 because it is possible to execute activities v_1 and v_2 with or without c_2.

Activity	C^u_v	C^d_v	C^i_v
v_1, v_2, v_3	$\varnothing$	$\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$	$\varnothing$
v_4	$\{c_2\}$	$\{c_1, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$	$\varnothing$
v_5, v_6, v_7, v_8	$\{c_1\}$	$\{c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}\}$	$\{c_2\}$
$v_9, v_{10}, v_{11}, v_{12}$	$\{c_1, c_3\}$	$\{c_5, c_6, c_7, c_8, c_9, c_{10}\}$	$\{c_2, c_4,\}$
v_{13}, v_{14}	$\{c_1, c_3, c_5\}$	$\{c_7, c_8, c_9, c_{10}\}$	$\{c_2, c_4, c_6\}$
v_{15}	$\{c_1, c_3, c_5 \vee c_6\}$	$\{c_7, c_8, c_9, c_{10}\}$	$\{c_2, c_4,\}$
v_{16}, v_{17}	$\{c_1, c_3, c_5 \vee c_6, c_7\}$	$\{c_9, c_{10}\}$	$\{c_2, c_4, c_8\}$
v_{18}	$\{c_1, c_3, c_5 \vee c_6, c_7, c_9\}$	$\varnothing$	$\{c_2, c_4, c_8, c_{10}\}$

Table 5. Upstream and Downstream Routing Constraint Sets

In a workflow, an activity depends on data produced by other activities, leading to activity dependencies. Next, we define two basic concepts, data dependency and activity dependency. There are two types of dependencies, mandatory and conditional, as defined below.

Definition 6 (Mandatory Input Data Set) The mandatory input data set for activity v is the set of data items that activity v always requires as input in order to produce an output and each of such data items is called an mandatory input data item of v. The mandatory input data set for activity v is denoted as $I^m_v=[i_1, i_2,\ldots, i_m]$, where i_j is the j^{th} mandatory

input data item and m is the total count of mandatory input data items needed by v.

Definition 7 (Conditional Input Data Set) The conditional input data set for activity v is the set of data items that activity v requires as input only when C is satisfied where $C \subseteq C_w$ and each of such data items is called a conditional input data item of v. The conditional input data set for activity v is denoted as $I^c_v=[i_1, i_2,\ldots, i_n]$, where i_j is the j^{th} conditional input data item and n is the total count of conditional input data items needed by v.

For instance, in the property loan approval process, the activities v_3, *decision node 1*, and v_4, *request missing info*, always use data item d_4, *application complete*, as data input. Hence, activities v_3 and v_4 have the same mandatory input data set, i.e., the set containing only d_4. It is different for activity v_{16}, *forward to loan officer for signature*. When the risk level is high, in addition to the other data items that v_{16} always needs, v_{16} also needs the data item d_{16}, *amount adjusted*, as input. Hence, the conditional input data set for v_{16} is the set containing d_{16} .

Definition 8 (Mandatory Data Dependency) The mandatory data dependency for activity v, denoted as $\lambda^m_v(I^m_v, O_v)$, represents the mandatory dependency of O_v on I^m_v, where I^m_v is the mandatory input data set for v (Definition 6) and $O_v=[o_1, o_2,\ldots, o_k]$ is the set of k output data items produced by v.

Definition 9 (Conditional Data Dependency) The conditional data dependency for activity v, denoted as $\lambda^c_v(I^c_v, O_v)$, represents the conditional dependency of O_v on I^c_v, where I^c_v is the conditional input data set for v (Definition 7) and $O_v=[o_1, o_2,\ldots, o_k]$ is the set of k output data items produced by v.

Mandatory data dependencies	λ^{m}_{v10} ([d_1, d_{12}], [d_{13}, d_{14}])
λ^{m}_{v1} (ϕ, [d_1, d_2, d_3, d_5, d_8, d_{12}])	λ^{m}_{v11} ([d_1, d_2, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{14}], [d_{15}])
λ^{m}_{v2} ([d_1, d_2, d_3, d_8, d_{12}], [d_4])	λ^{m}_{v12} ([d_{15}], ϕ)
λ^{m}_{v3} ([d_4], ϕ)	λ^{m}_{v13} ([d_1, d_{11}, d_{14}, d_{15}], [d_{16}])
λ^{m}_{v4} ([d_4], ϕ)	λ^{m}_{v14} ([d_1, d_2, d_{10}, d_{18}], [d_{17}, d_{19}])
λ^{m}_{v5} ([d_1, d_3], [d_6, d_{10}])	λ^{m}_{v15} ([d_{17}, d_{19}], ϕ)
λ^{m}_{v6} ([d_1], [d_7, d_{10}])	λ^{m}_{v16} ([d_1, d_2, d_{10}, d_{11}, d_{15}, d_{17}, d_{19}], [d_{20}])
λ^{m}_{v7} ([d_1, d_8], [d_9, d_{10}])	λ^{m}_{v17} ([d_{15}], ϕ)
λ^{m}_{v8} ([d_{10}], ϕ)	λ^{m}_{v18} ([d_1, d_2, d_5, d_{10}, d_{11}, d_{15}, d_{17}, d_{19}], [d_{21}])
λ^{m}_{v9} ([d_1, d_2, d_3, d_7, d_{14}], [d_{11}])	λ^{m}_{e} ([d_{10}], [d_{10}])
Conditional data dependency	
λ^{c}_{v9} ([d_{16}], [d_{11}])	λ^{c}_{v17} ([d_2, d_{16}], ϕ)
λ^{c}_{v14} ([d_{16}], [d_{17}, d_{19}])	λ^{c}_{v18} ([d_{16}], [d_{21}])
λ^{c}_{v16} ([d_{16}], [d_{20}])	λ^{c}_{e} ([d_1, d_{11}, d_{15}, d_{16}, d_{17}, d_{19}, d_{20}, d_{21}], [d_1, d_{11}, d_{15}, d_{16}, d_{17}, d_{19}, d_{20}, d_{21}])

Table 6. Data Dependencies for Property Loan Approval Workflow

In the property loan approval process, there is a conditional data dependency for activity v_{16}, *forward to loan officer for signature*, where the input data item d_{16}, *amount adjusted*, may be null depending on the risk level. When the risk level is low, activity v_{13}, *adjust the loan amount*, is not activated. Therefore, d_{16} is not initialized under this condition. But activity v_{16} can still be activated. Hence, v_{16} has a conditional dependency on d_{16}, i.e., v_{16} depends on d_{16} only when the risk level is high. This conditional dependency is denoted as $\lambda^{c}_{v16}([d_{16}], [d_{20}])$. Table 6 shows the mandatory and conditional data dependencies in the property loan approval process.

Definition 10 (Activity dependency[2]): Given two activities, v and u, v is dependent on u, denoted as $u \Rightarrow v$, if I) $\exists d$ such that $d \in O_u$, $d \notin E$, and $d \in I_v$, or II) $u \Rightarrow x$ and $x \Rightarrow v$ where x is

[2] In this , we limit our attention to activity dependencies that can be derived from a data dependency.

some activity.

Informally, we say that v is mandatorily dependent on u if u provides mandatory input data for activity v. If u provides conditional input data for activity v, we say that v is conditionally dependent on u. If there is no dependency between u and v, we denote the non-dependency between them as $u \infty v$. If x depends on u conditionally and v depends on x mandatorily, or if x depends on u mandatorily and v depends on x conditionally, there is conditional dependency between u and v. Mutual dependency, i.e., $u \Rightarrow v$ and $v \Rightarrow u$, is not allowed.

Definition 11 (Mandatory Requisite set) The set of activities Δ^u_v is the mandatory requisite set for activity v if, for any activity $x \in \Delta^u_v$, there exists a data item d such that $d \in O_x$, $d \notin E$, and $d \in I^m_v$.

Definition 12 (Conditional Requisite set) The set of activities Δ^c_v is the conditional requisite set for activity v if for any activity $x \in \Delta^c_v$ there exists a data item d, such that $d \in O_x$, $d \notin E$, and $d \in I^c_v$.

Activity dependencies can be derived from data dependencies. For example, the mandatory input data set for activity v_5, *verify employment status*, includes data item d_1 and d_3. As shown in Figure 2, d_1 and d_3 are the data output from activity v_1, *receive application*. Therefore, activity v_5 depends on activity v_1 through d_1 and d_3. The requisite set for activity v is the set of activities that provide the data set used by activity v as either mandatory input or conditional input. Table 7 shows the requisite sets for the activities in the property loan approval process. From Table 7, we know that most activities depend on activity v_1, *receive application*, to provide necessary input data. Moreover, activities

v_9, v_{14}, v_{16}, v_{17}, and v_{18} depend on activity v_{13}, *adjust the loan amount*, conditionally.

Activities	Mandatory Requisite Set	Conditional Requisite Set
v_1	$\varnothing \Rightarrow v_1$	
v_2	$\{v_1\} \Rightarrow v_2$	
v_3	$\{v_2\} \Rightarrow v_3$	
v_4	$\{v_2\} \Rightarrow v_4$	
v_5	$\{v_1\} \Rightarrow v_5$	
v_6	$\{v_1\} \Rightarrow v_6$	
v_7	$\{v_1\} \Rightarrow v_7$	
v_8	$\{v_5, v_6, v_7\} \Rightarrow v_8$	
v_9	$\{v_1, v_6, v_{10}\} \Rightarrow v_9$	$\{v_{13}\} \Rightarrow v_9$
v_{10}	$\{v_1\} \Rightarrow v_{10}$	
v_{11}	$\{v_1, v_5, v_6, v_7, v_9, v_{10}\} \Rightarrow v_{11}$	
v_{12}	$\{v_{11}\} \Rightarrow v_{12}$	
v_{13}	$\{v_1, v_9, v_{10}, v_{11}\} \Rightarrow v_{13}$	
v_{14}	$\{v_1, v_5, v_6, v_7\} \Rightarrow v_{14}$	$\{v_{13}\} \Rightarrow v_{14}$
v_{15}	$\{v_{14}\} \Rightarrow v_{15}$	
v_{16}	$\{v_1, v_5, v_6, v_7, v_9, v_{11}, v_{14}\} \Rightarrow v_{16}$	$\{v_{13}\} \Rightarrow v_{16}$
v_{17}	$\{v_{11}\} \Rightarrow v_{17}$	$\{v_1, v_{13}\} \Rightarrow v_{17}$
v_{18}	$\{v_1, v_5, v_6, v_7, v_9, v_{11}, v_{14}\} \Rightarrow v_{18}$	$\{v_{13}\} \Rightarrow v_{18}$

Table 7. Activity Dependencies for the Property Loan Approval Process

Definition 13 (Instance Set) For any successful enactment of a workflow, a set of activities is executed in a specified order from the start activity s to the end activity e, requiring a set of routing conditions$C \subseteq C_w$ to be satisfied. We refer to this set of activities as an instance set, denoted as Γ. The corresponding set C is called the routing constraint set of Γ.

Table 8 shows some examples of instance sets for the property loan approval process. Essentially, each instance set is the set of activities that are executed in one workflow instance. For example, when a complete application is received but the applicant is not

qualified, only the activities in the instance set Γ_1 are executed. Table 8 also shows the corresponding routing constraint set for each instance set.

Instance Set	Corresponding Routing Constraint Set
$\Gamma_1=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, e\}$	$C_1=\{c_1, c_4\}$
$\Gamma_2=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, e\}$	$C_2=\{c_1, c_3, c_5, c_8\}$
$\Gamma_3=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{15}, v_{16}, v_{17}, v_{18}, e\}$	$C_3=\{c_1, c_3, c_6, c_7, c_9\}$

Table 8. Instance Sets for the Property Loan Approval Process

4.2 Dataflow Verification Rules

Dataflow verification is the process of analyzing dataflow requirements and detecting dataflow anomalies in a business process. Next, we present lemmas and theorems that give rise to dataflow verification rules, which lay the foundation for discovering dataflow anomalies in business processes. For the relatively simple lemmas, we only provide an informal discussion, but for the more complex lemmas and theorems, a formal proof is given.

Lemma 1 (Condition for Absence of Initialization) Given a data item d that $d \notin (E \cup (\bigcup_{i=1}^{n} O_i))$ and $\exists\, i\ d \in I_i (i=1, 2, \ldots, n)$ or $d \in O_0$, a missing data anomaly occurs in at least one instance of the workflow W.

Discussion: Given $d \in I_i$ or $d \in O_0$, d is used as input by some activities or required as the final output from at least one workflow instance of W. Given $d \notin (E \cup (\bigcup_{i=1}^{n} O_i))$, d is neither provided by any external resources nor produced by any activities. Therefore, d would not be initialized, even though it is used as input, leading to missing data anomaly.

Lemma 2. (Condition for Delayed Initialization) Given that data item $d \in O_v$, $d \notin E$, and

$d \in I_u$, i.e., $v \Rightarrow u$, and $\exists \Gamma$ such that $v \in \Gamma$, $u \in \Gamma$, and activity u precedes activity v, a missing data anomaly occurs.

Discussion: Given that $d \in O_v$, $d \notin E$, and $d \in I_u$, d is produced as an output by v and used as an input by u. Because u precedes v, d has not been initialized by v when u uses d as an input. Therefore, a missing data anomaly occurs due to delayed initialization.

Lemma 3. (Condition for Uncertain Availability) Given that data item $d \in O_v$, $d \notin E$, and $d \in I_u$, i.e., $v \Rightarrow u$, and $\exists \Gamma$ such that the precedence of activity v and activity u cannot be determined until Γ is enacted at run time, missing data anomalies can occur.

Discussion: Given $d \in O_v$, $d \notin E$, and $d \in I_u$, d is produced as an output by v and used as an input by u. Because the precedence of activity v and activity u cannot be determined until run time, it is possible that when u uses d as an input, d has not been initialized by v. Therefore, a missing data anomaly occurs due to uncertain availability.

Lemma 4. (Condition for Improper Routing) If there exists two activities x and y such that $C^{u}_{x} \neq C^{u}_{y}$ and we can find a data item d that $d \in O_x$, $d \notin E$, and $d \in I_y$ and a routing constraint β that $\beta \in C^{u}_{x}$ but $\beta \notin C^{u}_{y}$, a missing data anomaly occurs.

Proof: We prove this lemma by contradiction. Assume that there is no missing data anomaly, meaning that activity x must be executed whenever y is executed in order to pass the input data d from x to y. We call this "the coupled execution requirement". However, $\beta \in C^{u}_{x}$ indicates that x is executed only when β is satisfied and $\beta \notin C^{u}_{y}$ means that y is executed regardless if β is satisfied or not. Therefore, it is possible that y is executed when x is not. This violates the coupled execution requirement. Thus, a

contradiction occurs. Lemma 4 holds. ♦

Lemma 5. (Finite Instance Sets) A workflow with a finite set of activities can only have a finite number of instance sets.

Discussion: Given that W is an acyclic workflow with n decision nodes, and at each decision node there exists a maximum of m possible routing branches, then there exists a maximum of m^n instance activity sets. When W contains cycles, we only need to consider the instance that executes the cycle once, since the instance activity set is the same no matter how many times the cycle is executed. Hence, lemma 5 is correct.

Theorem 1. (Missing Data Verification) A workflow W is free from missing data anomalies if the following conditions are satisfied: (1) $\forall d$ if $d \in (\bigcup_{i=1}^{n} I_i) \cup O_0$, then $d \in E \cup (\bigcup_{i=1}^{n} O_i)$; (2) given activity dependency $u \Rightarrow v$, $\forall \Gamma_w$ if $v \in \Gamma_w$, then $u \in \Gamma_w$ and u precedes v at least once under the routing constrain set C of Γ_w.

Proof: Let Lemmas 1, 2, 3, and 4 be the only conditions under which missing data anomalies can occur. We use enumeration to prove that a workflow W will avoid these four situations if it satisfies the two conditions: 1) $\forall d$ if $d \in (\bigcup_{i=1}^{n} I_i) \cup O_0$, then $d \in E \cup (\bigcup_{i=1}^{n} O_i)$, and 2) given activity dependency $u \Rightarrow v$, $\forall \Gamma_w$ if $v \in \Gamma_w$, then $u \in \Gamma_w$ and u precedes v at least once under the routing constrain set C of Γ_w.

1) When $d \in E \cup (\bigcup_{i=1}^{n} O_i)$ holds for every d that satisfies $d \in (\bigcup_{i=1}^{n} I_i) \cup O_0$, every input data item required by each activity and every final output data item from W are either

provided by the external data input set E (i.e., $d \in E$) or produced by certain activities in W (i.e., $d \in \bigcup_{i=1}^{n} O_i$). Therefore, W is free from Absence of Initialization according to Lemma 1.

2) We prove that W is free from Delayed Initialization and Uncertain Availability through contradiction. Suppose that there exists Delayed Initialization or Uncertain Availability when the set of routing constrains C is satisfied. Since a missing data anomaly occurs under C, we can find two activities, v and u, in Γ_w, such that v depends on activity u for some dataset D, i.e., $u \Rightarrow v$, and the precedence of u and v can be either v precedes u (Lemma 2) or the precedence cannot be determined until run time (Lemma 3). In either case, u does not precede v. This contradicts the hypothesis. Hence, when C is satisfied, there exists no Delayed Initialization or Uncertain Availability. By lemma 5, there are a finite number of instance activity sets in W. For each instance activity set Γ_w, if there exists an activity dependency between two activities v and u, u precedes v. Therefore, we conclude W is free from Delayed Initialization or Uncertain Availability under all the routing conditions.

3) Given the activity dependency $u \Rightarrow v$, $\forall \Gamma_w$ if $v \in \Gamma_w$, then $u \in \Gamma_w$ and u precedes v at least once under the routing constrain set C of Γ_w. Hence, we cannot find a set of routing conditions under which v is executed whereas u is not when u and v have dependency $u \Rightarrow v$. Therefore, W is free from Improper Routing (Lemma 4). Hence, Theorem 1 holds. ♦

Lemmas 1 to 4 above provide mathematical descriptions for the conditions under

which the four scenarios, *absence of initialization*, *delayed initialization*, *uncertain availability*, and *improper routing*, may cause missing data anomalies. Lemma 5 proves that a workflow can only have a finite number of instance sets and therefore it is possible to examine every instance set in a workflow model. According to Theorem 1, in order for a workflow model to be free of missing data anomalies, the following two conditions must hold for every workflow instance. First, every data item used as input data by an activity or required as the final output data from the workflow needs to be initialized. Second, if activity *u* produces a data item that is used as input by another activity *v*, then when *v* is executed for the first time, *u* must already be executed to guarantee the data item needed by *v* has been produced.

Lemma 6. (Condition for Inevitable Redundancy) If the following inequality holds, $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) \neq \varnothing$, redundant data anomalies occur in at least one instance of the workflow *W*.

Discussion: Given $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) \neq \varnothing$, there exists at least one data item *d* such that *d* is either from some external resources or produced as output by some activities in *W*, but *d* is not needed as input by any activities or required as final output in at least one instance Γ. Therefore, *d* is a redundant data item in Γ.

Lemma 7. (Condition for Contingent Redundancy) Given $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) = \varnothing$, and $\exists \Gamma$ that satisfies $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) \neq \varnothing$, a redundant data anomaly occurs in

the workflow *W*.

Discussion: Given $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) \neq \varnothing$ in instance set Γ, we know by Lemma 6 there exists at least one redundant data item *d* for the workflow instance corresponding to Γ.

Theorem 2 (Redundant Data Verification) A workflow *W* is free from redundant data anomalies if $\forall \Gamma$: $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) = \varnothing$.

Proof: Since $\forall \Gamma$ $((\bigcup_{i=1}^{n} O_i) \cup E) \setminus ((\bigcup_{i=1}^{n} I_i) \cup O_0) = \varnothing$, in each instance every output data item is used as input or required as the final output. Therefore, by Lemma 6, there is no Inevitable Redundancy and by Lemma 7 there is no Contingent Redundancy ♦

Lemmas 6 and 7 describe the conditions under which redundant data anomalies can occur because of the two scenarios, *inevitable redundancy* and *contingent redundancy*. According to Theorem 2, a workflow is free from redundant data anomalies if, for every instance, all the output data items are consumed by other activities or taken as the final output of the workflow.

Lemma 8. (Condition for Multiple Initializations) Given two different activities *x* and *y*, if $O_x \cap O_y \neq \varnothing$ and $C^u_x \subseteq C^u_y$ or $C^u_y \subseteq C^u_x$, a conflicting data anomaly occurs.

Discussion: Since $C^u_x \subseteq C^u_y$, *x* is executed when *y* is executed. Because $C^u_y \subseteq C^u_x$, *y* is executed when *x* is executed. Therefore, under either $C^u_x \subseteq C^u_y$ or $C^u_y \subseteq C^u_x$, it is possible that both *x* and *y* are executed. Because of $O_x \cap O_y \neq \varnothing$, we can assume a data item $d \in (O_x \cap O_y)$. When both *x* and *y* are executed, *x* and *y* can initialize *d* different values. As

such, a conflicting data anomaly occurs.

Theorem 3. (Conflicting Data Verification) A workflow W is free from conflicting data anomalies if the following conditions holds: 1) for any two activities x and y $O_x \cap O_y = \varnothing$, or 2) given $O_x \cap O_y \neq \varnothing$, $\forall \Gamma_w$ if $x \in \Gamma_w$, then $y \notin \Gamma_w$.

Proof: 1) If for any two activities x and y $O_x \cap O_y = \varnothing$, then no two activities initialize the same data item. Therefore, W is free from conflicting data anomalies.

2) Given $O_x \cap O_y \neq \varnothing$, x and y initialize the same data item. However, x and y are executed in different Γ_w since $\forall \Gamma_w$ if $x \in \Gamma_w$, then $y \notin \Gamma_w$. By Lemma 8, we know the workflow is free from conflicting data anomalies. ♦

Lemma 8 and Theorem 3 specify that in a workflow instance, each data item can only be initialized by one activity in order to avoid conflicting data anomalies. In summary, the three theorems above have established the correctness criteria for a dataflow model. A workflow model with correct dataflow specification should satisfy all these criteria, which can be summarized as: 1) Any input data item should be initialized before it is used; 2) Any output data item should be either used as input for some other activity or required as final output; 3) No two activities that produce the same data item can be executed in the same workflow instance.

4.3 Dataflow Verification Algorithms

In this section, we present the dataflow verification algorithms that are based on the theorems we have proven in the previous section. The verification algorithms consist of three procedures as shown in Figure 3, which can be used as a road map for

implementing a dataflow verification component in a workflow system.

The procedure *Checking_Missing_Data*, based on Theorem 1, first finds the mandatory and conditional requisite sets for each activity v in each instance of a workflow W. If the requisite sets cannot provide all the required data input for v at the time v is performed, or if some activities in the requisite sets are not included in the instance, then missing data is detected. The procedure *Checking_Redundant_Data,* based on Theorem 2, examines each instance set of a workflow. If in an instance some data items are produced, but they are not used as input of other activities or needed as final output, then redundant data is detected. The procedure *Checking_Conflicting_Data,* based on Theorem 3, examines each instance for activities that initialize the same data. If such a situation is discovered, then conflicting data is detected.

Next, we give a high-level analysis of computational complexity. First, we examine the procedure *Checking_Missing_Data.* When a workflow has n activities and each activity uses $(n+a)$ data items as input, it takes $n(n+a)$ steps to find all the mandatory requisite sets for each activity in the workflow. If the workflow has $(n+b)$ instances, it takes $n(n+a)(n+b)$, i.e., $(n^3+(a+b)n^2+abn)$, steps to find the conditional requisite set for each activity and to check whether the activities in the requisite sets of each activity v are executed before v in each instance. Assuming a and b can both be positive or negative and have a small absolute value relative to n, the complexity of the procedure *Checking_Missing_Data* is proportional to $[n(n+a) + (n^3+(a+b)n^2+abn)]$, which is equivalent to $(n^3+(a+b+1)n^2+a(b+1)n)$, or approximately $O(n^3)$. In the procedures of *Checking_Redundant_Data* and *Checking_Conflicting_Data*, the number of iterations is

determined by the number of instances and the number of activities. Moreover, in the procedure *Checking_Redundant_Data*, the number of iterations is decided by the number of input data items that each activity requires, as opposed to being decided by the number of output data items, as in the procedure *Checking_Conflicting_Data.* Therefore, the complexity for both procedures is approximately O(n^3), similar to that of the procedure of *Checking_Missing_Data*.

```
PROCEDURE Checking_Missing_Data(W, M, E) {
Δ^u= ∅
Γ_w= findAllInstanceSets(W)
for i = 1 to n { /*check every activity in a workflow*/
        I_vi = findMandatoryInput(M, v_i )
         /*find the mandatory data input of activity v_i from dataflow matrix*/
        for each d∈I_vi and d∉E{
                V_temp= findReqActivity(d) /*find the activity that sets the initial value of d*/
               if (V_temp=null) /*check whether any activity initialize d*/
                        print "Miss data occurs for activity v_i "
                            /*if d is not initialized, missing data occurs due to absence
                                of  initialization*/
                            /* Violation of  the first condition of Theorem 1 */
                 else add v_temp to Δ^u /*find mandatory requisite set for activity v_i */ }
        for each Γ∈Γ_w{
                if (v_i∈Γ) {
                        for each v_j ∈Δ^u {
                                if (v_j∉Γ or v_j isExecutedAfter v_i or v_j isParallelWith v_j )
                                    print "Missing data occurs for v_i in Γ"
                                     /* Violation of  the second condition of Theorem 1 */}
                        C= findRoutingConstraintSet(W, Γ) /*find the routing constraint set for Γ*/
                        I_vi = findConditionalInput(M, v_i , C) /*find conditional input for v_i under C*/
                        for d∈I_vi and d∉E {
                                v_temp = findReqActivity(d)
                                /*find the activity that sets the initial value of d*/
                                if (v_temp=null or v_temp ∉Γ or v_temp isExecutedAfter v_i
                                    or v_temp isParallelWith v_i )
                                    print "Missing data occurs for v_i in Γ"
                                     /* Violation of  the second condition of Theorem 1 */} } } } }

PROCEDURE Checking_Redundant_Data(W, M, E, O_0) {
Γ_w= findAllInstanceSets(W)
for each Γ∈Γ_w { /* check each instance set */
        for each v∈Γ{  D= findOutputData(v, M)  /* find all the output of v */
```

```
                        add D to E  /* find all the output produced in Γ */ }
        for each d∈O0 if (d ∈E) remove d from E /* check if d is required as final output in Γ */
        for each v∈Γ{   D= findInputData(v, M)  /* find all the input of v */
                        for each d∈D   /* check if d is used as input by some activities in Γ */
                            if (d∈E)  remove d from E   }
        if (E ≠∅) print "Redundant data occurs for in  Γ" /* Violation of  Theorem 2 */} }

PROCEDURE  Checking_Conflicting_Data(W, M, O0) {
Vtemp = ∅
Γw= findAllInstanceSets(W)
for i = 1 to n { /*check every activity in a workflow*/
        Ivi = findAllOutput(M, vi )
        for each d∈Ivi if (d ∉ O0) add d to O0 /* find all output data*/}
for each Γ ∈Γw{
        for each d∈O0 {
          Vtemp= findInitializingActivity(d, Γ) /*find all the activities that initialize d in Γ*/
          if ( NumberOfElement(Vtemp) >1) print "Conflicting data occurs for in  Γ"
            /* Violation of  Theorem 3 */} } } }
```

Figure 3. Dataflow Verification Algorithm

4.4 Validation of the Dataflow Verification Framework

In Chapter 3, we presented seven examples in which different dataflow anomalies occur for different reasons. Next, we explain how the dataflow verification theorems can help detect the dataflow anomalies in the seven examples.

- *Detection of Missing Data in Example 1*

As shown in Table 2, no activity initializes data item d_{18}, *property insured,* but activity v_{14} reads d_{18} as input, i.e., $d_{18} \in \bigcup_{i=1}^{18} I_i$ and $d_{18} \notin E \cup (\bigcup_{i=1}^{18} O_i)$, which violates Theorem 1. Therefore, missing data is detected.

- *Detection of Missing Data in Example 2*

From Table 7, we know that there is an activity dependency $v_{13} \Rightarrow v_9$. In the instance set $\Gamma_2 = \{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, e\}$ shown in Table 6, v_9 precedes v_{13} (Figure 1). Therefore, according to Theorem 1, missing data is detected.

- *Detection of Missing Data in Example 3*

From Table 7, we know that there is an activity dependency $v_{10} \Rightarrow v_9$. In both instance sets $\Gamma_2=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, e\}$ and $\Gamma_3=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{15}, v_{16}, v_{17}, v_{18}, e\}$ shown in Table 8, the execution order between v_9 and v_{10} are nondeterministic at design time (Figure 1). Therefore, a violation of Theorem 1 is detected.

- *Detection of Missing Data in Example 4*

From Table 7, we know that there is an activity dependency $v_{14} \Rightarrow v_{15}$. In the instance set $\Gamma_3=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{15}, v_{16}, v_{17}, v_{18}, e\}$ shown in Table 8, v_{15} is executed and v_{14} is not, leading to a violation of Theorem 1.

- *Detection of Redundant Data in Example 5*

As Table 2 shows, data item d_{13}, *current owner of property*, produced by activity v_{10}, *request appraisal info*, is not used by any other activities, i.e., $d_{13} \in (\bigcup_{i=1}^{18} O_i \cup E)$ and $d_{13} \notin (\bigcup_{i=1}^{18} I_i \cup O_0)$. Therefore, $((\bigcup_{i=1}^{18} O_i) \cup E) \setminus ((\bigcup_{i=1}^{18} I_i) \cup O_0) \neq \varnothing$, leading to a violation of Theorem 2. As such, redundant data is detected.

- *Detection of Redundant Data in Example 6*

Data item d_5, *application summary*, is produced by v_1, *receive application*, and only required by activity v_{18}, *forward to manager for signature*. Therefore, for the instance sets $\Gamma_1=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, e\}$ and $\Gamma_2=\{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, e\}$ in Table 8, $((\bigcup_{i=1}^{18} O_i) \cup E) \setminus ((\bigcup_{i=1}^{18} I_i) \cup O_0) = \varnothing$ does not hold since $d_5 \in ((\bigcup_{i=1}^{18} O_i) \cup E)$

and $d_5 \notin ((\bigcup_{i=1}^{18} I_i) \cup O_0)$. Hence, there is a violation of Theorem 2.

- *Detection of Conflicting Data in Example 7*

From Table 2 we know $d_{10} \in O_5$, $d_{10} \in O_6$, and $d_{10} \in O_7$ where O_5, O_6, and O_7 are the output data from v_5, v_6, and v_7, respectively. For all the three instance sets in Table 8, i.e., $\Gamma_1 = \{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, e\}$, $\Gamma_2 = \{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, e\}$, and $\Gamma_3 = \{s, v_1, v_2, v_3, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{15}, v_{16}, v_{17}, v_{18}, e\}$, $O_5 \cap O_6 \neq \varnothing$, $O_5 \cap O_7 \neq \varnothing$, and $O_6 \cap O_7 \neq \varnothing$. Therefore, there is a violation of Theorem 3.

4.5 Conclusions

In this chapter, we provide an analytical approach for detecting and eliminating the three types of dataflow errors. The verification rules we develop include lemmas specifying the conditions for different dataflow anomalies to occur and verification theorems describing the sufficient conditions for a workflow to be free from dataflow anomalies. The example of property loan approval process is used to demonstrate that the dependency-based approach is capable of detecting the dataflow anomalies under different scenarios.

We have limited our attention to the five basic workflow patterns, namely sequence, AND-Split, AND-Join, XOR-Split and XOR-Join. More advanced workflow patterns (Aalst et al. 2003b) such as an OR vertex (either Split or Join) can be simulated by a combination of AND and XOR vertices (Bi and Zhao, 2004a and 2004b). Therefore, the correctness of dataflow where an OR vertex is involved can be determined by analyzing the equivalent constructs consisting of AND and XOR vertices.

5. A DEPENDENCY ANALYSIS BASED APPROACH TO WORKFLOW DESIGN: CONCEPTS AND PRINCIPLES

In this chapter, we propose an analytical method of workflow design based on data dependency analysis. The basic idea is to decide how activities should be sequenced in a workflow by examining the transformation from input data to output data via a sequence of activities. The set of activities and their data input and output can be identified through interviews and discussion sessions. Information about dataflow can also be collected from existing forms and documents, such as product specifications (Reijers et al. 2003). Thus, dataflow in a business process can be determined before the activity sequences are identified. The information contained in a dataflow model can then be used to determine activity sequences in the resulting workflow.

The design framework proposed in this chapter starts with identification of a set of business activities and their input and output data. After a dataflow model is created, activity execution relations can be derived from the dataflow model and then used to identify the workflow model. We further present a matrix based design procedure to help generate workflow models. The design procedure is characterized with the capability of handling complex workflow models including unstructured workflow and overlapping patterns.

5.1 Basic Concepts

Next, we extend the concepts of data dependency analysis introduced in Chapter 4 and the concepts of workflow modeling along with some related concepts including inline

blocks. These concepts are fundamental to determining activity sequences. To illustrate some of these concepts, an order processing example is introduced.

5.1.1 Dataflow concepts: Data Dependencies and Activity Dependencies

Definition 14 (Routing condition) A routing condition c specifies that a set of activities V will be executed when a condition clause $f(D)$ is determined to be true, and is denoted as $c=f(D):Execute(V)$, where D is a set of data items and $f(D)$ is a logic expression of D.

An example of a routing condition is that when the condition "travel expense d is greater than $5,000" is true, the activity v, "approve the travel application by a director," will be executed. This routing condition can be written as: $c=(d>5000)$: $Execute(v)$. Note that the concept of routing condition is similar to the concept of routing constraint defined in Definition 3. But they serve different purposes. Given a workflow model, each routing constraint is associated with a link leaving from a decision node. The activities, which can be executed when a routing constraint is satisfied, are implied in the workflow model. In workflow design, a routing condition specifies not only the routing constraint but also the set of activities executed when a routing constraint is satisfied.

In addition to the mandatory and conditional dependency, we introduce another type of dependency that needs to be considered in workflow design, i.e., execution dependency.

Definition 15 (Execution Data Dependency) Execution data dependencies for activity v, denoted as $\lambda^{e}_{v}[I^{e}_{v},O_{v}]$, is the dependency of activity v on I^{e}_{v} in order to produce the output data set O_v where I^{e}_{v} represents the set of data that $\forall d \in I^{e}_{v}$ there exists a routing condition

$c=f(D)$:*Execute*(V) such that $d \in D$ and $v \in V$; In another word, the value of d determines whether v is executed.

Moreover, if $d \in I^e_v$ and $d \in O_u$, v has an execution dependency on u, denoted as $u \Rightarrow_e v$.

For simplicity, we use $\lambda_v[I^m_v|I^c_v|I^e_v, O_v]$ as an abbreviation for the totality of mandatory data dependency $\lambda^m_v[I^m_v, O_v]$, conditional data dependency $\lambda^c_v[I^c_v, O_v]$, and execution data dependency $\lambda^e_v[I^e_v, O_v]$.

Definition 16 (Dataflow) Given a set of business activities V, dataflow Λ is the set of data dependencies for all activities in V, denoted as $\Lambda=\{\lambda_v[I^m_v|I^c_v|I^e_v, O_v] \mid v \in V\}$ or $\Lambda=\{\lambda_v[I_v, O_v] \mid v \in V\}$.

Definition 17 (Direct Requisite Set Δ_v) A set of activities Δ_v is the direct requisite set for activity v if for any activity $x \in \Delta_v$, there exists a data item d such that $d \in O_x$, $d \in I_v$, where I_v is the input data set of activity v, O_x is the output data set of x, and $x \neq v$.

Definition 18 (Completeness of Δ_v) Given activity v and Δ_v, if $\forall d \in I_v$, there exists u such that $u \in \Delta_v$ and $d \in O_u$, then Δ_v is complete.

Essentially, the direct requisite set Δ_v for activity v is the set of activities that produce the data directly used by v as input. It includes both the mandatory requisite set defined in Definition 11 and the conditional requisite set defined in Definition 12. We say Δ_v is complete if for any d used by activity v as input we can always find an activity from the direct requisite set Δ_v that produces d as output. Moreover, if Δ_v is complete, any data needed by v as input is produced by an activity in Δ_v or given as external input.

Definition 19 (Full Requisite Set Γ_v) Given a set of activities V, their data dependencies

$\Lambda=\{\lambda_v[I_v,O_v]|\ v\in V\}$, the full requisite set Γ_v for $v\in V$ is a subset of V such that $u\Rightarrow v$ iff $u\in\Gamma_v$.

The full requisite set Γ_v for v is the set of activities that transform the external input data available at the various activities of a workflow into the data needed by v as input. Some of the activities in Γ_v may not produce data used by v directly as input but provide some intermediate data needed to produce the data items in I_v. Note $\Gamma_v\supseteq\Delta_v$.

5.1.2 Order Processing Example

Activities	**Data Items**
v_1: process order	d_0: quantity available
v_2: check product availability	d_1: product IDs
v_3: send replenishment order	d_2: order ID
v_4: verify credit	d_3: customer ID
v_5: order approval by sales manager	d_4: quantity ordered
v_6: process down payment	d_5: product available
v_7: confirm order	d_6 down payment amount
v_8: send back order notice	d_7: customer credit rating
v_9: update product inventory	d_8: approved by manager
v_{10}: make shipment	d_9: down payment paid
v_{11}: process payment	d_{10}: updated availability
v_{12}: notify of the sales manager's "No" decision	d_{11}: replenishment quantity
	d_{12}: replenishing date
v_{13}: notify order fulfillment	d_{13}: confirmation number
e: end activity	d_{14}: shipping date
s: start activity	d_{15}: back order notice status
	d_{16}: payment made
	d_{17}: order fulfilled
	d_{18}: order approval notified

Figure 4. Symbols Used in the Order Processing Workflow

Assume that we are to design an order processing workflow. Our main task of workflow design is to determine a correct sequence for the given activities in consideration of their input and output data as well as the routing conditions. Figure 4 shows the relevant

activities and key data items that have been identified from the existing documents. In this workflow, activity v_9, *update product inventory*, always uses the product quantity ordered by a customer as input in order to update the product availability. Therefore, data item d_4, *quantity ordered*, is the mandatory data input of v_9 and data item d_{10}, *updated availability*, is the output of v_9, i.e., $d_4 \in I^m_{v9}$ and $d_{10} \in O_{v9}$.

Business Rule	**Routing condition**
If a customer orders more than what is available, a replenish order should be sent to the manufacturer and a back order notice should be sent to the customer.	c_1={d_5 = "*No*": *Execute*(v_3, v_8)}
If there is enough product in the inventory, the customer's credit will be checked.	c_2={d_5 = "*Yes*": *Execute*(v_4)}
If the credit rating of a customer is bad, the order needs to be approved by a sales manager	c_3={d_7="*bad*": *Execute*(v_5)}
The down payment can be processed only if the credit rating of a customer is good or the order is approved by a sales manager.	c_4={(d_7="*good*"): *Execute*(v_6)} c_5={(d_8="*Yes*"): *Execute*(v_6)}
If the order is disapproved by the sales manager, the customer will be notified of the sales manager's decision.	c_6={(d_8="*No*"): *Execute*(v_{12})}

Table 9. Routing Conditions for the Order Processing Workflow

Table 9 shows the routing conditions for the order processing workflow. Given the routing conditions, we can identify the execution data dependency based on Definition 15. For instance, given c_1={d_5= "*Yes*": *Execute*(v_3, v_8)}, we know $d_5 \in I^e_{v3}$ and $d_5 \in I^e_{v8}$.

In addition, we need to examine whether an activity can still be enacted if an input data item is null. If that is the case, the dependency of the activity on the input data item is considered as conditional. For example, d_8, *approved by manager*, is the input for activities v_6, v_7 and v_{11}. When v_6, v_7 and v_{11} are enacted, the manager's approval (d_8) is

Activity	I^m_v	I^c_v	I^e_v	O_v
s	d_0	$\varnothing$	$\varnothing$	d_0
e	d_1, d_5, d_6	$d_9, d_{10}, d_{11}, d_{13}, d_{14}, d_{15}, d_{16}$	$\varnothing$	$d_1, d_5, d_6, d_9, d_{10}, d_{11}, d_{13}, d_{14}, d_{15}, d_{16}$
v_1	$\varnothing$	$\varnothing$	$\varnothing$	d_1, d_2, d_3, d_4, d_6
v_2	d_0, d_2, d_4	$\varnothing$	$\varnothing$	d_5
v_3	d_1	$\varnothing$	d_5	d_{11}
v_4	d_3	$\varnothing$	d_5	d_7
v_5	d_1, d_2, d_3, d_4, d_7	$\varnothing$	d_7	d_8
v_6	d_1, d_2, d_3, d_4, d_6	d_8	d_7, d_8	d_9
v_7	d_2, d_3, d_7, d_9	d_8	$\varnothing$	d_{13}
v_8	d_1, d_2, d_3	$\varnothing$	d_5	d_{15}
v_9	d_0, d_4, d_{14}	$\varnothing$	$\varnothing$	d_{10}
v_{10}	$d_1, d_2, d_3, d_4, d_{13}$	$\varnothing$	$\varnothing$	d_{14}
v_{11}	$d_1, d_2, d_3, d_4, d_6, d_7, d_{13}, d_{14}$	d_8	$\varnothing$	d_{16}
v_{12}	d_2, d_7, d_8	$\varnothing$	d_8	d_{18}
v_{13}	$d_1, d_2, d_3, d_{14}, d_{16}$	$\varnothing$	$\varnothing$	d_{17}

Table 10. Data Dependencies in the Order Processing Workflow

needed only if the customer does not have a good credit rating. When the *customer credit rating* (d_7) is good, v_6, v_7 and v_{11} can be enacted without the manager's approval (d_8). As such, v_6, v_7 and v_{11} will still be enacted even if d_8 is null, i.e., $d_8 \in I^c_{v6}$, $d_8 \in I^c_{v7}$ and $d_8 \in I^c_{v11}$. Moreover, it is possible that $I^e_v \cup I^c_v \neq \varnothing$, i.e., there exists a data item d that determines whether activity v will be executed only under some conditions. For instance, given the routing condition $c_5=\{(d_8=\text{“}Yes\text{”}): Execute(v_6)\}$ as shown in Table 9, the execution of v_6 depends on the value of d_8. By Definition 3, we have $d_8 \in I^e_{v6}$. On the other hand, even if

d_8 is null, v_6 can still be executed if d_7="*good*", given the routing condition c_4={(d_7="*good*"): *Execute*(v_6)}. By Definition 3, we have $d_8 \in I^c_{v6}$. Therefore, $I^e_{v6} \cup I^c_{v6} \neq \varnothing$.

Activity	Mandatory ($\Rightarrow_m$)	Conditional($\Rightarrow_c$)	Execution($\Rightarrow_e$)
v_1	*None*	*None*	*None*
v_2	$s \Rightarrow_m v_2$, $v_1 \Rightarrow_m v_2$	*None*	*None*
v_3	$v_1 \Rightarrow_m v_3$	*None*	$v_2 \Rightarrow_e v_3$
v_4	$v_1 \Rightarrow_m v_4$	*None*	$v_2 \Rightarrow_e v_4$
v_5	$v_1 \Rightarrow_m v_5$	*None*	$v_4 \Rightarrow_e v_5$
v_6	$v_1 \Rightarrow_m v_6$	$v_5 \Rightarrow_c v_6$	$v_4 \Rightarrow_e v_6$, $v_5 \Rightarrow_e v_6$
v_7	$v_1 \Rightarrow_m v_7$, $v_4 \Rightarrow_m v_7$, $v_6 \Rightarrow_m v_7$	$v_5 \Rightarrow_c v_7$	*None*
v_8	$v_1 \Rightarrow_m v_8$	*None*	$v_2 \Rightarrow_e v_8$
v_9	$s \Rightarrow_m v_9$, $v_1 \Rightarrow_m v_9$, $v_{10} \Rightarrow_m v_9$	*None*	*None*
v_{10}	$v_1 \Rightarrow_m v_{10}$, $v_7 \Rightarrow_m v_{10}$	*None*	*None*
v_{11}	$v_1 \Rightarrow_m v_{11}$, $v_4 \Rightarrow_m v_{11}$, $v_7 \Rightarrow_m v_{11}$, $v_{10} \Rightarrow_m v_{11}$	$v_5 \Rightarrow_c v_{11}$	*None*
v_{12}	$v_1 \Rightarrow_m v_{12}$, $v_4 \Rightarrow_m v_{12}$, $v_5 \Rightarrow_m v_{12}$	*None*	$v_5 \Rightarrow_e v_{12}$
v_{13}	$v_1 \Rightarrow_m v_{13}$, $v_{10} \Rightarrow_m v_{13}$, $v_{11} \Rightarrow_m v_{13}$	*None*	*None*
e	$v_1 \Rightarrow_m e$, $v_2 \Rightarrow_m e$	$v_3 \Rightarrow_c e$, $v_6 \Rightarrow_c e$, $v_7 \Rightarrow_c e$, $v_8 \Rightarrow_c e$, $v_9 \Rightarrow_c e$, $v_{10} \Rightarrow_c e$, $v_{12} \Rightarrow_c e$	*None*

Table 11. Activity Dependencies in the Order Processing Workflow

Table 10 shows the data dependencies of each activity in the order processing workflow. Table 11 shows the activity dependencies in the order processing workflow, and Table 12 lists the direct requisite set Δ_v and the full requisite set Γ_v for all the activities in the order processing workflow.

5.1.3 Workflow Concepts

In this section, we formalize the concept of activity relations to represent the activity execution structures in a control flow.

Activity	Δ_v	Γ_v
s	$\varnothing$	$\varnothing$
v_1	$\varnothing$	$\varnothing$
v_2	$\{s, v_1\}$	$\{s, v_1\}$
v_3	$\{v_1, v_2\}$	$\{s, v_1, v_2\}$
v_4	$\{v_1, v_2\}$	$\{s, v_1, v_2\}$
v_5	$\{v_1, v_4\}$	$\{s, v_1, v_2, v_4\}$
v_6	$\{v_1, v_4, v_5\}$	$\{s, v_1, v_2, v_4, v_5\}$
v_7	$\{v_1, v_4, v_5, v_6\}$	$\{s, v_1, v_2, v_4, v_5, v_6\}$
v_8	$\{v_1, v_2\}$	$\{s, v_1, v_2\}$
v_9	$\{s, v_1, v_{10}\}$	$\{s, v_1, v_2, v_4, v_5, v_6, v_7, v_{10}\}$
v_{10}	$\{v_1, v_7\}$	$\{s, v_1, v_2, v_4, v_5, v_6, v_7\}$
v_{11}	$\{v_1, v_4, v_5, v_7, v_{10}\}$	$\{s, v_1, v_2, v_4, v_5, v_6, v_7, v_{10}\}$
v_{12}	$\{v_1, v_4, v_5\}$	$\{s, v_1, v_2, v_4, v_5\}$
v_{13}	$\{v_1, v_{10}, v_{11}\}$	$\{s, v_1, v_2, v_4, v_5, v_6, v_7, v_{10}, v_{11}\}$
e	$\{v_1, v_2, v_3, v_6, v_7, v_8, v_9, v_{10}, v_{12}\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}\}$

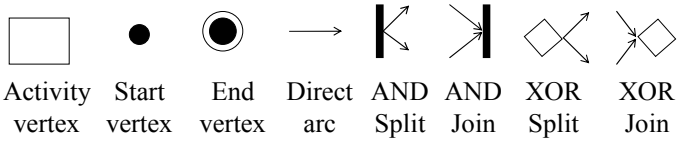
Table 12. Direct Requisite Sets (Δ_v) and Full Requisite Set (Γ_v) in the Order Processing Workflow

Definition 20 (Workflow) A workflow W is a 7-tuple $<A, s, e, R, L, \Lambda, C>$, where

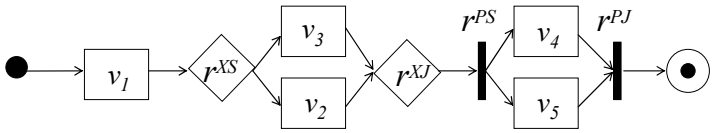
- A is a finite set of business activities,
- $s \in A$ is the start activity where $InDegree(s)=0$, $OutDegree(s)>=1$,
- $e \in A$ is the end activity where $InDegree(e)>=1$, $OutDegree(e)=0$,
- R is a finite set of routing activities and $R=R^{XS} \cup R^{XJ} \cup R^{PS} \cup R^{PJ}$ where R^{XS} is a set of XORSplits, R^{XJ} is a set of XORJoins, R^{PX} is a set of ANDSplits, and R^{PJ} is a set of ANDJoins,
- $L \subseteq (A \cup R) \times (A \cup R)$ is a set of directed arcs among activities,
- Λ is the dataflow of A, i.e., $\Lambda=\{\lambda_v [I^m{}_v | I^c{}_v | I^e{}_v, O_v] \mid v \in A\}$,

- $C=\{c=f(D):Execute(V_c) \mid D \subseteq \{O_v \mid v \in A\}$ and $V_c \subseteq A\}$ is a set of routing conditions.

Figure 5 shows the graphic representation of a workflow model, where v_1, v_2, v_3, v_4, and v_5 are business activities and r^{XS}, r^{XJ}, r^{PS}, and r^{PJ} are routing activities.



(a) Elements of workflow model



(b) A simple workflow example

Figure 5. Activity Based Workflow Modeling Using the UML Activity Diagram Notation

Definition 21 (Dot Notation[3]) Let $W = <A, s, e, R, L, \Lambda, C>$ be a workflow model. $v_j \in v_i\bullet$ and $v_i \in \bullet v_j$ iff $v_i, v_j \in A \cup R$ and there exists a direct arc from v_i to v_j in L.

Definition 22 (Firing Sequence) Given a workflow model $W=<A, s, e, R, L, \Lambda, C>$ and activities $v_i, v_j \in A \cup R$, a firing sequence between v_i and v_j, denoted as $\sigma(v_i, v_j)$, is a process instance of firing v_i and all other activities before firing v_j. If $v_j \in v_i\bullet$, then $\sigma(v_i, v_j)$ contains only v_i.

Definition 23 (Firing Rule) Let $W = <A, s, e, R, L, \Lambda, C>$ be a workflow model. Let

[3] We borrow the concepts of dot notation, firing rule, and firing sequence from Petri nets, but redefine them in the context of activity-based modeling.

$v_i \in A \cup R$:

- if $v_i \in R^{PS}$, every activity in $v_i\bullet$ can be fired after v_i is fired,
- if $v_i \in R^{PJ}$, v_i can be fired only after every activity in $\bullet v_i$ is fired,
- if $v_i \in R^{XS}$, only one activity in $v_i\bullet$ can be fired after v_i is fired,
- if $v_i \in R^{XJ}$, v_i can be fired after any activity in $\bullet v_i$ is fired,
- if $v_i \in A$, $|\bullet v_i|>1$, and $\cap_j \sigma(s, v_j)=\sigma(s, v_z)$ where $v_j \in \bullet v_i$ and $v_z \in R^{XS}$, v_i can be fired after any $v_j \in V$ is fired,
- if $v_i \in A$, $|\bullet v_i|>1$, and $\cap_j \sigma(s, v_j)=\sigma(s, v_z)$ where $v_j \in \bullet v_i$ and $v_z \in R^{PS}$, v_i can be fired after all $v_j \in V$ is fired,
- otherwise, v_i can be fired after any activity in $\bullet v_i$ is fired.

Definition 24 (Activity Relations) Given an acyclic workflow model $W=<A, s, e, R, L, \Lambda, C>$, and $v_i, v_j \in A \cup R$, seven types of activity relations are defined as follows:

- immediate precedence (*): $v_i * v_j$ iff $v_j \in v_i\bullet$,
- weak precedence ($>_w$): $v_i >_w v_j$ iff there exists a firing sequence $\sigma(v_i, v_j) \neq \emptyset$,
- strong precedence ($>_s$): $v_i >_s v_j$ iff $\forall \sigma(s, v_j)$ that $\sigma(v_i, v_j) \subseteq \sigma(s, v_j)$,
- conditional precedence ($>^k_C$): $v_i >^k_C v_j$ iff $v_i * v_y$, $v_y * v_j$, and $v_y \in R^{XS}$, the superscript k is used to specify the unique routing condition between v_i and v_j,
- XOR-parallel ($\vee$): $v_i \vee v_j$ iff $v_x \in \bullet v_i$, $v_x \in \bullet v_j$, and $\forall \sigma(s, v_i)$ and $\sigma(s, v_j)$, $\sigma(s, v_i) \cap \sigma(s, v_j)=\sigma(s, v_x)$ holds where $v_x \in R^{XS}$,

- AND-parallel ($\wedge$): $v_i \wedge v_j$ iff $v_x \in \bullet v_i$, $v_x \in \bullet v_j$, and $\forall \sigma(s, v_i)$ and $\sigma(s, v_j)$, $\sigma(s, v_i) \cap \sigma(s, v_j) = \sigma(s, v_x)$ holds where $v_x \in R^{PS}$,
- non-sequential (∞): $v_i \infty v_j$ iff $\sigma(v_i, v_j) = \varnothing$ and $\sigma(v_j, v_i) = \varnothing$

Relations "$*$", "$>_w$", "$>_s$", and "$>^k_C$" indicate sequential relations. $v_i * v_j$ indicates that v_j is to be fired right after v_i is fired, and $v_i >_w v_j$ and $v_i >_s v_j$ indicate that v_i precedes v_j but v_j does not necessarily follow v_i immediately. $v_i >_w v_j$ indicates that there is a firing sequence from v_i to v_j but v_i may not be included in some firing sequences to v_j. $v_i >_s v_j$ indicates that v_i is included in all the firing sequences from s to v_j. For example, in Figure 5b, v_1 must be fired before v_4 whenever v_4 is executed; therefore $v_1 >_s v_4$. Since there is a firing sequence from v_3 to v_4 and it is possible that v_3 is not executed when v_4 is executed, we have $v_3 >_w v_4$. $v_i >^k_C v_j$ indicates that v_i is followed by an XORsplit, which immediately precedes v_j. Relation "$\wedge$" indicates the beginning of parallelism. If $v_i \wedge v_j$, then v_i and v_j are executed in parallel. "$\vee$" indicates the beginning of conditional routing. If $v_i \vee v_j$, then either v_i or v_j is executed but not both. Relation "∞" indicates that two activities are irrelevant in terms of firing sequence.

5.1.4 Concepts of Inline blocks

WfMC (1999) defined the concept of *inline block* as a collection of activities that has one entry point and one exit point. An inline block may be condensed into a block activity corresponding to a sub-process containing the inline block. In this chapter, we consider only sequential inline blocks, which is sufficient for our purpose.

Definition 25 (Sequential Inline Block) Given an acyclic workflow model $W=<A, s, e, R,$

$L, \Lambda, C>$, a set of activities $V^s_{u,v} \subseteq A$ forms a sequential inline block if there exists only one firing sequence $\sigma(u, v)$ and $\forall x \in V^s_{u,v}$, $x \in \sigma(u, v) \cup \{v\}$ and $|x\bullet| = |\bullet x| = 1$. That is, we have a single chain of activities starting with u and ending with v. Activity u is called the *source* of the block and activity v is called the *sink* of the block.

When creating an inline block V, we need to figure out the activity relations between V and other activities outside V. We give the following property for this purpose.

Definition 26 (Relation Aggregation Property) A sequential inline block V^s aggregates the corresponding activity relations of the original activities contained in V^s, using the following scheme: Let v_i be the source of V^s, v_j be the sink of V^s, activities v_p, $v_q \notin V^s$, and θ be any of the seven activity relations defined in Definition 24. For each activity relation $v_p \theta v_i$, we have $v_p \theta V^s$. For each $v_j \theta v_q$, we have $V^s \theta\ v_q$. For any conditional precedence relation $v_x >^k_c v_q$ where $v_x \in V^s$, we will have $V^s >^k_c v_q$ because of relation transitivity. Note that when aggregating activity relations, we need to observe the priority among activity relations, i.e., "*" and "$>^k_c$" override "$>_s$", "$>_s$" overrides "$>_w$", "$\vee$" and "$\wedge$" overrides "∞".

5.2 Workflow Design Principles

5.2.1 Correctness of Dataflow

Workflow design requires the identification of correct activity sequences. As a prerequisite, we must ensure the dataflow specification is complete and concise as defined below.

Definition 27 (Completeness of Dataflow) Given a set of activities V and its dataflow

$\Lambda=\{\lambda_v[I_v, O_v] \mid v \in V\}$, if $\forall v \in V$, Δ_v is complete as defined in Definition 18, then Λ is complete.

Definition 28 (Conciseness of Dataflow) Given a set of activities V and its dataflow $\Lambda=\{\lambda_v[I_v, O_v] \mid v_i \in V\}$, Λ is concise if the following two conditions hold: 1) for each $d \in O_{vi}$, $v_i \in V$, there exists $v_j \in V$ such that $d \in I_{vj}$; and 2) for each $d \in I_{vi}$, $v_i \in V$, there exists only one $v_j \in V$ such that $d \in O_{vj}$.

Intuitively, dataflow needs to be complete, i.e., any input data not provided by external recourses must be produced as output by an activity in the workflow. Further, the dataflow must be concise, i.e., each output data item is used by at least one activity as input and no more than one activity produces the same output data[4]. Note that we cannot perform a complete verification of dataflow at this point since we have not yet generated the control flow. Our workflow design procedure proposed in this dissertation will prevent the occurrence of dataflow errors defined in (Sun et. al. 2006).

If the data dependencies are not complete, there are two possibilities. First, some activities may have been omitted and need to be identified. Second, the data dependencies may not be accurate and need to be refined. If the data dependencies are not concise, then some activities not needed for a workflow may have been included or some activities may have produced more data than necessary. In either case, the activities and data dependencies need to be refined.

---

[4] It is possible that under different routing situations, the same data item may be produced by different activities. For our purpose, they are considered as different data outputs.

5.2.2 Identification of Sequential Execution

Next, we present the principles for identifying the basic workflow constructs. The basic idea is straightforward. If activity v_j uses some input data d produced by activity v_i, then v_j cannot be executed before v_i is executed. Otherwise, data d will not be available for v_j to use. To make the lemmas and theorems easier to understand, we first explain each lemma and theorem using an example and then give the proof.

Lemma 9 (Weak Precedence Rule 1) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow model. Let Λ be complete and concise. For any $v_i, v_j \in A$: $v_i \Rightarrow v_j$ implies $v_i >_w v_j$.

EXAMPLE 8. In the order processing workflow, we know $v_5 \Rightarrow v_6$, $v_5 \Rightarrow v_{11}$, and $v_5 \Rightarrow v_{12}$ (Table 11), which means some data produced by v_5 are used as input by v_6, v_{11}, and v_{12}. Moreover, by transitivity, any activity depending on v_6, v_{11}, or v_{12}, such as v_7, has indirect dependencies on v_5. Therefore, in the workflow model for order processing there should exist firing sequences from v_5 to v_6, v_7, v_{11}, v_{12}, i.e., $v_5 >_w v_6$, $v_5 >_w v_7$, $v_5 >_w v_{11}$, and $v_5 >_w v_{12}$.

Proof: $v_i \Rightarrow v_j$ indicates two possibilities: (1) there exists a data item d such that $d \in O_{vi}$, and $d \in I_{vj}$; or (2) there exists a group of activities V_m such that $v_i \Rightarrow V_m$ and $V_m \Rightarrow v_j$, namely $\exists d_1$, d_2 that $d_1 \in O_{vi}$, $d_1 \in I_{Vm}$, $d_2 \in O_{Vm}$, and $d_2 \in I_{vj}$.

a) We first prove that under Condition (1), $v_i \Rightarrow v_j$ implies $v_i >_w v_j$. Assuming that $v_i >_w v_j$ does not hold, then by Definition 24, there exits no firing sequence $\sigma(v_i, v_j)$. Therefore, $v_i \notin \sigma(s, v_j)$ holds for every $\sigma(s, v_j)$. Since Λ is concise, there exists no other $v \in A$ that can produce d as output. Then d will not be available for v_j to use when it is executed. Therefore, $v_i >_w v_j$ must hold so that d can be produced and v_j can use d as input when v_j is

executed.

b) We then prove that under Condition (2), $v_i \Rightarrow v_j$ implies $v_i >_w v_j$. Since there exists a group of activities V_m such that $\exists d_1$, d_2 that $d_1 \in O_{vi}$, $d_1 \in I_{Vm}$, $d_2 \in O_{Vm}$, and $d_2 \in I_{vj}$, according to the proof above $v_i >_w V_m$ and $V_m >_w v_j$ must hold. Therefore, there exist $\sigma(v_i, V_m)$ and $\sigma(V_m, v_j)$. We can construct $\sigma(v_i, v_j) = \sigma(v_i, V_m) \cup \sigma(V_m, v_j)$. Therefore, $v_i >_w v_j$ holds.

By a) and b), we conclude $v_i \Rightarrow v_j$ implies $v_i >_w v_j$. ♦

Corollary 1 (Weak Precedence Rule 2) Let v_i and v_j be two activities. $v_i \in \Gamma_{vj}$ implies $v_i >_w v_j$.

Proof: Since $v_i \in \Gamma_{vj}$, by Definition 19, $v_i \Rightarrow v_j$. By Lemma 9 , we have $v_i >_w v_j$. ♦

Lemma 10 (Strong Precedence Rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow. Let Λ be complete and concise. For any v_i, $v_j \in A$, if $v_i \Rightarrow_m v_j$ or $v_i \Rightarrow_e v_j$, and the dependency between v_i and v_j are not conditional, then $v_i >_s v_j$.

EXAMPLE 9. In the order processing workflow, we know $v_5 \Rightarrow_m v_{12}$ and there is no conditional dependency between v_5 and v_{12} (Table 11), which means v_{12} uses some data items produced by v_5 as mandatory input. Therefore, v_5 must be included on every firing sequence to v_{12} in the workflow model.

Proof: Given $v_i \Rightarrow_m v_j$ or $v_i \Rightarrow_e v_j$, and the dependency between v_i and v_j are not conditional, by Definitions 10 and 15, there exists a data item d such that $d \in O_{vi}$, $d \in I^m_{vj}$ or $d \in I^e_{vj}$, and $d \notin I^c_{vj}$. Then, for every $\sigma(s, v_j)$, $v_i \in \sigma(s, v_j)$ holds; otherwise d will not be available for v_j to use and by Definitions 8 and 15 v_j cannot be executed since $d \in I^m_{vj}$ or $d \in I^e_{vj}$. By Definition 24, $v_i >_s v_j$. ♦

Theorem 4 (Immediate Precedence Rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow model. Let $v_i, v_j \in A$. $v_i * v_j$ occurs if 1) $v_i \Rightarrow_m v_j$, and v_j does not have execution dependency on v_i, and 2) there exists no $v_x \in A$ such that $v_i \in \Gamma_{vx}$ and $v_x \in \Gamma_{vj}$.

EXAMPLE 10. In the order processing workflow, we know $v_6 \Rightarrow_m v_7$ and v_7 has no execution dependency on v_6 (Table 11), which means v_7 uses some data items produced by v_6 as mandatory input. Then v_6 must be included on every firing sequence to v_7 in the workflow model. Moreover, given $\Gamma_{v6}=\{s, v_1, v_2, v_4, v_5\}$ and $\Gamma_{v7}=\{s, v_1, v_2, v_4, v_5, v_6\}$ (Table 12), there exists no activity v_x that $v_6 \in \Gamma_{vx}$ and $v_x \in \Gamma_{v7}$. That is, no activity needs to be placed between v_6 and v_7. Therefore, v_6 must be executed immediately before v_7, i.e., $v_6 * v_7$.

Proof: We prove that no other activities must be sequenced between v_i and v_j by contradiction. Assume that there exists activity $v_x \in A$ that must be placed between v_i and v_j. Then $v_i >_w v_x$ and $v_x >_w v_j$ must hold. By Lemma 9 , $v_i \Rightarrow v_x$ and $v_x \Rightarrow v_j$ must hold because otherwise $v_i >_w v_x$ and $v_x >_w v_j$ would not be necessary. Therefore, $v_i \in \Gamma_{vx}$ and $v_x \in \Gamma_{vj}$, which contradicts the assumption. Consequently, no activity $v_x \in A$ needs to be placed between v_i and v_j. As such, the theorem holds. ♦

Lemma 9 and Corollary 1 provide the basic principles on how to derive a weak precedence relation "$>_w$" based on data dependencies. Lemma 10 describes when a strong precedence relation "$>_s$" should be used. Theorem 4 describes the conditions when an immediate precedence relation "*" must be needed.

5.2.3 Identification of Conditional Routing and Parallelism

Next, we present the principles for deriving parallel relations, including conditional precedence "$>^{k}_{C}$", XOR-parallel "$\vee$", and AND-parellel "$\wedge$", between two activities, v_i and v_j.

Theorem 5 (Conditional Precedence Rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow. For any $v_i, v_j \in A$, $v_i >^{k}_{C} v_j$ occurs if $v_i \in \Gamma_{vj}$ and for each $v_y \in \Gamma_{vj} \backslash \Gamma_{vi}$, $\exists d \in O_{vy}$ such that there exists $c_k=f(D)$:*Execute* (V) where $d \in D$ and $v_j \in V$. That is, the set of $\Gamma_{vj} \backslash \Gamma_{vi}$ cannot contain v_x such that $v_i =>_m v_x$ and $v_x =>_m v_j$.

EXAMPLE 11. In the order processing workflow, we know $v_4 \in \Gamma_{v6}$ and $\Gamma_{v6} \backslash \Gamma_{v4} = \{v_4, v_5\}$ (Table 12). Moreover, we know $c_4=\{(d_7=$"*good*"): *Execute*$(v_6)\}$ and $c_5=\{(d_8=$"*Yes*"): *Execute*$(v_6)\}$ (Table 9) where $d_7 \in O_{v4}$ and $d_8 \in O_{v5}$ (Table 10), which means v_6 can be executed either after v_4 or v_5 is executed. Since there is no activity v_x such that $v_4 =>_m v_x$ and $v_x =>_m v_6$, no activity must be executed after v_4 and before v_6. As such, we have the firing sequence $\sigma(v_4, v_6)=\{v_4\}$ or $\{v_4, v_5\}$. In the workflow model, there should be an XORSplit between v_4 and v_6, i.e., $v_4 >^{4}_{C} v_6$.

Proof: Given $v_i \in \Gamma_{vj}$ and for each $v_y \in \Gamma_{vj} \backslash \Gamma_{vi}$, $\exists d \in O_{vy}$ such that there exists $c_k=f(D)$:*Execute* (V) where $d \in D$ and $v_j \in V$, we have $v_i \Rightarrow_e v_j$. Furthermore, we know that the set of $\Gamma_{vj} \backslash \Gamma_{vi}$ cannot contain v_x such that $v_i =>_m v_x$, $v_x =>_m v_j$. Therefore, by Theorem 4 there is no activity $v_x \in A$ such that $v_i * v_x$ and $v_x * v_j$ must holds. By Definition 23, v_i and v_j are separated by an *XORSplit*. By Definition 24, $v_i >^{k}_{C} v_j$ holds. ♦

By Theorem 5, given $v_i \in \Gamma_{vj}$, in order to decide if $v_i >^{k}_{C} v_j$ holds, we first need to

calculate $\Gamma_{vj}\backslash\Gamma_{vi}$, i.e., the difference between Γ_{vj} and Γ_{vi}. If v_j has only execution dependencies on every activity $v_y \in \Gamma_{vj}\backslash\Gamma_{vi}$, then $v_i >^{k}_{C} v_j$ should hold.

Lemma 11 (Non-Sequential Rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow. Given any two activities $v_i, v_j \in A$, $v_i \infty v_j$ holds if $v_i \notin \Gamma_{vj}$ and $v_j \notin \Gamma_{vi}$.

EXAMPLE 12. In the order processing workflow, we know $v_9 \notin \Gamma_{v11}$ and $v_{11} \notin \Gamma_{v9}$ (Table 12). Then v_9 cannot be included in any firing sequence to v_{11} and vice versa, i.e., $v_9 \infty v_{11}$.

Proof: We prove by contradiction. Assume $v_i \infty v_j$ holds if $v_i \in \Gamma_{vj}$ or $v_j \in \Gamma_{vi}$. By Corollary 1, we conclude that $v_i >_w v_j$ or $v_j >_w v_i$, i.e., if $v_i \in \Gamma_{vj}$ or $v_j \in \Gamma_{vi}$ there exist $\sigma(v_i, v_j) \neq \varnothing$ or $\sigma(v_j, v_i) \neq \varnothing$. As such, $v_i \infty v_j$ does not hold if $v_i \in \Gamma_{vj}$ or $v_j \in \Gamma_{vi}$. This contradicts the assumption. Therefore, Lemma 11 holds. ♦

Lemma 11 suggests that two activities independent of each other have a non-sequential relation "∞". Next, we present the conditions for XOR-parallel "$\vee$" and AND-parellel "$\wedge$".

Theorem 6 (XOR-Parallel rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow. Given any two activities $v_i, v_j \in A$, let $c_1=f_1(D_1)$:*Execute*(V_1) and $c_2=f_2(D_2)$:*Execute* (V_2) be the routing conditions for v_i and v_j to be executed, i.e., $v_i \in V_1$ and $v_j \in V_2$. $v_i \vee v_j$ occurs if (1) $\Gamma_{vi} = \Gamma_{vj}$, (2) $f_1(D_1)$ and $f_2(D_2)$ cannot be both true or both false at the same time, and (3) $\exists v_x \in \Gamma_{vi}$ such that $O_{vx} \cap D_1 \neq \varnothing$ and $O_{vx} \cap D_2 \neq \varnothing$.

EXAMPLE 13. In the order processing workflow, we know either c_1={d_5 = "*No*": *Execute*(v_3, v_8)} or c_2={d_5 = "*Yes*": *Execute*(v_4)} can be true but not both (Table 9). Moreover, we know $\Gamma_{v3}=\Gamma_{v4}$={s, v_1, v_2} (Table 12) and $d_5 \in O_{v2}$. As such, either v_3 or v_4 should be executed after v_2 is executed. Since $v_3 \notin \Gamma_{v4}$ and $v_4 \notin \Gamma_3$, v_3 does not need to be included in any firing sequence to v_4 and vice versa. In the workflow model, there should be an XORSplit between v_2 and {v_3, v_4}, i.e., $v_3 \vee v_4$.

Proof: Given the acyclic workflow W and $\Gamma_{vi}= \Gamma_{vj}$, we have $v_i \notin \Gamma_{vj}$ and $v_j \notin \Gamma_{vi}$. By Lemma 11, we have $v_i \infty v_j$. Given $\exists v_x \in \Gamma_{vi}$ such that $O_{vx} \cap D_1 \neq \varnothing$ and $O_{vx} \cap D_2 \neq \varnothing$, $v_x \Rightarrow_e v_i$ and $v_x \Rightarrow_e v_j$. By Lemma 10 , $v_x >_s v_i$ and $v_x >_s v_j$, i.e., $v_x \in \sigma(s, v_i)$ and $v_x \in \sigma(s, v_j)$ for every $\sigma(s, v_i)$ and $\sigma(s, v_j)$. Since $f_1(D_1)$ and $f_2(D_2)$ cannot be both true or both false at the same time, $v_i \notin \sigma(s, v_j) \cup \sigma(v_j, e)$ and $v_j \notin \sigma(s, v_i) \cup \sigma(v_i, e)$ always hold. By Definition 23, $\sigma(v_x, v_i) \cap \sigma(v_x, v_j) = \sigma(v_x, r^{XS})$ where r^{XS} is an *XORSplit*. Given $\Gamma_{vi}= \Gamma_{vj}$, r^{XS} should be placed after Γ_{vi} and before v_i and v_j. By Definition 24, we have $v_i \vee v_j$. ♦

Theorem 7 (AND-Parallel rule) Let W=<A, s, e, R, L, Λ, C> be an acyclic workflow. Given any two activities v_i, $v_j \in A$, $v_i \wedge v_j$ occurs if $\Gamma_{vi}=\Gamma_{vj}$ and either (1) there are no $v_x \in \Gamma_{vi}$ such that $v_x \Rightarrow_e v_j$ and $v_x \Rightarrow_e v_j$ or (2) there exists a routing condition $c=f(D)$:*Execute* (V) such that $v_i \in V$, $v_j \in V$.

EXAMPLE 14. In the order processing workflow, we know c_1={d_5 = "*No*": *Execute*(v_3, v_8)}(Table 9), which means whenever v_3 is executed v_8 is executed as well. Moreover, we know $\Gamma_{v3}=\Gamma_{v8}$={s, v_1, v_2} (Table 12). Since $v_3 \notin \Gamma_{v8}$ and $v_8 \notin \Gamma_{v3}$, v_3 does not need to be included in any firing sequence to v_8 and vice versa. In the workflow model, there should

be an ANDSplit between v_2 and $\{v_3, v_8\}$, i.e., $v_3 \wedge v_8$.

Proof: Given $\Gamma_{vi} = \Gamma_{vj}$, we have $v_i \notin \Gamma_{vj}$ and $v_j \notin \Gamma_{vi}$ by Definition 19. By Lemma 11, we have $v_i \infty v_j$, i.e., $\sigma(v_i, v_j)=\varnothing$ and $\sigma(v_j, v_i)=\varnothing$. Under either condition (1) or (2), v_i and v_j are either both executed or both not executed. Therefore, there exists a firing sequence $\sigma(s, e)$ that $v_i \in \sigma(s, e)$ and $v_j \in \sigma(s, e)$. By Definition 23, it is only possible that $\sigma(s, v_i) \cap \sigma(s, v_j) = \sigma(s, r^{PS})$ where r^{PS} is an *ANDSplit*. Given $\Gamma_{vi} = \Gamma_{vj}$, r^{XS} should be placed after Γ_{vi} and before v_i and v_j. By Definition 24, we have $v_i \wedge v_j$. ♦

Activity Relations	Design Principles	Implications
$v_i >_w v_j$	Occurs if $v_i \Rightarrow v_j$ or $v_i \in \Gamma_{vj}$ (By Lemma 9 and Corollary 1)	There exists a firing sequence from v_i to v_j
$v_i >_s v_j$	Occurs if $v_i \Rightarrow_m v_j$ or $v_i \Rightarrow_e v_j$ (By Lemma 10)	v_i is included in all firing sequences to v_j
$v_i * v_j$	Occurs if the following two conditions are both met: 1) $v_i \Rightarrow_m v_j$, and v_j does not have execution dependency on v_i, and 2) there exists no $v_x \in A$ such that $v_i \in \Gamma_{vx}$ and $v_x \in \Gamma_{vj}$ (By Theorem 4)	v_i immediately precedes v_j
$v_i >^k_C v_j$	Occurs if $v_i \in \Gamma_{vj}$ and v_j has only execution dependencies on every activity $v_y \in \Gamma_{vj} \backslash \Gamma_{vi}$ (By Theorem 5)	A XORSplit can be placed between v_i and v_j
$v_i \infty v_j$	Occurs if $v_i \notin \Gamma_{vj}$ and $v_j \notin \Gamma_{vi}$ (By Lemma 11)	Neither does v_i precede v_j and nor does v_j precede v_i
$v_i \vee v_j$	Occurs if (1) $\Gamma_{vi} = \Gamma_{vj}$, (2) $\exists v_x \in \Gamma_{vi}$, $v_x \Rightarrow_e v_i$ and $v_x \Rightarrow_e v_j$, and (3) when the routing condition c_p for v_i is true, the routing condition c_q for v_j is false or vice versa (By Theorem 6)	An XORSplit is before v_i and v_j
$v_i \wedge v_j$	Occurs if $\Gamma_{vi} = \Gamma_{vj}$ and there exists no routing conditions where v_i is executed and v_j is not (By Theorem 7)	An ANDSplit can be placed before v_i and v_j,

Table 13. Summary of Principles for Designing Various Activity Relations

Theorem 6 and Theorem 7 describe the situations where parallelism and conditional

routing start, respectively. Table 13 summarizes the principles for designing the seven types of activity relations.

Given a set of activity relations, we need to determine where to place the routing activities, including XORSplit, XORJoin, ANDSplit, and ANDJoin. Next, we present three more lemmas for such a purpose. Lemma 12 describes how to determine where to place an XORSplit. Lemmas 13 and 14 describe how to determine where to place an ANDJoin and an XORJoin, respectively.

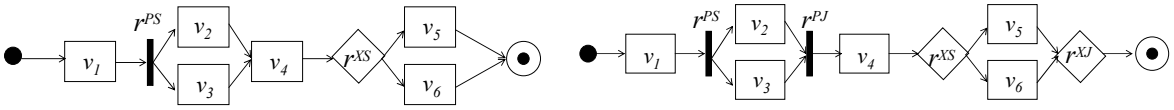
Lemma 12 (XORSplit rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow. Given activity $v_x \in A$, a set of activities $V \subset A$, if 1) $v_x >^{k}_{C} v_i$ always holds for every $v_i \in V$ and 2) the routing condition $c_i=f_i(D_i)$:*Execute* (V_i) for every $v_i \in V$ (i.e., $v_i \in V_i$) is mutually exclusive, namely only one of c_i (i=*1, 2, 3, ...n*) can be true, then there can be only one XORSplit, namely r^{XS}, between v_x and V such that $v_x * r^{XS}$ and $r^{XS} * v_i$ for every $v_i \in V$.

EXAMPLE 15. By applying Theorem 5 in the order processing workflow, we get $v_2 >^{1}_{C} v_3$ and $v_2 >^{2}_{C} v_4$, which correspond to the two routing conditions, $c_1=\{d_5=$"*No*": *Execute*(v_3, v_8)$\}$ and $c_2=\{(d_5=$"*Yes*": *Execute*(v_4)$\}$ (Table 9), respectively. Since c_1 and c_2 are mutually exclusive, we just need place one XORSplit, namely r^{XS}, between v_2 and $\{v_3, v_4\}$ such that $v_2 * r^{XS}$, $r^{XS} * v_3$, and $r^{XS} * v_4$.

Proof: Since $v_x >^{k}_{C} v_i$ always holds for every $v_i \in V$, there exists an XORSplit between v_x and every v_i. Since the routing condition c_i for every $v_i \in V$ (i.e., $v_i \in V_i$) is mutually exclusive, only one $v_i \in V$ can be fired. By Definition 23, there can be only one XORSplit, namely r^{XS}, between v_x and V such that $v_x * r^{XS}$ and $r^{XS} * v_i$ for every $v_i \in V$. ♦

Lemma 13 (ANDJoin rule) Let $W=<A, s, e, R, L, \Lambda, C>$ be an acyclic workflow, activity

$v_i \in A$, and $V = \bullet v_i$. If $|V|>1$ and $\cap_j \sigma(s, v_j) = \sigma(s, r^{PS})$ where $v_j \in V$, then an ANDJoin activity r^{PJ} can be added between V and v_i such that $\bullet r^{PJ} = V$ and $r^{PJ} \bullet = \{v_i\}$.



(a) A workflow model without Joins (b) A workflow model with Joins

Figure 6. An Simple Workflow Design Example

EXAMPLE 16. In Figure 6(a), $\bullet v_4 = \{v_2, v_3\}$ and $|\bullet v_4|>1$. Moreover, $\sigma(s, v_2) \cap \sigma(s, v_3) = \sigma(s, r^{PS})$. Therefore, an ANDJoin should be placed between $\{v_2, v_3\}$ and v_4, as shown in Figure 6(b).

Proof: Given $V = \bullet v_i$, $v_j \in V$, $\cap_j \sigma(s, v_j) = \sigma(s, r^{PS})$, by Definition 23, v_i can be fired after all $v_j \in V$ are fired. Therefore, adding an ANDJoin activity r^{PJ} between V and v_i is appropriate. ♦

Lemma 14. (XORJoin rule) Let $W = <A, s, e, R, L, \Lambda, C>$ be an acyclic workflow, activity $v_i \in A$, and $V = \bullet v_i$. If $|V|>1$ and $\cap_j \sigma(s, v_j) = \sigma(s, r^{XS})$ where $v_j \in V$, then an XORJoin activity r^{XJ} can be added between V and v_i such that $\bullet r^{XJ} = V$ and $r^{XJ} \bullet = \{v_i\}$.

EXAMPLE 17. In Figure 6(a), $\bullet e = \{v_5, v_6\}$ and $|\bullet e|>1$. Moreover, $\sigma(s, v_5) \cap \sigma(s, v_6) = \sigma(s, r^{XS})$. Therefore, an XORJoin can be placed between $\{v_5, v_6\}$ and e, as shown in Figure 6(b).

Proof: Given $V = \bullet v_i$, $v_j \in V$, $\cap_j \sigma(s, v_j) = \sigma(s, r^{XS})$, by Definition 23, v_i can be fired after any $v_j \in V$ is fired. Therefore, adding an XORJoin activity r^{XJ} between V and v_i is appropriate. ♦

Essentially, the lemmas and theorems above provide the principles for deriving activity relations from the given data dependencies and the principles for determining routing activities. These principles are validated with the example of order processing in the next section.

5.3 Design a Workflow Model for Order Processing Based on Data Dependency

Given the data dependencies (Table 10), the activity dependencies (Table 11), and the direct requisite sets and the full requisite sets (Table 12) for order processing, this section illustrates how to create a workflow model based on the design principles presented in Section 5.2.

5.3.1 Derive a Partial Activity Relation Matrix

Definition 29 (Partial Relation Matrix) Given $V=\{s, v_1, v_2, v_3, \ldots, v_n, e\}$, a partial relation matrix P is a $(n+1)\times(n+1)$ matrix, where the cell p_{ij} shows the activity relation between v_{i-1} and v_j where $i=1, 2, \ldots, n$, $j=1, 2, \ldots, (n+1)$, $v_0=s$, and $v_{n+1}=e$. If p_{ij} = "∞", "$\wedge$", or "$\vee$", then $p_{ji} = p_{ij}$. Otherwise, $p_{ji}= null$.

The following steps are needed to derive a partial relation matrix from data dependencies and activity dependencies in Tables 2, 3, and 4:

1) We derive the weak precedence relation "$>_w$" between two activities by applying Lemma 9 and Corollary 1 to the activity dependencies in Table 11 and Table 12. For example, from Table 12, we know that $v_1 \in \Gamma_{v2}$. By Corollary 1, we know that $v_1 >_w v_2$.
2) The strong precedence relation "$>_s$" between two activities are identified using Lemma 10. Given a set of activity dependencies as shown in Table 11, we apply

Lemma 10 to derive the strong precedence relations "$>_s$". For example, Table 11 gives $v_1 \Rightarrow_m v_2$. By Lemma 10, we know $v_1 >_s v_2$. Therefore, we can change the relation between v_1 and v_2 from "$>_w$" to "$>_s$". The strong precedence relation can be propagated by applying the sequential transitivity, namely, if $v_i >_s v_j$ and $v_j >_s v_m$, then $v_i >_s v_m$. For example, as Table 11 shows, $v_4 \Rightarrow_m v_7$ and $v_7 \Rightarrow_m v_{10}$ and therefore we have $v_4 >_s v_7$ and $v_7 >_s v_{10}$. By the transitivity property, we get $v_4 >_s v_{10}$.

3) We derive immediate precedence relations, i.e., "*", by applying Theorem 4. For example, Table 11 shows $v_1 \Rightarrow_m v_2$. From Table 12, we cannot find an activity v_x that $v_1 \in \Gamma_{vx}$ and $v_x \in \Gamma_{v2}$ given $\Gamma_{v1}=\varnothing$ and $\Gamma_{v2}=\{s, v_1\}$. Since v_2 does not have execution dependency on v_1 (Table 11), by Theorem 4, we have $v_1 * v_2$. Similarly, we find $v_6 * v_7$, $v_7 * v_{10}$, $v_{10} * v_9$, $v_{10} * v_{11}$, and $v_{11} * v_{13}$. Note that the activity relation $s * v_1$ is necessarily true because v_1 does not depend on any other activities and is therefore the leading activity in the workflow. Moreover, we have $v_3 * e$, $v_8 * e$, $v_9 * e$, $v_{12} * e$, and $v_{13} * e$ since only e depends on v_3, v_8, v_9, v_{12}, and v_{13}.

4) Next, we derive conditional precedence relations "$>^k c$" by applying Theorem 5. For example, given $\Gamma_{v4}=\{s, v_1, v_2\}$, $\Gamma_{v5}=\{s, v_1, v_2, v_4\}$, and $\Gamma_{v6}=\{s, v_1, v_2, v_4, v_5\}$, we find that $\Gamma_{v6} \backslash \Gamma_{v4}=\{v_4, v_5\}$ and $\Gamma_{v6} \backslash \Gamma_{v5}=\{v_5\}$. Moreover, we know the two routing conditions $c_4=\{(d_7=\text{"}good\text{"})$: $Execute(v_6)\}$ and $c_5=\{(d_8=\text{"}Yes\text{"})$: $Execute(v_6)\}$ (Table 9), and $d_7 \in O_{v4}$ and $d_8 \in O_{v5}$ (Table 10), and $v_4 \Rightarrow_e v_6$ and $v_5 \Rightarrow_e v_6$ (Table 11). According to Theorem 5, we have $v_4 >^4_c v_6$ and $v_5 >^5_c v_6$. Note that there is no other activity in either $(\Gamma_{v6} \backslash \Gamma_{v4})$ or $(\Gamma_{v6} \backslash \Gamma_{v5})$ except v_4 and v_5. Applying Theorem 5 further, we have five more conditional relations, $v_2 >^1_c v_3$, $v_2 >^1_c v_8$, $v_2 >^2_c v_4$, $v_4 >^3_c v_5$, and $v_5 >^6_c v_{12}$.

5) By Lemma 11, if two activities, v_i and v_j, satisfy the condition $v_i \notin \Gamma_{vj}$ and $v_j \notin \Gamma_{vi}$, then $v_i \infty v_j$. For example, we know $v_3 \notin \Gamma_{v5}$ and $v_5 \notin \Gamma_{v3}$ from Table 12, and therefore, $v_3 \infty v_5$.

6) We further refine the non-sequential relation "∞" and discover XOR-parallel relations, i.e., "$\vee$", from the non-sequential relation "∞" by applying Theorem 6. For example, given $\Gamma_{v3} = \Gamma_{v4}$ (Table 12), the two routing conditions $c_1 = \{d_5 =$ "*No*": *Execute*$(v_3, v_8)\}$ and $c_2 = \{d_5 =$"*Yes*": *Execute*$(v_4)\}$ (Table 9), and $d_5 \in O_{v2}$ (Table 10), i.e., $v_2 \Rightarrow_e v_3$ and $v_2 \Rightarrow_e v_4$ (Table 11), we have $v_3 \vee v_4$ according to Theorem 6. By the same token, we have $v_4 \vee v_8$ and $v_6 \vee v_{12}$.

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	e
s	*	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$
v_1	NA	*	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$
v_2		NA	$>^1_c$	$>^2_c$	$>_s$	$>_s$	$>_s$	$>^1_c$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$
v_3			NA	$\vee$	∞	∞	∞	$\wedge$	∞	∞	∞	∞	∞	*
v_4			$\vee$	NA	$>^3_c$	$>^4_c$	$>_s$	$\vee$	$>_s$	$>_s$	$>_s$	$>_s$	$>_s$	$>_w$
v_5			∞		NA	$>^5_c$	$>_w$	∞	$>_w$	$>_w$	$>_w$	$>^6_c$	$>_w$	$>_w$
v_6			∞			NA	*	∞	$>_s$	$>_s$	$>_s$	$\vee$	$>_s$	$>_w$
v_7			∞				NA	∞	$>_s$	*	$>_s$	∞	$>_s$	$>_w$
v_8			$\wedge$	$\vee$	∞	∞	∞	NA	∞	∞	∞	∞	∞	*
v_9			∞					∞	NA	∞	$\wedge$	∞	∞	*
v_{10}			∞					∞	*	NA	*	∞	$>_s$	$>_w$
v_{11}			∞					∞	$\wedge$		NA	∞	*	$>_w$
v_{12}			∞			$\vee$	∞	∞	∞	∞	∞	NA	∞	*
v_{13}			∞					∞	∞			∞	NA	*

Table 14. The Partial Relation Matrix for the Order Processing Workflow

7) We then discover AND-parallel relations "$\wedge$" by applying Theorem 7. For example, from Step 5 above, we have $v_3 \infty v_8$. Given $c_1 = \{d_5 =$ "*No*": *Execute*$(v_3, v_8)\}$ as shown in Table 9, we know v_3 and v_8 are both executed if $d_5 =$ "*No*". Further, we have $\Gamma_{v3} =$

Γ_{v8} from Table 12. By Theorem 7, $v_3 \wedge v_8$ holds. Applying Theorem 7 further, we have $v_9 \wedge v_{11}$.

Table 14 shows the resulting partial relation matrix for the order processing workflow, which contains all the possible activity relations derived from the data dependencies. For simplicity, in the following sections, we omit the weak and strong precedence relations, i.e., "$>_w$" and "$>_s$", which are not directly used in generating a workflow model.

5.3.2 Stage 1: Generate a Correct Workflow Model without Parallelism

To simplify the model derivation procedure, we derive the workflow model in three stages. In Stage 1, we deal with the correctness of the workflow model by ignoring parallelism. That is, we will randomly sequence activities that could potentially be put in parallel. Since parallelism is used only to improve efficiency, the resulting workflow model without parallelism is correct though not efficient. At this stage, we will treat AND-parallel relations as if they are immediate precedent relations. In Stage 2, we will take into account of the AND-parallel relations that were ignored in Stage 1. In Stage 3, we standardize the model by adding joins such that each business activity can have only one incoming link and one outgoing link.

1) Simplify the design process with simple sequential inline blocks

We apply the concept of sequential inline block (Definition 25) to simplify Table 14. Based on the immediate precedence relations (and the AND-parallel relations) in Table 14, we find three sequential inline blocks $V^S_1=\{v_1, v_2\}$, $V^S_2=\{v_3, v_8\}$, and $V^S_3=\{v_6, v_7, v_{10}, v_9, v_{11}, v_{13}\}$. Note that V^S_2 contains v_3 and v_8 because we sequence v_3 and v_8 in

their natural order and ignore parallelism in Stage 1. The same is true for V^S_3. More specifically, we have the following relations from Table 14, namely, v_6* v_7, v_7*v_{10}, v_{10}*v_9, v_9^v_{11}, v_{10}*v_{11}, and v_{11}*v_{13}. Since we are ignoring parallelism in Stage 1, we can replace v_9^v_{11} with v_9*v_{11} and then ignore v_{10}*v_{11} given v_{10}*v_9 and v_9*v_{11}. We derive Table 15 by replacing the activities v_1, v_2, v_3, v_6, v_7, v_8, v_9, v_{10}, v_{11}, and v_{13} in Table 14 with the three inline blocks. According to the relation aggregation property (Definition 26), the activity relations between the block activities V^S_1, V^S_2, and V^S_3 and other activities inherit the corresponding activity relations of the original activities contained in the block activities. For instance, s*V^S_1 because s*v_1.

	v_4	v_5	v_{12}	e	V^s_1	V^s_2	V^s_3
s					*		
v_4	NA	$>^3_c$				$\vee$	$>^4_c$
v_5		NA	$>^6_c$			∞	$>^5_c$
v_{12}			NA	*		∞	$\vee$
V^s_1	$>^2_c$				NA	$>^1_c$	
V^s_2	$\vee$	∞	∞	*		NA	∞
V^s_3			$\vee$	*		∞	NA

Table 15. A Simplified Partial Activity Relation Matrix

2) Determine XORSplit Activities

Following Lemma 12, we add three XORSplits, r^{XS}_1, r^{XS}_2, and r^{XS}_3. Note that r^{XS}_1 is used between V^s_1 and $\{v_4, V^s_2\}$ because $V^s_1 >^2_C v_4$ and $V^s_1 >^1_C V^s_2$, where c_1=$\{d_5$="*No*": *Execute*(v_3, v_8)$\}$ and c_2=$\{d_5$= "*Yes*": *Execute*(v_4)$\}$ are mutually exclusive. Similarly, r^{XS}_2 is used between v_4 and $\{v_5, V^S_3\}$ because $v_4 >^3_C v_5$ and $v_4 >^4_C V^S_3$, where c_3=$\{d_7$="*bad*": *Execute*(v_5)$\}$ and c_4=$\{(d_7$="*good*"): *Execute*(v_6)$\}$ are mutually exclusive. r^{XS}_3 is needed because $v_5 >^6_C v_{12}$ and $v_5 >^5_C V^s_3$, where c_5=$\{(d_8$="*Yes*"): *Execute*(v_6)$\}$ and c_6=$\{(d_8$="*No*"):

Execute(v_{12})} are mutually exclusive. Table 16 is the result of extending Table 15 with the three XORSplit activities.

	v_4	v_5	v_{12}	e	V^S_1	V^S_2	V^S_3	r^{XS}_1	r^{XS}_2	r^{XS}_3
s					*					
v_4	NA	$>^3_c$				$\vee$	$>^4_c$		*	
v_5		NA	$>^6_c$			∞	$>^5_c$			*
v_{12}			NA	*		∞	$\vee$			
V^S_1	$>^2_c$				NA	$>^1_c$		*		
V^S_2	$\vee$	∞	∞	*		NA	∞			
V^S_3			$\vee$	*		∞	NA			
r^S_1	*					*				
r^S_2		*					*			
r^S_3			*				*			

Table 16. A Simplified Partial Activity Relation Matrix with XORSplit Activities

Now, a workflow model can be created by illustrating graphically the immediate precedence relations in Table 16, leading to Figure 7.

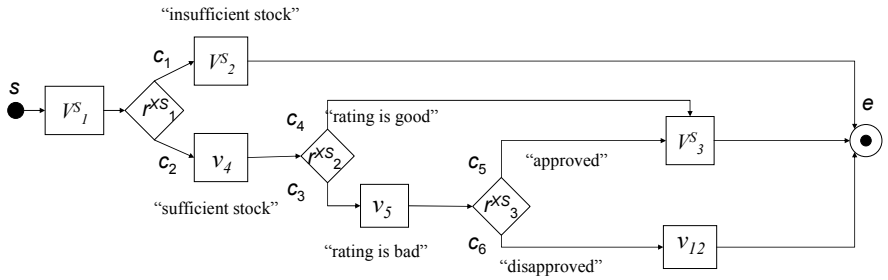


Figure 7. A Workflow Model with Sequential Inline Block Activities

Next, we expand the workflow model by exploding the three sequential inline blocks V^S_1={v_1, v_2}, V^S_2={v_3, v_8}, and V^S_3={v_6, v_7, v_{10}, v_9, v_{11}, v_{13}} (Figure 8).

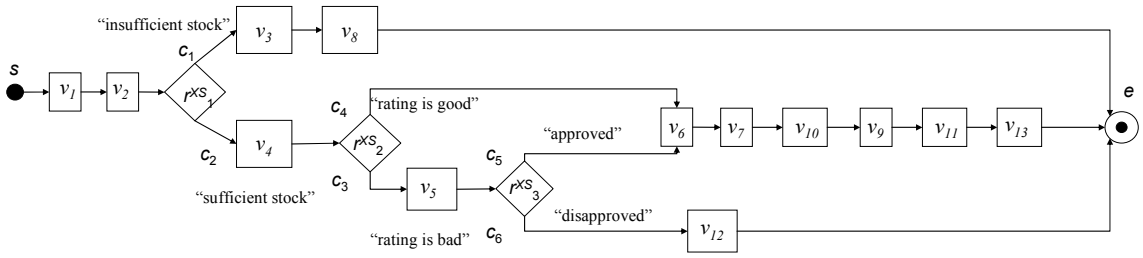


Figure 8. A Workflow Model without Parallelism

5.3.3 Stage 2: Add Parallelism to Achieve Efficiency

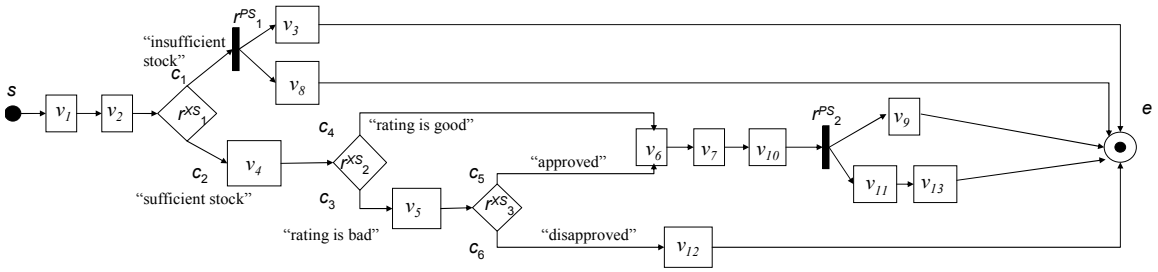


Figure 9. A Workflow Model with Parallelism

Recall that we ignored parallelism in Stage 1 above, which should now be rectified to make the workflow efficient. From Table 14, we know that $v_3 \wedge v_8$ and $v_9 \wedge v_{11}$. That is, v_3 and v_8 should be executed in parallel, and similarly v_9 and v_{11}. Note that the need for parallelism is confined within two inline blocks, $V^S_2=\{v_3, v_8\}$ and $V^S_3=\{v_6, v_7, v_{10}, v_9, v_{11}, v_{13}\}$, defined in Stage 1. For V^S_2, we need to place an ANDSplit activity between v_3 and v_8. V^S_3 contains two simple sequential inline blocks $\{v_6, v_7, v_{10}\}$ and $\{v_{11}, v_{13}\}$. Since $v_9 \wedge v_{11}$, we know that $v_9 \wedge \{v_{11}, v_{13}\}$ by the relation aggregation property (Definition 26). Consequently, we can place an ANDSplit between v_9 and $\{v_{11}, v_{13}\}$. Figure 9 shows the workflow model that is both correct and efficient. However, the model is not yet

standardized as some activities, such as v_6, have more than one incoming link.

5.3.4 Stage 3: Standardize the Model by Adding ANDJoins and XORJoins

We first identify where to place ANDJoins by applying Lemma 13. Given $\{v_3, v_8\} \subset \bullet e$ and $\sigma(s, v_3) \cap \sigma(s, v_8) = \sigma(s, r^{PS}_1)$ in Figure 9, an ANDJoin should be placed between $\{v_3, v_8\}$ and e. For convenience, we denote the simple sequential inline block $\{v_{11}, v_{13}\}$ as V^S_4. Given $\{v_9, V^S_4\} \subset \bullet e$ and $\sigma(s, v_9) \cap \sigma(s, V^S_4) = \sigma(s, r^{PS}_2)$ in Figure 9, an ANDJoin should be placed between $\{v_9, V^S_4\}$ and e. The result is shown in Figure 10.

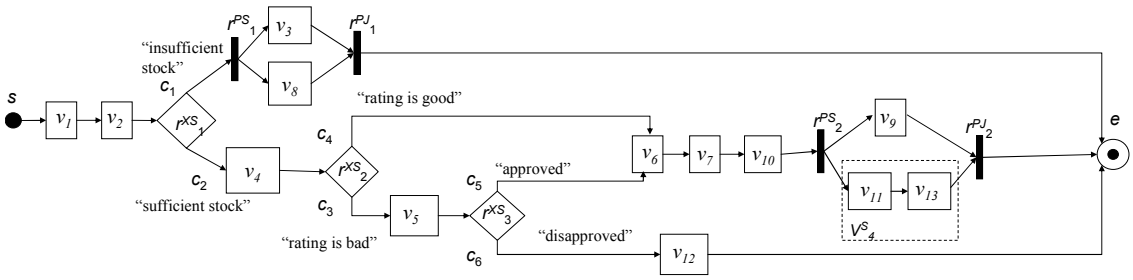


Figure 10. Workflow Model with ANDJoins

Then, according to Lemma 14, XORJoins should be placed before any activity with more than one incoming link that is traced back to an XORSplit. We add XORJoins before v_6 and e, leading to the final workflow model in Figure 11. Note Figure 11 is an unstructured model where not each split activity corresponds to a unique join activity. That is, our workflow design approach is capable of generating unstructured workflows. This important feature can be attributed to our idea of decoupling the model correctness from the model efficiency. By delaying the consideration of parallelism, the workflow design procedure is simplified significantly.

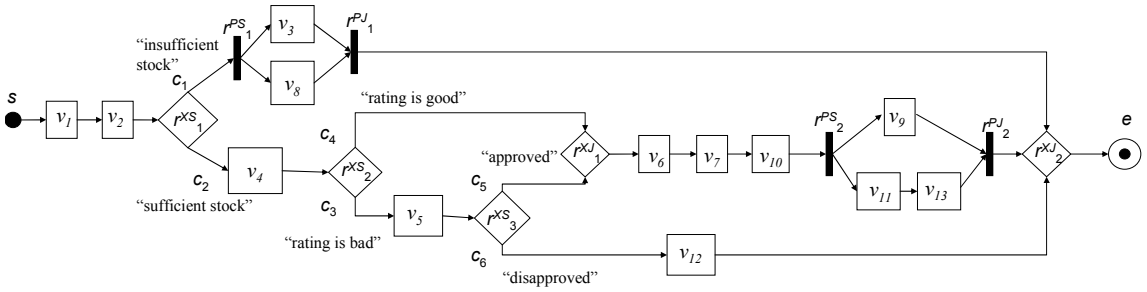


Figure 11. The Final Workflow Model

5.4 A Generic Procedure for Workflow Design

5.4.1 A Generic Procedure

In this section, we generalize the procedure for designing workflow models based on a dataflow specification. This procedure consists of four major steps as shown in Figure 12.

Step 1. Analyze the activity dependencies and derive activity relations from activity dependencies. At this step, the direct and full requisite sets for each activity are derived from a given set of activities and their input and output data. The activity relations between each pair of activities can be derived by applying Lemmas 9-11 and Theorems 4-7. A partial relation matrix is used to represent the activity relations.

Step 2. Generate a correct workflow model without considering parallelism. At this step we ignore parallelism by temporarily replacing activity relation "∧" with activity relation "*". Then, activities that can be grouped into sequential inline blocks are replaced with block activities. Furthermore, we determine where to add XORSplits by applying Lemma 12 and update the partial relation matrix by adding the new XORSplits into the matrix. At the end of this step, we replace all sequential inline blocks with the original activities and create an intermediate workflow model.

Step 3. Make the workflow model more efficient by adding parallelism. We can again simplify the workflow design by means of sequential inline blocks as needed. We determine where to place ANDSplits according to Theorem 7. At the end of this step, all sequential inline blocks are replaced with the original activities. The result of this step is a non-standard model without XORJoins and ANDJoins.

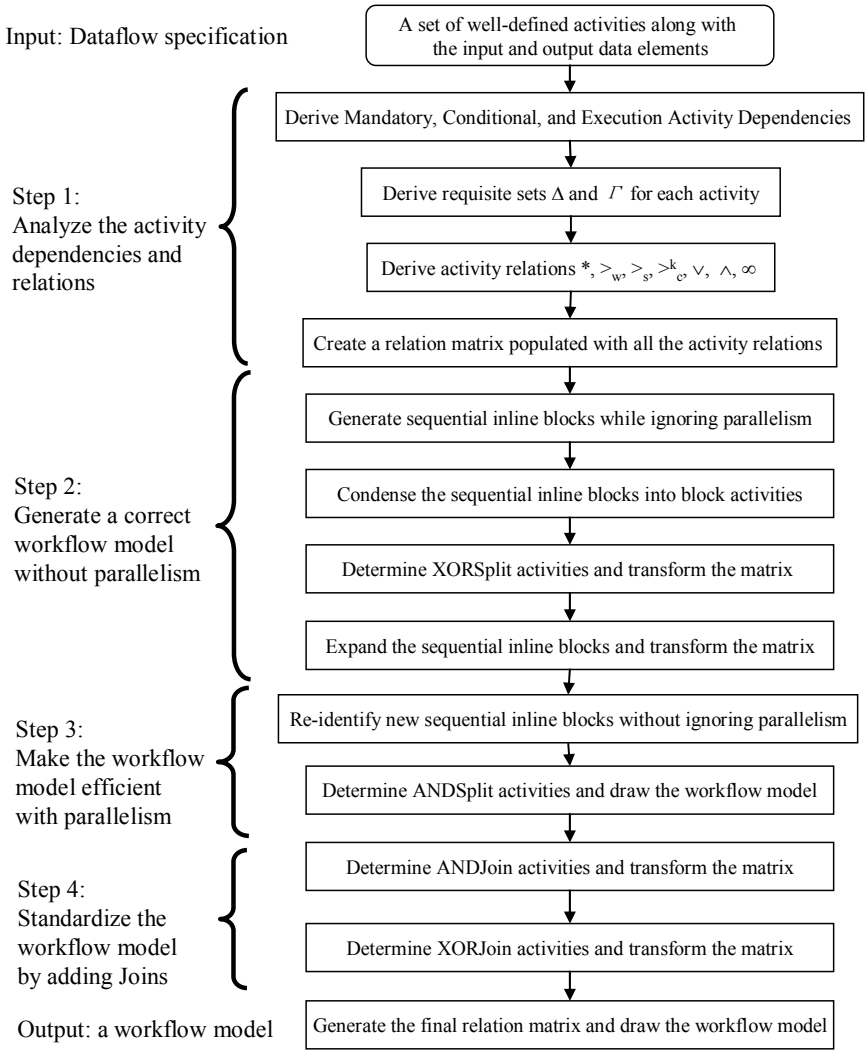


Figure 12. A Procedure for Workflow Design

Step 4. Standardize the model by first adding ANDJoins and then XORJoins. Each necessary join should be added according to Lemmas 13 and 14. After this step, we will have a standard workflow model.

Next, we use an example to illustrate that the procedure can be used to identify complex workflow models that contain overlapping structures where XORSplits and XORJoins are intertwined with ANDSplits and ANDJoins (Bi and Zhao 2004a and 2004b, Liu and Kumar, 2005).

5.4.2 Design an Workflow Model with Overlapping Structures

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	e
s	*										
v_1	NA	$>^1_c$	$>^2_c$								
v_2		NA	$\vee$	*	*	∞	∞				
v_3		$\vee$	NA	∞	∞	*	*				
v_4			∞	NA	$\wedge$	∞	∞	*	∞		
v_5			∞	$\wedge$	NA	∞	∞	∞	*		
v_6		∞		∞	∞	NA	$\wedge$	*	∞		
v_7		∞		∞	∞	$\wedge$	NA	∞	*		
v_8					∞		∞	NA	∞	*	
v_9				∞		∞		∞	NA	*	
v_{10}										NA	*

Table 17. The Partial Relation Matrix for a Workflow with Overlapping Structures

Table 17 shows a partial relation matrix with two mutually exclusive routing conditions, c_1 and c_2. For simplicity, we skip the analysis of data dependencies and the deviation of activity relations and show that a workflow model with overlapping structures can be

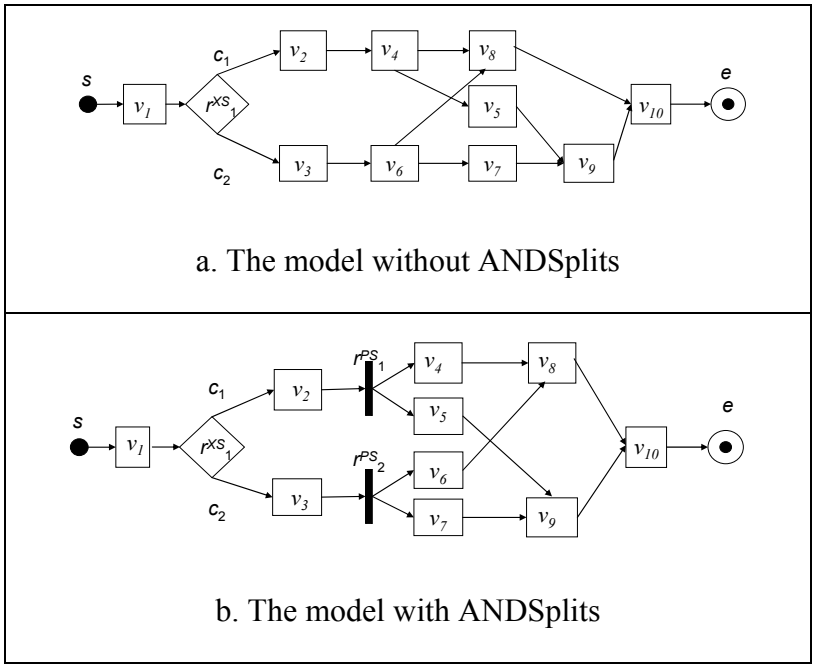
generated from Table 17 as follows.

Stage 1: Generate a Correct Workflow Model without Parallelism

At this step, we omit parallelism by replacing $v_4 \wedge v_5$ and $v_6 \wedge v_7$ with $v_4 * v_5$ and $v_6 * v_7$ and ignoring $v_2 * v_5$ and $v_3 * v_7$, which results in $v_2 * v_4$, $v_4 * v_5$ and $v_3 * v_6$, $v_6 * v_7$, respectively. Now, since we have $v_1 >^1_c v_2$, $v_1 >^2_c v_3$, and $v_2 \vee v_3$, by Lemma 12, an XORSplit is placed between v_1 and $\{v_2, v_3\}$, thus leading to Figure 13a.

Stage 2: Add Parallelism to Achieve Efficiency

Given $v_4 \wedge v_5$, $v_2 * v_4$, $v_2 * v_5$ (Table 17), we need an ANDSplit between v_2 and $\{v_4, v_5\}$. Given $v_6 \wedge v_7$, $v_3 * v_6$, $v_3 * v_7$ (Table 17), we need an ANDSplit between v_3 and $\{v_6, v_7\}$, resulting in Figure 13b.



a. The model without ANDSplits

b. The model with ANDSplits

Figure 13. Models for the Workflow with Overlapping Structures

Stage 3: Standardize the Model by Adding ANDJoins and XORJoins

Next, we add Joins before v_8, v_9, and v_{10}. Given that $\{v_8, v_9\} \subset \bullet v_{10}$ and $\sigma(s, v_8) \cap \sigma(s, v_9) = \sigma(s, r^{PS}_1)$ or $\sigma(s, r^{PS}_2)$, by Lemma 13, an ANDJoin activity r^{PJ} is needed between $\{v_8, v_9\}$ and v_{10}. Given $\{v_4, v_6\} \subset \bullet v_8$ and $\sigma(s, v_4) \cap \sigma(s, v_6) = \sigma(s, r^{xS}_1)$, by Lemma 14, XORJoin r^{XJ}_1 is added between $\{v_4, v_6\}$ and v_8. By the same token, r^{XJ}_2 is added between $\{v_5, v_7\}$ and v_9. Thus, we have Figure 14.

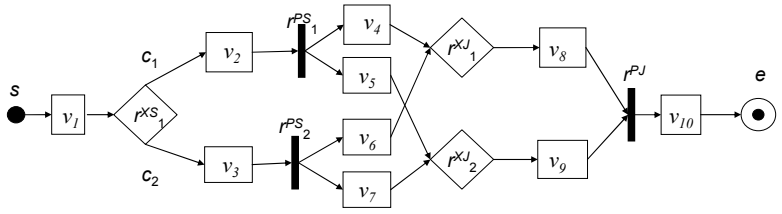


Figure 14. The Final Model for the Workflow with Overlapping Structures

5.5 Conclusions

In this chapter, we proposed the concept of activity relations and advocated the use of activity relations in designing workflow models through dataflow analysis. Furthermore, we presented a formal workflow design methodology in which the activity relations are derived from dataflow first and then the possible control flow structures are identified. We demonstrated that the design procedure is capable for generating complex workflow structures, including overlapping structures. The core part of our design approach is theoretically proved to be correct. In our approach, not only sequential structures but also parallel and conditional routing can be created through a formal procedure, leading to more rigorous workflow design.

6 IMPLEMENTING THE DEPENDENCY ANALYSIS BASED APPROACH TO WORKFLOW DESIGN

In this chapter, we present details on how to implement the dependency based workflow design. The implementation of the proposed method has three steps, requirements collection, requirement analysis, and workflow design. At the step of requirements collection, the concept of activity dependency tree is introduced for identifying the set of business activities involved in a process and the concept of condition-action table is introduced for routing conditions analysis. Various algorithms are presented for automating requirements analysis and workflow design. Furthermore, we propose a component based architecture for implementation. This chapter can be used as a roadmap for implementing the dependency analysis based workflow design to solve problems in practice.

6.1 A Framework for Workflow Design Based on Dependency Analysis

This section introduces a framework for implementing the dependency based workflow design. As shown in Figure 15, this framework consists of three major steps: 1) requirements collection, 2) requirements analysis, and 3) workflow design. The details of each step are described in the following sections.

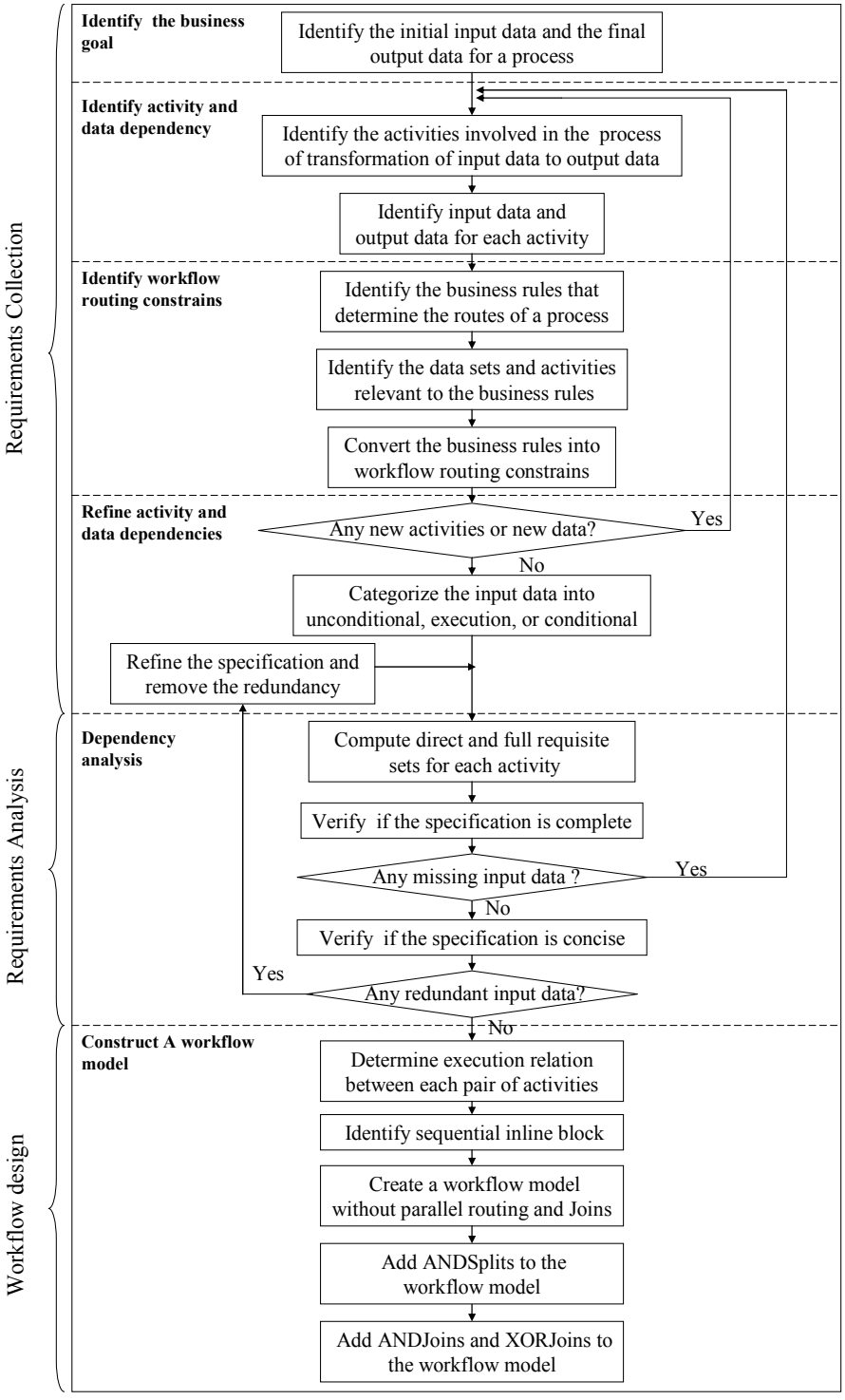


Figure 15. A Framework for Workflow Design Based on Dependency Analysis

6.2 Requirements Collection

This step aims to answer two questions: what activities should be included in a workflow model? What are the input and output data for each activity? To answer these two questions, we first determine the overall goal of a workflow process, from which we outline the set of activities that can be organized to achieve the goal and list the input data and output data of each activity. Then we analyze the business rules relevant to process routing and refine the set of activities according to the business rules.

6.2.1 Analyze Business Goal and Data and Activity Dependencies

From the data processing perspective, the goal of a workflow is to transform the set of input data available at the beginning of a workflow to the set of output data in need. For example, an insurance claim process takes customer's name, insurance policy number, damage cost as input and generates a data output that states the claim is either approved or rejected. As such, the initial input data and the final output data need to be decided at the beginning of the workflow design process. The initial input data and the final output data for a workflow process can be identified through collecting and analyzing the existing paperwork, including forms, documents, and product specifications and having meetings and interviews with people involved in the existing workflow process. In case that a process starts without any initial input data set, the first activity that produces the initial data set has to be identified first.

Once the initial input and final output data are known, the set of activities, which transform the initial input data into the final output data, can be traced in three steps: 1) The set of activities V that directly produce the final output data is identified first; 2) If

any activity in *V* needs some input data *d* neither provided by the initial input data set nor produced by any other activities in *V*, the activity producing data *d* as output needs to be identified and included in *V*; 3) repeat 2) until no more activities can be added into *V*.

An activity dependency tree can be used to facilitate the identification of the set of activities that need to be included in a workflow model.

Definition 29 (Activity dependency tree) An activity dependency tree *T* is a graphic structure where the root node represents the end activity (i.e., the last activity in a workflow model), all other nodes represent different activities that produce some output data used as input by their parent nodes.

An activity dependency tree can be created using the following procedure

(1) create the root node *e*;

(2) find all the set of activities *V* that directly produce the final output data;

(3) create a node for each activity in *V* and attach these nodes as the child nodes of *e* to the tree;

(4) for each activity *v* in *V*, if the input data required by *v* is produced by another activity *u* that does not have a corresponding node in the tree, create a node for *u* and attach the node as a child node of *v* to the tree;

(5) then add *u* to *V*

(6) repeat (4) and (5) until no more nodes can be added into the tree.

It is worth noting that the above procedure to create an activity dependency tree is very similar to the steps we need to identify the set activities to be included in a workflow model. An activity dependency tree essentially provides a visual tool to show the activity

dependency though data. Furthermore, activity dependency tree can assure that all the identified activities can be linked together through data and activity dependencies. This is important because it guarantees that all the identified activities can be added to one workflow model and they will not form unrelated groups when the design algorithms presented later in this chapter are applied. Note that when activities *u* and *v* both use some data produced by activity *x*, there can be different activity dependency trees since activity *x* can be the child node of either activity *u* or activity *v*.

We illustrate how to create an activity dependency tree through an order processing example, in which after a retailer receives an order from customer, the retailer processes the order and send confirmation to the customer. The initial input data for the process include *customer's name*, *product name*, *product quantity*, and *payment method*, which are produced by the activity *submit an order*. The final output is either an approval or a rejection, which is referred to as *order confirmation*. The order confirmation is actually the output data produced by the activity *sale manager approval*, which takes *product availability* and *customer's credit rating* as input. Furthermore, we identify the two activities, *check inventory* and *verify customer's credit*, produce *product availability* and *customer's credit rating* as output respectively. Given that the activity *check inventory* takes *product name* and *product quantity* as input, the activity *verify customer's credit* takes *customer's name* and *payment method* as input, and all these data items are provided as the initial input by the activity *submit an order*, we can create the activity dependency tree as shown in Figure 16.

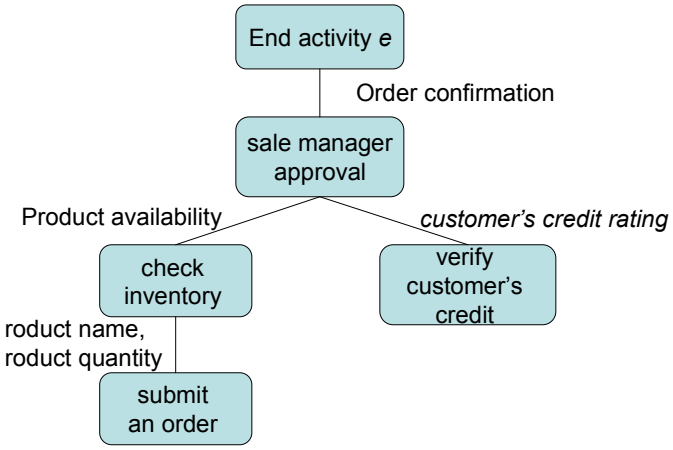


Figure 16. An Example of Activity Dependency Tree

6.2.2 Identify Workflow Routing Conditions and Refine Data and Activity Dependencies

Different workflow instances can take different paths under the guidance of business rules. To design a workflow, it is important to collect those business rules and then format them into workflow routing conditions. Business rules can be identified by interviews or extracted from existing policies (Wang and Zhao 2005). To format a business rules in plain English into workflow routing conditions, we can use a Condition-Action table, where a workflow routing condition is expressed as

Condition: Criteria(*D*) *Action:* Execute(*V*)

and *D* is a set of data, and *Criteria(D)* is a logical expression of *D*. The routing condition above means that when the set of data *D* meets certain criteria, the set of activity *V* is executed. Each data item in the data set *D* can be binary, categorical, or numerical data. That is, the criteria for *D* can be written a Boolean expression such as "$x = a$" or "$a \leq y < b$". For example, the business rules in an order processing example (Table 18) can be formatted into the Condition-Action table show in Table 19.

Rule 1	If a customer orders more than what is available, a replenish order should be sent to the manufacturer and a back order notice should be sent to the customer.
Rule 2	If there are enough products in the inventory, the customer's credit will be checked.
Rule 3	If the credit rating of a customer is good, the order needs to be approved by a sales manager

Table 18. Business Rules for the Order Processing Example

Routing Constraint	Condition	Action
R1	*Product availability* = "*No*"	*Execute*(*send replenish order*, *send back order notice*)
R2	*Product availability* = "*Yes*"	*Execute*(*verify customer's credit*)
R3	*customer's credit rating= "good"*	*Execute*(*sale's manager's approval*)

Table 19. The Condition-Action Table for The Order Processing Example

As defined in Definition 15, if the data item is included in a routing condition that determines when an activity should be executed, then the activity has an execution dependency on that data item. Analysis of routing conditions can help refine input and output data and further identify the set of activities to be included in a workflow model. From Table 19, we can identify two more activities for the ordering processing example, which are not included in the activity dependency tree in Figure 16. They are *send replenish order* and *send back order notice*. The output from these two activities, e.g., *replenishment quantity* and *back order notice status*, should be added to the final output data set produced by the workflow. Then the activity dependency tree in Figure 16 needs to be modified.

When a set of activities is well specified, then the input data required by each activity can be grouped into three categories, mandatory, conditional, and execution data input depending on the roles the data items play as input. Accurate categorization of data

dependencies in accordance with Definitions 6, 7 and 15 is fundamental to workflow design.

6.3 Requirements Analysis

At this step, activity dependencies are derived from the data dependencies identified in the previous step and then the direct and full requisite sets can be constructed for each activity. Moreover, given the set of activities and their data dependencies identified at the previous step, we need to make sure that adequate but succinct information has been collected for workflow design. Two measures are used to verify the correctness of the specification for workflow design, i.e., completeness and conciseness. As defined in Definition 27, completeness computes if the set of activities can produce the input data needed in the workflow. As defined in Definition 28, conciseness determines if the set of activities produce more data output than needed. If a specification for workflow design is not complete or concise, the step of requirement collection may be repeated in order to modify the specification, identify the activities that have been missed, and remove the redundant activities that produce useless data.

```
Procedure Construct_Direct_Requisite _Set
Input:  A set of activities V and the input data set and output data set of each v in V
for each activity v in V
        for each input data d required by v
                activity temp=findOutputActivity(d)
                  //Find the activity that produces d as output
                        if (temp≠null)  then //   the activity producing d as output can be found
                                add temp to the direct requisite set of v
                                if (d is mandatory input of v)
                                        then set dependencyc(v, temp)= "m"
                                if (d is conditional input of v)
                                        then set dependency(v, temp)= "c"
                                if (d is execution input of v)
                                        then set dependency(v, temp)= "e"
```

Figure 17. The Algorithm of Construct Direct Requisite Sets for a Set of Activities

```
Procedure Construct_Full_Requisite _Set

Input:  A set of activities V and the direct requisite set of each v in V
for each activity v in V
        create a stack
        for activity v in V
                set identified(v) = false
        for each activity u in the direct requisite set of v //initialize the stack
                set identified(u) = true
                stack.push(u)
        while (stack is not empty) do
                u=stack.pop()
                if(direct requisite set of u contain activities) then
                        for each activity z in the direct requisite set of u
                                if(not identified(z)) then
                                        stack.push(z)
                                        set identified(z) = true
                else add u to fullRequisiteSet(v)
```

Figure 18. The Algorithm of Construct Full Requisite Sets for a Set of Activities

We provide four algorithms to automate the step of requirements analysis. The procedure Construct_Direct_Requisite _Set shown in Figure 17 can help to automatically create direct requisite sets for a set of activities and categorize different types of activity dependency. For each data item *d* needed as input by an activity, the procedure tries to find the activity that produces data item *d* as output and create a dependency link between these two activities.

In order to create the full requisite sets, the procedure Construct_Full_ Requisite _Set shown in Figure 18 starts with a set of activities and their direct requisite sets and applies a depth first search in four steps. For each activity *v*, 1) the procedure marks the activities in the direct requisite set of *v* as identified and push them into a stack; 2) An activity *u* is popped from the stack and the procedure examines the direct requisite set of *u* if *u* has a

non-empty direct requisite set; 3) any activity *z* in the direct requisite set of *u* is marked as identified and pushed into the stack if *z* has not been marked as identified; 4) repeat the steps 2) and 3) until no more activities can be obtained from the stack.

```
Procedure Verify_Completeness
 Input:  A set of activities V and the direct requisite set of each v in V
 for each activity v in V
        for each input data d required by v
                set outputActivity(d) = "unknown"
                        //mark the activity producing d as unkown
for each activity v in V
        for each input data d required by v
                if(outputActivity(d) = "unknown") then
                        activity temp=findOutputActivity(d, directRequisiteSet(v))
                          //Find the activity that produces d as output from the
                            direct requisite set of v
                        if (temp=null)  then
                                set  outputActivity(d) = "none"
                                // the specification is not complete
```

Figure 19. The Algorithm of Verifying Completeness

The procedure Verify_Completeness shown in Figure 19 decides if the set activities can produce all the required input data, i.e., the completeness of the set of activities. At the beginning, the procedure labels the activities that produce the required input data as unknown. Then the procedure examines the direct requisite set of each activity and then changes the corresponding label to none if the activity that produces a required input data cannot be found.

The procedure Verify_Conciseness shown in Figure 20 first examines whether each output data item has been required either as input for an activity or final output from the workflow. Then the procedure compares all the output data and determines if a data item

has been produced multiple times. An error message is printed out if an output data does not contribute to the production of final output data or a data item is produced more than once.

```
Procedure Verify_Conciseness
 Input:  A set of activities V , the input data set and output data set of each v in V,
   and the final output O
 for each activity v in V
        for each output data d_v required by v
                activity temp=findInputActivity(d_v) //Find the activity that uses d as input
                if (temp=null and d_v is not included in O)
                        //   the activity producing d as output can be found
                then  print "no activity uses d as input"
                for each activity u in V and u≠v
                        for each output data d_u required by u
                                if (d_v is the same as d_u) //two activities produce the same data
                                then print "d_v produced by v is the same as d_u produced by u"
```

Figure 20. The Algorithm of Verifying Conciseness

6.4 Workflow Design

This section discusses how to generate a workflow model from activity and data dependencies resulted from the requirements collection and requirements analysis.

6.4.1 Identification of Activity Relation

Five basic workflow constructs, i.e., sequence, AND-Split, AND-Join, XOR-Split, and XOR-Join, have been defined as the essential workflow primitives by Workflow Management Coalition (WfMC). In order to create a workflow model, we need to answer a fundamental question, i.e. when a basic construct can be used. This section presents the algorithms for determining where to use sequence, XOR-Split, and AND-Split according to activity and data dependencies. Table 20 shows the design principles based on activity

dependency analysis and the corresponding algorithms that implement those principles. For details of the design principles, please refer to Chapter 5.

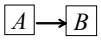
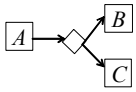
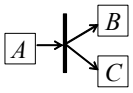
Workflow Construct	Activity Relation	Activity dependencies	Implementation Procedure
 (a) Sequence	*A* immediately precedes *B*	*B* has a mandatory but not execution dependency on *A* and there is not such an activity *C* such that *B* depends on *C* and *C* depends on *A* (Theorem 4)	IdentifyImmediateSequence (Shown in Figure 21)
 (b) XORSplit	*A* conditionally precedes *B*	*B* has an execution dependency on *A*. Further, *B* only has an execution dependency on any activity that is included in the full requisite set of *B* nut not included in the full requisite set of *A* (Theorem 5)	IdentifyCondtitionalPrecedence (Shown in Figure 22)
	B XOR-parallels with *C*	*B* and *C* have the same full requisite set and the routing conditionsfor *B* and *C* cannot be both true or both false (Theorem 6)	IdentifyXORParallel (Shown in Figure 23)
 (d) ANDSplit	*B* AND-parallels with *C*[5]	*B* and *C* have the same full requisite set. Moreover, no routing constraint keeps *B* and *C* from being both executed (Theorem 7)	IdentifyANDParallel (Shown in Figure 24)

Table 20. Workflow Constructs, Design Principles, and Implementation Algorithms

[5] We can consider the activity relation between *A* and *B* as *A* immediately precedes *B*.

```
Procedure IdentifyImmediateSequence
Input:  A set of activities V, the set of activity dependencies identified in V, the direct requisite
        set of each v in V, the full requisite set of each v in V
for each activity v in V
        for each activity u in the direct requisite set of v
                if (dependency(v, u)="m") then
                        set identified = true
                        for each activity z ≠(v or u) in V
                                if (identified and (u is included in the full requisite set of z)
                                  and ( z is included in the  full requisite set of v))   then
                                        set identified = false
                        if(identified)  then
                                        Sequence s=createSequence(u, v)
                                        sequenceSet. add(s)
```

Figure 21. The Algorithm of Identifying Immediate Precedence

```
Procedure IdentifyCondtitionalPrecedence
Input:  A set of activities V , the set of activity dependencies identified in V, the direct
         requisite set of each v in V, the full requisite set of each v in V
for each activity v in V
        for each activity u in the direct requisite set of v
                if (dependency(v, u)="e") then
                        activitySet=difference(fullRequisiteSet(v), fullRequisiteSet(u))
                        set identified = true
                        for each activity z in activitySet
                                if (identified and dependency(v, z)≠ "e") then
                                        set identified = false
                        if(identified)  then
                                        ConditionalSequence s=createConditionalSequence(u, v)
                                        conditionalSequenceSet. add(s)
```

Figure 22. The Algorithm of Identifying Conditional Precedence

```
Procedure IdentifyXORParallel
Input:  A set of activities V , the set of activity dependencies identified in V, the full requisite
        set of each v in V, the routing conditionsset R
for each activity v in V
        for each activity u≠ v in V
                if (fullRequisiteSet(v).equalsTo(fullRequisiteSet(u))) then
                        set identifiedXOR = true
                        conditionset_v=findConditions(R, v)
                          // find all the conditions related to activity v from R
                        conditionSet_u=findConditions(R, u)
                          // find all the conditions related to activity u from R
```

```
                    for each r_v in conditionSet_v
                            for each r_u in conditionset_u
                                    if (not r_v.isMutuallyExclusive(r_u)) then
                                            if (identifiedXOR) then set identifiedXOR =
                    false
                    if (identifiedXOR) then
                                    XORParalelle p=createXORParalelle (u, v)
                                    XORParalelleSet. add(p)
```

Figure 23. The Algorithm of Identifying XOR-Parallel

```
Procedure IdentifyANDParallel

Input: A set of activities V , the set of activity dependencies identified in V, the full requisite
 set of each v in V, the routing conditionsset R
for each activity v in V
        for each activity u≠v in V
                if (fullRequisiteSet(v).equalsTo(fullRequisiteSet(u))) then
                        set identifiedAND = true
                        conditionSet_v=findConditions(R, v)
                          // find all the conditions related to activity v from R
                        conditionSet_u=findConditions(R, u)
                          // find all the conditions related to activity u from R
                        for each r_v in conditionSet_v
                                for each r_u in condiitonSet_u
                                        if (r_v.isMutuallyExclusive(r_u)) then
                                                if (identifiedAND)
                                                        then set identifiedAND = false
                        if (identifiedAND) then
                                ANDParalelle q=createANDParalelle (u, v)
                                ANDParalelleSet. add(q)
```

Figure 24. The Algorithm of Identifying AND-Paraellel

6.4.2 Identification of Sequential Inline Blocks

Identification of sequential inline blocks can help simplify workflow design. Given a set of immediate sequential relations, Sequential Inline Blocks can be created as follows. If we know activity *u* immediately precedes activity *v* and only activity *v* and no other activities immediately precedes *v*, then activities *u* and *v* forms a smallest sequential inline block. Further, if we know *v* immediately precedes activity *z* and only activity *z*

```
Procedure IdentifyInlineBlocks
Input:  A set of activities V, the set of immediate precedence PRE identified in V, and a
        particular immediate precedence pre(u, v) where u and v are both included in V
set inblockFound= true
for each pre(x, y)≠ pre(u, v) in PRE
        if(y equals to v or x equals to u )
                //examine if there is any other activity that immediately precedes v or
                is immediately preceded by u
                        set inblockFound=false
                        break
if (inblockFound) then
        set start=u and end end=v
        inlineBlock=createInlineBlock(start, end)
        // create an inline block starting from u and ending with v
        set blockGrowing= true, blockGrowingAtFront=true, and blockGrowingAtFront=true
        while (blockGrowing)
                set blockGrowing=false
                if(blockGrowingAtFront) then
                        set blockGrowingAtFront=false
                        activitySet=findAllPrecedingActivities(PRE, start)
                        // find all the activities that immediately precedes the start activity
                        if (activitySet has one element) then
                                set blockGrowingAtFront=true
                                start=getActivity(activitySet)
                                addActivityToFront(inlineBlock, start)
                        // add the activity immediately preceding u to the inline block
                if (blockGrowingAtEnd) then
                        set blockGrowingAtEnd=false
                        activitySet=findAllFollowingActivities(PRE, end)
                        // find all the activities that immediately follows the end activity
                        if (activitySet has one element) then
                                set blockGrowingAtEnd=true
                                end=getActivity(activitySet)
                                addActivityToEnd(inlineBlock, end)
                                // add the activity immediately following v to the end of the inline
                        block
                if(blockGrowingAtFront or blockGrowingAtEnd) then set blockGrowing=true
```

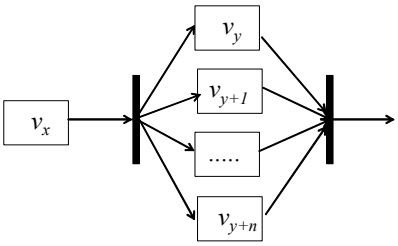
Figure 25. The Algorithm of Identifying Inline Blocks

and no other activities immediately precedes z, then activities u, v, and z form a sequential inline block. Figure 25 shows an algorithm, which help identify an inline

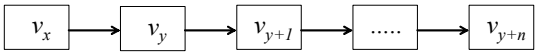
block from a set of immediate precedence relations given that activity *u* immediately precedes activity *v*. Note once a sequential inline block has been identified, the set of activity included in the block can be replaced by a block activity and the original set of activities identified at the steps of requirements collection and analysis can be simplified. Then the activity relation between the block activity and other activities can be derived by applying the relation aggregation property defined in Definition 26.

6.4.3 Create a Workflow Model without Parallelism and Joins

This section first presents two algorithms. One is for replacing AND-parallel and the other is for creating XOR-Splits. Then we describe the algorithm of creating a workflow model without parallelism and joins.



A. Activities in parallel execution



B. Activities in sequential execution

Figure 26. An Example for Replacing Parallelism with Sequential Execution

In Figure 26A, v_y AND-parallels with v_{y+1}, v_{y+2}, …, v_{y+n}. When we replace the parallel

execution with sequential execution, the AND-parallel relations between v_{y+i} and v_{y+i+1} is replaced by an immediate precedence relation. Moreover, the immediate precedence relation between v_x and v_{y+i} is ignored except for v_y, which immediately follows v_x in the sequential execution in Figure 26B. This is how the algorithm of replacing AND-Parallel (Figure 27) works. Note copies are made in order to keep the original immediate precedence relations and AND-Parallel relations since they are used later when AND-Splits are added into a workflow model.

```
Procedure RepalceANDParallel
Input:  A set of activities V, the set of immediate precedence PRE and the set of
        AND-Parallel PAR identified in V
copy PRE to PRE' and copy PAR to PAR'
for each activity v in V
        if (PAR' is not empty)
                activitySet=findActivities(PAR', v)
                //find all the activities that are in parallel with v
                if (activitySet is not empty) then
                        start=findStartActivity(PRE, v)
                                //find the activity immediately preceding v
                        second=v
                        for each activity u in activitySet
                                pre(second, u)=createPrecedence(second, u)
                                //create an immediate precedence relation
                                //between the second activity and u
                                add pre(second, u) into PRE'
                                delete par(start, u) from PAR'
                                set second=u
```

Figure 27. The Algorithm of Replacing AND-Parallel

The algorithm of creating XORSplits is developed based on the XORSplit rule proved in Lemma 12. It works as follows. Given that activity v conditional precedes activity u, an XORSplit r is placed between v and u. Then the algorithm finds the set of activities V that are conditionally preceded by activity v and may be executed whenever u is not executed. Moreover, the activities in V have to be mutually exclusive, i.e., only one of

them can be executed under a routing condition. The algorithm will set each activity in *V* to follow the XORSplit *r* immediately, i.e., the immediate precedence relation holds true for each activity in *V* and the XORSplit *r*. Figure 28 shows the details of this procedure.

```
Procedure CreateXORSplits
Input:  A set of activities V, the set of immediate precedence PRE identified in V,
        the set of conditional precedence CONDTIONAL_PRE identified in V, and
        the set of XOR-Parallel XOR_PAR
create a routingActivitySet R
for each activity v in V
        if (CONDTIONAL_PRE  is not empty)
                activitySet=findConditionalActivities(CONDTIONAL_PRE, v)
                        //find all the activities preceded conditionally by v
                while (activitySet is not empty) do
                        u=getActivity(activitySet)
                        // get an activity from activitySet and then remove that activity from
                activitySet
                        r=CreatXORSplit()
                        add r to R
                        pre(v, r)=CreateImmediatePrecedence(v, r)
                        pre(r, u)= CreateImmediatePrecedence(r, u)
                        add pre(v, r) and pre(r, u) to PRE
                        remove conditional_pre(v, u) from CONDTIONAL_PRE
                        mutuallyExclusiveActivitySet=findMutuallyExclusiveActivities(u,
                          CONDTIONAL_PRE )
                        //find from activitySet the mutually exclusive activities that may be
                          executed when u is not executed
                        for each x in mutuallyExclusiveActivitySet
                                pre(r, x)= CreateImmediatePrecedence(r, u)
                                add pre(r, x) to PRE
                                remove conditional_pre(v, x) from CONDTIONAL_PRE
if (RoutingActivitySet is not empty)  then add all elements in RoutingActivitySet to V
```

Figure 28. The Algorithm of Creating XORSplits

Now that the most important pieces that constitute the procedure of creating a workflow model without parallelism and Joins have been discussed, we present the overall procedure as shown in Figure 29. Note that the replacement of AND-Parallel relations with immediate precedence relations can result in more sequential inline blocks and inline blocks containing more activities. As such, for the purpose of further

simplification, the procedure *IdentifyInlineBlocks* needs to be repeated after the replacement of AND-Parallel relations. Then the procedure *CreateXORSplit* is called to create a set of XORSplits based on the conditional precedence identified by the procedure *IdentifyCondtitionalPrecedence* and the XOR-parallel relations identified by the procedure *IdentifyXORParallel*. At the end of this step, we can add all the links as follows. If activity *v* immediately precedes activity *u*, a link from *v* to *u* is added.

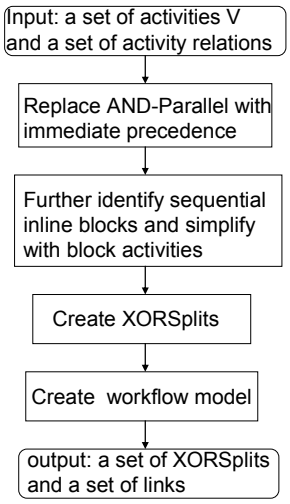


Figure 29. The High Level Procedure of Creating a Workflow Model without Parallelism and Joins

6.4.4 Add AND-Splits

Figure 30 shows the algorithm of adding AND-Split, which examines the sequential inline blocks resulting from the replacement of AND-Parallel relations with immediate precedence relations. Note that AND-Splits can be added only in those inline blocks. If an activity *v* is in AND-Parallel with other activities, the algorithm creates an ANDSplit, places the ANDSplit between *v* and the activity immediately preceding *v*, and updates the activity relations accordingly. Further, the algorithm sets all the activities in AND-

Parallel with *v* to be immediately preceded by the ANDSplit.

```
Procedure CreateANDSplits
Input:  A set of activities V, the set of immediate precedence PRE identified in V, the set
        of AND-Parallel PAR identified in V, and the set of sequential inline blocks BLOCKSET
BlockSetToBeChanged = findInlineBlocks(BLOCKSET)
         //find all the inline blocks where AND-Parallels were replaced
for each inline block block in BlockSetToBeChanged
         for each activity v in block
                  if (PAR  is not empty)
                           activitySet=findConditionalActivities(PAR, v)
                           //find all the activities AND-parallel with v
                           if (activitySet is not empty) then
                                    leadingActivty=findLeadingActivity(PRE, v)
                                    //find the activity which immediate precedes v
                                    r=CreatANDSplit()
                                    add r to RoutingActivitySet
                                    pre(leadingActivity,
                           r)=CreateImmediatePrecedence(leadingActivity, r)
                                    pre(r, v)=CreateImmediatePrecedence(r, v)
                                    add pre(leadingActivity, r) and pre(r, v) to PRE
                                    remove pre(leadingActivity, v) from PRE
                                    for each activity u in activitySet
                                             pre(r, u)= CreateImmediatePrecedence(r, u)
                                             add pre(r, u) to PRE
                                             remove pre(leadingActiivty, u) from PRE
                                             remove par(v, u) from PAR
if (RoutingActivitySet is not empty)  then add all elements in RoutingActivitySet to V
```

Figure 30. The Algorithm of Adding ANDSplits

6.6.5 Add AND-Joins and XOR-Joins

If an activity is preceded immediately by more than one activity, a Join should be added. Figure 31 shows the algorithm of adding ANDJoins, which is developed based on the ANDJoin rule (Lemma 13). For each activity *v*, the algorithm will search for the activities that immediately precede *v*. If more than one activity immediately precedes *v*, then the algorithm compares the firing sequences of those activities immediately preceding *v*. If the last activity shared by the firing sequences is an AND-Split, the algorithm places an

AND-Join between *v* and the activities immediately preceding *v* and updates the activity relations accordingly.

```
Procedure CreateANDJoins
Input:  A set of activities V and the set of immediate precedence PRE identified in V
for each activity v in V
        activitySet=findPrecedingActivities(PRE, v)
         //find all the activities that immediately precede v
        set ANDJoinAdded=true
        while ((activitySet has more than one activity) and (ANDJoinAdded)) do
                set ANDJoinAdded=false
                activityPairSet=createPairs(activitySet)
                // find all different combinations of activities x and y in activitySet
                while (activityPairSet is not empty)
                        pair(x, y)=removePairs(activityPairSet)
                        //get a pair of two activities x and y
                        firing_SequenceSet(x)=findFiringSequences(PRE, x)
                        //find all the firing sequences to x
                        firing_SequenceSet(y)=findFiringSequences(PRE, y)
                        //find all the firing sequences to y
                        set ANDSplitFound=false
                        for each combination of firing sequence fs_x in firing_SequenceSet(x)
                          and firing sequence fs_y in firing_SequenceSet(y)
                                lastActivity=findLastActivity(fs_x, fs_y)
                                // find the last activity fs_x and fs_y share
                                if (lastActivity is an ANDSplit)
                                        set ANDSplitFound=true
                                        break
                        if (ANDSplitFound) then
                                set ANDJoinAdded=true
                                r=CreatANDJoin()
                                add r to V
                                pre(r, v)=CreateImmediatePrecedence(r, v)
                                pre(x, r) =CreateImmediatePrecedence(x, r)
                                pre(y, r)=CreateImmediatePrecedence(y, r)
                                add pre(r, v), pre(x, r), pre(y, r)  to PRE
                                remove pre(x, v) and pre(y, v)  from PRE
                                for each activity u in activitySet
                                        firing_SequenceSet(u)=findFiringSequences(PRE, u)
                                        //find all the firing sequences to u
                                        lastActivitySet=findLastActivity(fs_x,
                        firing_SequenceSet(u))
                                        //find the last activities that fs_x and any firing sequence
                                        to u shares
                                        if (lastActivity is included in lastActivitySet) then
                                                pre(u, r)=CreateImmediatePrecedence(u, r)
                                                add pre(u, r) to PRE
```

```
                                        remove pre(u, v) from PRE
                                        remove pair(x, u) and pair(y, u)
                                          from activityPairSet
        activitySet=findPrecedingActivities(PRE, v)
```

Figure 31. The Algorithm of Adding ANDJoins

Once all the AND-Joins have been added, the algorithm of adding XORJoins is very straight forward. If more than one activity immediately precedes *v*, then an XORJoin can be place right before *v*. Note according to the XORJoin rule (Lemma 14), the firing sequences of those activities immediately preceding *v* should be compared in order to determine whether an ANDJoin or XORJoin is needed. In the algorithm of adding XORJoins, this step is skipped since all the ANDJoins have been added if the procedure CreateANDJoins is called first. Figure 32 shows the algorithm of adding XORJoins.

```
Procedure CreateANDJoins
Input:  A set of activities V and the set of immediate precedence PRE identified in V
for each activity v in V
        activitySet=findPrecedingActivities(PRE, v)
        //find all the activities that immediately precede v
        if (activitySet has more than one activity) then
                r=CreatANDJoin()
                add r to V
                pre(r, v)=CreateImmediatePrecedence(r, v)
                for each activity u in activitySet
                        pre(u, r) =CreateImmediatePrecedence(x, r)
                        add pre(u, r)  to PRE
                        remove pre(u, v) from PRE
```

Figure 32. The Algorithm of Adding XORJoins

6.5 A Component Based System Architecture

Figure 33 gives an outline of the component based architecture for implementing the proposed approach, which consists of five main components: *Interface*, *Dataflow Verifier*, *Dependency Analyzer*, *Workflow Modeler*, and *Design Coordinator*. The functions of

these components are discussed below. Note that the arrows indicate the interaction among different components.

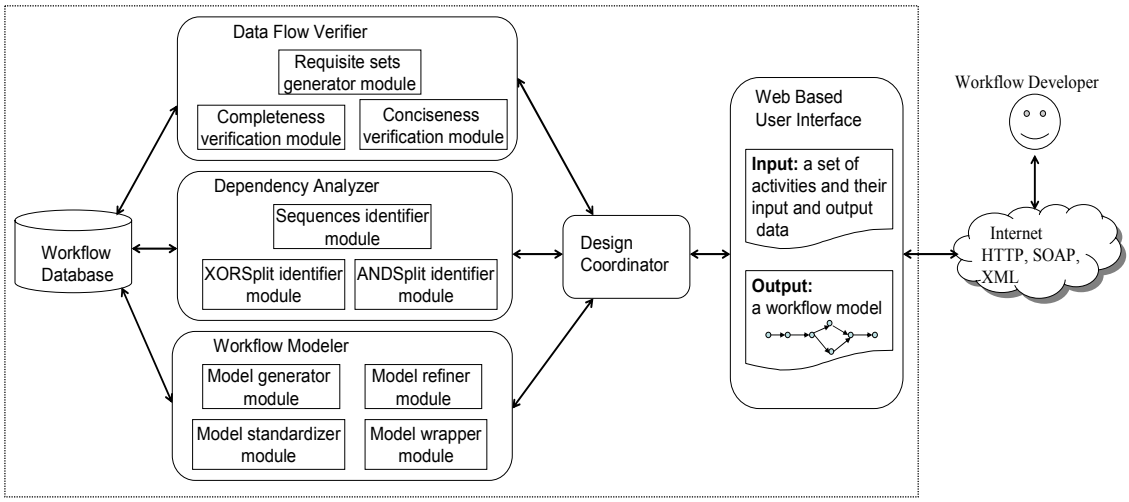


Figure 33. A High Level System Design of the Dependency Analysis Based Workflow Designer

- *Web Based User Interface*: an interface is provided for workflow developers to specify the set activities and their input and output data to the workflow designer. The interface sends the error message to workflow developers when the specification is incorrect. A workflow developer can access the interface through a web browser where the final model is displayed as a UML graph. The interface can also be invoked by a SOAP request and then the final model is sent back in a XML encoded file.

- *Dataflow Verifier*: the *Dataflow Verifier* examines whether the specification sent by a workflow developer, i.e., the set of activities and their input and output data, contains enough information for workflow design. It consists of three modules. The *requisite sets generator module* calculates the direct and full requisite sets for each activity. The *completeness verification module* analyzes the specification to assure all the input data

are provided either as the output from some activities or by external resources and there is no missing data. It implements the procedure *Verify_Completeness* (Figure 19). The *conciseness verification module* evaluates the specification for data redundancies, i.e., output data not required and repetitious production of the same data item. It implements the procedure *Verify_Conciseness* (Figure 20).

- *Dependency Analyzer:* The *Dependency Analyzer* analyzes the specification sent by a workflow developer and generates a set of activity relations that represent the possible execution steps in the workflow model. It has three modules: *sequences identifier module* helps determine the activities that can be executed sequentially, including sequential inline blocks; *XORSplit identifier module* and *ANDSplit identifier module* helps decide where to place XORSplit and ANDSplit..
- *Workflow Modeler*: The *Workflow Modeler* takes the set of activity relations as input and generates a standard workflow model. The *model generator module* is responsible for creating a basic workflow model without considering parallelism and Joins. The *model refiner module* incorporates parallelism into the workflow model. The *model standardizer module* identifies where ANDJoins and XORJoins are needed. The *model wrapper module* generates a XML document, which describes the structure of a final workflow model.
- *Design Coordinator:* the interactions among different components are directed by *Design Coordinator.*

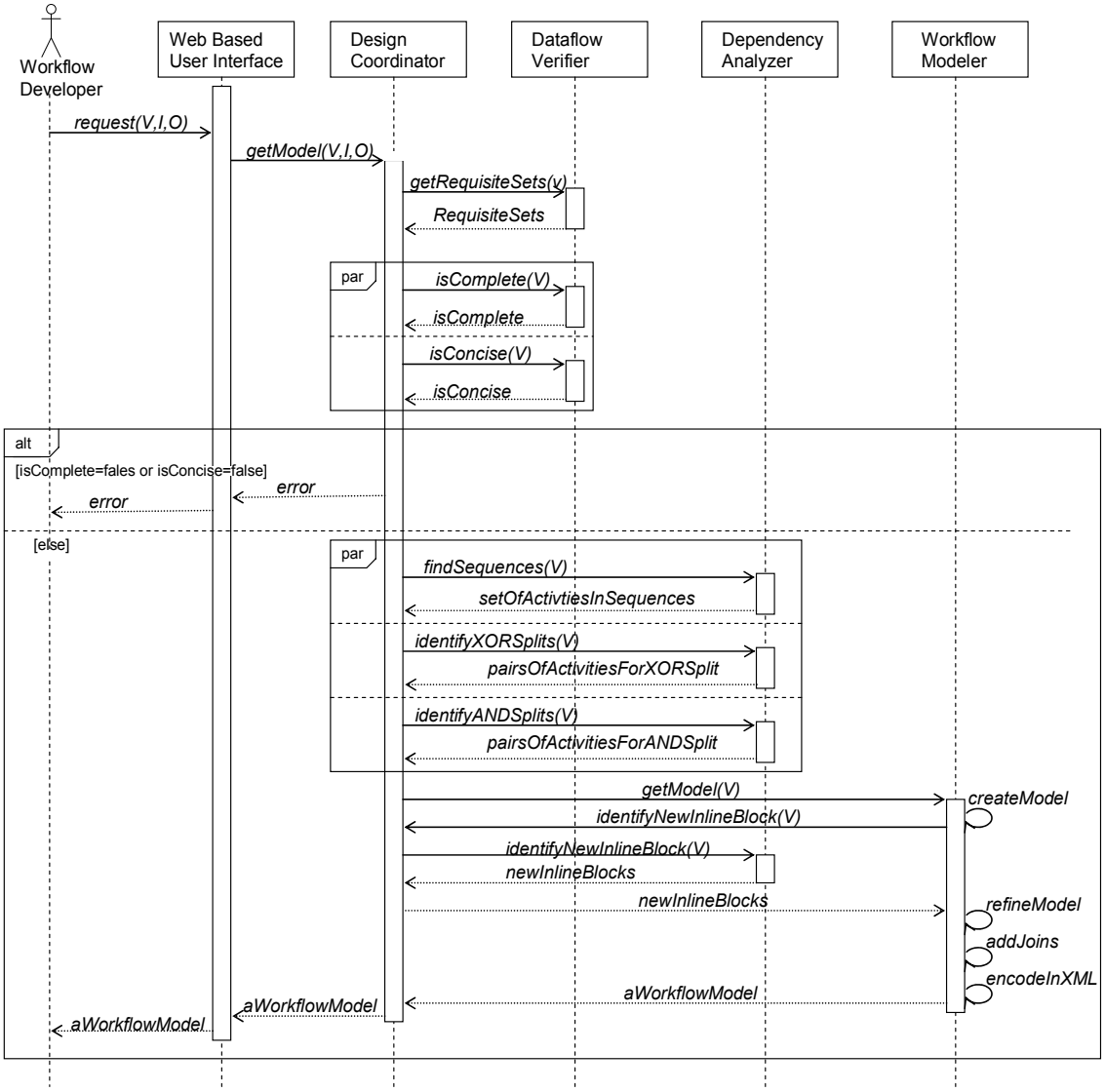


Figure 34. A UML Sequence Diagram Illustrating the Interaction among the Components of the Workflow Designer

A UML sequence diagram illustrating the interaction among the components of the workflow designer is shown in Figure 34. As shown in Figure 34, once the *Web Based User Interface* receives a request for generating a workflow model from a user, the request is then directed to the *Design Coordinator*, which is in charge of the activation of

various components in the system. Upon receiving a request from the interface, the *Design Coordinator* calls the *Dataflow Verifier* to compute the requisite sets for each activity and then verify whether the specification from a user is complete and concise. If the specification is not complete or concise, an error message is sent to the user through the *Web Based User Interface*. Otherwise, the *Design Coordinator* activates the *Dependency Analyzer* in order to identify sequential structures, parallel routing, and conditional routing. Then the *Workflow Modeler* is triggered to generate an overall workflow model and the model is sent back to the user by the *Design Coordinator* again through the *Web Based User Interface*.

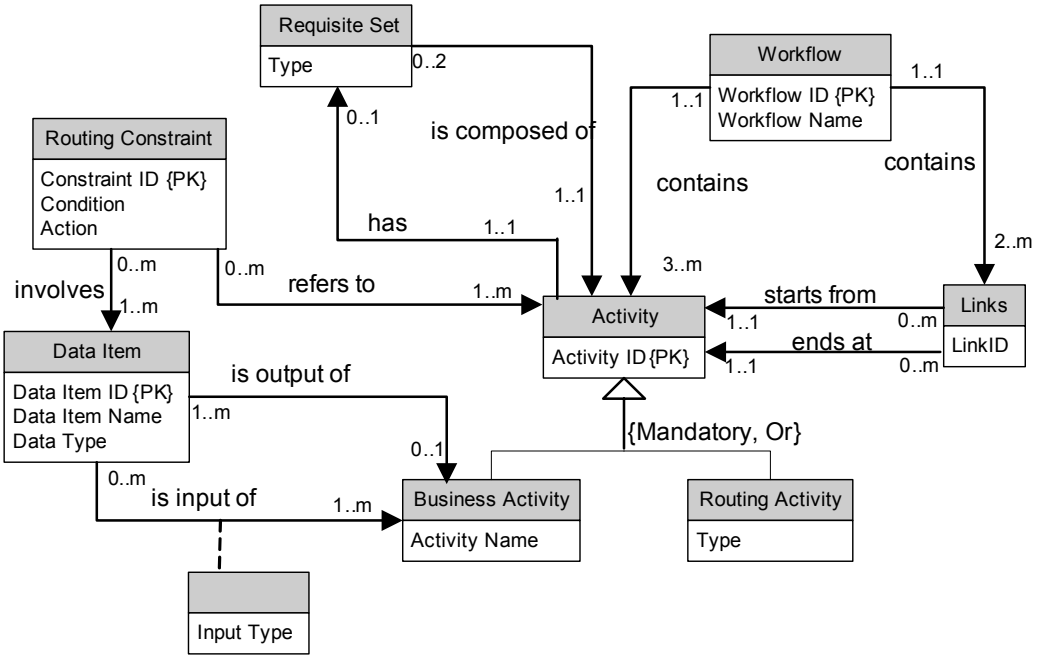


Figure 35. A Database Model for the Workflow Designer

A relational database is used to store all the design information. Figure 35 shows the design of the database and each important concept in workflow design is represented as

an entity in this database model. Note that for the purpose of simplification of coding, we use a table to store both direct and full requisite sets once they have been calculated.

6.6 A Proof-of- Concept Implementation

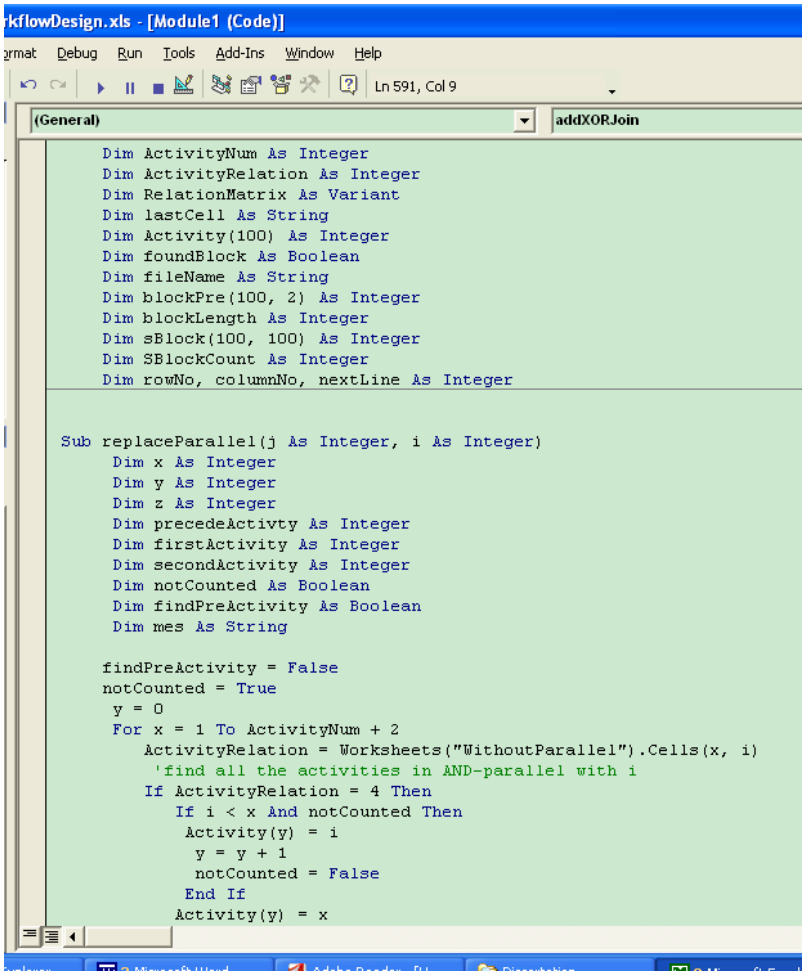
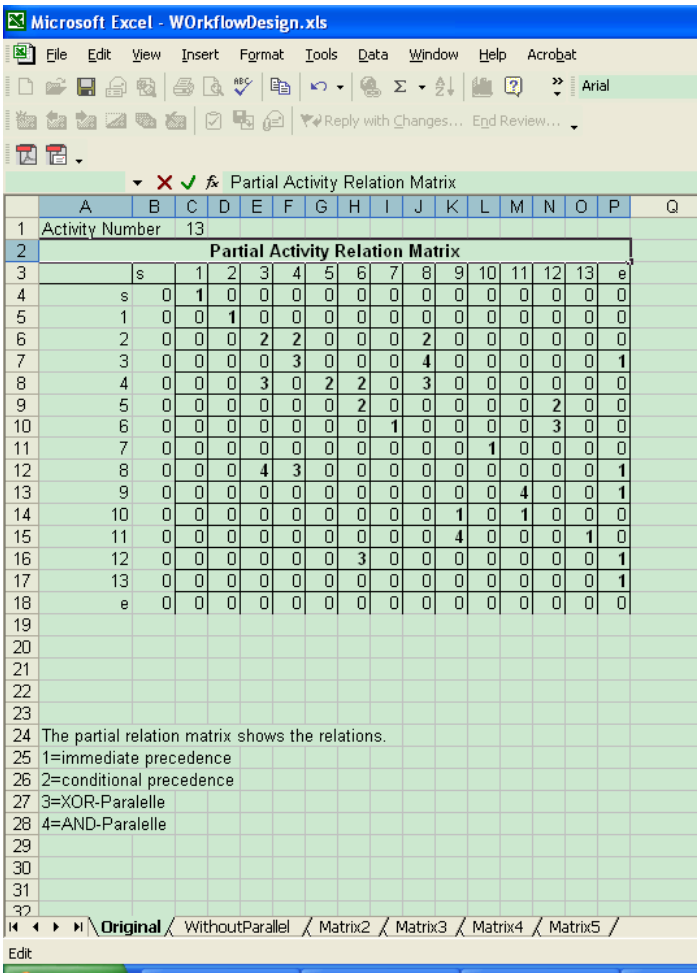


Figure 36. Microsoft Visual Basic Programming Environment

In order to test the core algorithms for constructing a workflow model from activity relations, a proof-of-concept implementation has been done in Microsoft Visual Basic, an object oriented programming language designed to be easy to use with the flexibility to to

develop fairly complex applications. Figure 36 shows the programming evnvironment used for the implementation.

Microsoft Excel spreadsheets are used to store the partial activty relation matrixes as shown Figure 37. We choose Excel spreadsheets as the primary data storage because Microsoft Visual Basic enables direct manipulation of Excel spreadsheets. When we develop a system with full functionalities in the future, a relational database will be developed.



	s	1	2	3	4	5	6	7	8	9	10	11	12	13	e
s	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	**2**	**2**	0	0	0	**2**	0	0	0	0	0	0
3	0	0	0	0	**3**	0	0	0	**4**	0	0	0	0	0	**1**
4	0	0	0	**3**	0	**2**	**2**	0	**3**	0	0	0	0	0	0
5	0	0	0	0	0	0	**2**	0	0	0	0	0	**2**	0	0
6	0	0	0	0	0	0	0	**1**	0	0	0	0	**3**	0	0
7	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
8	0	0	0	**4**	**3**	0	0	0	0	0	0	0	0	0	**1**
9	0	0	0	0	0	0	0	0	0	0	0	**4**	0	0	**1**
10	0	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	0	0
11	0	0	0	0	0	0	0	0	0	0	**4**	0	0	0	**1**	0
12	0	0	0	0	0	0	**3**	0	0	0	0	0	0	0	**1**
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 37. Partial Activity Relation Matrix Stored in Excel Spreadsheet

Five major subroutines are developed for the purpose of generating a correct workflow model from activity relations. Figure 38 shows the six subroutines and their interactions. The subroutines *omitParallel* and *replaceParallel* work together to replace AND-parallel and to update the partial activity relation matrix accordingly. They implement the algorithm of replacing AND-parallel shown in Figure 27. The subroutine *inlineBlock* implements the algorithm of identifying sequential inline blocks shown in Figure 25. The subroutines *addXORSplit* and *addXORJoin* implement the algorithm of creating XORSplits (Figure 28) and adding XORJoins (Figure 32) respectively. The steps of adding ANDSplits and ANDJoins are skipped in this proof-of-concept implementation because our purpose in this preliminary implementation is to test the proposed procedure is capable of generating a valid workflow model from data and activity dependency analysis.

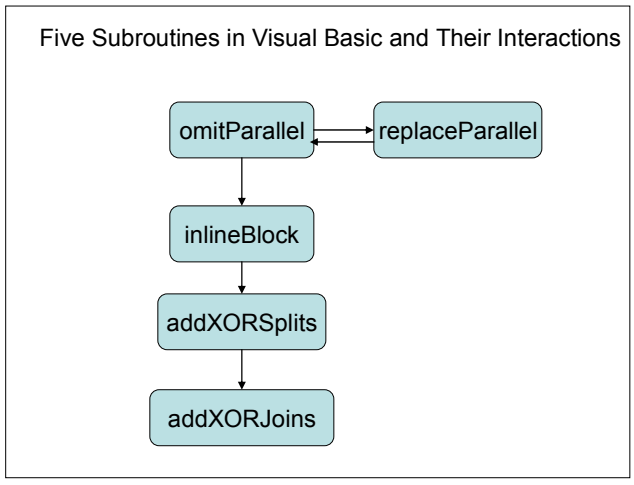


Figure 38. A Proof-of-Concept Implementation of Dependency Analysis Based Workflow Design in Visual Basic

6.7 Conclusions

Given the complexity of workflow design, implementing the approach proposed in previous chapter can still be a challenge even though at the theoretical level the approach is feasible. In this chapter, a detailed framework is provided for implementing the proposed approach. Various tools and detailed algorithms are developed for every step from collecting requirements to creating a final workflow model. Most of these implementation algorithms are guided by the theoretical results that have been proven analytically. Moreover, they provide guidance on how to transform the theoretical results in to programming code, which can potentially helps minimize the difficulties of applying the proposed approach in practice. In order to test the core algorithms for constructing a correct workflow model from activity relations, a proof-of concept implementation has been developed in Visual Basic.

7 CONCLUSIONS

Dataflow analysis and workflow design are critical steps in business process management and workflow automation. However, the existing literature and commercial workflow systems do not provide analytical tools for discovering the dataflow errors in a workflow model due to the lack of formalisms for dataflow analysis. Moreover, formal workflow design methods are not well adopted to enable efficient development of correct workflow models that satisfy business requirements and avoid various business process errors.

In this dissertation, we first propose a formal approach of dataflow analysis, which consists of dataflow specification and dataflow verification. To help specify the dataflow details in a business process, we propose dataflow matrix and process data diagram. A dataflow matrix is a two-dimensional table that shows the operation each activity performs on each data item. With a dataflow matrix, it is easy to identify how each data item is processed in a workflow. A process data diagram extends the UML activity diagram by specifying the input and output data for each activity.

More importantly, we propose a formal approach of dataflow verification for discovering dataflow anomalies based on dependency analysis. In the dataflow verification framework we have developed, the conditions necessary for different dataflow anomalies to occur are mathematically formulated and theoretically proved. The verification principles are presented with theoretical proofs to help determine if a workflow model to be free from these dataflow anomalies. We demonstrated how to apply the verification principles through real world business process examples. Based on

the verification rules, algorithms are developed to help guide implementations. To the best of our knowledge, this research is the first attempt to formally establish the correctness criteria for dataflow analysis in workflow management.

We then proposed a new workflow design approach based on several innovative techniques, leading to a rigorous design methodology for workflow development. First, our design method starts with data dependency analysis, thereby eliminating workflow problems stemming from dataflow errors. Second, the design procedure is capable of handling unstructured workflows and overlapping structures due to the decoupling of model correctness and model efficiency. Third, the design procedure is formally defined as the foundation for developing an automated workflow design tool. Forth, we use inline blocks to reduce the design complexity and simplify the design procedure. Detailed guidelines for requirements collection and analysis are given as the direction for applying this design methodology in practice. Various algorithms and a component-based system architecture are presented to help implementation. Further, we validate the design methodology through a proof-of-concept system develop in Visual Basic. We believe our approach will help to significantly improve the state of the art in workflow automation by enabling systematical elimination of dataflow anomalies and rigorous design of workflow models.

We are currently continuing our work in a number of directions. First, the design methodology currently does not provide mechanisms for handling workflows containing loops and OR splits and OR joins. In the next step, we plan to develop mechanisms to design more complex workflow structures including loops and OR nodes.

Furthermore, it is worth noting that other factors, such as resource limitations and cost optimization, need to be taken into consideration when determining the final model. A more comprehensive framework for workflow design can be developed through empirical studies, which can also help further validate the design methodology proposed in this dissertation.

Third, we plan to develop a formal methodology to help correct dataflow anomalies. Once dataflow anomalies are identified from a workflow model through dataflow verification, the next step is to remove the identified dataflow anomalies. We need to modify not only the dataflow but also the control flow in some cases. We will develop guidelines, which can suggest the appropriate modification when different dataflow anomalies are found.

Fourth, we will develop methods and guidelines for identification of candidate activity sets, which is the prerequisite step to apply the proposed design methodology. In addition, dataflow has been studied in the context of Web Services orchestration (Chafle et al., 2005). As an extension of our current work, we will investigate how dataflow analysis can be used in Web Services orchestration.

REFERENCES

Aalst, van der W.M.P. (1998) The application of petri nets to workflow management, *The Journal of Circuits Systems and Computers,* 8, 1, 21-66.

Aalst, van der W.M.P. (2001) Re-engineering knock-out processes. *Decision Support System*s, 30(4), 451-468.

Aalst, van der W.M.P., and Dongen. van B.F. (2002) Discovering Workflow Performance Models from Timed Logs. In Y. Han, S. Tai, and D. Wikarski, editors, *International Conference on Engineering and Deployment of Cooperative Information Systems* (EDCIS 2002), Lecture Notes in Computer Science, 2480: 45-63. Springer-Verlag, Berlin.

Aalst, van der W.M P.and Hee, van K. (2002) *Workflow management: Models, methods, and systems,* The MIT Press, Cambridge, Massachusetts, London, England.

Aalst, van der W.M.P. and Hofstede, ter A. (2000) Verification of Workflow Task Structures: A Petri-Net-Based Approach. *Information Systems*, 25, 1, 43-69.

Aalst, van der W.M.P., Dongen, van B.F., Herbst, J., Maruster, L., Schimm, G., and Weijters, A.J.M.M.(2003a)Workflow Mining: a Survey of Issues and Approaches, *Data and Knowledge Engineering*, 47(2):237-267,

Aalst, van der W.M.P., Hofsted, ter A.H.M., Kiepuszewski, B., and Barros, A.P (2003b) Workflow Patterns. *Distributed and Parallel Databases*, 14(1), 5-51.

Aalst, van der W.M.P., and Medeiros, de A.K.A. (2005) Process Mining and Security: Detecting Anomalous Process Executions and Checking Process Conformance. *Electronic Notes in Theoretical Computer Science*, 121:3-21.

Aalst, W. van der, Weijters, T., Maruster, L. (2004) "Workflow Mining: Discovering Process Models from Event logs", *IEEE Trans. on Knowledge and Data Engg*. (16) 9), 1128-1142

Abecker, A., Bernardi, A., Hinkelmann, K., Ku¨hn, O. and Sintek, M. (2000) Context-aware, proactive delivery of task-specific information: the KnowMore project, *Information Systems Frontiers,* 2, 3/4, 253-276.

Agarwal, R., and Tanniru, M. (1992a) A Petri-Net based approach for verifying the integrity of production systems. *International Journal of Man-Machine Studies* 36(3): 447-468.

Agarwal, R., and Tanniru, M. (1992b) A structured methodology for developing production systems. *Decision Support Systems* 8(6): 483-499.

Bajaj, A., Ram, S. (2002) SEAM: A state-entity-activity-model for a well-defined workflow development methodology, *IEEE Trans. on Knowledge and Data Engg.,* 14(2), 415-431.

Basu, A. and Blanning, R. W. (1994a) Metagraphs: A tool for modeling decision support systems, *Management Science*, 40, 12, 1579-1600.

Basu, A. and Blanning, R. W. (1994b) Model integration using metagraphs, *Information Systems Research*, 5, 3, 195-218.

Basu, A. Blanning, R. W., Shtub, A. (1997) Metagraphs in hierarchical modeling, Management Science 43, 5, 623-639.

Basu, A. and Blanning, R. W. (1998) The analysis of assumptions in model bases using metagraphs, Management Science 44, 7, 982-995.

Basu, A. and Blanning, R. W. (2000) A formal approach to workflow analysis, *Information Systems Research,* 11, 1, 17-36.

Basu, A. and Kumar, A. (2002) Workflow management issues in e-business, *Information Systems Research,* 13, 1, 1-14.

Berg, H. K., Boebert, W. E., Franta, W. R., Moher, T. G. (1982) *Formal Methods of Program Verification and Specifcation.* Prentice Hall, Englewood Cliffs, NJ.

Bi, H.H., and Zhao, J.L. (2003a) Mending the lag between commercial needs and research prototypes: A logic-based workflow verification approach," *Proceedings of the Proc. of the 8th INFORMS Computing Society Conference*, Chandler, AZ, pp. 191-212.

Bi, H. H., Zhao, J. L. (2003b) A formal classification of process anomalies for workflow verification, *13th Workshop on Information Technology & Systems,* Dec. 13-14, Seattle, WA.

Bi, H., and Zhao, L. (2004a) Process logic for verifying the correctness of business process models. *Proceedings of the International Conference on Information Systems* (*ICIS* 2004), Washington, D.C.

Bi, H. H. and Zhao, J. L. (2004b) Applying propositional logic to workflow verification, *Information Technology and Management,* 5, 3-4, 293-318.

Chafle, G. Chandra, S. Mann, V. Nanda, M.G. (2005) Orchestrating composite Web services under data flow constraints. Proceedings. 2005 IEEE International Conference on Web Services, (ICWS 2005), 211-218.

Choi, Y., and Zhao, J.L. (2002) Matrix-Based Abstraction & Verification for e-Business Processes. *Proceedings of the Proc. of 1st Workshop on e-Bus*., Barcelona, Spain, pp. 154-165.

Choi, Y., and Zhao, J.L. (2003) Handling Cycles in Workflow Verification by Feedback Identification and Partition. *Proceedings of the 2003International Conference on Information and Knowledge Engineering*, Las Vegas, Nevada, June 23-26.

Curran, T., Keller, G., and Ladd, A. (1998) SAP R/3 Business Blueprint: Understanding the Business Process Reference Model, Prentice Hall PTR, Upper Saddle River

Curtis, B., Kellner, M. I, and Over, J. (1992) Process modeling. *CACM,* 35, 9, 75-90.

Davenport, T.H. (1993) Process Innovation. Harvard Business School Press, Massachusetts.

Davidson, E. J. (1999) Joint application design (JAD) in practice. *Journal of Systems and Software* (45:3) pp. 215-223.

Earl, M. J., Sampler, J. L., Short, J.E. (1995). Strategies for Business Process Reengineering: Evidence from Field Studies. *Journal of Management Information Systems* 12(1): 31-56.

Georgakopoulos, D., Hornick, M. and Sheth, A. (1995) An overview of workflow management: From process modeling to workflow automation infrastructure, *Distributed and Parallel Database,* 3, 119-153.

Greco, G., Guzzo, A., Manco, G. Sacca, D. (2005) Mining and Reasoning on Workflows. IEEE Transactions on Knowledge and Data Engineering, 17(4), 519-534.

Guaspari, D., Marceau, C., and Polak, W. (1990) Formal Verification of Ada Programs. *IEEE Transactions on Software Engineering,* 16, 9, 1058-1075.

Herbst. J. (1999) An Inductive Approach to the Acquisition and Adaptation of Workflow Models. *Proc. Workshop Intelligent Workflow and Process Management: The New Frontier for AI in Business*, M.Ibrahim and B. Darbble, eds, 52-57.

Herbst, J. and Karagiannis, D. (2000) Integrating Machine Learning and Workflow Management to Support Acquisition and Adaptation of Workflow Models. *Int'l J. Intelligent System in Accounting, Finance, and Management*, 9:67-92.

Herrmann, T., and Walter, T. (1998) The relevance of showcases for the participative improvement of business processes and workflow management. *Proc. of Participatory Design Conf*., 117–127.

Hofacker, I., and Vetschera, R. (2001) Algorithmical approaches to business process design. *Computers & Operations Research* 28(13), 1253-1275.

Kappel, G., Lang, P., Rausch-Schott, S., Retschitzegger, W. (1995) Workflow management based on objects, rules, and roles, *IEEE Data Engineering Bulletin*. 18, 1, 11-18.

Krauskopf, R., Rash, F. (1990) Independent Verification and Validation. *IEEE Potentials*, 9(2), 12-14.

Kumar A, and Zhao, J. L. (1999) Dynamic routing and operational control in workflow management systems. *Management Science*, 35(2) 253-272.

Kumar, A. and Zhao, J. L. (2002) Workflow support for electronic commerce applications, *Decision Support System*, 32, 265-278.

Kwan, M.M., and Balasubramanian, P.R. (1997) Dynamic workflow management: a framework for modeling workflows. *Proceedings of HICSS 1997,* Maui, HI, USA, 7-10 January 1997, vol. 4, IEEE Computer Society Press, pp. 367 – 376.

Kwan, M.M., and Balasubramanian, P.R.(1998) Adding workflow analysis techniques to the IS development toolkit," *Proceedings of the Hawaii International Conference on System Sciences* (4), 312-321.

Lee, H. B., Kim J. W., and Park, S. J. (1999) KWM: Knowledge-based Workflow Model for Agile Organization, *Journal of Intelligent Information Systems*, 13, 261-278.

Lin, F. R., Yang, M. C., and Pai, Y. H. (2002) "A Generic Structure for Business Process Modeling," *Business Process Management Journal* (8:1), pp. 19-41.

Liu, R. and Kumar, A. (2005) "An Analysis and Taxonomy of Unstructured Workflows," *Third International Conference on Business Process Management (BPM 2005)* Nancy, France, September.

Mendling, J., Moser, M., Neumann, G., Verbeek, H.M.W., Dongen, B.F. van, and Aalst, W.M.P. van der (2006) Faulty EPCs in the SAP Reference Model. *Lec. Notes in Comp. Sci*., V 4102, pp. 451-457.

Mili, A. (1985) *An Introduction to Formal Program Verification*. Van Nostrand Reinhold Company. New York, NY.

OMG (2003) Unified Modeling Language, Version 1.5, formal/03-03-01," Object Management Group, Needham, MA, p. 736

Panzarasa, S., Maddè, S., Quaglini, S., Pistarini, C. and Stefanelli, M. (2002) Evidence-based careflow management systems: The case of post-stroke rehabilitation, Journal of Biomedical Informatics, 35, 2, 123-139.

Reijers, H.A., Limam, S., Aalst, van der W.M.P. (2003) Product-based workflow design. *Journal of Management Information Systems* (20) 1, 229-262.

Reijers, H.A., and Mansar, S. L. (2005) Best practices in business process redesign: an Overview and qualitative evaluation of successful redesign heuristics. *Omega, International Journal of Management Science*, 33 (4) , 283-306

Reuter, A. and Schwenkreis, F. (1995) Contracts: A low level mechanism for building general purpose workflow management systems, *IEEE Data Engg. Bulletin*, 18,1, 41-47.

Sadiq, S., Orlowska, M., Sadiq, W., Foulger, C. (2004) Data flow and validation in workflow modeling. *Proc. of the 15th Australasian Database Conference,* Jan.18 - 22, pp. 207-214.

Sarnikar, S., Zhao, J. L., and Kumar, A. (2004) Organizational knowledge distribution: An experimental evaluation", *Proceedings of AMCIS 2004,* August 5-8, New York, 2305-2314.

Smith, H. and Fingar, P. (2003) IT Doesn't Matter? Business Processes Do, Meghan-

Kiffer Press, Tampa, Florida, USA.

Stohr, E. A. and Zhao, J. L. (2001) Workflow automation: Overview and research issues, *Information Systems Frontiers,* 3, 3, 281-296.

Sun, S.X., Zhao, J.L., Sheng, O.R. (2004) Data flow modeling and verification in business process management. *Proceedings of AMCIS 2004*, Aug. 6-8, New York, NY, 4064-4073.

Sun, S. X. and Zhao, J.L. (2004) A data flow approach to workflow design. *Proceedings of WITS 2004,* Dec 11-12, Washington D.C., 80-85.

Sun, S.X., Zhao, J.L., Nunamaker, J. and Sheng, O.R. (2006) "Formulating the Data Flow Perspective for Business Process Management", *Information Systems Research*, December, pp. 374-391.

Wang, H. J., Zhao J. L., Zhang L. J. (2006) "Policy-Driven Process Mapping (PDPM): Towards Process Design Automation", *Proc. of the Intl. Conf. on Information Systems,* Dec. 10-13, Milwaukee, USA.

WfMC (1999) Workflow Management Coalition Interface 1: Process Definition Interchange Process Model. Document Number WfMC TC-1016-P.

Wallace, D. R. and Fujii, R.U. (1989) Software Verification and Validation: an Overview. *IEEE Software*, 6, 3, 10 - 17.

Wirtz, G., Weske, M., and Giese, H. (2001) The OCoN approach to workflow modeling in object-oriented systems, *Information Systems Frontiers,* 3 (3), 357-376.

Yourdon, E. and Constantine, L. L. (1979) *Structured design,* Prentice Hall.

Zhao, J. L. and Kumar, A. (1999) Data management issues for large scale, distributed workflow systems on the Internet, *ACM SIGMIS Data Base*, 29, 4, 22-32.

Zhao, J. L. and Stohr, E (2000) Workflow-centric information distribution through email, *Journal of Management Information Systems*, 17, 3, 45-72.

Zhao J. L. and Bi, H. H. (2003) On the Completeness of Logic-Based Workflow Verification, *the 13th Workshop on Information Technology and Systems*, Dec. 13-14, Seattle, WA.