GRAPHICAL AND LOGICAL FORMALISMS FOR
BUSINESS PROCESS MODELING AND VERIFICATION

by

Henry Haidong Bi

Copyright © Henry Haidong Bi 2004

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

2004

UMI Number: 3132200

Copyright 2004 by
Bi, Henry Haidong

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3132200

Copyright 2004 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

The University of Arizona ®
Graduate College

As members of the Final Examination Committee, we certify that we have read the
dissertation prepared by Henry Haidong Bi
entitled Graphical and Logical Formalisms for Business Process Modeling and
Verification
and recommend that it be accepted as fulfilling the dissertation requirement for the
Degree of Doctor of Philosophy

J. Leon Zhao

5/19/04

Date

Jay F. Nunamaker, Jr.

5/19/04

Date

Daniel D. Zeng

5/19/04

Date

Cole Smith

5/19/04

Date

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.

Dissertation Director: J. Leon Zhao

6/15/04

Date

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 acknowledgment 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: _____

A handwritten signature in black ink, consisting of a large capital 'O' followed by a lowercase 'r' and a stylized 'mg'.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank the members of my dissertation committee: Dr. J. Leon Zhao, Professor Jay F. Nunamaker, Jr., Dr. Daniel D. Zeng, Dr. Cole Smith, and Dr. Olga Yiparaki. I am indebted to them for the efforts they put into giving me invaluable, thoughtful criticisms and suggestions on completing this dissertation and related research. In particular, I am extremely grateful to my advisor, Dr. J. Leon Zhao, who has been a true mentor to me. Without his vision, guidance, support, and professional advice, this dissertation would not have been possible.

I must extend special thanks to Professor Jay F. Nunamaker, Jr. who makes me realize that persistence is the single most important component of research, and guides me to look to seeing “forests” beyond this dissertation. I am truly thankful to Dr. Daniel D. Zeng who provides a number of frank and constructive comments and suggestions. Special thanks are also to Dr. Cole Smith and Dr. Olga Yiparaki who provide valuable comments on my research in applying graph theory and logic to the area of business process management.

I am grateful to Professor Hsinchun Chen for his support during my first two years of doctoral study. His advice – stay healthy, concentrate efforts by goal orientation, and work hard – is a simple yet practical guide in my daily research activities.

Finally, I would like to thank my fellow doctoral students for their friendship and support. Particularly, I would like to thank Harry J. Wang, Surendra Sarnikar, and those who worked in the Artificial Intelligence Lab at the University of Arizona for their stimulating ideas and useful comments on various occasions.

DEDICATION

This dissertation is dedicated

to my wife, Lei Liu,

whose love, understanding, and encouragement

have carried me through this trying experience,

to my father, Zhiming Bi, my mother, Mali Jiang, and my brother, Xudong Bi,

for their love and persistent support

over the course of my long and winding academic journey, and

to all teachers in my life,

especially Zili Zhou who was my six-year high school math teacher,

for their dedication to education.

TABLE OF CONTENTS

LIST OF FIGURES	10
LIST OF TABLES	13
LIST OF DEFINITIONS	16
LIST OF LEMMAS, THEOREMS, AND COROLLARIES	18
SUMMARY OF NOTATION	19
ABSTRACT	22
1. INTRODUCTION	24
2. LITERATURE REVIEW	29
2.1. Process Modeling	29
2.1.1. Control Flow Modeling	29
2.1.2. Existing Process Modeling Methods	31
2.2. Process Verification.....	34
2.2.1. Existing Process Verification Methods	36
2.2.2. Research on Process Anomalies	40
3. APPLYING PROPOSITIONAL LOGIC TO WORKFLOW	
VERIFICATION	42
3.1. Foundation of Logic-Based Workflow Verification	42
3.1.1. Activity-Based Workflow Modeling	42

TABLE OF CONTENTS – *Continued*

3.1.2. Logic-Based Process Inference	45
3.1.3. Constrained Truth Table	48
3.1.4. Proofs of Process Inference Laws via Constrained Truth Table	52
3.2. Logic-Based Detection of Workflow Anomalies	54
3.2.1. Deadlocks	54
3.2.2. Lack of Synchronization	55
3.2.3. Activities without Termination or without Activation	56
3.2.4. Structurally Infinite Cycles	57
3.2.5. The Correctness of Logic-Based Verification	58
3.3. Logic-Based Workflow Verification Algorithm	59
3.3.1. The Verification Algorithm	59
3.3.2. Handling Cyclic Workflow Models	65
3.3.3. A Workflow Example	68
3.4. Instance Process Inference	75
4. PROCESS GRAPHS	82
4.1. Process Models	82
4.2. Definitions of Process Graphs	83
4.2.1. Syntax of Process Graphs	83
4.2.2. Semantics of Process Graphs	86
4.2.3. Standard Process Graphs	88

TABLE OF CONTENTS – *Continued*

4.2.4. Process Constructs	89
4.2.5. Subprocesses	91
4.2.6. Adjacency Matrix	94
4.2.7. Instance Walks	95
4.2.8. Isomorphism	101
4.3. Properties of Process Graphs	102
4.3.1. Use of Explicit Control Vertices	102
4.3.2. Use of <i>OR</i> Vertices	104
4.3.3. Correctness of Process Graphs	107
4.4. Analysis Methods for Process Graphs	109
4.4.1. Adjacency-Matrix-Based Analysis	110
4.4.2. Logic-Based Analysis	110
5. PROCESS LOGIC	111
5.1. Syntax of Process Logic	112
5.2. Semantics of Process Logic	121
5.2.1. Truth Values of Simple Process Propositions	121
5.2.2. Semantics of Logical Operators	122
5.2.3. Truth Tables for <i>Sequence</i> Wffs	131
5.2.4. Truth Tables for Process Argument Forms	132
5.3. Relationship between Process Graphs and Process Logic	135

TABLE OF CONTENTS – *Continued*

6. APPLICATIONS OF PROCESS GRAPHS AND PROCESS LOGIC ..	138
6.1. Applying Process Graphs to Defining and Classifying Process Anomalies	138
6.1.1. Anomalies Violating the Definition of Process Graphs	138
6.1.2. Anomalies Related to Improper Uses of Control Vertices	140
6.2. Applying Process Logic to Analyzing Structures of Process Models	143
6.2.1. Applying Process Logic to Detecting Process Anomalies	143
6.2.1.1. Deadlock	143
6.2.1.2. Lack of Synchronization	144
6.2.2. Applying Process Logic to Verifying the Correctness of Process Models ..	145
7. CONCLUSION AND FUTURE RESEARCH	149
REFERENCES	152

LIST OF FIGURES

FIGURE 1. A Vertex Series-Parallel Digraph	32
FIGURE 2. <i>AND/XOR</i> Graphs	33
FIGURE 3. Symbols for Activity-based Workflow Models	42
FIGURE 4. Basic Workflow Constructs	44
FIGURE 5. Adding a Dummy Activity between Two Control Vertices	45
FIGURE 6. An Invalid Workflow Structure due to Lack of Synchronization	51
FIGURE 7. Workflow Deadlocks	54
FIGURE 8. Lack of Synchronization	55
FIGURE 9. An Activity without Termination or without Activation	56
FIGURE 10. Structurally Infinite Cycles	57
FIGURE 11. Basic Cycles	66
FIGURE 12. An Overlapping Workflow Pattern	68
FIGURE 13. A Wrongly Designed Hotel Reservation Workflow Model	69
FIGURE 14. A Workflow Model after Adding Dummy Activities to FIGURE 13	69
FIGURE 15. A Correctly Designed Hotel Reservation Workflow Model	72
FIGURE 16. A Workflow Model after Adding Dummy Activities to FIGURE 15	73
FIGURE 17. A Workflow Model Containing a Basic Cycle	74
FIGURE 18. An Overlapping Workflow Pattern (Same as FIGURE 12)	75
FIGURE 19. Basic Constructs of Acyclic Workflows (Same as FIGURE 4(a) – (g)) ...	77
FIGURE 20. A Graphical Representation of Instance Process Inference	80

LIST OF FIGURES – *Continued*

FIGURE 21. An Overlapping Pattern Containing Lack of Synchronization	80
FIGURE 22. Graphical Notation for Process Graphs (Same as FIGURE 3)	85
FIGURE 23. A Process Graph P_1	86
FIGURE 24. A Process Graph P_2	88
FIGURE 25. Standard Process Graph P_{S2} of P_2 in FIGURE 24	89
FIGURE 26. Four Process Constructs	90
FIGURE 27. Process Constructs of the Standard Process Graph P_{S2} in FIGURE 25	91
FIGURE 28. A Process Graph P and Its Subprocesses	92
FIGURE 29. Three Blocks in FIGURE 28(b)	93
FIGURE 30. Three Process Instances of a Process Graph P_3	95
FIGURE 31. A Process Graph P_4	96
FIGURE 32. A Process Graph P_5	100
FIGURE 33. A Process Graph P_6	100
FIGURE 34. Two Isomorphic Process Graphs – Example 1	101
FIGURE 35. Two Isomorphic Process Graphs – Example 2	102
FIGURE 36. A Process Model without Using Control Vertices	103
FIGURE 37. Use of <i>OR</i> Vertices	106
FIGURE 38. A Process Graph P_7	107
FIGURE 39. Examples of Incorrect Process Graphs	108
FIGURE 40. A Standard Process Graph P_{S8}	113

LIST OF FIGURES – *Continued*

FIGURE 41. Structures in Standard Process Graphs	119
FIGURE 42. A Process Instance in a Standard Process Graph P_{S9}	122
FIGURE 43. <i>and</i> , <i>xor</i> , and <i>or</i> Structures with Three Activity Vertices	127
FIGURE 44. A Process Structure	131
FIGURE 45. A Standard Process Graph P_{S10} (Same as FIGURE 23)	134
FIGURE 46. A Process Model Containing Multiple Anomalies	140
FIGURE 47. Deadlocks	141
FIGURE 48. Lack of Synchronization (Same as FIGURE 8)	142
FIGURE 49. Structurally Infinite Cycles (Same as FIGURE 10)	142
FIGURE 50. Unnecessary Non-determinism	143
FIGURE 51. An Example of Deadlock	144
FIGURE 52. An Example of Lack of Synchronization	145
FIGURE 53. A Standard Process Graph P_{S11} (Same as FIGURE 12)	147

LIST OF TABLES

TABLE 1. Comparisons of Three Process Verification Approaches	40
TABLE 2. Logical Operators and Symbols for Logic-Based Workflow Verification	47
TABLE 3. Rules of Converting Basic Workflow Constructs into Logical Formulas	48
TABLE 4. Truth Table for $a_1 \rightarrow (a_2 \oplus a_3)$ in Propositional Logic	49
TABLE 5. Constrained Truth Table for $a_1 \rightarrow (a_2 \oplus a_3)$	50
TABLE 6. Applying Rule 2 to LHS of $(a \oplus b) \rightarrow (a \wedge b)$	50
TABLE 7. Applying Rule 1 to RHS of $(a \oplus b) \rightarrow (a \wedge b)$	51
TABLE 8. Logical Laws	52
TABLE 9. Proof of $(p \oplus q) \leftrightarrow (q \oplus p)$	53
TABLE 10. Proof of $((p \oplus q) \oplus r) \leftrightarrow (p \oplus (q \oplus r))$	53
TABLE 11. Proof of $p \rightarrow q, (q \wedge r) \rightarrow s \vdash (p \wedge r) \rightarrow s$	53
TABLE 12. Proof of $p \rightarrow q, (q \vee r) \rightarrow s \vdash (p \vee r) \rightarrow s$	53
TABLE 13. Proof of $p \rightarrow q, (q \oplus r) \rightarrow s \vdash (p \oplus r) \rightarrow s$	54
TABLE 14. Logical Formulas for the Workflow Model in FIGURE 14	70
TABLE 15. Transformation of Logical Formulas in TABLE 14	71
TABLE 16. Logical Formulas for the Workflow Model in FIGURE 16	73
TABLE 17. Transformation of Logical Formulas in TABLE 16	74
TABLE 18. Workflow Constructs and Their Logical formulas	76
TABLE 19. Adjacency Matrix for P_1 in FIGURE 23	94

LIST OF TABLES – *Continued*

TABLE 20. Truth Table of Material Conditional $\alpha \rightarrow \beta$ in Propositional Logic	112
TABLE 21. Compound Wffs	118
TABLE 22. Truth Table of <i>Sequence</i> $\alpha \rightarrow \beta$	126
TABLE 23. Truth Table of Join Wffs with Three Components	127
TABLE 24. Truth Table of <i>AND-Split</i> with Three Components	128
TABLE 25. Truth Table of <i>XOR-Split</i> with Three Components	128
TABLE 26. Truth Table of <i>OR-Split</i> with Three Components	129
TABLE 27. Truth Table of <i>AND-Split</i> ($n \geq 2$)	130
TABLE 28. Truth Table of <i>XOR-Split</i> ($n \geq 2$)	130
TABLE 29. Truth Values of c in $c \rightarrow \wedge(a_2, a_3, a_4)$	131
TABLE 30. Truth Values of $\wedge(a_2, a_3, a_4)$ in $c \rightarrow \wedge(a_2, a_3, a_4)$	132
TABLE 31. Truth Values of a_2, a_3 , and a_4 in $c \rightarrow \wedge(a_2, a_3, a_4)$	132
TABLE 32. Verification of the Validity of the Process Argument Form F of P_{S8} in FIGURE 40	133
TABLE 33. Cycle Is Not Yet Involved: the First Truth Table	134
TABLE 34. Cycle Is Not Yet Involved: the Second Truth Table	134
TABLE 35. Cycle Is Involved: the First Truth Table	135
TABLE 36. Cycle Is Involved: the Second Truth Table	135
TABLE 37. Rules of Converting Process Constructs into Wffs	137
TABLE 38. The First Truth Table for Finding the Deadlock in FIGURE 51	144

LIST OF TABLES – *Continued*

TABLE 39. The Second Truth Table for Finding the Deadlock in FIGURE 51	144
TABLE 40. Truth Table for Finding Lack of Synchronization in FIGURE 52	145
TABLE 41. The First Truth Table for FIGURE 53	147
TABLE 42. The Second Truth Table for FIGURE 53	148

LIST OF DEFINITIONS

DEFINITION 1	82
DEFINITION 2	84
DEFINITION 3	86
DEFINITION 4	86
DEFINITION 5	87
DEFINITION 6	89
DEFINITION 7	90
DEFINITION 8	91
DEFINITION 9	93
DEFINITION 10	94
DEFINITION 11	96
DEFINITION 12	97
DEFINITION 13	98
DEFINITION 14	101
DEFINITION 15	107
DEFINITION 16	113
DEFINITION 17	113
DEFINITION 18	113
DEFINITION 19	114
DEFINITION 20	114

LIST OF DEFINITIONS – *Continued*

DEFINITION 21	115
DEFINITION 22	115
DEFINITION 23	116
DEFINITION 24	116
DEFINITION 25	119
DEFINITION 26	119
DEFINITION 27	120
DEFINITION 28	121
DEFINITION 29	123
DEFINITION 30	124
DEFINITION 31	124
DEFINITION 32	124
DEFINITION 33	125
DEFINITION 34	125
DEFINITION 35	125
DEFINITION 36	126
DEFINITION 37	132

LIST OF LEMMAS, THEOREMS, AND COROLLARIES

LEMMA 1	58
THEOREM 2	59
LEMMA 3	66
THEOREM 4	67
THEOREM 5	103
COROLLARY 6	104
THEOREM 7	109
THEOREM 8	110
THEOREM 9	136
COROLLARY 10	137
THEOREM 11	145

SUMMARY OF NOTATION

■	end of a definition/anomaly
▲	end of a lemma/theorem/corollary
□	end of a proof
$\mapsto$	cause
$\wedge$	(logic) and
$\vee$	(logic) or
$\oplus$	(logic) exclusive or (xor)
$\rightarrow$	(logic) sequence
$\therefore$	therefore
$\vdash$	assertion sign
$\forall$	for all
$\exists$	for at least one
$\cong$	isomorphic
Σ	original formula set
Γ	resulting formula set
A	activity set
$a_1, a_2, a_3, \dots$	activity vertices
$A(P)$	directed arc set of P
B	block

SUMMARY OF NOTATION – *Continued*

$\bar{C}$	cycle
$c_1, c_2, c_3, \dots$	control vertices
$\bar{C}_v$	virtual cycle
$d_1, d_2, d_3, \dots$	dummy activity vertices
$d^-(v)$	in-degree of v
$d^+(v)$	out-degree of v
$d(v)$	degree of v
e	end vertex
F	process argument form
M	process model
$M(P)$	adjacency matrix of P
$N^-(v)$	in-neighborhood of v
$N^+(v)$	out-neighborhood of v
$N(v)$	neighborhood of v
O	execution order
P	process graph
$\vec{P}$	path
P_S	standard process graph
$\bar{P}_v$	virtual path

SUMMARY OF NOTATION – *Continued*

s	start vertex
$T_C(c)$	type of control vertex c
$u, v, w, \dots$	vertices
$uv, vw, wx, \dots$	arcs
$V(P)$	vertex set of P
$V_A(P)$	activity vertex set of P
$V_C(P)$	control vertex set of P
W	workflow
$\overline{W}$	walk
$\overrightarrow{W}_I$	instance walk
$\overrightarrow{W}_V$	virtual walk

ABSTRACT

Process models are an essential component of business process management and are found in various information systems such as workflow management systems, enterprise resource planning systems, and supply chain management systems. Process modeling and analysis are key steps in business process management. However, most existing activity-based process modeling paradigms found in process management tools lack a mathematical formalism, have limited expressive power, or have little analytical capability. Consequently, process modeling and analysis in the industry remain an art rather than a science.

In this dissertation, we first propose a logic-based workflow verification approach by applying propositional logic with constraints to verifying the correctness of both acyclic and cyclic workflow models. We demonstrate that this approach is capable of detecting process anomalies in workflow models.

We then propose process graphs as a graphical and mathematical tool for business process modeling and analysis. We formally define the syntax and semantics of process graphs and their constructs. We show that process graphs can not only model all types of execution order of activities, but also support multi-level abstraction, modular modeling, and analysis of the correctness of process models. We apply process graphs to defining and classifying process anomalies, and demonstrate that the proper use of process graphs can prevent certain process anomalies.

We also propose process logic as a logical formalism and mathematical method to represent process models for the purpose of process verification. We formally define the syntax and semantics of process logic to reflect the characteristics of process structures in a more precise way. We establish a formal relationship between process logic and graphical representations of process models, and transform the problem of verifying the correctness of process models into the problem of determining the validity of process argument forms in process logic. We demonstrate that process logic can be used to verify completely the correctness of activity-based process models.

Process graphs and process logic provide a theoretical foundation for the modeling, analysis, and verification of activity-based process models that are most widely used in the applications of business process management.

Keywords: process graph, process logic, business process management, process modeling, process analysis, process verification, workflow verification, process anomaly, workflow anomaly

1. INTRODUCTION

Processes are ordered sequences of activities (also called tasks, steps, transitions, actions, or events) and are found in various business areas such as manufacturing, software development, and product ordering and fulfillment. Well defined and unique business processes become enterprise assets (Pall 2000), enhance competitive advantages (Meade and Rogers 2001), and are a focus of continuous efforts for performance improvement (Davenport and Stoddard 1994; Kettinger et al. 1997). In the last two decades, workflow management systems (WfMSs) have been developed to specify and execute business processes (Georgakopoulos et al. 1995; Stohr and Zhao 2001). Business processes are also automated in many other information systems such as enterprise resource planning systems (Al-Mashari et al. 2003; Sheth et al. 1999; Stijn and Wensley 2001) and supply chain management systems (Larson and Rogers 1998; Mejza and Wisner 2001). In any of these applications, process models must be designed and implemented to satisfy specific business needs.

Although many process modeling paradigms exist, there is no unified process modeling framework (Basu and Kumar 2002; Sheth et al. 1999). Because processes are generally defined as ordered sequences of activities and process management practitioners tend to describe their work by activities (Davenport 1993; Lin et al. 2002; Mackenzie 2000; Sheth et al. 1999), activity-based process modeling methods are most widely used in the practice of process management. In general, however, existing activity-based modeling paradigms lack formalisms and mechanisms for rigorous

analysis and have limited expressive power (Sheth et al. 1999). For instance, few of existing activity-based methods are capable of both representing a process model that enables customers to “book flight”, “book hotel”, and/or “book car” in any possible combinations, and verifying mathematically the correctness of process models before they are put into action. Furthermore, many existing efforts to develop and standardize process modeling languages focus on the syntax of languages rather than the descriptions of process semantics (Sheth et al. 1999). Without defining formal semantics of process modeling languages, the correctness of process models cannot be defined and verified mathematically. Thus, research is needed to develop an activity-based modeling language that is expressive yet simple enough to represent all possible process scenarios and has a mathematical formalism to support rigorous analysis of the correctness of process models.

The purpose of process verification is to verify the correctness of process models during design time. Process verification is important because detecting process anomalies before process models are implemented can help reduce high costs of breakdown, debugging, and fixing during runtime. A process anomaly is simply an improper design that causes execution errors. Process verification remains an open research area. Existing research efforts focus on finding ways to verify completely the correctness of process models; that is, we need to develop a verification approach that is capable of detecting all potential process anomalies in all possible process structures. Some process structures can be very complex. IDEF3 standard, for instance, supports modeling parallel (*AND*), exclusive or (*XOR*), and inclusive or (*OR*) ordering relationships among activities (Mayer et al. 1995). Thus, no matter how complex a process model is, such a verification

approach should be able to assert, at the end of the verification procedure, whether the process model is completely correct or contains process anomalies.

Process structural analysis includes process validation, process verification, and data usage analysis (Basu and Kumar 2002). This dissertation focuses on process verification. We first propose a framework of applying propositional logic with constraints to workflow verification. A workflow is a process model that can be automated (WfMC 1999b). In this dissertation, we use process models and workflow models interchangeably. We then propose process graphs as a graphical and mathematical tool for process modeling and analysis. We also propose process logic as a logical formalism and mathematical method for process verification. Most contents of this dissertation are from four refereed research articles that are published or forthcoming (Bi and Zhao 2003a, 2003b, 2004; Zhao and Bi 2003) and two working papers (Bi and Zhao; Bi et al.).

We claim six contributions. First, we propose a logic-based workflow verification approach by applying propositional logic with constraints to verifying the correctness of workflow models. We demonstrate that logic-based workflow verification is capable of detecting process anomalies in both acyclic and cyclic activity-based workflow models.

Second, we formally define the syntax and semantics of process graphs and their standardized modeling units, subprocesses, instance walks, isomorphism, and correctness. We define process graphs such that in addition to capturing *AND* and *XOR* precedence constraints of activities, process graphs can be used to directly model *OR* precedence constraints that are left out in most existing process modeling approaches. Without losing

rigor at the computational level, the *OR* construct greatly increases the modeling efficiency at the presentation level (van der Aalst and Kumar 2003).

Third, we apply process graphs to formally defining and classifying process anomalies, and demonstrate that the proper use of process graphs can prevent certain anomalies in process models. The classification of process anomalies is a step towards a framework for process modeling and verification.

Fourth, we formally define the syntax and semantics of process logic such that process logic can be used to precisely represent process structures such as join and split structures. We also establish a formal relationship between process logic and process graphs, and transform the problem of verifying the correctness of graphical process models into the problem of determining the validity of process argument forms in process logic. We demonstrate that process logic can be used to verify *completely* the correctness of activity-based process models via truth tables.

Fifth, we claim a contribution to the application of graph theory. As graph theory has been applied to information system development (Couger et al. 1982), process graphs are an application of directed graphs (or digraphs) that usually have one type of vertices (Bang-Jensen and Gutin 2001; Chartrand and Lesniak 1996; Wilson 1996). Process graphs have two types of vertices, with the second type of vertices explicitly depicting the precedence constraints of activities. We use “process graphs,” although already existing in literature for different purposes, to indicate that they are graphs used to study process phenomena.

Finally, we demonstrate the applicability of logic to process analysis. We propose process logic as a logical formalism that can formally describe structures of process models. We use “process logic”, although already existing in literature for different purposes, to indicate that it is a logical formalism used to study process phenomena.

We restrict our research scope to modeling and analysis issues of processes and do not emphasize implementation and runtime issues, although we do make sure that the modeling constructs that we propose can be implemented. A runtime example is that for two parallel searches for the same information, the completion of one search activity dynamically causes the result of the other to be ignored (van der Aalst et al. 2003). In addition, we focus on the precedence constraints of activities that reflect the structures of process models and neglect other constraints such as timing and duration of activities and processes.

The rest of the dissertation proceeds as follows. In Section 2, we give an overview of process modeling, and review existing process modeling and verification approaches with emphasis on activity-based process models. In Section 3, we propose a framework of applying propositional logic with constraints to workflow verification. We define process graphs and demonstrate their properties and analysis methods in Section 4. In Section 5, we define the syntax and semantics of process logic, and establish a formal relationship between process logic and process graphs. In Section 6, we apply process graphs to defining and classifying process anomalies, and apply process logic to verifying the correctness of process models. We conclude the dissertation in Section 7 with future research directions.

2. LITERATURE REVIEW

2.1. Process Modeling

This section discusses requirements for process modeling and analysis and reviews existing process modeling approaches.

2.1.1. Control Flow Modeling

Davenport (1993) defines a process as “a structured, measured set of activities designed to produce a specified output for a particular customer or market.” Specifically, a process model comprises four basic perspectives (Basu and Blanning 2000; Curtis et al. 1992; Jablonski and Bussler 1996; Stohr and Zhao 2001; van der Aalst et al. 2003). The *control flow perspective* (also called *behavioral* or *transactional perspective*) specifies activities and their execution order such as sequence, split, join, parallelism, and iteration. The execution order of a process model is the precedence constraints specifying what activities must be executed before each activity in the process model is executed. In the domain of software engineering, control flow is “the sequence in which operations are performed during the execution of a computer program” (IEEE 1990). In the area of process management, the control flow of a process model is the sequence in which activities are executed in the model. The *informational perspective* represents data and information that are generated, manipulated, consumed, and transferred by activities. The *organizational perspective* reflects the relationship between a process model and organizational structures in terms of roles, people, and resources involved in activities.

The *functional perspective* describes functions and outcomes accomplished by activities and a process model.

A process model may contain other perspectives such as security, quality, and failure recovery (Jablonski and Bussler 1996), and therefore can be very complex. Because the execution order of activities is the basis of process models on which other perspectives are built, the control flow perspective is arguably the most basic perspective and is the focus of most existing process modeling efforts (Malone et al. 1999; Sadiq and Orłowska 2000; Sheth et al. 1999; van der Aalst et al. 2003). In this dissertation, we center on the control flow perspective of activity-based process models, and hereafter we simply use process models to denote their control flow perspective.

Based on the studies of process modeling principles (Curtis et al. 1992; Ellis and Nutt 1980; Green 1982; Huckvale and Ould 1995; Jablonski and Bussler 1996; Lin et al. 2002; Luo and Tung 1999), we generalize the requirements for a process modeling language as follows:

- (1) *Formality*. The syntax and formal semantics of all of its constructs must be unambiguously defined such that it can precisely represent processes and define the correctness of process models. Its properties must support mathematical analysis and verification of process models. Formal and precise process models are enactable on a machine.
- (2) *Expressiveness*. Its constructs should be sufficient to express succinctly all possible types of execution order of activities.

- (3) *Scalability*. It should be able to represent large and complex processes by means of multi-level abstraction. Multi-level abstraction provides building blocks that make it easier to design complex process models with stepwise refinement.
- (4) *Modularity*. It should specify standardized units for flexible design and manipulation. Process models can be subdivided into standardized units such that processes described by such models can be easily recognized.
- (5) *Ease of use*. Graphical representations intuitively illustrate the constructs of the modeling language and generally support ease of use and readability. A user-friendly modeling language should be simple enough to allow non-technical users to easily comprehend, design, and modify process models.

2.1.2. Existing Process Modeling Methods

Process modeling methods can be categorized according to their modeling objectives and focuses. Although various methods have their own strengths and are used in different situations, we focus on evaluating their capabilities of modeling control flows because “how good a notation is may well depend on what it is used for” (Green 1982). As a rigorous modeling tool with a mathematical foundation and strong analytical capabilities, metagraphs are used to represent and analyze the roles and interactions of the informational, functional, and organizational perspectives of process models (Basu and Blanning 1999, 2000, 2001, 2003). There are many other methods such as communication-oriented modeling (Medina-Mora et al. 1993; Winograd and Flores 1986), event-based modeling (Kumar and Zhao 1999), object-oriented modeling (Kueng et al. 1996), and goal-oriented modeling (Downs and Lunn 2002; Yu 1993). We do not stress

them in this dissertation because they place less emphasis on control flow modeling. Underlining graphical and formal representations of the control flow perspective, we examine two prevailing categories of process modeling methods – activity-based modeling and state-based modeling.

With emphasis on modeling activities and their execution order, *activity-based process modeling* methods represent activities as vertices (or nodes) and causal relations between activities as directed arcs (Georgakopoulos et al. 1995). There are many variations in activity-based modeling. A vertex series-parallel digraph (Lawler 1978; Monma and Sidney 1979) as shown in FIGURE 1 can model only sequential and parallel (*AND*) orderings among activities. Because vertex series-parallel digraphs are acyclic digraphs, they are not suitable for representing processes that contain cycles. *AND/OR* graphs (Nilsson 1980), which are actually *AND/XOR* graphs, mark all arcs in an *AND* structure (FIGURE 2). However, as will be explained in Section 4.3, without using explicit vertices to model control flows, *AND/XOR* graphs cannot model *OR* orderings or scenarios involving mixtures of multiple splits and joins of control flows. Consisting of actions and tests, conventional flowcharts have no built-in modularization, lack rigor, and have an impoverished notation (Green 1982).

FIGURE 1. A Vertex Series-Parallel Digraph

FIGURE 2. *AND/XOR* Graphs

There are many other activity-based modeling methods, such as Workflow Management Coalition (WfMC) standards (WfMC 1999a, 1999b), activity diagrams of Unified Modeling Language (Bastos and Ruiz 2001; OMG 2003; Rumbaugh et al. 1998), IDEF3 (Mayer et al. 1995), role activity diagramming (Huckvale and Ould 1995), and Windows-Explorer-like hierarchy of activities (Cheung et al. 1999). All of these methods lack mechanisms for verifying the correctness of process models, and most of them do not specify how to graphically model *OR* orderings. Although IDEF3 has explicit symbols to model *AND*, *XOR*, and *OR* orderings, it has not explored theoretically whether it can represent all types of execution order of activities.

Among *state-based process modeling* methods, state transition diagrams (Grosu et al. 1996; Klein et al. 1997) and statecharts (Harel 1987; Harel 1988) are directed graphs with vertices denoting states and arcs denoting transitions or events. Because in a state diagram, a single transition may relate to different states and appear more than once, the ordering relationships among transitions are difficult to interpret. Although Petri nets are traditionally classified as a state-based modeling tool (Murata 1989; Peterson 1981), they combine activity-based and state-based modeling by using two types of vertices – transitions and places, where places hold tokens to represent states. With a special source

place and a special sink place, workflow nets are an extension to Petri nets (van der Aalst 1997, 1998). A mathematical formalism and rich analysis tools are strengths of Petri nets. Nevertheless, because Petri nets translate processes into transitions, places, and tokens, Petri-nets-based process models are not as easy to understand as those based on activities and explicit control flow constructs. As a result, Petri nets are not yet widely used in commercial applications such as WfMSs (Sadiq and Orlowska 2000; van der Aalst 1998).

In sum, activity-based modeling methods are the most appropriate for representing control flows of process models and have been most widely used in commercial process management systems. However, existing activity-based modeling methods lack a mathematical foundation and analytical capabilities.

2.2. Process Verification

In the domain of software engineering, verification and validation are two closely related concepts. Verification is defined as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase” (IEEE 1990) or “confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled” (IEEE 1998). Validation is defined as “the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements” (IEEE 1990) or “confirmation by examination and provisions of objective evidence that the particular requirements for a specific intended use are fulfilled” (IEEE 1998). The purpose of verification and validation is to catch errors as early as possible (IEEE 1986). In addition to verification and validation,

methods have been developed to examine whether system requirements satisfy specified logical conditions that enable satisfactory system design and implementation in later stages (Ho 1974).

In the field of workflow and process management, three aspects of process structural analysis include process validation, process verification, and data usage analysis (Basu and Kumar 2002). Process validation is to validate whether a process model satisfy design requirements; that is, process validation is used for semantics examination. Process verification is to verify whether a process model is structurally correct; that is, process verification is used for syntax examination. Data usage analysis is to analyze the pattern of data access, and prevent two concurrent tasks from writing to the same data object at the same time.

The purpose of process verification is to examine whether there are any anomalies in process models during design time so as to avoid much higher costs of breakdown, debugging, and fixing during runtime (Sadiq and Orłowska 2000). Instead of detecting errors at the testing stage after implementation, verification at the design stage can shorten the development cycle (Schneider and Taubner 1992).

The verification and validation tools for computer-aided software engineering (CASE) include static analyzers that analyze a computer program without running it, dynamic analyzers that analyze a program by monitoring program execution, and correctness proof assistants that prove mathematically that a program satisfies its specifications (Fuggetta 1993). Proof of correctness is “a formal technique used to prove mathematically that a computer program satisfies its specified requirements” (IEEE 1990). In terms of process

verification, process graphs and process logic proposed in this dissertation are mathematical and static methods that verify the structural correctness of process models.

2.2.1. Existing Process Verification Methods

Verification has become an indispensable step in system or product development, such as network protocols (Lai 1995), distributed computer systems (Yau and Hong 1988), manufacturing systems (Vyatkin and Hanisch 2001), and web service components (Foster et al. 2003). Existing verification techniques include state transition technique (Foster et al. 2003; Lai 1995), Petri nets (Lai 1995), process algebra (Schneider and Taubner 1992), logic programming (Aagaard and Leiser 1995; Carchiolo and Faro 1990; Sarkar and Sarkar 1989), model-based simulation (Vyatkin and Hanisch 2001), and so on.

Many existing verification methods used in various domains are formal analysis and design tools whose syntax and semantics are formally defined (Fuggetta 1993). However, most existing formal analysis and verification tools adopt a state-based approach and, unfortunately, have not yet been used to the verification of activity-based process models that are most widely used in the practice of process management.

In this subsection, we review four existing process verification approaches, Petri nets, graph reduction, matrix-based method, and transaction logic.

Petri nets have been used to represent and verify workflow models (Adam et al. 1998; Murata 1989; van der Aalst 1997, 1998). A Petri net consists of three main components: transitions that represent activities or tasks, places that hold tokens representing states, and directed arcs that link transitions and places. The existing research on Petri-net-based workflow is related to a concept called workflow net (WF-net) (van der Aalst 1997). A

Petri net is a WF-net if and only if (1) it has a source place and a sink place and (2) if a transition that connects the sink place and the source place, the resulting Petri net is strongly connected. The verification of WF-nets focuses on verifying the soundness of workflows and related issues such as liveness, boundedness, safeness, deadlock, livelock, and dead activity (van der Aalst 1997, 1998, 1999). The main advantages of the Petri net formalism include a formal theory base, a token-based representation of workflow states, and its rigorous analysis and verification tools. Tokens play a key role in simulating control flows within a workflow model, representing states of the model at any given time, and providing an instrument to verify the model. However, since Petri nets translate workflows into transitions, places, and tokens, Petri-net-based workflow representation is not as easy to understand as those based on activities and explicit control constructs. As a result, most existing WfMSs do not use Petri nets as the modeling formalism (Sadiq and Orłowska 2000; van der Aalst 1998). Furthermore, the approach of Petri-nets-based verification for activity-based process models requires model translation. The verification methods that require translating a process model expressed by a modeling language (such as an activity-based process model) to a model expressed by another modeling language (such as a state-based process model) before verification face the challenge of validating whether two models are exactly identical.

Graph reduction was developed to identify two types of structural anomalies – deadlock and lack of synchronization (Sadiq and Orłowska 2000). It does so by removing the definitely correct structures that do not contain any structural anomaly, thus reducing the process model. This is accomplished by iteratively applying five reduction rules to

remaining vertices in the model. The five reduction rules are terminal reduction, sequential reduction, adjacent reduction, closed reduction, and overlapping reduction. The model cannot be completely reduced to an empty graph, if it contains any of these two types of anomalies. Graph reduction improves computational efficiency by reducing the graph iteratively. The worst-case complexity of the main graph reduction algorithm is $O(n^2)$. The main weakness of the graph reduction technique is that it is not applicable to process models that contain cycles (Sadiq and Orlowska 2000). Furthermore, although graph reduction can verify a special overlapping structure, it may not be able to handle general overlapping structures.

The matrix-based process verification integrates process abstraction and process verification (Choi and Zhao 2002, 2003). This approach uses the adjacency matrix to represent the process model and applies the concept of inline block to reduce computational complexity. An inline block is a collection of vertices satisfying the blocked transition property and is free from structural conflicts to verify process models. The blocked transition property requires that any inward transition to the inline block can only occur to the start vertex and that any outward transition from the inline block can only occur at the end node. A matrix-based algorithm is applied to identifying subsets of a process model that can be represented as inline blocks. So far, the matrix-based approach has been shown to be effective in identifying deadlock and lack of synchronization problems in cyclic process models. Additionally, this approach can verify complex process structures (including any overlapping process structures) by analyzing all instance flows within each inline block. However, like most existing

WfMSs, this approach has not addressed how to handle process models that contain *OR* vertices. In addition, the computational efficiency of matrix-based verification has not been reported in the literature.

Transaction logic (Bonner and Kifer 1994; Kifer 1996) has been proposed to analyze state-based process models, in which the notion of states corresponds to the notion of database states. Nevertheless, most process modeling paradigms in existing information systems apply activity-based modeling (Davenport 1993; Lin et al. 2002).

The first three approaches are compared in TABLE 1, where *OR* vertices are *OR-Split* and *OR-Join* vertices. ‘n/a’ means that the information is not found in the referenced papers. A state-based approach can capture the states of a process, whereas an activity-based approach does not model the states of a process.

TABLE 1. Comparisons of Three Process Verification Approaches

	Petri nets	Graph reduction	Matrix-based
Model Type	State-based	Activity-based	Activity-based
Anomalies detected or properties ensured	<ul style="list-style-type: none"> • Deadlock • Liveness • Boundedness • Safeness • Livelock • Dead activity 	<ul style="list-style-type: none"> • Deadlock • Lack of synchronization 	<ul style="list-style-type: none"> • Deadlock • Lack of synchronization
Cyclic models	Yes	No	Yes
OR vertices	n/a	n/a	n/a
Overlapping structures	Yes	Yes (partially)	Yes
Complexity	n/a	$O(n^2)$	n/a
Tools developed	Woflan	FlowMake	n/a

2.2.2. Research on Process Anomalies

A formal and systematic classification of process anomalies can help identify and avoid anomalies in business process design and provide guidelines for the verification of process models. However, because of the lack of such a classification, there are no general standards for examining process verification methods and tools. As a result,

researchers in process verification simply choose certain common anomalies as their focuses, as exhibited in verification methods such as graph reduction techniques, matrix-based abstraction and verification, and Petri-nets-based verification. Although Petri-nets-based process models can be thoroughly verified (Verbeek et al. 2001), there is no formal categorization of process anomalies from this line of research.

Little research in process anomalies has been found in literature. Sadiq and Orlowska discuss twelve syntactical errors, but they give only high-level descriptions of process anomalies without precise definitions and systematic classification (Sadiq and Orlowska 1997). Therefore, a comprehensive study in process anomalies needs to be conducted to classify formally all possible process anomalies. Such a classification of process anomalies can provide a guideline for developing verification methods.

3. APPLYING PROPOSITIONAL LOGIC TO WORKFLOW VERIFICATION

In this section, we apply proposition logic with constraints to verifying the correctness of workflow models.

3.1. Foundation of Logic-Based Workflow Verification

We introduce the activity-based workflow modeling formalism and the concept of logic-based workflow verification.

3.1.1. Activity-Based Workflow Modeling

FIGURE 3 contains a set of symbols for the activity-based modeling formalism. There are two types of vertices, activity vertices and control vertices. There are two special activity vertices. The start vertex represents the start point of the workflow and the end vertex stands for the end point. There are three kinds of control vertices, *AND* vertex, *OR* vertex, and *XOR* vertex. Finally, directed arcs are used to link vertices.

FIGURE 3. Symbols for Activity-based Workflow Models

An activity-based workflow is composed of a set of activities $A = \{a_1, a_2, \dots, a_l\}$, optional control vertices, and directed links. We use eight basic workflow constructs as shown in FIGURE 4, where a_i , a_j , a_s , and a_t are activities, i.e., $a_i, a_j, a_s, a_t \in A$.

- (a) *Sequence*: a construct in which an activity leads to another activity.
- (b) *AND-Split*: a construct in which multiple threads are generated. These threads can be executed in parallel or in any order.
- (c) *AND-Join*: a construct in which multiple parallel threads converge with synchronization.
- (d) *XOR-Split*: a construct in which exactly one of multiple threads is to be executed.
- (e) *XOR-Join*: a construct in which any one of multiple activities causes the following activity to be executed.
- (f) *OR-Split*: a construct in which one or more of multiple activities are executed. The number of activities that are actually triggered depends on runtime conditions.
- (g) *OR-Join*: a construct in which one or more of multiple threads converge.
- (h) *Cycle*: a construct in which one or more activities are executed iteratively until a certain condition is met.

FIGURE 4. Basic Workflow Constructs

These eight basic constructs are sufficient to model most workflow structures. Six of these constructs, (a) – (e) and (h), have been defined as the essential workflow primitives by the Workflow Management Coalition (WfMC 1999b) and are also defined in the form of Petri nets (van der Aalst 1998, 1999). However, *OR-Split* and *OR-Join* vertices have not been defined by WfMC. It is worth noting that *OR-Split* and *OR-Join* in conventional Petri nets are actually *XOR-Split* and *XOR-Join* under our framework, respectively. We

include *OR-Split* and *OR-Join* constructs in the activity-based workflow modeling formalism to make our models more expressive.

We can use the basic workflow constructs to build structures that are more complex. In addition, we allow two control vertices to connect directly to each other, such as c_1 and c_2 in FIGURE 5(a). A dummy activity can be added between two consecutive control vertices, such as the dummy activity d_1 in FIGURE 5(b).

FIGURE 5. Adding a Dummy Activity between Two Control Vertices

3.1.2. Logic-Based Process Inference

The logic-based workflow verification rests on the analogy between workflow models and logical deductive arguments. If there is no structural anomaly in an activity-based workflow model, every workflow instance can walk its way from the start vertex of the model through some activities to the end vertex, and there is no activity left inactivated. This is similar to a deductive argument: if all premises of a deductive argument are true, the conclusion of the argument will be true (Nolt et al. 1998).

In mathematical logic, an argument can be written in the form

$$\text{Premise 1, Premise 2, ..., Premise N} \vdash \text{Conclusion} \quad (1)$$

where $\vdash$ is called an assertion sign and is read as “therefore”. According to the aforementioned analogy, a workflow model can be expressed in the argument form

$$\text{Formula 1, Formula 2, ..., Formula N} \vdash s \rightarrow e \quad (2)^1$$

where logical formulas are obtained by translating constructs in the workflow model into logical statements, and ' $s \rightarrow e$ ' stands for from the start vertex s of the model to the end vertex e and is referred to as the conclusion. The left-hand side of ' $\rightarrow$ ' is called the LHS and the right-hand side the RHS.

Form (2) should be considered as a deductive argument whose conclusion $s \rightarrow e$ follows necessarily from its premises (logical formulas). *In other words, it is impossible for $s \rightarrow e$ to be false while all logical formulas are true.* If all logical formulas are true, the conclusion $s \rightarrow e$ will certainly be reached. If the conclusion $s \rightarrow e$ cannot be reached, it can be concluded that one or more logical formulas (i.e., one or more constructs in the model) are wrong.

It is noteworthy to mention two important points. First, we consider workflow models initially that do not contain cycles and then extend the ideas to cyclic workflows in later sections. Second, in order to detect all potential workflow anomalies, we require all formulas to be included in the deduction. This is different from classical propositional logic where even if some logical formulas in Form (2) are false, Form (2) can still be valid.

¹ The conclusion $s \rightarrow e$ is for an acyclic workflow model. For a cyclic model, the conclusion $(s \oplus \Phi) \rightarrow (e \oplus \Phi)$ is used, where Φ is a formula with exclusive disjunction of activities as addressed in later sections of the dissertation.

TABLE 2 lists the logical operators and symbols (Nolt et al. 1998; Sipser 1997) used for logic-based workflow verification. In addition, each activity is assigned a truth value. In workflow, if an activity is executed, its truth value is 1 (true); if an activity is not executed, its truth value is 0 (false). Furthermore, for any pair of activities a_i and a_j , $a_i \rightarrow a_j$ means if activity a_i is executed, then activity a_j will certainly be executed.

TABLE 2. Logical Operators and Symbols for Logic-Based Workflow Verification²

Logical Operator	Logical Symbol
and	$\wedge$
or	$\vee$
exclusive or (xor)	$\oplus$
if ... then	$\rightarrow$

The conversion rules for seven workflow constructs are summarized in TABLE 3, where a_i , a_j , a_s , and a_t are activities. Because a cycle construct in FIGURE 4(h) can be modeled using two *XOR* constructs, we do not include the cycle construct in TABLE 3.

² Symbol $\leftrightarrow$ is not a formal logical symbol used in logic-based workflow verification. It is used in $\alpha \leftrightarrow \beta$ only for the purpose of conveniently standing for both $\alpha \rightarrow \beta$ and $\beta \rightarrow \alpha$.

TABLE 3. Rules of Converting Basic Workflow Constructs into Logical Formulas

No.	Workflow Construct	Graph Representation	Logical Formula
1	<i>Sequence</i>	FIGURE 4(a)	$a_i \rightarrow a_j$
2	<i>AND-Split</i>	FIGURE 4(b)	$a_i \rightarrow ((a_j \wedge a_s) \wedge a_t)$
3	<i>AND-Join</i>	FIGURE 4(c)	$((a_i \wedge a_j) \wedge a_s) \rightarrow a_t$
4	<i>XOR-Split</i>	FIGURE 4(d)	$a_i \rightarrow ((a_j \oplus a_s) \oplus a_t)$
5	<i>XOR-Join</i>	FIGURE 4(e)	$((a_i \oplus a_j) \oplus a_s) \rightarrow a_t$
6	<i>OR-Split</i>	FIGURE 4(f)	$a_i \rightarrow ((a_j \vee a_s) \vee a_t)$
7	<i>OR-Join</i>	FIGURE 4(g)	$((a_i \vee a_j) \vee a_s) \rightarrow a_t$

The objective of workflow verification is to detect structural anomalies. Given a logical representation of a workflow model (or simply the logical model), we can verify its correctness by reducing the logical model by logical substitution, or simply logical reduction. We refer to the process of verifying the logical representation of a workflow model via logical reduction as *process inference*.

3.1.3. Constrained Truth Table

In propositional logic, truth table is used to prove inference laws. However, a workflow construct does not require the complete enumeration of all possible truth values because the nature of workflow says that if a LHS activity is not activated, its RHS activities will not be activated. In other words, a LHS activity precedes its RHS activities. That is, it is not meaningful in process inference to say that the premise is 1, but the

consequent is 0. As a result, we proposed a new concept called “constrained truth table” to derive new inference rules that are applicable to workflow verification.

A constrained truth table excludes the truth values of impossible situations in a workflow model. For example, TABLE 4 is the truth table in propositional logic for formula $a_1 \rightarrow (a_2 \oplus a_3)$. There are eight entry rows in TABLE 4. However, in workflow, activity a_2 or a_3 will not be executed until activity a_1 is executed. Thus, the second, third, and fourth rows will not be considered. Additionally, because the relationship between a_2 and a_3 is exclusive or (*XOR*), after the execution of a_1 , exactly one of a_2 and a_3 will be executed. Therefore, the fifth and eighth rows will not be considered either. Finally, we ignore the first row, which is a trivial case for workflow. The resulting constrained truth table is given in TABLE 5.

TABLE 4. Truth Table for $a_1 \rightarrow (a_2 \oplus a_3)$ in Propositional Logic

a_1	a_2	a_3	$a_2 \oplus a_3$	$a_1 \rightarrow (a_2 \oplus a_3)$
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

TABLE 5. Constrained Truth Table for $a_1 \rightarrow (a_2 \oplus a_3)$

a_1	a_2	a_3	$a_2 \oplus a_3$	$a_1 \rightarrow (a_2 \oplus a_3)$
1	0	1	1	1
1	1	0	1	1

The following *constrained truth table rules* (CTT Rules) are applicable when building a constrained truth table, where α and β are logical formulas that are not identical:

Rule 1: For $\alpha \wedge \beta$, α and β must both be true or both be false.

Rule 2: For $\alpha \oplus \beta$, α and β cannot both be true.

Rule 3: For $\alpha \rightarrow \beta$, α and β must both be true or both be false.

Rule 4: For $\alpha \leftrightarrow \beta$, α and β must both be true or both be false.

Note that the rules above should be applied consistently, and in case there is a conflict between applying the rules to the LHS and the RHS, one should only apply the rules to the LHS, which we refer to as the *Consistency Principle* when constructing a constrained truth table. For instance, given a logical formula $(a \oplus b) \rightarrow (a \wedge b)$, we construct a constrained truth table. As shown in TABLE 6, Rule 2 is applied to the LHS of the formula, indicating that this formula is invalid.

TABLE 6. Applying Rule 2 to LHS of $(a \oplus b) \rightarrow (a \wedge b)$

a	b	$a \oplus b$	$a \wedge b$
0	1	1	0
1	0	1	0

Note that in this example, if Rule 2 is applied to the LHS and Rule 1 to the RHS of the same formula, this will lead to a conflict and the values of a and b cannot be determined since Rule 1 says both a and b must be 1 and Rule 2 says that a and b cannot both be 1.

Another potential error when applying the rules is to apply Rule 1 to the RHS only as shown in TABLE 7. Although it correctly shows that the formula is invalid since not both the LHS and the RHS are true, this should not be allowed since the LHS cannot be 0 in a non-trivial case.

TABLE 7. Applying Rule 1 to RHS of $(a \oplus b) \rightarrow (a \wedge b)$

a	b	$a \oplus b$	$a \wedge b$
1	1	0	1

In sum, the given formula indicates a structural anomaly, namely lack of synchronization under a cyclic structure as shown in FIGURE 6. Although this situation rarely arises in a real world workflow, it illustrates that we should not allow the application of multiple rules that cause inconsistency.

FIGURE 6. An Invalid Workflow Structure due to Lack of Synchronization

3.1.4. Proofs of Process Inference Laws via Constrained Truth Tables

The purpose of process inference is to reduce a logical model towards its conclusion by logical substitution. Logical reduction requires the application of several logical laws such as commutative laws, transitive laws, and associative laws (Nolt et al. 1998). The most common among these three types of laws are the transitive laws, which reduce the logical model by logical substitution. We show that if a workflow model is reduced to the conclusion, then the workflow model is free from process anomalies.

We create constrained truth tables by applying the CTT Rules given above to remove non-applicable rows from the conventional truth tables used in propositional logic. Since the logical laws found in the literature do not contain *xor*, we add a number of new laws as proven next using constrained truth tables. The three types of logical laws are given in TABLE 8.

TABLE 8. Logical Laws

Commutative Laws	Associative Laws	Transitive Laws
$(p \wedge q) \leftrightarrow (q \wedge p)$	$((p \wedge q) \wedge r) \leftrightarrow (p \wedge (q \wedge r))$	$p \rightarrow q, q \rightarrow r \vdash p \rightarrow r$
$(p \vee q) \leftrightarrow (q \vee p)$	$((p \vee q) \vee r) \leftrightarrow (p \vee (q \vee r))$	$p \rightarrow q, (q \wedge r) \rightarrow s \vdash (p \wedge r) \rightarrow s^3$
$(p \oplus q) \leftrightarrow (q \oplus p)^3$	$((p \oplus q) \oplus r) \leftrightarrow (p \oplus (q \oplus r))^3$	$p \rightarrow q, (q \vee r) \rightarrow s \vdash (p \vee r) \rightarrow s^3$
		$p \rightarrow q, (q \oplus r) \rightarrow s \vdash (p \oplus r) \rightarrow s^3$

³ Nolt, Rohatyn, and Varzi (1998) do not give these laws. We proved them next using constrained truth tables.

Five new laws are proven in constrained truth tables TABLE 9 through TABLE 13.

TABLE 9. Proof of $(p \oplus q) \leftrightarrow (q \oplus p)$

p	q	$p \oplus q$	$q \oplus p$	$(p \oplus q) \leftrightarrow (q \oplus p)$
0	1	1	1	1
1	0	1	1	1

TABLE 10. Proof of $((p \oplus q) \oplus r) \leftrightarrow (p \oplus (q \oplus r))$

p	q	r	$(p \oplus q) \oplus r$	$p \oplus (q \oplus r)$	$((p \oplus q) \oplus r) \leftrightarrow (p \oplus (q \oplus r))$
0	0	1	1	1	1
0	1	0	1	1	1
1	0	0	1	1	1

TABLE 11. Proof of $p \rightarrow q, (q \wedge r) \rightarrow s \vdash (p \wedge r) \rightarrow s$

p	q	r	s	$p \rightarrow q$	$(q \wedge r) \rightarrow s$	$(p \wedge r) \rightarrow s$
1	1	1	1	1	1	1

TABLE 12. Proof of $p \rightarrow q, (q \vee r) \rightarrow s \vdash (p \vee r) \rightarrow s$

p	q	r	s	$p \rightarrow q$	$(q \vee r) \rightarrow s$	$(p \vee r) \rightarrow s$
0	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

TABLE 13. Proof of $p \rightarrow q, (q \oplus r) \rightarrow s \vdash (p \oplus r) \rightarrow s$

p	q	r	s	$p \rightarrow q$	$(q \oplus r) \rightarrow s$	$(p \oplus r) \rightarrow s$
0	0	1	1	1	1	1
1	1	0	1	1	1	1

Note that the distributive laws such as $((p \vee q) \wedge r) \leftrightarrow (p \wedge r) \vee (q \wedge r)$ and the law of exportation i.e., $((p \wedge q) \rightarrow r) \leftrightarrow (p \rightarrow (q \rightarrow r))$ are not allowed in process inference because their usage results in the alteration of the original workflow models.

3.2. Logic-Based Detection of Workflow Anomalies

In this section, we show how to detect several well-known workflow anomalies. Throughout this subsection, we assume a workflow model has a start vertex and an end vertex, as defined in most workflow modeling paradigms.

3.2.1. Deadlocks

A deadlock refers to a situation in which a workflow instance gets into a stalemate such that no activity can be executed (Verbeek et al. 2001). FIGURE 7 shows two types of deadlocks. FIGURE 7(a) is a deterministic deadlock. Semantically, the deadlock occurs because only one of a_1 and a_2 is executed after the *XOR-Split* vertex, the *AND-Join* vertex will wait forever and thus block the continuation of the process.

FIGURE 7. Workflow Deadlocks

The logical representation of the model in FIGURE 7(a) includes two logical formulas, $s \rightarrow (a_1 \oplus a_2)$ and $(a_1 \wedge a_2) \rightarrow e$. However, $s \rightarrow e$ cannot be reached by process inference. FIGURE 7(b) is a non-deterministic deadlock. When both a_1 and a_2 are executed after the *OR-split* vertex, there is no problem. But if only one of a_1 and a_2 is executed, $a_1 \wedge a_2$ will block any further execution. Logically, we cannot obtain $s \rightarrow e$ by process inference because $a_1 \vee a_2$ and $a_1 \wedge a_2$ are not equivalent.

3.2.2. Lack of Synchronization

Lack of synchronization refers to a situation in which the concurrent activities are joined by an *XOR* vertex, resulting in unintentional multiple executions of the end vertex e that follows the *XOR* vertex (Sadiq and Orłowska 2000). FIGURE 8 shows two kinds of lack of synchronization: (1) a deterministic one and (2) a non-deterministic one.

FIGURE 8. Lack of Synchronization

The logical formulas for the workflow model in FIGURE 8(a) are $s \rightarrow (a_1 \wedge a_2)$ and $(a_1 \oplus a_2) \rightarrow e$. Similarly, the logical model for FIGURE 8(b) is $s \rightarrow (a_1 \vee a_2)$ and $(a_1 \oplus a_2) \rightarrow e$. It is easy to see that the conclusion $s \rightarrow e$ cannot be reached by process inference in both cases. In terms of workflow execution, if only one of a_1 and a_2 is executed after s , then e is executed once and there is no problem; but if both a_1 and a_2 are executed, then e is triggered twice.

3.2.3. Activities without Termination or without Activation

Each well-defined activity in a workflow should be on a path from the start to the end vertex. If a path from an activity cannot lead to the end, this activity is an activity without termination. If an activity on a path that does not begin with the start, we say that it is an activity without activation.

For example, in FIGURE 9(a), activity a_2 is an activity without termination. This anomaly can be detected based on process inference as follows. Given the logical formulas $s \rightarrow a_1$, $a_1 \rightarrow (a_2 \wedge a_3)$, and $a_3 \rightarrow e$, by process inference, we obtain $s \rightarrow (a_2 \wedge e)$. The resulting logical formula indicates that a_2 is an activity without termination. Similarly, we can use process inference to determine that a_2 is an activity without activation in FIGURE 9(b).

FIGURE 9. An Activity without Termination or without Activation

As mentioned previously, process inference is different from conventional logical inference. We use the “activity without termination” problem in FIGURE 9(a) to illustrate this point more explicitly. It is interesting to note that a conventional propositional logical validation will not be able to identify the problem. From FIGURE 9(a), we have the formulas: $s \rightarrow a_1$, $a_1 \rightarrow (a_2 \wedge a_3)$, and $a_3 \rightarrow e$. In propositional logic, by adding a logical inference rule $(a_2 \wedge a_3) \rightarrow a_3$, it is easy to show that the conclusion $s \rightarrow e$

can be reached. That is to say, the conventional inference of propositional logic cannot detect that there is a structural problem in the original model. This is because the logical formula $(a_2 \wedge a_3) \rightarrow a_3$, though valid in propositional logic, adds a link to the workflow model and amounts to a modification of the original model. Consequently, we should not allow any addition of logical formulas that add new links to the original model.

3.2.4. Structurally Infinite Cycles

A structurally infinite cycle is a structural anomaly that causes some activities in a workflow model to be repeatedly executed forever. For instance, FIGURE 10(a) illustrates a deterministic infinite cycle in which after a_1 is executed, both e and a_2 are executed. The execution of a_2 triggers a_1 and itself again, thus causing an infinite cycle. This anomaly can be detected by logical inference. Two logical formulas can be obtained from FIGURE 10(a): $(s \oplus a_2) \rightarrow a_1$ and $a_1 \rightarrow (e \wedge a_2)$. By the transitive law, we obtain $(s \oplus a_2) \rightarrow (e \wedge a_2)$. This formula indicates that the execution of a_2 triggers itself and e , resulting in an infinite cycle. Similarly, FIGURE 10(b) shows a non-deterministic infinite cycle. That is, although long cycles are possible, once only e is triggered after c_2 , the cycle terminates.

FIGURE 10. Structurally Infinite Cycles

It is worth noting that the logic-based verification can be used to test the reachability of a workflow model as well. Reachability between any two vertices is defined as the connectivity between them (Maruta et al. 1998). In our context, we define *the reachability of a workflow model* as the connectivity between any vertex and the end vertex. Note that three of the aforementioned anomalies, deadlock, lack of synchronization, and activity without activation or termination can cause reachability problems in a workflow model.

3.2.5. The Correctness of Logic-Based Verification

We present one lemma and one theorem to illustrate formally the correctness of the logic-based workflow verification. LEMMA 1 is based on the anomalies defined above, and THEOREM 2 specifies the sufficient condition for an acyclic workflow to be correct.

LEMMA 1. Given an *acyclic* workflow $W = (f_1, f_2, \dots, f_n)$ that contains at least one anomaly such as deadlock, lack of synchronization, and activity without termination or activation, the corresponding logical representation must contain at least one logical formula that causes the inference by substitution to fail, or at least one original or intermediate formula that contains an anomaly. ▲

Discussion. As we have explained in the previous subsections, each type of workflow anomalies (not including structurally infinite cycles) can be detected based on their logical expressions. First, a deadlock (Section 3.2.1) or lack of synchronization (Section 3.2.2) anomaly causes inference by substitution to fail because of asymmetry of logical operators. Second, an activity without termination or activation (Section 3.2.3) can cause a logical formula to fail to link to other formulas that lead to the end vertex e . Third,

certain original or intermediate formulas can also contain anomalies such as deadlocks (Section 3.2.1). Consequently, LEMMA 1 is correct. Note that since we are dealing with acyclic workflows here, we need not consider the structurally infinite cycle anomalies (Section 3.2.4). $\square$

THEOREM 2. An acyclic workflow model $W = (f_1, f_2, \dots, f_n)$ is free from anomalies if all formulas $f_1, f_2, \dots, f_n$ are combined by substitution into $s \rightarrow e$, where s is the start vertex and e is the end vertex of W . $\blacktriangle$

Proof by Contradiction. Assume THEOREM 2 is false, that is, W contains at least one anomaly. By LEMMA 1, there is at least one formula in $\{f_1, f_2, \dots, f_n\}$ that causes the inference to fail or contains an anomaly. Consequently, it is impossible to achieve the conclusion $s \rightarrow e$. This contradicts the given condition that $s \rightarrow e$ has been reached. Thus, THEOREM 2 must be true if the condition holds. $\square$

3.3. Logic-Based Workflow Verification Algorithm

In this subsection, we first present the verification algorithm and then illustrate it using a real world workflow example.

3.3.1. The Verification Algorithm

The following notations are used in the algorithm:

W : a workflow model

s : start vertex

e : end vertex

a_i : an activity

Σ : a set of logical formulas for a given workflow

Γ : a set of terminal logical formulas resulting from inference

$\varphi(x_1, x_2, x_3, \dots, x_d)$: a logical expression in the form of $((((x_1 \theta x_2) \theta x_3) \theta \dots) \theta x_d)$, $\theta \in \{\wedge, \oplus, \vee\}$, $d \geq 1$. Note that $x_1, x_2, x_3, \dots, x_d$ can be either an activity or a logical expression

f : a logical formula in the form $lhs(f) \rightarrow rhs(f)$, and can also be written as $f(lhs(f), rhs(f))$

$lhs(f)$: the left hand side (LHS) of formula f in the form of $\varphi(p_1, p_2, p_3, \dots, p_k)$, $k \geq 1$

$rhs(f)$: the right hand side (RHS) of formula f in the form of $\varphi(q_1, q_2, q_3, \dots, q_m)$, $m \geq 1$

Φ : an exclusive disjunction of activities in the form of $((a_1 \oplus a_2) \oplus \dots) \oplus a_c$, $c \geq 1$

contains: a function indicating that an expression subsumes another expression, for instance, $lhs(f) = ((a_1 \wedge a_2) \wedge a_3) \wedge a_4$ contains $rhs(f') = (a_1 \wedge a_2) \wedge a_3$.

We use an example to illustrate some of these notations. Given a formula $f_1 = (((a_1 \wedge a_2) \wedge a_3) \wedge a_4) \rightarrow (a_5 \oplus a_6)$, we can write

$lhs(f_1) = \varphi(p_1, p_2, p_3, p_4) = (((a_1 \wedge a_2) \wedge a_3) \wedge a_4)$, where $p_1 = a_1, p_2 = a_2, p_3 = a_3, p_4 = a_4$

and

$rhs(f_1) = \varphi(q_1, q_2) = (a_5 \oplus a_6)$, where $q_1 = a_5$, and $q_2 = a_6$.

The way of expressing $lhs(f_1)$ and $rhs(f_1)$ as exemplified here enables us to specify various matching conditions between formulas. For instance, given another formula $f_2 = (\varphi(p^*_1) \rightarrow \varphi(q^*_1)) = (a_5 \rightarrow (a_7 \oplus a_8))$, where $p^*_1 = a_5$ and $q^*_1 = (a_7 \oplus a_8)$. We can say that the RHS of f_1 , $rhs(f_1)$, contains the LHS of f_2 , $lhs(f_2)$. Note that $rhs(f_1) = \varphi(q_1, q_2)$, $lhs(f_2) = \varphi(p^*_1)$ where $q_1 = p^*_1 = a_5$, and the superscript $*$ indicates that the expression is from formula f_2 . Consequently, formula f_1 and formula f_2 can be merged into

PROCEDURE MatchLHS(f, Σ) { // Search for formulas in Σ that matches the LHS of f

$Lhs_match(f) = \emptyset$

for each $f' \in \Sigma$ {

if ($lhs(f)$ contains $rhs(f')$ OR $rhs(f')$ contains $lhs(f)$) {

$Lhs_match(f) = Lhs_match(f) + f'$

$\Sigma = \Sigma - f'$

}

} return $Lhs_match(f)$

}

PROCEDURE MatchRHS(f, Σ) { // Search for formulas in Σ that matches the RHS of f

$Rhs_match(f) = \emptyset$

for each $f' \in \Sigma$ {

if ($lhs(f')$ contains $rhs(f)$ OR $rhs(f)$ contains $lhs(f')$) {

$Rhs_match(f) = Rhs_match(f) + f'$

$\Sigma = \Sigma - f'$

}

} return $Rhs_match(f)$

}

PROCEDURE Merge($f, Lhs_match(f), Rhs_match(f)$) {

// Merge formula f with its matching formulas

for each $f' \in Lhs_match(f)$ {

if ($lhs(f)$ contains $rhs(f')$) {

```

     $lhs(f) = \varphi(p_1, p_2, \dots, lhs(f'), \dots, p_k)$ 

    // replace part of the LHS of  $f$  by the LHS of  $f'$  based on the transitive law
} else {
     $rhs(f') = \varphi(q_1, q_2, \dots, rhs(f), \dots, q_m)$ 

    // replace part of the RHS of  $f'$  by the RHS of  $f$  based on the transitive law
     $f = f'$ 
}
}

for  $f' \in Rhs\_match(f)$  {
    if ( $rhs(f)$  contains  $lhs(f')$ ) {
         $rhs(f) = \varphi(q_1, q_2, \dots, rhs(f'), \dots, q_m)$ 
    } else {
         $lhs(f') = \varphi(p_1, p_2, \dots, lhs(f), \dots, p_k)$ 
         $f = f'$ 
    }
} return  $f$ 
}

PROCEDURE Identify( $\Gamma$ ) {           // Identify potential anomalies in resulting formulas
    if ( $|\Gamma| = 1$ ) {
        for  $f \in \Gamma$ 
            if ( $f = (s, e)$  OR  $f = ((s \oplus \Phi), (e \oplus \Phi))$ ) {
                print " $W$  contains no anomaly"
            }
        }
    }
}

```



```

    }
} else {
    for each  $f \in \Gamma$  {
        for each  $f' \in \Gamma, f' \neq f$  {
            if ( $lhs(f)$  contains  $((a_1 \oplus a_2) \oplus \dots) \oplus a_r$ ) AND  $rhs(f)$  contains  $((a_1 \wedge a_2) \wedge \dots) \wedge a_r$ ) {
                print “ $\mathcal{W}$  contains lack of synchronization at  $a_1, a_2, \dots, a_r$ ”
            } else if ( $lhs(f)$  contains  $((a_1 \wedge a_2) \wedge \dots) \wedge a_r$ ) AND  $rhs(f)$  contains  $((a_1 \oplus a_2) \oplus \dots) \oplus a_r$ ) {
                print “ $\mathcal{W}$  contains deadlock at  $a_1, a_2, \dots, a_r$ ”
            }
        }
    }

    if ( $f = ((s \wedge \varphi), (e \wedge \varphi))$ ) { //  $\varphi$  is a valid logical expression
        print “ $\mathcal{W}$  contains a deadlock at  $\varphi$ ”
    }

    if ( $f = ((s \oplus \varphi), (e \wedge \varphi))$ ) { print “ $\mathcal{W}$  contains a deterministic infinite cycle at  $\varphi$ ” }
    if ( $f = ((s \oplus \varphi), (e \vee \varphi))$ ) { print “ $\mathcal{W}$  contains a non-deterministic infinite cycle at  $\varphi$ ” }
    if ( $f = (s, (e \wedge \varphi))$  OR  $f = (s, (e \oplus \varphi))$  OR  $f = (s, (e \vee \varphi))$ ) {
        print “ $\mathcal{W}$  contains an activity without termination at  $\varphi$ ”
    }

    if ( $f = ((s \wedge \varphi), e)$  OR  $f = ((s \oplus \varphi), e)$  OR  $f = ((s \vee \varphi), e)$ ) {
        print “ $\mathcal{W}$  contains an activity without activation at  $\varphi$ ”
    }
}

```

```

    }
  }
}

```

The complexity of the algorithm can be estimated as follows. Although the procedure Verify contains two levels of do-loops, they combine together into one loop since the number of outer loop iteration is equal to the number of resulting formulas, and the number of inner loop iteration depends on the number of formulas in each resulting formula of process inference. At one extreme, when there is no anomaly found in the workflow, there will be only one resulting formula. In this case, the inner loop will iterate at most $n - 1$ times, and the outer loop will iterate only once, where n is the number of constructs in the workflow model as defined in FIGURE 4.

In sum, the two do-loops collectively contain $n - 1$ steps of iteration since at the first iteration the initial dimension of Σ is $n - 1$. Furthermore, procedures MatchLHS and MatchRHS also contain loops, with a maximum $2(n - 1)$ number of comparisons initially and decreasing thereafter. Consequently, the worst case complexity of the algorithm is proportional to $2(n - 1)^2$, or $O(n^2)$.

3.3.2. Handling Cyclic Workflow Models

In the verification algorithm, formulas $f_1, f_2, \dots, f_n$ are transformed through process inference to reach the conclusion $s \rightarrow e$ if the given workflow W is acyclic. However, if W is cyclic, we will not be able to reach $s \rightarrow e$. Instead, we need a new form of conclusion as explained next.

FIGURE 11. Basic Cycles

As shown in FIGURE 11, if a correct workflow contains a single cycle, process inference can reduce the workflow model to FIGURE 11(a). If a correct workflow contains multiple cycles, process inference can reduce the workflow model similar to FIGURE 11(a) or FIGURE 11(b). FIGURE 11(a) contains a basic cycle that can be expressed as $(s \oplus a_1) \rightarrow (e \oplus a_1)$. FIGURE 11(b) contains three basic cycles that can be represented by $(s \oplus (a_1 \oplus (a_2 \oplus a_3))) \rightarrow (e \oplus (a_1 \oplus (a_2 \oplus a_3)))$, which can be generalized as $(s \oplus \Phi) \rightarrow (e \oplus \Phi)$, where $\Phi = a_1 \oplus (a_2 \oplus a_3)$, an exclusive disjunction of two or more activities.

We prove the correctness of verification in the case of cyclic workflows next.

LEMMA 3. Given a cyclic workflow $W = (f_1, f_2, \dots, f_n)$ that contains at least one anomaly such as deadlock, lack of synchronization, activity without termination or activation, and structurally infinite cycle, the corresponding logical representation must contain at least one logical formula that causes the inference by substitution to fail, or at least one original or intermediate formula that contains an anomaly. ▲

Discussion. Similar to LEMMA 1, each type of workflow anomalies can be detected based on their logical expressions. In addition to anomalies found in acyclic workflows,

structurally infinite cycles can also be detected based on the original or intermediate logical formulas in a cyclic workflow. Consequently, LEMMA 3 is correct. $\square$

THEOREM 4. A cyclic workflow model $W = (f_1, f_2, \dots, f_n)$ is free from anomalies if all formulas $f_1, f_2, \dots, f_n$ are combined by substitution into $(s \oplus \Phi) \rightarrow (e \oplus \Phi)$, where s is the start vertex and e is the end vertex of W , and Φ is a formula with exclusive disjunction of activities. $\blacktriangle$

Proof by Contradiction. Assume THEOREM 4 is false, that is, W contains at least one anomaly. By LEMMA 3, there is at least one formula in $\{f_1, f_2, \dots, f_n\}$ that causes the inference to fail or contains an anomaly. Consequently, it is impossible to achieve the conclusion $(s \oplus \Phi) \rightarrow (e \oplus \Phi)$. This contradicts the given condition. Thus, THEOREM 4 must be true. $\square$

Note that THEOREM 4 proves the correctness of logic-based process inference, but not its completeness. In fact, the verification algorithm using propositional logic with constraints does not handle workflow models that contain certain types of overlapping patterns. An overlapping pattern is a structure in which two or more control vertices share two or more immediately preceding (or *parent*) control vertices. For instance, FIGURE 12 illustrates an overlapping pattern in which control vertices c_4 and c_5 share two parent control vertices, c_2 and c_3 .

FIGURE 12. An Overlapping Workflow Pattern

Applying process inference to the logical formulas obtained from this workflow model, we get two formulas, $s \rightarrow ((a_3 \wedge a_4) \oplus (a_5 \wedge a_6))$ and $((a_3 \oplus a_5) \wedge (a_4 \oplus a_6)) \rightarrow e$, which cannot be further transformed using existing logical rules. However, it can be shown that this workflow is structurally correct. In other words, not all overlapping patterns will present a problem. However, for example, if all *AND* control vertices in FIGURE 12 are replaced with *XOR* vertices, the process inference algorithm will be able to verify its correctness.

In Section 6.2.2, we will demonstrate that we can use process logic to verify the correctness of all process models, including models that contain overlapping patterns.

3.3.3. A Workflow Example

We use an online hotel reservation workflow (FIGURE 13) to illustrate the verification algorithm. The hotel reservation company allows a customer to name her or his price first and then contacts its partner hotels to find whether any hotels accept the requested reservation. For simplicity, we limit the number of hotels to three. FIGURE 14 displays the same workflow model after adding six dummy activities to FIGURE 13.

FIGURE 13. A Wrongly Designed Hotel Reservation Workflow Model

FIGURE 14. A Workflow Model after Adding Dummy Activities to FIGURE 13

We will show that this model contains a structural anomaly, assuming that the conversion rules given in TABLE 3 are used to create the logical formulas as shown in TABLE 14 from FIGURE 14.

TABLE 14. Logical Formulas for the Workflow Model in FIGURE 14

Formula No.	Formula	Formula No.	Formula
1	$(s \oplus d_1) \rightarrow a_1$	10	$(a_9 \oplus a_{10}) \rightarrow d_4$
2	$a_1 \rightarrow a_2$	11	$(a_{11} \oplus a_{12}) \rightarrow d_5$
3	$a_2 \rightarrow (d_2 \oplus d_3)$	12	$(a_{13} \oplus a_{14}) \rightarrow d_6$
4	$(a_5 \oplus d_2) \rightarrow a_3$	13	$((d_4 \wedge d_5) \wedge d_6) \rightarrow a_{15}$
5	$a_3 \rightarrow (a_4 \oplus d_1)$	14	$a_{15} \rightarrow (a_5 \oplus a_{16})$
6	$d_3 \rightarrow ((a_6 \vee a_7) \vee a_8)$	15	$a_{16} \rightarrow (a_{17} \wedge a_{18})$
7	$a_6 \rightarrow (a_9 \oplus a_{10})$	16	$(a_{17} \wedge a_{18}) \rightarrow a_{19}$
8	$a_7 \rightarrow (a_{11} \oplus a_{12})$	17	$(a_{19} \oplus a_4) \rightarrow e$
9	$a_8 \rightarrow (a_{13} \oplus a_{14})$		

The logical formulas in TABLE 14 can be transformed using logical laws defined in Section 3.1.4. In TABLE 15, for example, the transitive law $p \rightarrow q, q \rightarrow r \vdash p \rightarrow r$ is first applied to formulas 1 and 2, and $(s \oplus d_1) \rightarrow a_2$ is obtained. Then the same law is applied to $(s \oplus d_1) \rightarrow a_2$ and formula 3 to generate $(s \oplus d_1) \rightarrow (d_2 \oplus d_3)$. Other formulas in TABLE 15 are obtained in a similar way.

TABLE 15. Transformation of Logical Formulas in TABLE 14

Formula	Transformation	Sources
18	$(s \oplus d_1) \rightarrow a_2$	1,2
19	$(s \oplus d_1) \rightarrow (d_2 \oplus d_3)$	3,18
20	$(a_5 \oplus d_2) \rightarrow (a_4 \oplus d_1)$	4,5
21	$d_3 \rightarrow (((a_9 \oplus a_{10}) \vee (a_{11} \oplus a_{12})) \vee (a_{13} \oplus a_{14}))$	6,7,8,9
22	$(((a_9 \oplus a_{10}) \wedge (a_{11} \oplus a_{12})) \wedge (a_{13} \oplus a_{14})) \rightarrow a_{15}$	10,11,12,13
23	$a_{15} \rightarrow (a_5 \oplus (a_{17} \wedge a_{18}))$	14,15
24	$a_{15} \rightarrow (a_5 \oplus a_{19})$	16,23
25	$(((a_9 \oplus a_{10}) \wedge (a_{11} \oplus a_{12})) \wedge (a_{13} \oplus a_{14})) \rightarrow (a_5 \oplus a_{19})$	22,24
26	$(s \oplus d_1) \rightarrow (d_2 \oplus (((a_9 \oplus a_{10}) \vee (a_{11} \oplus a_{12})) \vee (a_{13} \oplus a_{14})))$	19,21
27	$(a_{19} \oplus a_4) \rightarrow e$	17

Now, we explain briefly how to verify the example using the verification algorithm.

Initially the formula set Σ contains 17 formulas as listed in TABLE 14. Starting with formula 1, $(s \oplus d_1) \rightarrow a_1$, for instance, the procedure MatchRHS finds that the LHS of formula 2, a_1 , contains the RHS of formula 1, a_1 . Then, the procedure Merge merges the LHS of formula 1, $(s \oplus d_1)$, into the LHS of formula 2 and obtains formula 18, $(s \oplus d_1) \rightarrow a_2$, in TABLE 15.

Continuing the algorithm, we derive the resulting formula set Γ that contains four formulas, that is, formulas 20, 25, 26, and 27 in TABLE 15. Because Γ contains four

terminal formulas, i.e., formulas that cannot be further merged, the workflow model must contain at least one anomaly. An anomaly, deadlock, is found among these four remaining formulas when the procedure Identify is applied to them. The *AND* relationship among $(a_9 \oplus a_{10})$, $(a_{11} \oplus a_{12})$, and $(a_{13} \oplus a_{14})$ in formula 25 does not match the *OR* relationship among them in formula 26. This indicates that the model contains a deadlock around these six activities, a_9 , a_{10} , a_{11} , a_{12} , a_{13} , and a_{14} .

This deadlock anomaly can be corrected by changing c_{12} to an *OR-Join* vertex. The new model is shown in FIGURE 15. FIGURE 16 gives the same workflow model after adding six dummy activities to FIGURE 15. The logical formulas for FIGURE 16 are given in TABLE 16.

FIGURE 15. A Correctly Designed Hotel Reservation Workflow Model

FIGURE 16. A Workflow Model after Adding Dummy Activities to FIGURE 15

TABLE 16. Logical Formulas for the Workflow Model in FIGURE 16

Formula No.	Formula	Formula No.	Formula
1	$(s \oplus d_1) \rightarrow a_1$	10	$(a_9 \oplus a_{10}) \rightarrow d_4$
2	$a_1 \rightarrow a_2$	11	$(a_{11} \oplus a_{12}) \rightarrow d_5$
3	$a_2 \rightarrow (d_2 \oplus d_3)$	12	$(a_{13} \oplus a_{14}) \rightarrow d_6$
4	$(a_5 \oplus d_2) \rightarrow a_3$	13	$((d_4 \vee d_5) \vee d_6) \rightarrow a_{15}$
5	$a_3 \rightarrow (a_4 \oplus d_1)$	14	$a_{15} \rightarrow (a_5 \oplus a_{16})$
6	$d_3 \rightarrow ((a_6 \vee a_7) \vee a_8)$	15	$a_{16} \rightarrow (a_{17} \wedge a_{18})$
7	$a_6 \rightarrow (a_9 \oplus a_{10})$	16	$(a_{17} \wedge a_{18}) \rightarrow a_{19}$
8	$a_7 \rightarrow (a_{11} \oplus a_{12})$	17	$(a_{19} \oplus a_4) \rightarrow e$
9	$a_8 \rightarrow (a_{13} \oplus a_{14})$		

TABLE 17. Transformation of Logical Formulas in TABLE 16

Formula	Transformation	Sources
18	$(s \oplus d_1) \rightarrow a_2$	1,2
19	$(s \oplus d_1) \rightarrow (d_2 \oplus d_3)$	3,18
20	$(a_5 \oplus d_2) \rightarrow (a_4 \oplus d_1)$	4,5
21	$d_3 \rightarrow (((a_9 \oplus a_{10}) \vee (a_{11} \oplus a_{12})) \vee (a_{13} \oplus a_{14}))$	6,7,8,9
22	$(((a_9 \oplus a_{10}) \vee (a_{11} \oplus a_{12})) \vee (a_{13} \oplus a_{14})) \rightarrow a_{15}$	10,11,12,13
23	$a_{15} \rightarrow (a_5 \oplus (a_{17} \wedge a_{18}))$	14,15
24	$a_{15} \rightarrow (a_5 \oplus a_{19})$	16,23
25	$(((a_9 \oplus a_{10}) \vee (a_{11} \oplus a_{12})) \vee (a_{13} \oplus a_{14})) \rightarrow (a_5 \oplus a_{19})$	22,24
26	$d_3 \rightarrow (a_5 \oplus a_{19})$	21,25
27	$(s \oplus d_1) \rightarrow (d_2 \oplus (a_5 \oplus a_{19}))$	19,26
28	$(s \oplus d_1) \rightarrow ((d_2 \oplus a_5) \oplus a_{19})$	27
29	$(s \oplus d_1) \rightarrow ((a_4 \oplus d_1) \oplus a_{19})$	20, 28
30	$(s \oplus d_1) \rightarrow ((a_{19} \oplus a_4) \oplus d_1)$	29
31	$(s \oplus d_1) \rightarrow (e \oplus d_1)$	17, 30

Applying the verification algorithm to the formulas in TABLE 16, we obtain a single resulting formula 31, $(s \oplus d_1) \rightarrow (e \oplus d_1)$, in TABLE 17. This resulting formula can be represented graphically as a workflow model containing a basic cycle without any

structural anomaly (FIGURE 17). According to THEOREM 4, we can declare that the revised model in FIGURE 15 contains no structural anomaly.

FIGURE 17. A Workflow Model Containing a Basic Cycle

3.4. Instance Process Inference

We have shown previously that logic-based workflow verification can verify the correctness of both acyclic and cyclic workflow models that do not contain overlapping workflow patterns. In this subsection, we propose the idea of instance process inference to detect structural anomalies in acyclic workflow models that contain overlapping workflow patterns.

By *overlapping workflow pattern*, it means that a workflow model contains such a structure that a split vertex leads to multiple immediate join vertices as exemplified in FIGURE 18. For instance, an *AND-Split* vertex c_1 leads to two immediate *XOR-Join* vertices, c_4 and c_5 , thus creating an overlapping pattern.

FIGURE 18. An Overlapping Workflow Pattern (Same as FIGURE 12)

TABLE 18 illustrates the logical formulas for the seven basic constructs of acyclic workflows in FIGURE 19. A logical formula is the logical representation of a workflow construct, and a set of instance logical formulas is the logical representation of all possible instances of a workflow. A workflow instance is a specific execution of a workflow.

TABLE 18. Workflow Constructs and Their Logical formulas

No.	Constructs	Definition Logical Formula	Instance Logical Formula
1	<i>Sequence</i>	$a_i \rightarrow a_j$	$a_i \rightarrow a_j$
2	<i>AND-Split</i>	$a_i \rightarrow ((a_j \wedge a_s) \wedge a_t)$	$a_i \rightarrow ((a_j \wedge a_s) \wedge a_t)$
3	<i>AND-Join</i>	$((a_i \wedge a_j) \wedge a_s) \rightarrow a_t$	$((a_i \wedge a_j) \wedge a_s) \rightarrow a_t$
4	<i>XOR-Split</i>	$a_i \rightarrow ((a_j \oplus a_s) \oplus a_t)$	$a_i \rightarrow a_j, a_i \rightarrow a_s, a_i \rightarrow a_t$
5	<i>XOR-Join</i>	$((a_i \oplus a_j) \oplus a_s) \rightarrow a_t$	$a_i \rightarrow a_t, a_j \rightarrow a_t, a_s \rightarrow a_t$
6	<i>OR-Split</i>	$a_i \rightarrow ((a_j \vee a_s) \vee a_t)$	$a_i \rightarrow a_j, a_i \rightarrow a_s, a_i \rightarrow a_t,$ $a_i \rightarrow (a_j \wedge a_s), a_i \rightarrow (a_j \wedge a_t), a_i \rightarrow (a_s \wedge a_t),$ $a_i \rightarrow ((a_j \wedge a_s) \wedge a_t)$
7	<i>OR-Join</i>	$((a_i \vee a_j) \vee a_s) \rightarrow a_t$	$a_i \rightarrow a_t, a_j \rightarrow a_t, a_s \rightarrow a_t,$ $(a_i \wedge a_j) \rightarrow a_t, (a_i \wedge a_s) \rightarrow a_t, (a_j \wedge a_s) \rightarrow a_t,$ $((a_i \wedge a_j) \wedge a_s) \rightarrow a_t$

FIGURE 19. Basic Constructs of Acyclic Workflows (Same as FIGURE 4(a) – (g))

We use “definition logical formulas” to refer to the logical formulas that correspond to the workflow definition and “instance logical formulas” to refer to those that correspond to the workflow instances. Note that the definition logical formulas and the instance logical formulas are the same for *Sequence*, *AND-Split*, and *AND-Join* constructs. The definition logical formulas of *XOR-Split*, *XOR-Join*, *OR-Split*, and *OR-Join* constructs are different from their instance logical formulas.

Although the definition process inference can verify the logical models of workflows, it fails when a workflow model contains an overlapping workflow pattern, where two or more split-join pairs are intertwined together. As a result, the process inference based on

definition logical formulas is not complete because it cannot guarantee logical reduction in the presence of overlapping patterns. Hence, we use instance process inference to break down the overlapping patterns, thereby verifying the correctness of workflow models that contain overlapping patterns.

The overlapping pattern in FIGURE 18 can be converted into the following logical formulas: $s \rightarrow (a_1 \oplus a_2)$, $a_1 \rightarrow (a_3 \wedge a_4)$, $a_2 \rightarrow (a_5 \wedge a_6)$, $(a_3 \oplus a_5) \rightarrow a_7$, $(a_4 \oplus a_6) \rightarrow a_8$, and $(a_7 \wedge a_8) \rightarrow e$. These six formulas can be transformed into two formulas via logical reduction, $s \rightarrow ((a_3 \wedge a_4) \oplus (a_5 \wedge a_6))$ and $((a_3 \oplus a_5) \wedge (a_4 \oplus a_6)) \rightarrow e$, which cannot be further reduced even though this workflow pattern is known to be correct as shown next using its instance logical formulas:

- 1) $s \rightarrow a_1$
- 2) $a_1 \rightarrow (a_3 \wedge a_4)$
- 3) $a_3 \rightarrow a_7$
- 4) $a_4 \rightarrow a_8$
- 5) $(a_7 \wedge a_8) \rightarrow e$
- 6) $s \rightarrow a_2$
- 7) $a_2 \rightarrow (a_5 \wedge a_6)$
- 8) $a_5 \rightarrow a_7$
- 9) $a_6 \rightarrow a_8$

These formulas can be reduced logically as follows:

- 10) $s \rightarrow (a_3 \wedge a_4)$ (1, 2)

11) $s \rightarrow (a_7 \wedge a_8)$	(3, 4, 10)
12) $s \rightarrow e$	(5, 11)
13) $s \rightarrow (a_5 \wedge a_6)$	(6, 7)
14) $s \rightarrow (a_7 \wedge a_8)$	(8, 9, 13)
15) $s \rightarrow e$	(5, 14)

Therefore, we conclude that the workflow structure in FIGURE 18 is correct because each logical formula has been used to derive two possible paths from s to e , as indicated by formulas 12 and 15. In sum, instance process inference verifies a workflow based on two principles: 1) all possible instance paths must lead from s to e , and 2) all logical formulas must be used by the logical reduction.

FIGURE 20 illustrates the rationale of instance process inference, with the highlighted elements representing a workflow instance in each of FIGURE 20(a) and (b). Essentially, if an instance process inference “walks through” successfully all possible paths from the start vertex s to the end vertex e of the workflow model, the workflow model is free from anomalies.

FIGURE 20. A Graphical Representation of Instance Process Inference

Similarly, instance process inference can be applied to the overlapping pattern in FIGURE 21. The instance logical formulas are: $s \rightarrow (a_1 \wedge a_2)$, $a_1 \rightarrow (a_3 \wedge a_4)$, $a_2 \rightarrow (a_5 \wedge a_6)$, $a_3 \rightarrow a_7$, $a_4 \rightarrow a_8$, $a_5 \rightarrow a_7$, $a_6 \rightarrow a_8$, and $(a_7 \wedge a_8) \rightarrow e$. By logical reduction, it is easy to derive $s \rightarrow (e \wedge e)$, indicating that the end vertex e is executed twice. That is, a lack of synchronization anomaly is found.

FIGURE 21. An Overlapping Pattern Containing Lack of Synchronization

The two examples of instance process inference show how instance logical formulas can be used to verify overlapping patterns because at the instance level, the overlap among control vertices is broken by instance logical formulas.

4. PROCESS GRAPHS

When we applied propositional logic with constraints to workflow verification in Section 3, we used *AND*, *OR*, and *XOR* control vertices to explicitly represent the control flow perspective of workflow models. In this section, we formalize such representation by proposing a process modeling language called process graphs. We define process graphs, study their properties and analysis methods, and demonstrate that the proper use of process graphs can prevent certain process anomalies such as activities without termination or activation.

4.1. Process Models

Before we use process graphs to represent the control flow perspective of process models (or simply process models), we give a precise definition of process models.

DEFINITION 1 (process model). A *process model* is a 2-tuple $M = (A, O)$, where

- (1) *activity set* A is a non-empty finite set of distinct activities, and
- (2) *execution order* O is a finite set of structural precedence constraints specifying what activities in A must be executed before or after the execution of each activity in A . ■

For example, for a process model $M = (A, O)$, $A = \{a_1, a_2\}$, $O = \{o_1, o_2, o_3\}$, $o_1 =$ (the process starts with a_1 that has no preceding activity), $o_2 =$ (after a_1 is executed, either a_2 is executed or the process terminates), and $o_3 =$ {after a_2 is executed, the process terminates}.

4.2. Definitions of Process Graphs

4.2.1. Syntax of Process Graphs

Before defining process graphs, we introduce the related concepts of graph theory (Bang-Jensen and Gutin 2001; Chartrand and Lesniak 1996; Wilson 1996). A directed graph $D = (V(D), A(D))$ consists of a vertex set $V(D)$ and an arc set $A(D)$ of ordered pairs of elements of $V(D)$. For a directed arc (or simply arc) vw , the first vertex v is its *tail*, i.e., $\text{tail}(vw) = v$, and the second vertex w is its *head*, i.e., $\text{head}(vw) = w$; v and w are also called *end-vertices* of vw ; we say that vw *leaves* v and *enters* w , and that vw is a *leaving arc* of v and an *entering arc* of w ; we say that v *dominates* w (or w is *dominated by* v) and denote it by $v \rightarrow w$; we say that v and w are *adjacent*, and that v is *adjacent to* w and w is *adjacent from* v ; and we say that v and w are *incident* with vw , and that vw *joins* v and w .

For a vertex v , the set $N^-(v)$ consists of all vertices adjacent to v , and the set $N^+(v)$ consists of all vertices adjacent from v . The sets $N^-(v)$, $N^+(v)$, and $N(v) = N^-(v) \cup N^+(v)$ are called the *in-neighborhood*, *out-neighborhood*, and *neighborhood* of v . The vertices in $N^-(v)$, $N^+(v)$, and $N(v)$ are called the *in-neighbors*, *out-neighbors*, and *neighbors* of v . The *in-degree* (*out-degree*) of v , denoted by $d^-(v)$ ($d^+(v)$), is the number of arcs with head (tail) v . The *degree* of a vertex v is the sum of its in-degree and out-degree, i.e., $d(v) = d^-(v) + d^+(v)$.

Subgraphs of a graph P are obtained by deleting arcs and vertices from P . If uv is an arc of P , we denote by $P - uv$ the graph obtained from P by deleting the arc uv . More generally, if F is any set of arcs in P , we denote by $P - F$ the graph obtained from P by deleting the arcs in F . Similarly, if w is a vertex of P , we denote by $P - w$ the graph

obtained from P by deleting the vertex w together with the arcs incident with w . More generally, if H is any set of vertices in P , we denote by $P - H$ the graph obtained from P by deleting the vertices in H and all arcs incident with any of the vertices in H .

A digraph D is *connected* if its underlying undirected graph G is connected. An undirected graph G is *connected* if and only if between each pair of vertices v and w in G , there is a path, denoted by vw -path.

DEFINITION 2 (process graph). A *process graph* is a 5-tuple $P = (V_A(P), V_C(P), A(P), s, e)$, where

- (1) *activity vertex set* $V_A(P)$ is a non-empty finite set of distinct elements called *activity vertices*, and $\forall v \in V_A(P), d^-(v) = d^+(v) = 1$,
- (2) *control vertex set* $V_C(P)$ is a finite set of distinct elements called *control vertices*, and $\forall v \in V_C(P), d^-(v) \geq 1, d^+(v) \geq 1, d(v) \geq 3$, and $T_C(v) \in \{AND, XOR, OR\}$ is the type of v ,
- (3) s is the *start vertex*, and $d^-(s) = 0, d^+(s) = 1$,
- (4) e is the *end vertex*, and $d^-(e) = 1, d^+(e) = 0$,
- (5) *directed arc set* (or simply *arc set*) $A(P)$ is a non-empty finite set of distinct ordered pairs called *arcs* of distinct elements of *vertex set* $V(P) = V_A(P) \cup V_C(P) \cup \{s, e\}$,
- (6) P is connected, and
- (7) $V_A(P) \cap V_C(P) = \emptyset, V_A(P) \cap \{s, e\} = \emptyset$, and $V_C(P) \cap \{s, e\} = \emptyset$. ■

DEFINITION 2 implies that *loops* (i.e., arcs whose head and tail coincide) or *parallel arcs* (i.e., pairs of arcs with the same tail and the same head) are not allowed in process

graphs. A vertex that is not a control vertex is also called a *non-control vertex*. Non-control vertices include activity vertices, the start vertex s , and the end vertex e . *AND*, *XOR*, and *OR* control vertices are three types of control vertices and are simply called *AND*, *XOR*, and *OR* vertices. FIGURE 22 gives the graphical notation for process graphs.

FIGURE 22. Graphical Notation for Process Graphs (Same as FIGURE 3)

A control vertex v is a *join control vertex* (or simply *join vertex*) and is denoted by $T_C(v)$ -*Join* if $d^-(v) \geq 2$, where $T_C(v) \in \{AND, XOR, OR\}$. Similarly, a control vertex v is a *split vertex* and is denoted by $T_C(v)$ -*Split* if $d^+(v) \geq 2$. A control vertex v can be both a join vertex and a split vertex of the same type, and is denoted by $T_C(v)$ -*Join-Split* that is treated as a combination of $T_C(v)$ -*Join* and $T_C(v)$ -*Split*. When we say a join vertex v or a split vertex w , we also mean that v or w may also be a join-split vertex unless there is a special specification.

For a process graph P_1 in FIGURE 23, which is the representation of the process model M stated in the example in Section 4.1, $V_A(P) = \{a_1, a_2\}$, $V_C(P) = \{c_1\}$, $A(P) = \{sa_1, a_1c_1, a_2c_1, c_1a_2, c_1e\}$; $T_C(c_1) = XOR$, and c_1 is an *XOR-Join-Split* vertex; $\text{tail}(a_1c_1) = a_1$, $\text{head}(a_1c_1) = c_1$; c_1a_2 and c_1e are leaving arcs of c_1 , and c_1e is an entering arc of e ; s is adjacent to a_1 , s and a_1 are incident with sa_1 , and sa_1 joins s and a_1 ; $d^-(c_1) = 2$, $d^+(c_1) = 2$, $d(c_1) = 4$; and $N^-(c_1) = \{a_1, a_2\}$, $N^+(c_1) = \{a_2, e\}$, $N(c_1) = \{a_1, a_2, e\}$.

FIGURE 23. A Process Graph P_1

4.2.2. Semantics of Process Graphs

Semantically, when a non-control vertex v in a process graph P is visited, we say that v is *executed* and denote it by $Executed(v)$; when a control vertex v takes effect, we say that v is *activated* and denote it by $Activated(v)$; and when an arc vw is in a condition under which it alone or together with other arcs causes w to be executed or activated, we say that vw is *active* and denote it by $Active(vw)$. The verb *cause* is denoted by $\mapsto$. $Unexecuted(v)$, $Inactivated(v)$, and $Inactive(vw)$ are defined as the opposites to $Executed(v)$, $Activated(v)$, and $Active(vw)$, respectively. Execution of non-control vertices, activation of control vertices, and active arcs are interdependent.

DEFINITION 3. In a process graph $P = (V_A(P), V_C(P), A(P), s, e)$, the *condition for a non-control vertex to be executed* is defined as

- (1) $Executed(s)$, and
- (2) $Active(vw) \mapsto Executed(w)$, where $vw \in A(P)$ and $w \in V_A(P) \cup \{e\}$. ■

According to DEFINITION 3, the execution of a non-control vertex w is caused by the arc with head w being active. Because the start vertex s has no entering arc, s is executed unconditionally when processes start.

DEFINITION 4. In a process graph $P = (V_A(P), V_C(P), A(P), s, e)$, the *condition for a control vertex c to be activated* is defined as

- (1) $\{Active(vc) \mid \forall vc \in A(P)\} \mapsto Activated(c)$, where $c \in V_C(P)$ and $T_C(c) = AND$,
- (2) $Active(vc)$ and $\{Inactive(wc) \mid \forall wc \in A(P) \text{ and } w \neq v\} \mapsto Activated(c)$, where $c \in V_C(P)$, $T_C(c) = XOR$, and $vc \in A(P)$, and
- (3) $\{Active(vc) \mid \exists vc \in A(P)\} \mapsto Activated(c)$, where $c \in V_C(P)$ and $T_C(c) = OR$. ■

DEFINITION 4 specifies that an *AND* vertex is activated when all of its entering arcs are active. Similarly, an *XOR* (*OR*) vertex is activated when exactly one (at least one) of its entering arcs is active. Particularly, when a control vertex is not a join vertex, it is activated when its only entering arc is active.

DEFINITION 5. In a process graph $P = (V_A(P), V_C(P), A(P), s, e)$, the *condition for an arc to become active* is defined as

- (1) $Executed(v) \mapsto Active(vw)$, where $v \in V_A(P) \cup \{s\}$ and $vw \in A(P)$,
- (2) $Activated(c) \mapsto \{Active(cv) \mid \forall cv \in A(P)\}$, where $c \in V_C(P)$ and $T_C(c) = AND$,
- (3) $Activated(c) \mapsto Active(cv)$ and $\{Inactive(cw) \mid \forall cw \in A(P) \text{ and } w \neq v\}$, where $c \in V_C(P)$, $T_C(c) = XOR$, and $cv \in A(P)$, and
- (4) $Activated(c) \mapsto \{Active(cv) \mid \exists cv \in A(P)\}$, where $c \in V_C(P)$ and $T_C(c) = OR$. ■

DEFINITION 5 indicates that arcs become active under four different conditions. The execution of a non-control vertex (except the end vertex e) causes its leaving arc to become active. The activation of an *AND* (*XOR*, *OR*) vertex causes all (exactly one, at least one) of its leaving arcs to become active. Particularly, when a control vertex is not a split vertex, its activation causes its only leaving arc to become active.

Based on DEFINITION 3, DEFINITION 4, and DEFINITION 5, sequences in which vertices are executed or activated can be generated. For simplicity without ambiguity, we exclude arcs in such execution and activation sequences. Vertices are executed or activated sequentially or in parallel. In sequential execution and activation, vertices are executed or activated one after another. For instance, $Executed(s) \mapsto Executed(a_1) \mapsto Activated(c_1) \mapsto Executed(e)$ in FIGURE 23 is an example of sequential execution and activation. In parallel execution and activation, vertices are executed or activated in parallel. The activation of an *AND-Split* (*OR-Split*) vertex causes all (at least one) of its leaving arcs to become active. These arcs are active independently of each other, thus causing their head vertices to be executed or activated in parallel. For example, after c_2 in FIGURE 24 is activated, a_2 and a_4 are executed in parallel.

FIGURE 24. A Process Graph P_2

4.2.3. Standard Process Graphs

Control vertices in process graphs can be adjacent. The process graphs in which control vertices are not adjacent are called standard process graphs, which have desirable characteristics.

DEFINITION 6 (standard process graph). A *standard process graph* P_S is a process graph P in which for any control vertex $c \in V_C(P)$, $N(c) \subseteq V_A(P) \cup \{s, e\}$. A process graph that is not a standard process graph is called a *non-standard process graph*. ■

In a standard process graph, each control vertex c is adjacent only to and from non-control vertices. For a process graph P , its corresponding standard process graph P_S can be generated as follows: for each pair of adjacent control vertices v and w joined by arc vw , remove arc vw , add a dummy activity vertex u , and add arcs vu and uw . A *dummy activity* is an artificial activity that is used to satisfy the definition of standard process graphs. FIGURE 25 gives the standard process graph of P_2 in FIGURE 24 after adding dummy activity vertices d_1, d_2, d_3 , and d_4 . Because there is only one way to add a dummy activity vertex between two adjacent control vertices, a process graph P has only one corresponding standard process graph P_S .

FIGURE 25. Standard Process Graph P_{S2} of P_2 in FIGURE 24

4.2.4. Process Constructs

Standard process graphs consist of process constructs that allow for modularized design and analysis of both acyclic and cyclic process graphs.

DEFINITION 7 (process construct). In a standard process graph P_S , a *process construct* (or simply *construct*) consists of (1) $u, v \in V_A(P_S) \cup \{s, e\}$, and uv or $vu \in A(P_S)$, or (2) $c \in V_C(P_S)$, $N(c)$, vc , and cw , $\forall v \in N^-(c)$ and $\forall w \in N^+(c)$. ■

FIGURE 26 illustrates four process constructs. A *Sequence construct* consists of two non-control vertices joined by an arc. An *AND (XOR, OR) construct* consists of an *AND (XOR, OR)* vertex with its neighbors, and arcs with which the *AND (XOR, OR)* vertex and its neighbors are incident. In a *Sequence* construct, the execution of a vertex causes the execution of the other vertex. In an *AND (XOR, OR)* construct, the *AND (XOR, OR)* vertex is activated by the execution of all (exactly one, at least one) of its in-neighbors, and then the activated control vertex, in turn, causes the execution of all (exactly one, at least one) of its out-neighbors.

FIGURE 26. Four Process Constructs

Process constructs are defined based on standard process graphs because it is usually difficult to decompose structures in which control vertices are adjacent. For example, because c_1 is adjacent to c_4 and from c_6 in FIGURE 24, it is difficult to decompose the structure containing c_1 into simpler ones. FIGURE 27 shows eight constructs obtained from decomposing P_{S2} in FIGURE 25.

FIGURE 27. Process Constructs of the Standard Process Graph P_{S2} in FIGURE 25

There is exactly one way to decompose a standard process graph into process constructs. Because an activity vertex has one entering arc and one leaving arc, each activity vertex appears in exactly two constructs. Exactly one standard process graph can be built from a set of constructs by combining each activity vertex in two constructs into one.

4.2.5. Subprocesses

The concept of subprocesses provides a stepwise means to model large-scale and complex processes.

DEFINITION 8 (subprocess, superprocess). In a process graph P , if an activity vertex $v \in V_A(P)$ is itself another process graph Q and $V(P) \cap V(Q) = \emptyset$, then Q is called a *subprocess* of P , and P is called the *superprocess* of Q . ■

A process graph may have multiple subprocesses, whereas a subprocess has only one superprocess. A subprocess may have its own subprocesses. A process graph and its subprocesses can be represented as a nested model. FIGURE 28(a) shows an example of

nested process graphs in which P has two subprocesses R and T , and R has one subprocess S .

FIGURE 28. A Process Graph P and Its Subprocesses

A subprocess can be incorporated into its superprocess by substituting the corresponding activity in the superprocess with the subprocess, with the subprocess' start and end vertices being deleted. There is only one way to incorporate each subprocess into its superprocess. FIGURE 28(b) shows the process graph after subprocesses in FIGURE 28(a) are incorporated into their superprocesses.

The purpose of expressing each subprocess as a complete process graph is to facilitate independent analysis of each subprocess, thus decreasing the complexity of analysis while maintaining the integrity of the entire process model. The key to representing a process graph P as nested process graphs is to identify blocks in P .

DEFINITION 9 (block). In a process graph P , $B = (V(B), A(B))$ is a *block*, denoted by (v_i, v_t) -*block*, if

- (1) $V(B) \subseteq V(P) - \{s, e\}$, $A(B) \subseteq A(P)$, $A(B) \neq \emptyset$, and $\forall vw \in A(B)$, $v \in V(B)$ and $w \in V(B)$, and
- (2) there is exactly one entering arc from $P - B$ to B and exactly one leaving arc from B to $P - B$.

We say that the vertex v_i in B that is incident with the entering arc from $P - B$ to B is the *initial vertex* of B , the vertex v_t in B that is incident with the leaving arc from B to $P - B$ is the *terminal vertex* of B , and v_i and v_t are *end-vertices* of B . ■

Because each block contains at least one arc, an activity vertex cannot itself be a block. In a process graph, blocks may be identified differently as long as they satisfy the definition of blocks. Because a block has one entering arc and one leaving arc, it becomes a process graph after a start vertex and an end vertex are added. FIGURE 29 illustrates three blocks identified in FIGURE 28(b).

FIGURE 29. Three Blocks in FIGURE 28(b)

In a process graph P , subprocesses can be modeled in the following procedure: (1) identify blocks $B_1, B_2, \dots, B_k$ ($k \geq 1$) in P ; (2) replace each block with a new activity vertex in P ; and (3) add a start vertex and an end vertex to each block with a copy of the

entering arc and the leaving arc of the block to make the block a subprocess. If multi-level subprocesses are to be modeled, the above procedure can be again applied to each subprocess.

4.2.6. Adjacency Matrix

A process graph can be stored in a computer in the form of an adjacency matrix.

DEFINITION 10 (adjacency matrix). For a process graph P with vertex set $V(P) = \{v_1, v_2, \dots, v_n\}$, *adjacency matrix* $M(P) = [m_{ij}]$ is an $n \times n$ matrix such that $m_{ij} = 1$ if $v_i v_j \in A(P)$, and $m_{ij} = 0$ otherwise, where $1 \leq i, j \leq n$. ■

For instance, TABLE 19 gives the adjacency matrix for the process graph P_1 in FIGURE 23. The adjacency matrix is a convenient and systematic tool to check whether two vertices are adjacent. We will show later how to use an adjacency matrix to analyze a process graph.

TABLE 19. Adjacency Matrix for P_1 in FIGURE 23

$M(P)$	s	a_1	a_2	c_1	e
s	0	1	0	0	0
a_1	0	0	0	1	0
a_2	0	0	0	1	0
c_1	0	0	1	0	1
e	0	0	0	0	0

4.2.7. Instance Walks

Before we define instance walks, we need to introduce process instances. In a process graph, a *process instance* is an instantiation of a process definition and can be expressed as the executions or activations of activities and control vertices in a specified order. A process instance comprises activities and control vertices that are actually executed or activated during runtime. FIGURE 30 illustrates all three process instances, which are highlighted, of a process graph P_3 .

FIGURE 30. Three Process Instances of a Process Graph P_3

The concepts of walks, paths, and cycles in digraphs are applicable to process graphs. In a process graph P , a *walk from v_1 to v_k* , or a (v_1, v_k) -*walk*, is a finite sequence $\bar{w} = v_1 v_2 \dots v_k$ such that $v_i v_{i+1} \in A(P)$ for every $i = 1, 2, \dots, k-1$. A *path $\bar{p}$* is a walk in which all vertices are distinct. A *cycle $\bar{c}$* is a closed path $v_1 v_2 \dots v_k$ where $v_1 = v_k$ and $k \geq 2$. If v

is a vertex in $\vec{C}$, then we say that $\vec{C}$ is a *cycle through* v . If $\vec{C}$ is a cycle through v and v is a join (split) control vertex, then we say that v is an *entering (exit) vertex* of $\vec{C}$.

For instance, in a process graph P_4 in FIGURE 31, $sc_1a_1c_2a_2c_3e$ is a walk. Cycle $c_1a_1c_2a_3c_1$ has one exit vertex c_2 , and cycle $c_1a_1c_2a_3a_4c_1$ has two exit vertices c_2 and c_3 .

FIGURE 31. A Process Graph P_4

Because of parallel execution and activation of vertices, instance walks are distinctive and complex features of process graphs in addition to walks. Instance walks are an important concept on which many analysis methods are built. Before we define instance walks, it is necessary to define virtual walks that are useful for studying parallel execution of activity vertices.

DEFINITION 11 (virtual walk, virtual path, virtual cycle). $v_1v_2 \dots v_k$ is a (v_1, v_k) -virtual-walk $\vec{W}_v$ if it is generated from a (v_1, v_k) -walk with control vertices being omitted from the (v_1, v_k) -walk. A *virtual path* $\vec{P}_v$ is a virtual walk in which all vertices are distinct. A *virtual cycle* $\vec{C}_v$ is a closed virtual path $v_1v_2 \dots v_k$ where $v_1 = v_k$ and $k \geq 2$. ■

For example, $\mathbf{a_1a_2a_3a_1a_2a_3a_7}$ in FIGURE 24 is a virtual walk in which $\mathbf{a_1a_2a_3a_1}$ is a virtual cycle, with the bold parts representing a virtual cycle. In a (v_1, v_k) -virtual-walk, virtual cycles can be identified as follows:

Step 1: Start from the first vertex in the (v_1, v_k) -virtual-walk.

Step 2: If no vertex (including the current vertex) occurs twice in the rest of the (v_1, v_k) -virtual-walk, stop; otherwise, identify the first vertex v_j ($1 \leq j \leq k$) that occurs twice (e.g., a_1 in $a_1a_2a_3a_1a_2a_3a_7$). Two v_j 's and the vertices between v_j 's form a virtual cycle.

Step 3: Ignore the vertices between v_j 's (e.g., a_2 and a_3 between two a_1 's in $a_1a_2a_3a_1a_2a_3a_7$), start from the second v_j , and go to Step 2.

An (s, e) -virtual-walk $sa_1a_3a_1a_2a_4a_1a_3a_1a_2e$ in FIGURE 31 contains three virtual cycles in order: $a_1a_3a_1$, $a_1a_2a_4a_1$, and $a_1a_3a_1$, with the bold parts representing virtual cycles.

DEFINITION 12 (instance walk). In a process graph P , an *instance walk from V_1 to V_m* , or a (V_1, V_m) -instance-walk, is a finite feasible execution sequence $\overrightarrow{W}_I = V_1V_2 \dots V_m$ with vertex set $V(\overrightarrow{W}_I) \subseteq V_A(P) \cup \{s, e\}$ such that each of $V_1, V_2, \dots, V_m$ is

- (1) a non-control vertex, or
- (2) a set of activity vertices in a (v_i, v_t) -block B , where activity vertices in B are executed

in parallel and expressed as $\begin{bmatrix} U_1 \\ U_2 \\ \dots \\ U_n \end{bmatrix}$ ($n \geq 2$) in which U_i ($1 \leq i \leq n$) is a virtual walk

between v_i and v_t , and

only after the execution of vertices in V_i is completed, can vertices in V_j be executed, where $1 \leq i < j \leq m$.

An instance walk $\overrightarrow{W}_I$ is *closed* if $\forall v, v \in V_1$ and $v \in V_m$, and *open* otherwise. If $\overrightarrow{W}_I$ is open, then we say that the vertices in V_1 are the *initial vertices* of $\overrightarrow{W}_I$, the vertices in V_m are the *terminal vertices* of $\overrightarrow{W}_I$, and the vertices in V_1 and V_m are *end-vertices* of $\overrightarrow{W}_I$.

An instance walk $\overrightarrow{W}_I$ is an (s, e) -instance-walk if $V_1 = \{s\}$ and $V_m = \{e\}$.

An instance walk $\overrightarrow{W}_I$ is *minimal* if $\overrightarrow{W}_I$ contains no virtual cycles or every virtual cycle exists in $\overrightarrow{W}_I$ no more than once.

The *length of an instance walk* $\overrightarrow{W}_I$ is the length of its longest virtual walk.

The *longest instance walk* of P is the minimal instance walk of maximum length in P .

If $V_1V_2 \dots V_{k-1}V_k$ is a (V_1, V_k) -instance-walk and $V_kV_{k+1} \dots V_{m-1}V_m$ is a (V_k, V_m) -instance-walk, then we say that $V_1V_2 \dots V_{k-1}V_kV_{k+1} \dots V_{m-1}V_m$ is a (V_1, V_m) -instance-walk *extended* from the (V_1, V_k) -instance-walk or from the (V_k, V_m) -instance-walk, or the (V_1, V_k) -instance-walk or the (V_k, V_m) -instance-walk is *extendable* to a (V_1, V_m) -instance-walk. ■

DEFINITION 13 (cyclic process graph). A process graph P is *acyclic* if it contains no virtual cycles, and *cyclic* otherwise. ■

The definition of instance walks considers only non-control vertices because only they are actually executed in process instances that are instantiations of process models. The “feasible execution sequence” in DEFINITION 12 refers to an execution sequence that actually exists in a process graph. For a generic (s, e) -instance-walk $sV_2V_3 \dots V_{m-1}e$, only after the execution of vertices in V_i ($2 \leq i \leq m-1$) is completed, can e be executed.

For example, $a_1 \begin{bmatrix} a_2 a_3 \\ a_4 \end{bmatrix} a_7$ in FIGURE 24 is a feasible execution sequence and thus an $(a_1,$

$a_7)$ -instance-walk. $a_1a_2a_3a_7$ is an (a_1, a_7) -virtual-walk, but is not a feasible execution sequence because in this case, a_7 cannot be executed without a_4 being executed. sa_5a_7e is an (s, e) -instance-walk in FIGURE 24.

For an (s, e) -instance-walk $sV_2V_3 \dots V_{m-1}e$ in a process graph P , each V_i ($2 \leq i \leq m-1$) is either an activity vertex or a set of virtual walks in a (v_i, v_i) -block B in which activity vertices are executed in parallel. v_i and v_i must be *AND* or *OR* vertices where the parallel execution of activity vertices in B originates and terminates, and v_i is the first vertex at which all virtual walks originating from v_i terminate. In addition, B cannot be a subgraph of another block in which the parallel execution involves both activity vertices in B and those not in B , or otherwise the virtual walks in B cannot be expressed as a V_i ($2 \leq i \leq m-1$) item in the $sV_2V_3 \dots V_{m-1}e$ instance walk. Thus, each V_i ($2 \leq i \leq m-1$) in an $sV_2V_3 \dots V_{m-1}e$ instance walk can be unambiguously determined.

For example, the only (s, e) -instance-walk in FIGURE 32 is $s \begin{bmatrix} a_1a_4 \\ a_1a_5 \\ a_2a_3a_4 \\ a_2a_3a_5 \\ a_2a_6 \end{bmatrix} e$, in which a_1a_4 ,

a_1a_5 , $a_2a_3a_4$, $a_2a_3a_5$, and a_2a_6 are five virtual walks in a (c_1, c_4) -block. In other words, all of these five virtual walks originate from c_1 and terminate at c_4 in the (c_1, c_4) -block. The length of this (s, e) -instance-walk is 5 because its longest (s, e) -virtual-walks $sa_2a_3a_4e$ and $sa_2a_3a_5e$ are of length 5. In FIGURE 33, there are two blocks involved in an (s, e) -

instance-walk $s \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} a_4 \\ a_5 \end{bmatrix} e$, a (c_1, c_4) -block and a (c_5, c_6) -block. In the (c_1, c_4) -block, a_1 , a_2 ,

and a_3 are executed in parallel. Note that c_4 , not c_5 or c_6 , is the first vertex at which all virtual walks originating from c_1 terminate. Because a (c_2, c_3) -block is a subgraph of the

(c_1, c_4) -block, the (c_2, c_3) -block cannot be written as a V_i ($2 \leq i \leq m - 1$) item in this $sV_2V_3 \dots V_{m-1}e$ instance walk.

FIGURE 32. A Process Graph P_5

FIGURE 33. A Process Graph P_6

Minimal instance walks contain no virtual cycles or contain each virtual cycle no more than once. Minimal instance walks are a useful concept because they are sufficient to reveal the structure of a process graph. The use of minimal instance walks avoids considering instances walks that contain repeated virtual cycles. Two minimal instance walks that contain identical virtual cycles in different orders are considered different minimal instance walks. For instance, the process graph P_4 in FIGURE 31 contains five minimal instance walks: sa_1a_2e , $sa_1a_3a_1a_2e$, $sa_1a_2a_4a_1a_2e$, $sa_1a_3a_1a_2a_4a_1a_2e$, and $sa_1a_2a_4a_1a_3a_1a_2e$, with the bold parts representing virtual cycles. The last two minimal instance walks contain two virtual cycles $a_1a_3a_1$ and $a_1a_2a_4a_1$ in different orders, and thus they are different minimal instance walks.

4.2.8. Isomorphism

DEFINITION 14 (isomorphic process graphs). Two process graphs P_1 and P_2 are *isomorphic*, denoted by $P_1 \cong P_2$, if there is a one-to-one correspondence between the minimal (s, e) -instance-walks of P_1 and those of P_2 such that each pair of the corresponding minimal (s, e) -instance-walks is identical. ■

For example, the two process graphs in FIGURE 34 are isomorphic since they contain

the same minimal (s, e) -instance-walk $s \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} e$. The two process graphs in FIGURE 35 are

also isomorphic because each of them contains three minimal (s, e) -instance-walks, i.e., sa_1a_2e , sa_3e , and sa_4e .

FIGURE 34. Two Isomorphic Process Graphs – Example 1

FIGURE 35. Two Isomorphic Process Graphs – Example 2

Different designers may map a given set of activities with specified execution order differently. The concept of isomorphism of process graphs provides an effective way to determine whether two process models are identical.

4.3. Properties of Process Graphs

In this subsection, we discuss the properties of process graphs.

4.3.1. Use of Explicit Control Vertices

In comparison with conventional digraphs, the most distinguishing feature of process graphs is that they contain two types of vertices, activity vertices (the start and end vertices can be treated as a special type of activity vertices) and control vertices. Without explicit control vertices, many process scenarios cannot be easily modeled. For instance, without using control vertices, FIGURE 36 tries to model the same process scenarios shown in FIGURE 24 by using descriptions among arcs. In FIGURE 24, multiple splits and joins of control flows, expressed by control vertices c_1 and c_4 and by c_3 , c_5 , and c_6 ,

are mixed together. It is obvious that the meanings represented by vertices c_1 and c_6 in FIGURE 24 cannot be easily added to FIGURE 36.

FIGURE 36. A Process Model without Using Control Vertices

We prove next that any arbitrary process model can be represented by a process graph.

THEOREM 5 (Completeness Theorem). A process model $M = (A, O)$ can be represented by a process graph $P = (V_A(P), V_C(P), A(P), s, e)$, where A is a non-empty finite set of distinct activities, and O is the execution order of M that is a finite set of structural precedence constraints specifying what activities in A must be executed before the execution of each activity in A . ▲

Proof. Each activity in the activity set A of a process model $M = (A, O)$ can be represented by an activity vertex of a process graph P . Because the execution order O specifies the structural precedence constraints in M that include five cases: (1) activities are executed sequentially, (2) activities are executed in parallel, (3) exactly one of multiple activities is executed, (4) at least one of multiple activities is executed, and (5) combinations are made of the first four cases. These five cases can be modeled in P with (1) *Sequence* constructs, (2) *AND* vertices, (3) *XOR* vertices, (4) *OR* vertices, and (5)

combinations of *Sequence* constructs and control vertices, respectively. Because P can represent both the activity set A and the execution order O of M , P can represent M . $\square$

COROLLARY 6. A process model $M = (A, O)$ can be represented by a process graph P such that P has a single start vertex s and a single end vertex e . $\blacktriangle$

Proof (by construction). When a process model M is represented by a process graph P , if M has only one start activity, adding a start vertex s and an arc from s to this start activity will be sufficient. If M has more than one start activity, and if it is required to execute all (exactly one, at least one) of these start activities in order to start a process instance that is an instantiation of M , then add an *AND-Split* (*XOR-Split*, *OR-Split*) vertex and a start vertex s , add an arc from s to this new control vertex, and add an arc from this control vertex to each start activity of M . As a result, M can be represented by P that has a single start vertex s . Similarly, we can prove that M can be represented by P that has a single end vertex e . $\square$

By COROLLARY 6, multiple start points and multiple end points of a process model can be combined into one start vertex and one end vertex in a process graph, thus ensuring the tractability of the process graph.

4.3.2. Use of *OR* Vertices

There are evident advantages of using *OR* vertices. First, many business scenarios call for flexible business processes that enable customers to make payment in any possible combinations of cash, check, and credit card, to choose m ($m \geq 1$) out of a group of reviewers to review papers, and so on. The use of *OR* vertices is a solution to the problem of representing these business processes efficiently. Second, although at the

computational level, an *OR* vertex c can be substituted with a combination of *XOR* and *AND* vertices, such substitution is awkward at the graphical representation level even when $d^+(c)$ or $d^-(c)$ is not very large. FIGURE 37 gives an example of replacing *OR* vertices with *XOR* and *AND* vertices; that is, the process graph in FIGURE 37(a) is isomorphic to that in FIGURE 37(b). After an *OR-Split* (*OR-Join*) vertex c with $d^+(c) = k$ ($d^-(c) = k$), such as c_1 (c_2) in FIGURE 37(a), is substituted with *XOR* and *AND* vertices, the out-degree (in-degree) of the corresponding *XOR-Split* (*XOR-Join*) vertex, such as c_1 (c_{16}) in FIGURE 37(b), is $C_k^1 + C_k^2 + \dots + C_k^k = 2^k - 1$, which increases exponentially with the increase of $d^+(c)$ ($d^-(c)$). Third, the use of *OR* vertices gives a concise graphical representation that avoids the cumbersome numeration of all possible combinations of a set of activities when (at least) m out of n ($1 \leq m \leq n$) activities are executed (van der Aalst and Kumar 2003). Fourth, at the computational level, *OR* vertices can be implemented in a way that all possible scenarios with specified constraints and runtime conditions are examined. Therefore, the succinct representation at the graphical level and the executability at the computational level make *OR* vertices desirable.

The implementation of *OR* vertices depends on domain requirements. For instance, if FIGURE 37(a) represents the process of selecting two out of three reviewers to review papers, a constraint can be added to c_1 such that after it is activated, exactly two leaving arcs of c_1 will become active. If FIGURE 37(a) represents the process of booking a flight, booking a hotel, and/or booking a car in any possible combinations, after c_1 is activated, one or more leaving arcs of c_1 will become active, depending on the information that c_1

receives during runtime. In either case, c_2 can be implemented in a way such that before it is activated, it will receive the information about the number of active leaving arcs of c_1 .

FIGURE 37. Use of *OR* Vertices

However, if *OR-Join* vertices are abused, they may cause unnecessary non-determinism. For an *OR-Join* vertex to be activated, how many arcs and which arcs should become active are determined depending on design specifications and runtime conditions. Therefore, although *OR-Join* vertices are essentially a generalization of *AND-Join* and *XOR-Join* vertices, unnecessary non-determinism will be introduced if an *OR-Join* vertex is used where an *AND-Join* or *XOR-Join* vertex is sufficient.

The case of structural unnecessary non-determinism does not include *OR-Split* vertices because the improper use of *OR-Split* vertices is more related to the validation of process models. For example, suppose that design specifications require that a_1 and a_2 in FIGURE 38 both be executed in all cases, and thus c_1 and c_2 should both be *AND* vertices. If both c_1 and c_2 in FIGURE 38 were *OR* vertices, then the unnecessary non-determinism introduced by the *OR-Split* vertex c_1 could not be recognized through structural analysis alone.

FIGURE 38. A Process Graph P_7

4.3.3. Correctness of Process Graphs

In this subsection, we show how process graphs can be used to analyze the correctness of process models.

DEFINITION 15 (correct process graph). A process graph P is *correct* if every instance walk of P with s as the initial vertex can be extended to an (s, e) -instance-walk with e being executed exactly once, and if every activity vertex in P is in at least one (s, e) -instance-walk. Otherwise, P is an *incorrect process graph*. ■

In FIGURE 39, the first condition (*terminability*) implies that every instance walk of P starting from s will eventually end at e , no vertices can be further executed or activated after e is executed, and e cannot be executed more than once. The second condition

(*nonredundancy*) implies that there is no redundant activity vertex in P that will never be executed.

Three examples of incorrect process graphs are given in FIGURE 39. FIGURE 39(a) violates the terminability condition because it contains a deadlock in which an *AND-Join* vertex c_2 is mismatched to an *XOR-Split* vertex c_1 . Since only one of a_1 and a_2 is executed after c_1 is activated, c_2 will wait forever and block the continuation of the process, and thus e can never be reached. FIGURE 39(b) also violates the terminability condition because it does not contain an (s, e) -instance-walk $sV_2V_3 \dots V_{m-1}e$, although its execution sequence can be written as $s \begin{bmatrix} a_1 e \\ a_1 a_2 a_1 \end{bmatrix}$ that is not an (s, e) -instance-walk. FIGURE 39(c) violates the nonredundancy condition because a_3 can never be executed due to a deadlock at c_2 , although both (s, a_1) -instance-walk and (s, a_2) -instance-walk can be extended to an (s, e) -instance-walk.

FIGURE 39. Examples of Incorrect Process Graphs

The following theorem establishes the relationship between the correctness of a process graph and that of its corresponding standard process graph.

THEOREM 7. A process graph P and its corresponding standard process graph P_S have the same correctness or incorrectness. ▲

Proof. Suppose that in P there are only two adjacent control vertices v and w incident with arc vw . P_S is created after a dummy activity vertex u is added between v and w as follows: remove vw , add u , and add arcs vu and uw . Because P and P_S are identical except that u is added to P_S , after v in both P and P_S is activated (remains inactivated), vu in P_S has the same active (inactive) status as vw in P . The active (inactive) vu in P_S causes u to be executed (unexecuted), and then uw becomes active (remains inactive). Because vu , u , and uw have the same effect on w in P_S as vw on w in P , semantically the addition of the dummy activity vertex u does not cause correct P to become incorrect and does not cause incorrect P to become correct. This proof can be generalized to the case when more than one dummy activity vertex is added. □

4.4. Analysis Methods for Process Graphs

In this subsection, we mention two process analysis methods of process graphs. When a process graph is analyzed, it is not mandatory to modify it into a standard process graph. If it is needed to decompose a process graph into process constructs or to convert a process graph to logical formulas, such modification is necessary. If it is only required to build subprocesses, identify instance walks, construct an adjacency matrix, or recognize isomorphism, the original process graph is sufficient.

4.4.1. Adjacency-Matrix-Based Analysis

An adjacency matrix can be used to check whether the definition of process graphs is satisfied.

THEOREM 8. For a process graph $P = (V_A(P), V_C(P), A(P), s, e)$ with vertex set $V(P) = \{v_1, v_2, \dots, v_n\}$, P is incorrect if its adjacency matrix $M(P) = [m_{ij}]_{n \times n}$ ($1 \leq i, j \leq n$) violates any of the following eight criteria: (1) The number of rows (columns) of $V_A(P) \geq 1$. (2) $0 \leq m_{ij} \leq 1$ ($1 \leq i, j \leq n$). (3) $\forall v_k \in V_A(P), \sum_{j=1}^n m_{kj} = \sum_{i=1}^n m_{ik} = 1$. (4) $\forall v_k \in V_C(P), \sum_{j=1}^n m_{kj} \geq 1, \sum_{i=1}^n m_{ik} \geq 1, \sum_{j=1}^n m_{kj} + \sum_{i=1}^n m_{ik} \geq 3$. (5) For $v_k = s$ ($1 \leq k \leq n$), $\sum_{j=1}^n m_{kj} = 1, \sum_{i=1}^n m_{ik} = 0$. (6) For $v_k = e$ ($1 \leq k \leq n$), $\sum_{j=1}^n m_{kj} = 0, \sum_{i=1}^n m_{ik} = 1$. (7) $m_{ii} = 0$ ($1 \leq i \leq n$). $\blacktriangle$

Proof. These seven criteria can be directly obtained from DEFINITION 2. $\square$

THEOREM 8 states a necessary condition for a process graph to be correct, but not a sufficient condition. THEOREM 8 can be used to check whether the definition of process graphs is satisfied. A process graph can be incorrect even if it satisfies the definition of process graphs, such as those in FIGURE 39. In order to detect such process anomalies, other analysis methods must be applied.

4.4.2. Logic-Based Analysis

Logic-based process analysis will be addressed in Section 5.3.

5. PROCESS LOGIC

In Section 3, we used propositional logic with constraints to verify activity-based workflow models. The purpose was to take advantage of the logical formalism and deduction to reason about the correctness of process models. However, propositional logic has some limitations in describing process phenomena. For instance, from a formula of propositional logic, it cannot tell whether the represented process structure is a join structure in which multiple activities come to a point or a split structure in which multiple activities leave from a point. In addition, the definition of the material conditional $\alpha \rightarrow \beta$ (Nolt et al. 1998) in propositional logic, as shown in TABLE 20, is of no practical use in describing process phenomena. If $\alpha \rightarrow \beta$ is used to represent a *sequence* process structure in which activity β is executed (true) after α is executed (true), it is impossible that β is executed (true) but α is not executed (false). Considering these process phenomena, constraints have been added on applying propositional logic to verifying workflow models in Section 3.

TABLE 20. Truth Table of Material Conditional $\alpha \rightarrow \beta$ in Propositional Logic

α	β	$\alpha \rightarrow \beta$
1	1	1
1	0	0
0	1	1
0	0	1

In this section, we propose process logic by defining the syntax and semantics of process logic to reflect the characteristics of process structures in a more precise way. We differentiate the definitions of join structures from those of split structures. We also refine the notations of logical operators *and*, *or*, and *xor* to reflect that they are n -ary ($n \geq 2$) operators in process logic. Process logic proposed in this section is a natural step and a significant extension to our research presented in Section 3.

Process logic is the study of process phenomena by means of a logical formalism. Process logic shares many similarities with propositional logic (Hodel 1997; Nolt et al. 1998). However, process logic has unique features that are distinct from those of propositional logic.

5.1. Syntax of Process Logic

Process logic is concerned with the analysis of process arguments regarding process models.

DEFINITION 16 (process argument). A *process argument* is a sequence of process propositions of which one is intended as a *conclusion* and the others, the *premises*, are intended to prove the conclusion. ■

DEFINITION 17 (simple process proposition). A *simple process proposition* is a declarative sentence, which is either true or false and has no operators, regarding whether a vertex in a process model is executed or activated. ■

DEFINITION 18 (process proposition). A *process proposition* is a declarative sentence that either is a simple process proposition or is built up from simple process propositions using one or more operators *and*, *or*, *exclusive or (xor)*, and *sequence*. ■

For instance, the following argument is regarding part of a process model represented with a standard process graph P_{S8} in FIGURE 40:

- (1) After the start vertex s is executed, a control vertex c_1 is activated.
- (2) The start vertex s is executed.
- ∴ (3) The control vertex c_1 is activated.

In this process argument, (1) and (2) are premises, and (3) is the conclusion. (2) and (3) are simple process propositions, and (1) is a process proposition. The symbol ∴ means ‘therefore’.

FIGURE 40. A Standard Process Graph P_{S8}

DEFINITION 19 (language of process logic). The language that consists of symbolic notation to represent process models is called the *language of process logic*. ■

The language of process logic is described in two steps: symbols of the language and formulas of the language.

DEFINITION 20 (symbols). The *symbols* of the language of process logic are

(1) Process variables: *Process variables* are interpreted as simple process propositions and are denoted by lowercase letters with or without numerical subscripts: $a, b, c, \dots, a_1, a_2, a_3, \dots, b_1, b_2, b_3, \dots, c_1, c_2, c_3, \dots$, where, for example, a_1, a_2, a_3 , etc., are different from a .

(2) Logical operators: $\wedge, \vee, \oplus, \rightarrow$, whose names and interpretations are as follows:

Symbol	Name and Interpretation
$\wedge$	<i>and</i>
$\vee$	<i>or</i>
$\oplus$	<i>exclusive or (xor)</i>
$\rightarrow$	<i>sequence</i>

(3) Parentheses: $(,)$, which are used for punctuation.

These three sets of symbols constitute the *vocabulary* of the language of process logic.

■

Some literature uses uppercase letters to represent propositional variables. In this dissertation we reserve uppercase letters for other purposes.

In process logic, we interpret a process variable v as a simple process proposition that ‘A vertex v is executed (or activated)’. For example in FIGURE 40, we interpret s as ‘The start vertex s is executed’, and interpret c_1 as ‘A control vertex c_1 is activated’.

or implies ‘at least one’ and is the *inclusive* sense of ‘or’. *xor* implies ‘exact one’ and is the *exclusive* sense of ‘or’. In propositional logic, the inclusive sense of ‘or’ is standard, whereas in process phenomena, the exclusive sense of ‘or’ is quite common. Although in propositional logic, a proposition involving *xor* can be equivalently expressed by a proposition using *or*, *and*, and *not* (note that *not* is not included in process logic), this is not a case in process logic. Hence, in addition to *or*, *xor* is a necessary logical operator in process logic.

The vocabulary of the language of process logic is divided into logical and nonlogical symbols.

DEFINITION 21 (logical and nonlogical symbols). The symbols $\wedge$, $\vee$, $\oplus$, $\rightarrow$, $($, and $)$ are *logical symbols* that are always interpreted in the same way. Process variables are *nonlogical symbols* that have different interpretations in different contexts. ■

For instance, a process variable a_1 may stand for ‘Activity of submitting an application is executed’ in one process model and ‘Activity of making payment is executed’ in another.

DEFINITION 22 (formula). A formula of the language of process logic is any sequence of elements of the vocabulary of the language of process logic. ■

Formulas can be nonsense sequences such as $(\wedge)(\vee a_2))$ that are not meaningful in depicting process phenomena. Thus, we define well-formed formulas as follows.

DEFINITION 23 (well-formed formula). The *well-formed formulas* (or simply *wffs*) of the language of process logic are defined inductively by three *formation rules* that constitute the grammar of the language of process logic:

R1: Each process variable is a wff.

R2: If α , β , α_1 , α_2 , ..., and α_n are wffs, then so are $(\alpha_1, \alpha_2, \dots, \alpha_n)\wedge$, $\wedge(\alpha_1, \alpha_2, \dots, \alpha_n)$, $(\alpha_1, \alpha_2, \dots, \alpha_n)\vee$, $\vee(\alpha_1, \alpha_2, \dots, \alpha_n)$, $(\alpha_1, \alpha_2, \dots, \alpha_n)\oplus$, $\oplus(\alpha_1, \alpha_2, \dots, \alpha_n)$, $(\alpha \rightarrow \beta)$.

R3: Every wff is obtained by a finite number of applications of *R1* and *R2*. ■

Greek letters α , β , α_1 , α_2 , ..., and α_n do not belong to the vocabulary of the language of process logic and are used to denote arbitrary wffs. For example, $(a_2 \rightarrow a_3)$ and $((a_5, a_9)\oplus \rightarrow \oplus(a_{10}, a_{11}))$ are both instances of $(\alpha \rightarrow \beta)$.

DEFINITION 24 (atomic wff, compound wff, subwff, component). Process variables are called *atomic wffs*. Wffs that are not process variables are called *compound wffs*. A *subwff* is a part of a wff that is itself a wff. Each wff is also a subwff of itself. If a wff β is a subwff of a wff α and is not a subwff of any subwff (except β itself) of α , then β is called a *component* of α . ■

For instance, $(a_5, a_9)\oplus$ is a subwff of $(a_5, a_9)\oplus \rightarrow \oplus(a_{10}, a_{11})$.

The wffs of process logic either are process variables or are built up from simpler wffs using operators $\wedge$, $\vee$, $\oplus$, and $\rightarrow$. In order to represent join and split structures in process models, wffs in process logic incorporate the concepts of join and split. A wff $(\alpha_1, \alpha_2, \dots, \alpha_n)\wedge$ is called *AND-Join* with components α_1 , α_2 , ..., and α_n . A wff $\wedge(\alpha_1,$

$\alpha_2, \dots, \alpha_n$) is called *AND-Split* with components $\alpha_1, \alpha_2, \dots$, and α_n . *AND-Join* wffs and *AND-Split* wffs are called *and* wffs. A wff $(\alpha_1, \alpha_2, \dots, \alpha_n)\vee$ is called *OR-Join* with components $\alpha_1, \alpha_2, \dots$, and α_n . A wff $\vee(\alpha_1, \alpha_2, \dots, \alpha_n)$ is called *OR-Split* with components $\alpha_1, \alpha_2, \dots$, and α_n . *OR-Join* wffs and *OR-Split* wffs are called *or* wffs. A wff $(\alpha_1, \alpha_2, \dots, \alpha_n)\oplus$ is called *XOR-Join* with components $\alpha_1, \alpha_2, \dots$, and α_n . A wff $\oplus(\alpha_1, \alpha_2, \dots, \alpha_n)$ is called *XOR-Split* with components $\alpha_1, \alpha_2, \dots$, and α_n . *XOR-Join* wffs and *XOR-Split* wffs are called *xor* wffs. A wff $(\alpha \rightarrow \beta)$ is called *sequence* with *antecedent* α and *consequent* β . *AND-Join* wffs, *OR-Join* wffs, and *XOR-Join* wffs are also called *join wffs*. *AND-Split* wffs, *OR-Split* wffs, and *XOR-Split* wffs are also called *split wffs*.

The outer pair of parentheses in a wff is not actually necessary to make the meaning of the wff clear. Thus, as a convention in process logic, such outer pair of parentheses may be omitted. For instance, $(\alpha \rightarrow \beta)$ may be written as $\alpha \rightarrow \beta$. TABLE 21 summarizes compound wffs that are built up from simpler wffs using logical operators.

TABLE 21. Compound Wffs

Name	wff	Graphical Structure
<i>AND-Join</i>	$(\alpha_1, \alpha_2, \dots, \alpha_n) \wedge$	FIGURE 41(a)
<i>AND-Split</i>	$\wedge(\alpha_1, \alpha_2, \dots, \alpha_n)$	FIGURE 41(b)
<i>XOR-Join</i>	$(\alpha_1, \alpha_2, \dots, \alpha_n) \oplus$	FIGURE 41(c)
<i>XOR-Split</i>	$\oplus(\alpha_1, \alpha_2, \dots, \alpha_n)$	FIGURE 41(d)
<i>OR-Join</i>	$(\alpha_1, \alpha_2, \dots, \alpha_n) \vee$	FIGURE 41(e)
<i>OR-Split</i>	$\vee(\alpha_1, \alpha_2, \dots, \alpha_n)$	FIGURE 41(f)
<i>sequence</i>	$\alpha \rightarrow \beta$	FIGURE 41(g)

$\rightarrow$ is a binary operator, and $\wedge$, $\vee$, and $\oplus$ are n -ary ($n \geq 2$) operators. For operators $\wedge$, $\vee$, and $\oplus$, the order of components $\alpha_1, \alpha_2, \dots, \alpha_n$ is insignificant. This corresponds to the fact that activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ in each of FIGURE 41(a) through FIGURE 41(f) can be placed in any order.

FIGURE 41. Structures in Standard Process Graphs

DEFINITION 25 (scope of an occurrence of a logical operator). In a wff, an occurrence of a logical operator, together with the part of the wff to which that occurrence of the logical operator applies, is called the *scope* of that occurrence of the logical operator. ■

Essentially, the scope of an occurrence of a logical operator in a wff is the smallest subwff that contains that occurrence. For example, in a wff $(a_5, a_9) \oplus \rightarrow \oplus(a_{10}, a_{11})$, the scope of the first occurrence of $\oplus$ is $(a_5, a_9) \oplus$, the scope of the second occurrence of $\oplus$ is $\oplus(a_{10}, a_{11})$, and the scope of $\rightarrow$ is the entire wff.

DEFINITION 26 (main operator). Each wff has exact one logical operator whose scope is the entire wff. Such a logical operator is called the *main operator* of that wff. ■

As shown in TABLE 21, a main operator, along with the consideration for join or split, is reflected in the name of a wff.

DEFINITION 27 (process argument form). For a standard process graph P_S , the corresponding *process argument form* F of the language of process logic is written as

$$s, P_1, P_2, \dots, P_n \vdash e$$

where

- (1) s and e are the start and end vertices in P_S , and represent two process variables in process logic,
- (2) $P_1, P_2, \dots, P_n$ constitute a unique, finite set of *sequence* wffs,
- (3) $s, P_1, P_2, \dots, P_n$ constitute a finite set of premises, separated by commas, of F ,
- (4) e is the conclusion of F ,
- (5) premises are intended to prove the conclusion, and
- (6) the symbol ' $\vdash$ ' is called an assertion sign. ■

The statement that “ $P_1, P_2, \dots, P_n$ constitute a unique, finite set of *sequence* wffs” will be proved by .

For instance, the entire standard process graph P_{S8} in FIGURE 40 can be expressed as the following process argument:

- (1) The start vertex s is executed.
- (2) After the start vertex s is executed, a control vertex c_1 is activated.
- (3) After the control vertex c_1 is activated, both an activity vertex a_1 is executed and an activity vertex a_2 is executed.
- (4) After the activity vertex a_2 is executed, an activity vertex a_3 is executed.
- (5) After both the activity vertex a_1 is executed and the activity vertex a_3 is executed, a control vertex c_2 is activated.

(6) After the control vertex c_2 is activated, the end vertex e is executed.

$\therefore$ (7) The end vertex e is executed.

Because we interpret a process variable v as ‘A vertex v is executed (or activated)’, these seven process propositions can be converted into seven wffs as follows:

(1) s

(2) $s \rightarrow c_1$

(3) $c_1 \rightarrow \wedge(a_1, a_2)$

(4) $a_2 \rightarrow a_3$

(5) $(a_1, a_3) \wedge \rightarrow c_2$

(6) $c_2 \rightarrow e$

$\therefore$ (7) e

This process argument can be written in a process argument form:

$$s, s \rightarrow c_1, c_1 \rightarrow \wedge(a_1, a_2), a_2 \rightarrow a_3, (a_1, a_3) \wedge \rightarrow c_2, c_2 \rightarrow e \vdash e$$

5.2. Semantics of Process Logic

In this subsection, we define the semantics of process logic by means of truth values.

5.2.1. Truth Values of Simple Process Propositions

DEFINITION 28 (truth value of a simple process proposition). For a standard process graph P_S , if a vertex v is executed (or activated) in a process instance, then a simple process proposition that ‘ v is executed (or activated)’ is *true* for this process instance; otherwise, the simple process proposition that ‘ v is executed (or activated)’ is *false* for this process instance. The truth or falsity of a simple process proposition is called the

truth value of the simple process proposition. 1 and 0 are selected to be truth values, and 1 stands for ‘*true*’ and 0 stands for ‘*false*’. ■

In process logic, every simple process proposition is either true or false but not both. That is, every simple process proposition has a unique truth value. This feature is called the *principle of bivalence* (Nolt et al. 1998).

For example, the highlighted elements in a standard process graph P_{S9} in FIGURE 42 illustrate a process instance. For this process instance, a simple process proposition that ‘ s (c_1 , a_1 , c_3 , or e) is executed (or activated)’ is true, and a simple process proposition that ‘ a_2 (a_3 , c_2 , or a_4) is executed (or activated)’ is false.

FIGURE 42. A Process Instance in a Standard Process Graph P_{S9}

Because a process variable is interpreted as a simple process proposition, each process variable in a wff is either true or false but not both.

5.2.2. Semantics of Logical Operators

In this subsection, we give a rigorous formulation of the semantics, which is the intended interpretation, of logical operators.

The definition of the semantics of a logical operator depends on the type of a wff in which the operator is a main operator. Three cases exist: (1) The semantics of a logical operator as a main operator in a *join* wff is given by a set of rules for determining the

truth value of the wff involving that operator on the basis of the truth values of the components. (2) The semantics of a logical operator as a main operator in a *split* wff is given by a set of rules for determining the truth values of the components of the wff involving that operator on the basis of the truth value of the wff. (3) The semantics of the *sequence* operator in a *sequence* wff is given by a set of rules for determining the truth value of the consequent on the basis of the truth value of the antecedent.

For each of the following definitions, we first give an illustrative discussion on the basis that every component of each compound wff is an activity vertex. Then we generalize that every component of each compound wff can be an arbitrary wff.

For an *AND-Join* structure in FIGURE 41(a), a control vertex c is activated after all activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ are executed. c is not activated when at least one of $\alpha_1, \alpha_2, \dots, \alpha_n$ is not executed. Generally, an *AND-Join* wff is true when all of its components are true.

DEFINITION 29 (*AND-Join*). The truth value of an *AND-Join* wff is defined as:

$$(\alpha_1, \alpha_2, \dots, \alpha_n)^\wedge = \begin{cases} 1, & \text{if } \alpha_1 = \alpha_2 = \dots = \alpha_n = 1 \\ 0, & \text{otherwise} \end{cases} \quad (n \geq 2). \blacksquare$$

For an *AND-Split* structure in FIGURE 41(b), after a control vertex c is activated, all activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ are executed. If c is not activated, then none of $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. In any case, it is impossible that only some but not all of $\alpha_1, \alpha_2, \dots, \alpha_n$ are executed. Generally, if an *AND-Split* wff is true, then all of its components are true.

DEFINITION 30 (*AND-Split*). The truth values of the components of an *AND-Split*

wff are defined as: $\alpha_1 = \alpha_2 = \dots = \alpha_n = \begin{cases} 0, & \text{if } \wedge(\alpha_1, \alpha_2, \dots, \alpha_n) = 0 \\ 1, & \text{if } \wedge(\alpha_1, \alpha_2, \dots, \alpha_n) = 1 \end{cases} \quad (n \geq 2). \blacksquare$

For an *XOR-Join* structure in FIGURE 41(c), a control vertex c is activated after exact one of activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed; otherwise, c is not activated.

Generally, an *XOR-Join* wff is true when exact one of its components is true.

DEFINITION 31 (*XOR-Join*). The truth value of an *XOR-Join* wff is defined as:

$$(\alpha_1, \alpha_2, \dots, \alpha_n) \oplus = \begin{cases} 1, & \text{if } \alpha_i = 1 (1 \leq i \leq n) \text{ and } \alpha_j = 0 (j \in \{1, 2, \dots, n\} - i) \\ 0, & \text{otherwise} \end{cases} \quad (n \geq 2). \blacksquare$$

For an *XOR-Split* structure in FIGURE 41(d), after a control vertex c is activated, exact one of activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. If c is not activated, then none of $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. In any case, it is impossible that more than one of $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. Generally, if an *XOR-Split* wff is true, then exact one of its components is true.

DEFINITION 32 (*XOR-Split*). The truth values of the components of an *XOR-Split* wff are defined as:

If $\oplus(\alpha_1, \alpha_2, \dots, \alpha_n) = 0$, then $\alpha_1 = \alpha_2 = \dots = \alpha_n = 0 \quad (n \geq 2)$.

If $\oplus(\alpha_1, \alpha_2, \dots, \alpha_n) = 1$, then $\alpha_i = 1 \quad (1 \leq i \leq n)$ and $\alpha_j = 0 \quad (j \in \{1, 2, \dots, n\} - i) \quad (n \geq 2)$.

■

For an *OR-Join* structure in FIGURE 41(e), a control vertex c is activated after at least one of activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. c is not activated when none of

$\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. Generally, an *OR-Join* wff is true when at least one of its components is true.

DEFINITION 33 (*OR-Join*). The truth value of an *OR-Join* wff is defined as:

$$(\alpha_1, \alpha_2, \dots, \alpha_n) \vee = \begin{cases} 0, & \text{if } \alpha_1 = \alpha_2 = \dots = \alpha_n = 0 \\ 1, & \text{otherwise} \end{cases} \quad (n \geq 2). \blacksquare$$

For an *OR-Split* structure in FIGURE 41(f), after a control vertex c is activated, at least one of activity vertices $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. If c is not activated, then none of $\alpha_1, \alpha_2, \dots, \alpha_n$ is executed. Generally, if an *OR-Split* wff is true, then at least one of its components is true.

DEFINITION 34 (*OR-Split*). The truth values of the components of an *OR-Split* wff are defined as:

If $\vee(\alpha_1, \alpha_2, \dots, \alpha_n) = 0$, then $\alpha_1 = \alpha_2 = \dots = \alpha_n = 0$ ($n \geq 2$).

If $\vee(\alpha_1, \alpha_2, \dots, \alpha_n) = 1$, then $\exists \alpha_i$ ($1 \leq i \leq n$), $\alpha_i = 1$ ($n \geq 2$). $\blacksquare$

For a *sequence* structure in FIGURE 41(g), if α and β are both activity vertices, after α is executed, β will definitely be executed. If α is not executed, then β will not be executed. In any case, it is impossible that α is executed but β is not executed, or that α is not executed but β is executed. Generally, the antecedent and consequent of a *sequence* wff always have the same truth value.

DEFINITION 35 (*sequence*). The truth value of the consequent of a *sequence* wff α

$$\rightarrow \beta \text{ is defined as: } \beta = \begin{cases} 0, & \text{if } \alpha = 0 \\ 1, & \text{if } \alpha = 1 \end{cases}. \blacksquare$$

The semantics of logical operators can also be expressed using truth tables.

DEFINITION 36 (truth table). A *truth table* is a tabular description of all feasible truth value assignments involved in wffs and is used to display calculations of truth values. ■

TABLE 22 gives the truth table of *sequence*. As a distinctive feature in process logic, the truth value of a *sequence* $\alpha \rightarrow \beta$ itself is of no interest. α and β are either both true or both false, thus always giving a ‘true’ truth value to a *sequence* $\alpha \rightarrow \beta$ in the sense of propositional logic.

In a truth table, on the left-hand side of double vertical lines are all possible truth value assignments that are used for calculation, and on the right-hand side are the calculation results on the basis of the logical operator involved and the truth values on the left-hand side. Given bivalence, it is possible that a truth table completely describes the truth values in every feasible situation for a wff.

TABLE 23 through TABLE 25 show the truth tables of *and*, *xor*, and *or* wffs that contain three components. Truth tables of *AND-Join*, *XOR-Join*, and *OR-Join* that contains n ($n \geq 2$) components comprise 2^n rows, excluding the heading rows that are not an official part of a truth table.

TABLE 22. Truth Table of *Sequence* $\alpha \rightarrow \beta$

α	β
0	0
1	1

FIGURE 43. *and*, *xor*, and *or* Structures with Three Activity Vertices

TABLE 23. Truth Table of Join Wffs with Three Components

α_1	α_2	α_3	FIGURE 43(a)	FIGURE 43(c)	FIGURE 43(e)
			<i>AND-Join</i>	<i>XOR-Join</i>	<i>OR-Join</i>
			$(\alpha_1, \alpha_2, \alpha_3) \wedge$	$(\alpha_1, \alpha_2, \alpha_3) \oplus$	$(\alpha_1, \alpha_2, \alpha_3) \vee$
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	1	1
0	0	1	0	1	1
1	1	0	0	0	1
1	0	1	0	0	1
0	1	1	0	0	1
1	1	1	1	0	1

TABLE 24. Truth Table of *AND-Split* with Three Components

FIGURE 43(b)			
<i>AND-Split</i>	α_1	α_2	α_3
$\wedge(\alpha_1, \alpha_2, \alpha_3)$			
0	0	0	0
1	1	1	1

TABLE 25. Truth Table of *XOR-Split* with Three Components

FIGURE 43(d)			
<i>XOR-Split</i>	α_1	α_2	α_3
$\oplus(\alpha_1, \alpha_2, \alpha_3)$			
0	0	0	0
1	1	0	0
1	0	1	0
1	0	0	1

TABLE 26. Truth Table of *OR-Split* with Three Components

FIGURE 43(f)			
<i>OR-Split</i>	α_1	α_2	α_3
$\vee(\alpha_1, \alpha_2, \alpha_3)$			
0	0	0	0
1	1	0	0
1	0	1	0
1	0	0	1
1	1	1	0
1	1	0	1
1	0	1	1
1	1	1	1

Double vertical lines in a truth table separate the truth values used in calculation that are on the left-hand side of the table from the calculation results that are on the right-hand side of the table.

TABLE 27 and TABLE 28 show truth tables of *AND-Split* and *XOR-Split* wffs that comprise n ($n \geq 2$) components. All components of *AND-Split* are false when *AND-Split* is false, and true when *AND-Split* is true. Hence, there are only two rows in the truth table of an *AND-Split* wff that comprises n ($n \geq 2$) components. All components are false when *XOR-Split* is false, and exact one of components is true when *XOR-Split* is true. Thus, there are $n + 1$ rows in the truth table of an *XOR-Split* wff that comprises n ($n \geq 2$)

components. The truth table of an *OR-Split* wff that contains n ($n \geq 2$) components has 2^n rows.

TABLE 27. Truth Table of *AND-Split* ($n \geq 2$)

<i>AND-Split</i>	α_1	α_2	α_3	...	α_n
$\wedge(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n)$					
0	0	0	0	...	0
1	1	1	1	...	1

TABLE 28. Truth Table of *XOR-Split* ($n \geq 2$)

<i>XOR-Split</i>	α_1	α_2	α_3	...	α_n
$\oplus(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n)$					
0	0	0	0	...	0
1	1	0	0	...	0
1	0	1	0	...	0
1	0	0	1	...	0
...	...	...	...	...	...
1	0	0	0	...	1

5.2.3. Truth Tables for *Sequence* Wffs

We give an intuitive example of constructing truth tables for an arbitrary *sequence* wff. The process structure in FIGURE 44 can be converted into two wffs: (1) $a_1 \rightarrow c$, and (2) $c \rightarrow \wedge(a_2, a_3, a_4)$. Consider the second wff $c \rightarrow \wedge(a_2, a_3, a_4)$. TABLE 29 gives two possible truth values, 0 and 1, of c in $c \rightarrow \wedge(a_2, a_3, a_4)$. On the basis of truth values of c , the truth values of $\wedge(a_2, a_3, a_4)$ are calculated in TABLE 30 according to DEFINITION 35. According to DEFINITION 30, the truth values of $\wedge(a_2, a_3, a_4)$, in turn, determines the truth values of a_2, a_3 , and a_4 in TABLE 31, which is the final result.

FIGURE 44. A Process Structure

TABLE 29. Truth Values of c in $c \rightarrow \wedge(a_2, a_3, a_4)$

c
0
1

TABLE 30. Truth Values of $\wedge(a_2, a_3, a_4)$ in $c \rightarrow \wedge(a_2, a_3, a_4)$

c	$\wedge(a_2, a_3, a_4)$
0	0
1	1

TABLE 31. Truth Values of $a_2, a_3,$ and a_4 in $c \rightarrow \wedge(a_2, a_3, a_4)$

c	$\wedge(a_2, a_3, a_4)$	a_2	a_3	a_4
0	0	0	0	0
1	1	1	1	1

5.2.4. Truth Tables for Process Argument Forms

Truth tables can also provide a rigorous evaluation of deductive validity of process argument forms.

DEFINITION 37 (valid process argument form). A process argument form F is *valid* if and only if its conclusion e is true when its premise s is true, no other components involved in the same consequent as e in a *sequence* wff are true when e is true, and all premises and all process variables are used in proving the conclusion e . ■

Truth tables are actually exhaustive lists of all possible situations. Thus, we can use truth tables to verify whether a process argument form is valid.

We illustrate with examples the procedure of verifying the validity of a process argument form. A standard process graph P_{S8} in FIGURE 40 can be converted into a unique process argument form $F: s, s \rightarrow c_1, c_1 \rightarrow \wedge(a_1, a_2), a_2 \rightarrow a_3, (a_1, a_3) \wedge \rightarrow c_2, c_2 \rightarrow$

$e \vdash e$. F can be rewritten to show the number of each premise: (1) s , (2) $s \rightarrow c_1$, (3) $c_1 \rightarrow \wedge(a_1, a_2)$, (4) $a_2 \rightarrow a_3$, (5) $(a_1, a_3) \wedge \rightarrow c_2$, (6) $c_2 \rightarrow e \vdash e$.

TABLE 32 displays the verification of the validity of F , with the top heading row shows the number of each premise involved in the truth value calculation. In a standard process graph, the start vertex s is executed unconditionally when processes start. Hence, at the beginning of calculation, only s is set 1. The truth values of all other wffs must be calculated. The truth value of c_1 in premise (2) is determined on the basis of s , and then the truth value of $\wedge(a_1, a_2)$ in premise (3) is determined on the basis of c_1 , and so on. Finally, we obtain the truth value of the conclusion e , which is 1. Therefore, it can be concluded that F is valid, because e is 1 when s is 1, and all premises and all process variables are used in proving the conclusion e .

TABLE 32. Verification of the Validity of the Process Argument Form F of P_{S8} in

FIGURE 40

(1)	(2)	(3)	(3)		(4)	(5)	(5)	(6)
s	c_1	$\wedge(a_1, a_2)$	a_1	a_2	a_3	$(a_1, a_3) \wedge$	c_2	e
1	1	1	1	1	1	1	1	1

For another example, a standard process graph P_{S10} in FIGURE 45 can be converted into a unique process argument form F : $s, s \rightarrow a_1, (a_1, a_2) \oplus \rightarrow c_1, c_1 \rightarrow \oplus(a_2, e) \vdash e$, which can be rewritten as: (1) s , (2) $s \rightarrow a_1$, (3) $(a_1, a_2) \oplus \rightarrow c_1$, (4) $c_1 \rightarrow \oplus(a_2, e) \vdash e$.

FIGURE 45. A Standard Process Graph P_{S10} (Same as FIGURE 23)

FIGURE 45 contains a cycle that must be taken care of during the verification of the validity of F . In FIGURE 45, a_2 cannot be executed before c_1 is activated. Hence a_2 is set 0 in TABLE 33. TABLE 34 has to be created because there are two rows, whereas there is only one row in TABLE 33. The truth value of $\oplus(a_2, e)$ in TABLE 34 is copied from that in TABLE 33. In TABLE 34, according to DEFINITION 32 (*XOR-Split*), either a_2 or e is 1 when $\oplus(a_2, e)$ is 1.

TABLE 33. Cycle Is Not Yet Involved: the First Truth Table

(1)	(2)	(3)	(3)	(3)	(4)
s	a_1	a_2	$\oplus(a_1, a_2)$	c_1	$\oplus(a_2, e)$
1	1	0	1	1	1

TABLE 34. Cycle Is Not Yet Involved: the Second Truth Table

(4)	(4)
$\oplus(a_2, e)$	a_2 e
1	1 0
1	0 1

TABLE 35 shows the calculation after a_2 is executed and thus the cycle is involved. Note that a_1 is no longer executed and is set 0. TABLE 36 gives the same truth table as TABLE 34 after a_2 is executed. TABLE 36 tells that the execution of a_2 can eventually cause e to be executed, unless the cycle is a semantically infinite cycle, which is out of the scope of this dissertation. Therefore, it can be concluded that F is valid, because e is 1 when s is 1, no other component (a_2) involved in the same consequent as e is 1 when e is 1, and all premises and all process variables are used in proving the conclusion e .

TABLE 35. Cycle Is Involved: the First Truth Table

(3)	(3)	(3)	(3)	(4)
a_1	a_2	$\oplus(a_1, a_2)$	c_1	$\oplus(a_2, e)$
0	1	1	1	1

TABLE 36. Cycle Is Involved: the Second Truth Table

(4)	(4)
$\oplus(a_2, e)$	a_2 e
1	1 0
1	0 1

5.3. Relationship between Process Graphs and Process Logic

In this subsection, we formalize the relationship between process graphs proposed in Section 4 and process logic.

In a standard process graph, a *sequence* construct is converted into a wff (such as $a_2 \rightarrow a_3$), and a construct involving a control vertex is converted into two wffs (such as $s \rightarrow c_1$ and $c_1 \rightarrow \wedge(a_1, a_2)$). Thus, we can obtain the following theorem that establishes a formal relationship between standard process graphs and process logic.

THEOREM 9. For a standard process graph P_S , each *sequence* process construct can be converted into a unique *sequence* wff and each process construct involving a control vertex can be converted into two unique *sequence* wffs such that there is a unique, finite set of *sequence* wffs $\Phi = \{\varphi_1, \varphi_2, \dots, \varphi_k\}$ corresponding to P_S . ▲

Proof. A standard process graph P_S can be decomposed into a unique, finite set of process constructs. According to the conversion rules in TABLE 37, each *sequence* process construct can be converted into a unique *sequence* wff, and each process construct involving a control vertex can be converted into two unique *sequence* wffs. As a result, P_S can be converted into a unique, finite set of *sequence* wffs. □

TABLE 37. Rules of Converting Process Constructs into Wffs

Process Construct	Graphical Representation	Wffs
<i>Sequence</i>	FIGURE 26(a)	$v \rightarrow w$
<i>AND</i>	FIGURE 26(b)	$(v_1, v_2, \dots, v_m) \wedge \rightarrow c,$ $c \rightarrow \wedge(w_1, w_2, \dots, w_n)$
<i>XOR</i>	FIGURE 26(c)	$(v_1, v_2, \dots, v_m) \oplus \rightarrow c,$ $c \rightarrow \oplus(w_1, w_2, \dots, w_n)$
<i>OR</i>	FIGURE 26(d)	$(v_1, v_2, \dots, v_m) \vee \rightarrow c,$ $c \rightarrow \vee(w_1, w_2, \dots, w_n)$

COROLLARY 10. Each standard process graph P_S has a unique corresponding *process argument form* F . ▲

Proof. COROLLARY 10 can be directly derived from DEFINITION 27 (process argument form) and THEOREM 9. □

After a process construct is converted into a *sequence* wff(s), the antecedent of a sequence wff is always a process variable or a join wff, and the consequent of a sequence wff is always a process variable or a split wff.

6. APPLICATIONS OF PROCESS GRAPHS AND PROCESS LOGIC

In this section, we discuss two applications of process graphs and process logic: applying process graphs to defining and classifying anomalies in process models, and applying process logic to analyzing structures of process models.

6.1. Applying Process Graphs to Defining and Classifying Process Anomalies

As stated in THEOREM 5, process graphs can represent all types of structural orderings of activities. Hence, process graphs can be used to completely and formally define and classify structural process anomalies that may be found in process models.

DEFINITION 15 (process anomaly). A *process anomaly* is an error or conflict in a graph P that makes P violate the definition of process graphs or makes P an incorrect process graph. ■

We classify structural process anomalies into two categories.

6.1.1. Anomalies Violating the Definition of Process Graphs

Each of the following anomalies does not satisfy the definition of process graph P :

Anomaly 1. A vertex is not incident to any arc, i.e., $\exists v \in V(P), d(v) = 0$. This vertex is not on any “walk through” path from the start vertex s to the end vertex e and thus will not be executed or activated for any process instance. ■

Anomaly 2. An activity or control vertex has no entering arc or no leaving arc, i.e., $\exists v \in V(P) - \{s, e\}, d^-(v) = 0$ or $d^+(v) = 0$. This vertex is on an incomplete path because it is not on a “walk through” path from s to e . ■

Anomaly 3. An activity vertex has more than one entering arc or more than one leaving arc, i.e., $\exists v \in V_A(P)$, $d^-(v) > 1$ or $d^+(v) > 1$. There is a missing control vertex that should be adjacent to or from this activity vertex. ■

Anomaly 4. A control vertex has only one entering arc and only one leaving arc, i.e., $\exists v \in V_C(P)$, $d^-(v) = 1$ and $d^+(v) = 1$. This is either an unnecessary control vertex or a modeling error. ■

Anomaly 5. There is no start vertex or no end vertex, or there is more than one start vertex or more than one end vertex, i.e., $|\{s\}| = 0$ or $|\{s\}| \geq 2$ or $|\{e\}| = 0$ or $|\{e\}| \geq 2$. ■

Anomaly 6. The start vertex has entering arcs or has more than one leaving arc, i.e., $d^-(s) \geq 1$ or $d^+(s) \geq 2$. ■

Anomaly 7. The end vertex has leaving arcs or has more than one entering arc, i.e., $d^+(e) \geq 1$ or $d^-(e) \geq 2$. ■

Anomaly 8. A loop exists on a vertex, i.e., $\exists v \in V(P)$, $vv \in A(P)$. This is not allowed in typical activity-based process models. ■

Anomaly 9. Multiple arcs exist between two vertices, i.e., $\exists vw, xy \in A(P)$, $\text{tail}(vw) = \text{tail}(xy)$ and $\text{head}(vw) = \text{head}(xy)$. Again, this is not allowed in activity-based process models. ■

For example in FIGURE 46, a_9 is not incident to any arc (Anomaly 1); a_8 has no entering arc (Anomaly 2); a_1 , a_3 , and a_5 have more than one entering arc or more than one leaving arc (Anomaly 3); c_7 has only one entering arc and only one leaving arc (Anomaly 4); there are two start vertices (Anomaly 5); one of the start vertices has two leaving arcs

(Anomaly 6); the end vertex e has a leaving arc (Anomaly 7); a_3a_3 is a loop (Anomaly 8); and two arcs exist between a_5 and c_5 (Anomaly 9).

FIGURE 46. A Process Model Containing Multiple Anomalies

6.1.2. Anomalies Related to Improper Uses of Control Vertices

Such anomalies include deadlock, lack of synchronization, structurally infinite cycles, and unnecessary non-determinism caused by the improper uses of control vertices of six kinds: *AND-Split*, *AND-Join*, *XOR-Split*, *XOR-Join*, *OR-Split*, and *OR-Join*. Although some of these process anomalies have been addressed in Section 3.2, in this subsection we give them more formal definitions.

Anomaly 10 (deadlock). A *deadlock* refers to a situation in which a process instance gets into a stalemate such that no activity can be further executed (Verbeek et al. 2001). ■

There are a variety of situations in which a deadlock occurs. First, a deadlock occurs or may occur when an *AND-Join* vertex is mismatched to an *XOR-Split* vertex or an *OR-Split* vertex. FIGURE 47(a) and FIGURE 47(b) show these two types of deadlocks caused by mismatched control vertices. The deadlock in FIGURE 47(a) is a deterministic one in which, because only one of a_1 and a_2 is executed after c_1 , c_2 will wait forever and thus block the continuation of the process. The deadlock in FIGURE 47(b) is a non-

deterministic one in which, when both a_1 and a_2 are executed after c_1 , there is no problem. But if only one of a_1 and a_2 is executed, c_2 will block any further execution.

FIGURE 47. Deadlocks

There are other situations where a deadlock may arise. For example in FIGURE 47(c), suppose for a process instance, after c_1 is activated, c_2 is activated and then a_1 and a_2 are executed. But c_4 and c_5 will never be activated because a_3 and a_4 are not executed. This results in a deadlock.

Anomaly 11 (lack of synchronization). *Lack of synchronization* refers to a situation in which the concurrent activities are joined by an *XOR-Join* vertex, resulting in unintentional multiple executions of activities that follow the *XOR-Join* vertex (Sadiq and Orłowska 2000). ■

The reason for the occurrence of lack of synchronization is that for a process instance, more than one entering arc of an *XOR-Join* vertex becomes active, thus activating the *XOR-Join* vertex and triggering the subsequent activities multiple times. FIGURE 48(a) shows deterministic lack of synchronization where the executions of a_1 and a_2 cause e to

be executed twice. FIGURE 48(b) illustrates non-deterministic lack of synchronization; that is, if only one of a_1 and a_2 is executed, then e is executed once and there is no problem; but if both a_1 and a_2 are executed, then e is triggered twice.

FIGURE 48. Lack of Synchronization (Same as FIGURE 8)

Anomaly 12 (structurally infinite cycle). A *structurally infinite cycle* originates from structural errors in which an exit vertex of a cycle is an *AND-Split* or *OR-Split* vertex instead of an *XOR-Split* vertex. ■

A structurally infinite cycle may cause some activities in a process model to be repeatedly executed forever. For instance, FIGURE 49(a) illustrates a deterministic infinite cycle in which after a_1 is executed, both e and a_2 are executed. The execution of a_2 triggers a_1 again, thus causing an infinite cycle. Similarly, FIGURE 49(b) shows a non-deterministic infinite cycle; that is, after c_2 is activated, the cycle terminates when only e is triggered, but the cycle goes forever when both a_2 and e are executed repeatedly.

FIGURE 49. Structurally Infinite Cycles (Same as FIGURE 10)

Anomaly 13 (unnecessary non-determinism). *Unnecessary non-determinism* occurs if an *OR-Join* vertex is used where an *XOR-Join* vertex or an *AND-Join* vertex is sufficient.

■

An *OR-Join* vertex waits for one or more entering arcs to become active in order to be activated. How many and which entering arcs become active before the activation of the *OR-Join* vertex are non-deterministic and dependent on runtime conditions. Hence, unnecessary non-determinism is introduced if an *OR-Join* vertex is used where an *XOR-Join* vertex (FIGURE 50(a)) or an *AND-Join* vertex (FIGURE 50(b)) is sufficient. ■

FIGURE 50. Unnecessary Non-determinism

6.2. Applying Process Logic to Analyzing Structures of Process Models

6.2.1. Applying Process Logic to Detecting Process Anomalies

As an alternative way to the application of propositional logic with constraints to detecting process anomalies in Section 3.2, process logic can also be used to detect process anomalies such as deadlock and lack of synchronization in process graphs.

6.2.1.1. Deadlock

As shown in FIGURE 51, when an *AND-Join* vertex c_2 is mismatched to an *XOR-Split* vertex c_1 , a deadlock occurs at c_2 . The process argument form F corresponding to

the standard process graph P_S in FIGURE 51 is $s, s \rightarrow c_1, c_1 \rightarrow \oplus(a_1, a_2), (a_1, a_2)^\wedge \rightarrow c_2, c_2 \rightarrow e \vdash e$, for which truth tables are constructed in TABLE 38 and TABLE 39.

FIGURE 51. An Example of Deadlock

TABLE 38. The First Truth Table for Finding the Deadlock in FIGURE 51

s	c_1	$\oplus(a_1, a_2)$
1	1	1

TABLE 39. The Second Truth Table for Finding the Deadlock in FIGURE 51

$\oplus(a_1, a_2)$	a_1	a_2	$(a_1, a_2)^\wedge$	c_2	e
1	1	0	0	0	0
1	0	1	0	0	0

Because e is 0 when s is 1, F is invalid and thus P_S is incorrect. The problem is that $(a_1, a_2)^\wedge$ is always 0 when s is 1; in other words, the *AND-Join* vertex c_2 can never be activated, thus blocking the process continuation.

6.2.1.2. Lack of Synchronization

In FIGURE 52, when an *XOR-Join* vertex c_2 is mismatched to an *AND-Split* vertex c_1 , lack of synchronization occurs at c_2 . The process argument form F corresponding to the

standard process graph P_S in FIGURE 52 is $s, s \rightarrow c_1, c_1 \rightarrow \wedge(a_1, a_2), (a_1, a_2) \oplus \rightarrow c_2, c_2 \rightarrow e \vdash e$, for which truth values are calculated in TABLE 40.

FIGURE 52. An Example of Lack of Synchronization

TABLE 40. Truth Table for Finding Lack of Synchronization in FIGURE 52

s	c_1	$\wedge(a_1, a_2)$	a_1	a_2	$(a_1, a_2) \oplus$	c_2	e
1	1	1	1	1	0	0	0

Because e is 0 when s is 1, F is invalid and thus P_S is incorrect. Because $(a_1, a_2) \oplus$ is 0 when s is 1, the *XOR-Join* vertex c_2 can never be activated because of lack of synchronization.

6.2.2. Applying Process Logic to Verifying the Correctness of Process Models

We can use process logic to verify the correctness of process graphs.

THEOREM 11. If the corresponding process argument form F of a standard process graph P_s is valid, then P_s is correct. ▲

Proof. According to COROLLARY 10, each standard process graph P_S has a unique corresponding *process argument form* F . According to DEFINITION 37, if F is valid, then (1) its conclusion e is true when its premise s is true, which means that every instance walk of P_S starting from s will eventually end at e ; (2) no other components involved in the same consequent as e are true when e is true, which means that no other

vertices can be further executed or activated after e is executed, and thus e will not be executed more than once; and (3) all premises and all process variables are used in proving the conclusion e , which means that there is no redundant activity vertex in P_S that will never be executed. Therefore, according to DEFINITION 15, if F is valid, then P_S is correct. $\square$

Because truth tables exhaustively list all possible situations for a process argument form, we can use truth tables to determine whether a process argument form is valid and thus the underlying process model is correct. Truth tables constitute a systematic and exhaustive approach to process verification.

An arbitrary process model can be represented by a process graph (THEOREM 5), and a process graph and its corresponding standard process graph have the same correctness (THEOREM 7). Hence, process logic can be used to verify the correctness of any process models.

We use the standard process graph P_{S11} in FIGURE 53 as an example to demonstrate that process logic can verify the correctness of process models. The corresponding process argument form F of P_{S11} is: (1) s , (2) $s \rightarrow c_1$, (3) $c_1 \rightarrow \oplus(a_1, a_2)$, (4) $a_1 \rightarrow c_2$, (5) $c_2 \rightarrow \wedge(a_3, a_4)$, (6) $a_2 \rightarrow c_3$, (7) $c_3 \rightarrow \wedge(a_5, a_6)$, (8) $(a_3, a_5) \oplus \rightarrow c_4$, (9) $(a_4, a_6) \oplus \rightarrow c_5$, (10) $c_4 \rightarrow a_7$, (11) $c_5 \rightarrow a_8$, (12) $(a_7, a_8) \wedge \rightarrow c_6$, (13) $c_6 \rightarrow e \vdash e$.

FIGURE 53. A Standard Process Graph P_{S11} (Same as FIGURE 12)

TABLE 41 and TABLE 42 are two truth tables for F . Because e is 1 when s is 1, F is valid and thus P_{S11} is correct.

TABLE 41. The First Truth Table for FIGURE 53

(1)	(2)	(3)
s	c_1	$\oplus(a_1, a_2)$
1	1	1

7. CONCLUSION AND FUTURE RESEARCH

Process management is a very important aspect of business management. However, although existing activity-based process modeling paradigms are widely used in the area of process management, they lack a mathematical formalism and rigorous analysis mechanisms. This deficiency indicates an imperative need for the development of effective process modeling and analysis methodologies and tools. Process validation, process verification, and data usage analysis are three aspects of process structural analysis. This dissertation concentrates on the modeling and verification of the control flow perspective of process models.

We applied propositional logic with constraints to verifying the correctness of both acyclic and cyclic activity-based workflow models. Then we perceived the needs to develop graphical and logical formalisms to more precisely represent process models and verify their correctness.

We proposed process graphs for formal business process modeling and verification. The main value of process graphs is their capabilities of analyzing activity-based process models mathematically, thus enabling machine-based process design and analysis. The correctness of process-graph-based process models can be defined unambiguously. We demonstrated that process graphs satisfy the five requirements for a process modeling language: (1) The well-defined semantics of process graphs supports both precise representations of processes and formal analysis of the correctness of process models. (2) Process graphs are expressive enough to represent succinctly all possible types of

execution order of activities using explicit control vertices. (3) Process graphs are capable of representing large and complex processes through multi-level abstraction using subprocesses. (4) Four process constructs are standardized units of process graphs to support flexible process design and manipulation. (5) Explicit control vertices and other graphical symbols in process graphs are easy to comprehend and use. In this sense, we say that process graphs provide a theoretical foundation for activity-based process design and analysis.

We also proposed process logic as a logical formalism and mathematical tool to verify the correctness of process models. Although process logic borrows a great amount from propositional logic, process logic is defined to specifically describe process phenomena such as join and split structures in process models. We demonstrated that process logic is capable of verifying completely the correctness of an arbitrary activity-based process model by converting a standard process graph into a process argument form, and then determining the validity of the process argument form through truth tables.

In addition to applying process graphs and process logic to verifying the correctness of process models, we used process graphs to define and classify process anomalies. The systematic study in process anomalies can have a significant impact on developing principles for designing correct process models and for verifying process models.

We outline our future research as follows:

- (1) Our next step in fully developing the process logical formalism is to define logical inference rules that can be used to verify process models via logical deduction. The well defined syntax and semantics of process logic provide a theoretical foundation

for further research on the development of more efficient process verification approaches.

- (2) Another step in our future research is to develop a process verification system, based on process graphs and process logic, which integrates adjacency-matrix-based analysis and logic-based analysis.
- (3) On the basis of the study in process anomalies, we will develop principles for designing correct process models. We will also evaluate the verification capabilities and efficiency of existing process verification approaches against the classification of process anomalies. Such evaluation will provide guidelines for improving existing verification approaches and developing new verification approaches.
- (4) In the future, we will combine the modeling and analysis of data and information flow with that of control flow, and developing more sophisticated process modeling and analysis methodologies.

REFERENCES

- Aagaard, M., Leaser, M. 1995. Verifying a logic-synthesis algorithm and implementation: a case study in software verification. *IEEE Transactions on Software Engineering*. **21**(10) 822-833.
- Adam, N.R., Atluri, V., Huang, W.K. 1998. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*. **10**(2) 131-158.
- Al-Mashari, M., Al-Mudimigh, A., Zairi, M. 2003. Enterprise resource planning: A taxonomy of critical factors. *European Journal of Operational Res.* **146**(2) 352-364.
- Bang-Jensen, J., Gutin, G. 2001. *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, London, UK.
- Bastos, R.M., Ruiz, D.D.A. 2001. Extending UML activity diagram for workflow modeling in production systems. *Proc. of the 35th Annual Hawaii International Conference on System Sciences*. 3818-3827.
- Basu, A., Blanning, R.W. 1999. Metagraphs in workflow support systems. *Decision Support Systems*. **25**(3) 199-208.
- . 2000. A formal approach to workflow analysis. *Info. Systems Res.* **11**(1) 17-36.
- . 2001. Workflow analysis using attributed metagraphs. *Proc. of the 34th Annual Hawaii International Conference on System Sciences*. 3735-3743.
- . 2003. Synthesis and decomposition of processes in organizations. *Info. Systems Res.* **14**(4) 337-355.
- Basu, A., Kumar, A. 2002. Research commentary: Workflow management issues in e-Business. *Info. Systems Res.* **13**(1) 1-14.

- Bi, H.H., Zhao, J.L. Process graphs: a formal language for business process modeling and analysis. *Working Paper*.
- . 2003a. A formal classification of process anomalies for workflow verification. *Proc. of the 13th Workshop on Info. Technology and Systems*, Seattle, WA, Dec. 13-14, 2003. 207-212.
- . 2003b. Mending the lag between commercial needs and research prototypes: A logic-based workflow verification approach. *Proc. of the 8th INFORMS Computing Society Conference*, Chandler, AZ, Jan. 8-10, 2003. 191-212.
- . 2004. Applying propositional logic to workflow verification. *Info. Technology and Management: Special Issue on Workflow and E-business (Forthcoming)*.
- Bi, H.H., Zhao, J.L., Yiparaki, O. Process logic for verifying the correctness of business process models. *Working Paper*.
- Bonner, A.J., Kifer, M. 1994. An overview of transaction logic. *Theoretical Computer Science*. **133**(2) 205-265.
- Carchiolo, V., Faro, A. 1990. Knowledge-based approach for parallel software verification. *Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, Tel-Aviv, Israel, May 8-10, 1990. 322-328.
- Chartrand, G., Lesniak, L. 1996. *Graphs & Digraphs, 3rd ed.* Chapman & Hall, London, UK.
- Cheung, T.Y., Shi, L., Ye, X. 1999. A Windows-Explorer-like model for workflow specification. *Proc. of the 5th Asia-Pacific Conference on Communication*, Beijing, China, Oct. 1999. 1091-1094.
- Choi, Y., Zhao, J.L. 2002. Matrix-Based Abstraction & Verification for e-Business Processes. *Proc. of 1st Workshop on e-Bus.*, Barcelona, Spain, Dec.14-15, 2002. 154-165.

- . 2003. Handling Cycles in Workflow Verification by Feedback Identification and Partition. *Proceedings of the 2003 International Conference on Information and Knowledge Engineering*, Las Vegas, Nevada, June 23-26, 2003.
- Couger, J.D., Colter, M.A., Knapp, R.W. 1982. *Advanced System Development/Feasibility Techniques*. John Wiley & Sons.
- Curtis, B., Kellner, M.I., Over, J. 1992. Process modeling. *Communications of the ACM*. **35**(9) 75-90.
- Davenport, T.H. 1993. *Process Innovation: Reengineering Work through Information Technology*. Harvard Business School Press, Boston, MA.
- Davenport, T.H., Stoddard, D.B. 1994. Reengineering: Business change of mythic proportions? *MIS Quarterly*. **18**(2) 121-127.
- Downs, D., Lunn, K. 2002. Analysis and design for process support systems using goal-oriented business process modeling. *Workshop on Goal-Oriented Business Process Modeling*, London, UK, Sept. 2, 2002.
- Ellis, C.A., Nutt, G.J. 1980. Office information systems and computer science. *ACM Computing Surveys*. **12**(1) 27-60.
- Foster, H., Uchitel, S., Magee, J., Kramer, J. 2003. Model-based verification of Web service compositions. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Oct. 6-10, 2003. 152-161.
- Fuggetta, A. 1993. A classification of CASE technology. *Computer*. **26**(12) 25-38.
- Georgakopoulos, D., Hornick, M., Sheth, A. 1995. An overview of workflow management. *Distributed and Parallel Databases*. **3**(2) 119-153.
- Green, T.R.G. 1982. Pictures of programs and other processes, or how to do things with lines. *Behavior and Info. Technology*. **1**(1) 3-36.

- Grosu, R., Klein, C., Rumpe, B., Broy, M. 1996. *State transition diagrams. Technical Report TUM-19630*. Technische Universität München, Garching, Germany. 27 pages.
- Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*. 8(3) 231-274.
- . 1988. On visual formalisms. *Communications of the ACM*. 31(5) 514-530.
- Ho, T.I.M. 1974. *Toward a Formal Theory for the Requirements Statement, Analysis, and Design of Information Systems*. Doctoral dissertation. Purdue University.
- Hodel, R.E. 1997. *An Introduction to Mathematical Logic*. PWS Publishing Co., Boston, MA.
- Huckvale, T., Ould, M. 1995. Process modelling – who, what and how: Role activity diagramming. in: *Business Process Change: Reengineering Concepts, Methods, and Technologies*, V. Grover and W.J. Kettinger (eds.), Idea Group Publishing, Harrisburg, PA. 330-349.
- IEEE. 1986. *IEEE Std 1012-1986, IEEE standard for software verification and validation plans*. Software Engineering Technical Committee of the IEEE Computer Society, USA.
- . 1990. *IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology*. Standards Coordinating Committee of the IEEE Computer Society, USA.
- . 1998. *IEEE Std 1012-1998, IEEE standard for software verification and validation*. Software Engineering Standards Committee of the IEEE Computer Society, USA. viii+71.
- Jablonski, S., Bussler, C. 1996. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK.
- Kettinger, W.J., Teng, J.T.C., Guha, S. 1997. Business process change: A study of methodologies, techniques, and tools. *MIS Quarterly*. 21(1) 55-80.

- Kifer, M. 1996. Transaction logic for the busy workflow professional. *Unpublished manuscript*.
- Klein, C., Prehofer, C., Rumpe, B. 1997. Feature specification and refinement with state transition diagrams. *4th IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, IOS-Press. 284-297.
- Kueng, P., Kawalek, P., Bichler, P. 1996. How to compose an object-oriented business process model? *Proc. of the IFIP WG8.1/WG8.2 Working Conference on Method Engineering*, Chapman & Hall, Atlanta, GA., Aug. 26-28, 1996.
- Kumar, A., Zhao, J.L. 1999. Dynamic routing and operational controls in workflow management systems. *Management Science*. **45**(2) 253-272.
- Lai, R. 1995. Formal specification and verification of a procedural protocol: case study. *Software Engineering Journal*. **10**(3) 97-104.
- Larson, P.D., Rogers, D.S. 1998. Supply chain management: Definition, growth and approaches. *Journal of Marketing Theory and Practice*. **6**(4) 1-5.
- Lawler, E.L. 1978. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*. **2** 75-90.
- Lin, F.R., Yang, M.C., Pai, Y.H. 2002. A generic structure for business process modeling. *Business Process Management Journal*. **8**(1) 19-41.
- Luo, W.H., Tung, Y.A. 1999. A framework for selecting business process modeling methods. *Industrial Management & Data Systems*. **99**(7) 312-319.
- Mackenzie, K.D. 2000. Processes and their frameworks. *Management Science*. **46**(1) 110-125.
- Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborn, C.S., Bernstein, A., Herman, G., Klein, M., O'Donnell, E. 1999. Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science*. **45**(3) 425-443.

- Maruta, T., Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N. 1998. A deadlock detection algorithm for business processes workflow models. *IEEE International Conference on Systems, Man, and Cybernetics*, October 11-14, 1998. 611-616.
- Mayer, R.J., Menzel, C.P., Painter, M.K., deWitte, P.S., Blinn, T., Perakath, B. 1995. *Information Integration for Concurrent Engineering: IDEF3 Process Description Capture Method Report*. Knowledge Based Systems, Incorporated, College Station, TX. 236 pages.
- Meade, L.M., Rogers, K.J. 2001. Selecting critical business processes: A case study. *Engineering Management Journal*. 13(4) 41-46.
- Medina-Mora, R., Winograd, T., Flores, R., Flores, F. 1993. The ActionWorkflow approach to workflow management technology. *The Info. Society*. 9(4) 391-404.
- Mejza, M.C., Wisner, J.D. 2001. The scope and span of supply chain management. *International Journal of Logistics Management*. 12(2) 37-55.
- Monma, C.L., Sidney, J.B. 1979. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Res.* 4(3) 215-224.
- Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*. 77(4) 541-580.
- Nilsson, N.J. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto, CA. pp. 40-41.
- Nolt, J., Rohatyn, D., Varzi, A. 1998. *Schaum's Outline of Logic, 2nd Edition*. McGraw-Hill Companies, Inc.
- OMG. 2003. *Unified Modeling Language, Version 1.5, formal/03-03-01*. Object Management Group, Needham, MA. 736 pages.
- Pall, G.A. 2000. *The Process-Centered Enterprise: The Power of Commitments*. St. Lucie Press, Boca Raton, Florida. p. 201.

- Peterson, J.L. 1981. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ.
- Rumbaugh, J., Jacobson, I., Booch, G. 1998. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Harlow, England. pp. 81-84.
- Sadiq, W., Orlowska, M.E. 1997. On Correctness Issues in Conceptual Modeling of Workflows. *Proc. of 5th Euro. Conf. on Info. Sys.*, Cork, Ireland, June 19-21, 1997. 943-964.
- . 2000. Analyzing process models using graph reduction techniques. *Info. Systems*. **25**(2) 117-134.
- Sarkar, D.C., Sarkar, S.C.D. 1989. Some inference rules for integer arithmetic for verification of flowchart programs on integers. *IEEE Transactions on Software Engineering*. **15**(1) 1-9.
- Schneider, H.-A., Taubner, D. 1992. Process algebra techniques for verification of SDL-diagrams. *Proceedings of the Eighth International Conference on Software Engineering for Telecommunication Systems and Services*, Florence, Italy, March 30 - April 1, 1992. 107-111.
- Sheth, A.P., van der Aalst, W.M.P., Arpinar, I.B. 1999. Processes driving the networked economy. *IEEE Concurrency*. **7**(3) 18-31.
- Sipser, M. 1997. *Introduction to the Theory of Computation*. PWS Publishing Company.
- Stijn, E.V., Wensley, A. 2001. Organizational memory and the completeness of process modeling in ERP systems. *Business Process Management Journal*. **7**(3) 181-194.
- Stohr, E.A., Zhao, J.L. 2001. Workflow automation: overview and research issues. *Info. Systems Frontiers: Special Issue on Workflow Automation*. **3**(3) 281-296.
- van der Aalst, W.M.P. 1997. Verification of workflow nets. *The 18th International Conference on Application and Theory of Petri Nets*, Toulouse, France, June 1997. 407-426.

- . 1998. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*. **8**(1) 21-66.
- . 1999. Three Good Reasons for Using a Petri-net-based Workflow Management System. *Information and Process Integration in Enterprises: Rethinking Documents*, T. Wakayama et al., eds., Kluwer Academic Publishers, Norwell, Mass. 161-182.
- van der Aalst, W.M.P., Kumar, A. 2003. XML-based schema definition for support of inter-organizational workflow. *Info. Systems Res.* **14**(1) 23-46.
- van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P. 2003. Workflow patterns. *Distributed and Parallel Databases*. **14**(3) 5-51.
- Verbeek, H.M.W., Basten, T., van der Aalst, W.M.P. 2001. Diagnosing Workflow Processes Using Woflan. *The Computer Journal*. **44**(4) 246-279.
- Vyatkin, V., Hanisch, H.-M. 2001. Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems. *Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Antibes-Juan les Pins, France, Oct. 15-18, 2001. 113-118.
- WfMC. 1999a. *Interface 1 - Process Definition Interchange, WfMC TC-1016-P, Version 1.1*. Workflow Management Coalition, Winchester, Hampshire, UK. 103 pages.
- . 1999b. *Workflow Management Coalition Terminology & Glossary, WfMC-TC-1011, Issue 3.0*. Workflow Management Coalition, Winchester, Hampshire, UK. 65 pages.
- Wilson, R.J. 1996. *Introduction to Graph Theory, 4th ed.* Longman Group Ltd, Harlow, UK.
- Winograd, T., Flores, F. 1986. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex Publishing Corporation, Norwood, NJ.
- Yau, S.S., Hong, W. 1988. Verification of concurrent control flow in distributed computer systems. *IEEE Transactions on Software Engineering*. **14**(4) 405-417.

- Yu, E.S.K. 1993. Modeling organizations for information systems requirements engineering. *Proc. of IEEE International Symposium on Requirements Engineering*, San Diego, CA, Jan. 4-6, 1993. 34-41.
- Zhao, J.L., Bi, H.H. 2003. On the completeness of logic-based workflow verification. *Proc. of the 13th Workshop on Info. Technology and Systems*, Seattle, WA, Dec. 13-14, 2003. 73-78.