

**FORMAL TECHNIQUES FOR
ANALYZING BUSINESS
PROCESS MODELS**

By

ESWAR SIVARAMAN

Bachelor of Technology
National Institute of Foundry & Forge Technology
Ranchi, India
1996

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1998

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of

DOCTOR OF PHILOSOPHY
MAY 2003

Thesis
2003D
S1624f

Copyright

by

ESWAR SIVARAMAN

May, 2003

FORMAL TECHNIQUES FOR
ANALYZING BUSINESS
PROCESS MODELS

Thesis Approved:

Manjunnath Kannath
Thesis Advisor

David R. Pratt

Spencer Kook

Timothy A. Petton
Dean of the Graduate College

Acknowledgments

I would like to share with the reader a poem that I came across at the Museum of Fine Arts, Houston – it was titled INNOCENCE.

Young, naive, curious, and pure,
new to life and all the hardships to endure.

Untarnished by the world's offenses and lies,
he sees nothing but good through his unbiased eyes.

Waiting to be molded, he's impressionable as clay,
receiving characteristics and features day by day.

I know he'll soon grow up and find his way,
without too many bruises I hope and pray.

Innocence is a virtue pure and clean,
childhood to adulthood, somewhere in between.

I am deeply grateful to the faculty and staff of the School of Industrial Engineering & Management for letting me retain my innocence and grow up without too many bruises. The School has been a home away from home, and that which I lost in not being under my father's shadow, I have more than made up for, within the walls of the department. The sum total of who I am, my values, and my unflinching belief in goodness and honesty, all of which I have been fortunate enough to preserve while here at OSU, I owe to my parents; were it not for their support and blessings, this would not have been possible. It is not possible to acknowledge their sacrifices with a simple 'thank you' – my debt is eternal.

I have admired the personality, the professionalism, and the teaching style of many a professor, and I most earnestly aspire to see their shadows in mine. I would like to take this opportunity to record my admiration for the following professors – Drs. Kenneth E. Case, David B. Pratt, William J. Kolarik (Ind. Engg), Martin T. Hagan (Elec.

Engg), Wade B. Brorsen (Ag. Econ), Dave Witte, Alan A. Adolphson, John Wolfe, Dennis Bertholf (Mathematics), James Cain (Philosophy), Leroy J. Folks, Mark Payton (Statistics), and Blayne Mayfield (Computer Science). To each, I owe a debt of gratitude – I have learnt a lot by observing them, both inside and outside the classroom, and I am happy to say that most of my mannerisms have been influenced (or tempered) by the greatest gift that I could have ever received from them, that of being their student.

I would like to thank the members of my dissertation advisory committee – Dr. David B. Pratt, Dr. William J. Kolarik, and Dr. Syam Menon for their time and interest in helping me succeed. I have benefited immensely from my observations and interactions with all of them – at times, from a distance, but, close nevertheless.

This research was supported, in part, by the National Science Foundation, under the Scalable Enterprise Systems initiative (Grant# DMI-0075588).

I am (probably) the worst student anyone can ever have, and Dr. Manjunath Kamath is the best advisor anyone can ever have – he has allowed me tremendous freedom of thought, expression, and unlimited opportunity to explore my creativity and curiosity. It has been a privilege to have earned his trust, friendship, and cherished regard. There is a dialogue from the movie **Nixon** that I often recall – “a man must sometimes go down to the deepest and darkest depths of despair before he can begin to appreciate the glory of standing at the highest peak” – I am grateful that I had the benefit of Dr. Kamath’s counsel and wisdom to pull me out from my many frequent trips to the abyss. Growing up is not easy; growing up alone is even worse – Dr. Kamath is the closest surrogate that I have had for a father, while here in the U.S., and to him, I most fondly dedicate this dissertation with these words:¹

The time has come for closing books and long last looks must end
And as I leave I know that I am leaving my best friend behind.

A friend who taught me right from wrong and weak from strong
That’s a lot to learn, but what can I give you in return?

ESWAR SIVARAMAN
Stillwater, Oklahoma
May 2003

¹Lyrics from the title song of the movie “To Sir, With Love” (1967)

Table of Contents

1	Introduction	1
1.1	Business Process Modeling: Purpose and Scope	1
1.2	Motivation for this Research	4
1.3	The Control Flow Problem	6
1.3.1	The Control Flow Model	6
1.3.2	Control Flow: Statement of the Problem	9
1.3.3	Control Flow: Overview of Research	11
1.4	The Resource-Sharing Problem	12
1.4.1	Formalism for Specifying Resource Requirements	12
1.4.2	Resource-Sharing: Statement of the Problem	13
1.4.3	Resource-Sharing: Overview of Research	15
1.5	Summary	17
2	Review of the Literature	19
2.1	Business Process Modeling: Major Issues	19
2.2	Business Processes - General Classification	21
2.3	Business Process Modeling Methodologies	22
2.4	Workflow Management	24
2.4.1	The Modeling Phase	24
2.4.2	The Execution Phase	26
2.4.3	Implementation Issues in Workflow Management	27
2.4.4	Process & Workflow Meta-Models – Basic Concepts	27
2.4.5	Summary	28
2.5	Control Flow Verification: Research Review	29
2.5.1	Petri-net Formalizations	29
2.5.2	Graph-theoretic Reductions	31

2.5.3	Model-theoretic Event Algebras	32
2.5.4	Other Related Network Models	33
2.5.5	Summary	33
2.6	Resource-Sharing Correctness: Research Review	34
2.6.1	Summary	36
3	The Korreктness Algorithm	37
3.1	Introduction & Background	37
3.2	The Korreктness Algorithm	38
3.2.1	Definitions	39
3.2.2	The Control Flow Problem is NP-Complete	41
3.2.3	Construction of Meta-Paths	46
3.2.4	The Komplete Korreктness Algorithm	50
3.2.5	Resolving Cycles in the Control Flow Model	50
3.2.6	Complexity Analysis	55
3.2.7	Diagnostic Checking of the Control Flow Model	62
3.3	Summary	64
4	The Resource-Sharing Problem	65
4.1	Introduction & Background	65
4.2	The Control Flow Model Revisited	66
4.3	The Resource-Sharing Problem: Some Ideas	70
4.4	Single-Instance Verification	71
4.4.1	Identifying Circular-Waits Within Concurrent Sets	72
4.4.2	Identifying Circular-Waits Across Concurrent Sets	74
4.5	The Static-Design Net Representation	77
4.5.1	The Static-Design Net	77
4.5.2	Computing Minimum Resource Requirements	82
4.5.3	Summary	83
4.6	Multiple-Instance Verification	84
4.7	Summary	87
5	Modeling and Analysis of Business Process Models	89
5.1	MAPS: Proof-Of-Concept Implementation	89
5.2	Modeling Interface	90

5.3	Analysis and Verification Capabilities of MAPS	94
5.3.1	Syntax Verification	94
5.3.2	Control Flow Verification	94
5.4	Summary	97
6	Summary & Research Contributions	98
6.1	Summary	98
6.2	Research Contributions	100
6.3	Future Research Directions	101
A	Petri Nets: A Primer	104
A.1	What is a Petri Net?	104
A.2	Basic Definitions	106
A.3	Additional Concepts	108
A.4	Invariant Analysis	109
	Bibliography	118

List of Figures

1.1	A Sample Control Flow Model	7
1.2	Mapping an OR logical operand with XORs and ANDs	8
2.1	Classification of Business Processes	21
2.2	Example of a Workflow Specification	25
2.3	Workflow Management System - Reference Model	26
2.4	Process Meta-Model	28
2.5	Petri Net Representation - An Example (control-flow only)	30
2.6	Two Examples – (a) Non Free-Choice Net, and (b) Free-Choice Net	31
2.7	Metagraphs - An Example	35
2.8	Representational Equivalence of Metagraphs & Petri nets - An Example	35
3.1	An Illustration of Paths and Meta-paths	39
3.2	(a) XOR-Representation, (b) AND-Repr., and (c) Path-Repr. Graphs	42
3.3	A Control Flow Model with the Maximum Number of S - F Paths	45
3.4	Connectivity between Vertices in V^* and $\mathbf{V} \setminus V^*$	52
3.5	Control Flow Sub-model Possibilities for the Case $V^* \cap \mathbf{A}_J \neq \emptyset$	53
3.6	Illustration of Multiple Paths from an AND-Split to an XOR-Join	54
3.7	Expected Number of Valid Meta-Paths in a Random Control Flow Model	58
3.8	Incorrect Process Model - An Example	63
4.1	Partitioning the Control Flow Model – An Example	67
4.2	Two different classes of Circular-Wait (CW) problems	72
4.3	Circular-Wait Within A Concurrent Set – Another Example	74
4.4	Petri Net Representation of the Process Model – An Example	76
4.5	The Static-Design Net Representation – An Example	78
4.6	Another Example of a Deadlock-Free Process Model	81

4.7	Example of a Process with Problems of Deadlock across Multiple Instances	85
5.1	A Sample Screen-shot of MAPS	91
5.2	Identifying the Set of Valid Meta-paths	95
5.3	Identifying the Set of Valid Meta-paths	96
A.1	An Example of a Petri Net Model	105

List of Tables

1.1	Types of Business Processes	2
1.2	Incorrect Control Flow Models – Some Illustrations	10
1.3	Incorrect Resource Allocation Models – Some Illustrations	14
2.1	A Comparison of Metagraphs and Petri nets	36
3.1	Control Flow Sub-models with Empty Cycles	51
4.1	Petri Net Mappings of Basic Routing Constructs	75

List of Algorithms

1	Path Enumeration	45
2	Meta-Path Enumeration	49
3	The Korreктness Algorithm	50
4	The Partition Algorithm	68

List of Notation

Nomenclature for the Control Flow Model

\mathbb{N}	Set of natural numbers, $\{0, 1, \dots\}$
\mathbf{V}	Set of all vertices in the control flow model
$N^+(x), N^-(x)$	Set of all vertices leading out from, and into, vertex x
S	The S tart node in the control flow model
F	The F inish node in the control flow model
\mathbf{T}	Set of all T asks in the control flow model
\mathbf{X}_S	Set of all XOR-Splits in the control flow model
\mathbf{X}_J	Set of all XOR-Joins in the control flow model
\mathbf{A}_S	Set of all AND-Splits in the control flow model
\mathbf{A}_J	Set of all AND-Joins in the control flow model
a_J, x_J	Number of AND- and XOR-Joins
a_S, x_S	Number of AND- and XOR-Splits
$\Delta_{A_S}, \Delta_{X_S}$	Maximum out-degree of AND- and XOR-Splits
$\Delta_{A_J}, \Delta_{X_J}$	Maximum in-degree of AND- and XOR-Joins
$\delta_{A_S}, \delta_{X_S}$	Minimum out-degree of AND- and XOR-Splits
$\delta_{A_J}, \delta_{X_J}$	Minimum in-degree of AND- and XOR-Joins
\mathbf{E}	Set of all edges in the control flow model
$c(x, y)$	Counter on edge (x, y) in the control flow model
\mathbf{P}	Set of all $S - F$ paths in the control flow model
\mathbf{M}	Set of all valid meta-paths
$V(m)$	Vertices covered by the paths in valid meta-path m
$CountersInAND(a)$	Set of all counters arriving at either AND-Join or -Split a
$PathsThrough(a)$	Set of all paths through AND-Join a
$ANDJoinsInPath(p)$	Set of all AND-Joins in path p

Nomenclature for the Resource-Sharing Model

R	The set $\{R_1, R_2, \dots, R_r\}$ of all resources
$R_i^\#$	Number of units available for resource R_i
$R_i^{Cap}(T_j)$	Number of units of resource R_i captured by task T_j
$R_i^{Rel}(T_j)$	Number of units of resource R_i released by task T_j
Con	Set of all concurrent sets $\{Con_i\}$ obtained from the control flow model
Con_i^m	Set of concurrent elements that occur on valid meta-path m
$Con _{V(m)}$	The set of concurrent sets $\{Con_i^m\}$ with vertices restricted to $V(m)$
SD_{Net}^m	Static-Design Net representation of valid meta-path m
C_m	Incidence matrix of SD_{Net}^m
σ^m	Transition sequence $Con_0 = \{S\}, Con_1, \dots, Con_q^m = \{F\}$ in SD_{Net}^m
M_0^m	Initial marking of SD_{Net}^m that enables σ^m
\mathcal{F}_m^+	Set of weighted arcs connecting transitions to places in SD_{Net}^m
\mathcal{F}_m^-	Set of weighted arcs connecting places to transitions in SD_{Net}^m

Chapter 1

Introduction

Chapter Overview

A business process is much like a recipe – it involves some tasks, ingredients, and resources, all coming together to create something that is useful (and hopefully palatable). While cooking involves mostly sequential activities, business processes are characterized by combinations of concurrency, choice, and asynchronism, the mix of which could lead to incorrect designs. The purpose of this chapter is to highlight challenges in the verification of business process designs, and to chart the scope and purpose of this research.

1.1 Business Process Modeling: Purpose and Scope

A business process is an ordered sequence of tasks/activities involving people, materials, energy, equipment, or information, designed to achieve some specific business outcome. Business processes are usually one of either a material, information, or a people process, or a combination thereof, the characteristics of which are presented in Table 1.1 [21].

Table 1.1: Types of Business Processes

	Process Type		
	Material (Things)	Information (Data)	People (Relationships)
Purpose	Transform and assemble raw materials and components into other components and finished products, using resources.	Store, retrieve, manipulate, display, and communicate structured and unstructured data and knowledge	Articulate and complete conditions of satisfaction between customers and performers
Characteristics	Based on the traditions of industrial engineering	Based on the traditions of computer science and software engineering	Based on structures of human communication and coordination
Verbs	Assemble, Inspect, Transform, Store, Transport	Send, Transact, Invoke, Save, Forward, Query	Request, Promise, Offer, Decline, Propose, Cancel, Measure

A business process specifies what a business does, and more importantly, determines how well the business does what it does [65]. To this end, irrespective of the type or the context of the business process, it is imperative that it be well-designed, to ensure that it is both effective and efficient. The *effectiveness* of a business process is a function of the match between the process’s operational objectives and the customer’s needs, and *efficiency* is an assessment of the process’s performance and the level of resource utilization, and depends on its configuration (i.e., its design). The standard approach to designing and implementing business processes is to rely on a domain expert to develop a process configuration that is subsequently “tuned-up” and configured using descriptive (e.g., simulation, queuing models) and/or prescriptive (e.g., optimization) techniques. However, there is a subtle, but significant question that is often never asked, namely – “what is the guarantee that the process’s configuration is correct?” This question has

become increasingly important, given the growing interest in process automation [18, 21] and enterprise integration [62]. Problems, if any, in the design of a process, are usually detected by simulating the run-time behavior of a process. The purpose of this dissertation is to develop generic techniques for verifying the correctness of a process's design, by focusing exclusively on its static structural definitions, without recourse to any simulated executions.

The study of business processes requires that a description of the business process be prepared by a domain expert, namely, one that is context-specific and rich in detail, and accommodative of different operational perspectives, occurrence scenarios, etc. – SAP's EPC [52], Baan's DEM¹, and IDEF3 [65] are all examples of process description languages that allow a domain expert to represent his/her understanding of the business process with complete conceptual clarity [47]. Business process modeling (BPM) is the collective term for the process of specifying business process descriptions. To quote Vernadat [93]:

“Enterprise (process) modeling is concerned with the representation and specification of the various aspects of an enterprise's operations, namely, *functional* aspects that describe what things are to be done, and in what order; *informational* aspects that describe which objects are used or processed; *resource* aspects that describe what or who performs things and according to which policy; and *organizational* aspects that describe the organizational structure within which things are to be done.”

The purpose of this research is to develop techniques for verifying that the *functional* and *resource* aspects of a process's design are correct. The major questions addressed in this research are briefly summarized below:

1. FUNCTIONAL ASPECTS – is the logic of the process correct, i.e., does the flow of control within the process ensure that the process will execute correctly from initiation to completion?

¹<http://www.dynaflow-dem.com>

2. RESOURCE ASPECTS – is the release and capture of shared resources among different tasks, either in the same or in different instances of a process, well-designed so as to avoid conflict?

1.2 Motivation for this Research

The need for re-designing existing business processes, improving process efficiencies, coordinating technology with distributed manpower and material resources, and enforcing rapid process development and design makes it imperative to adequately represent, study, and when possible, automate business processes [22, 33]. This is especially significant in the context of today’s growing interest in *workflow management*, which promises automated control and coordination of business processes, made possible by the numerous advances in information technology [18, 79, 62]. Consequently, it is important that design errors, if any, be identified and eliminated before the process is *deployed*, i.e., implemented for execution by an automated system. Unlike manual implementation and coordination of a process, where human intuition can readily respond to errors and inconsistencies, *automated solutions require that the process, by design, be correct*. This will guarantee that any delays or errors in the automated execution arise only from sources like data inconsistency, failure of supporting IT infrastructure, etc., and not for anything lacking in the design of the process.

Process descriptions are generally developed using graphical languages that include constructs for modeling concurrency (AND operands) and choice (XOR operands), the combination of both of which can result in incorrect process designs – this is discussed further in Section 1.3.2. The verification of control flow correctness has received much attention recently, in that several restricted classes of the control flow problem have been addressed to date [84, 1, 3, 75, 61]; additionally, the general control flow problem has been shown

to be NP-complete [40]. As regards the study of resource allocation policies, there has been some work, especially in understanding connectivity issues, using metagraphs [11]. However, this work does not extend to answering questions about the detection of potential deadlock possibilities in business process definitions. These are the questions dealt with, in this research.

In the larger context of business process modeling, the research presented herein is an essential step in developing an integrated framework for the modeling and analysis of business processes that [49, 23]:

- Reduces the gap between the domain expert and the business process analyst.
- Allows for the design and analysis of business processes to be simultaneous, with analysis influencing the design of effective and efficient processes.
- Clarifies ambiguities in the domain expert's interpretation, experience, and expectations of the business process through immediate qualitative analysis.
- Provides a seamless, almost invisible translation between the description of the business process and the formalization that feeds the underlying analysis.
- Provides linkages to other analysis techniques that can be used to derive summary metrics about the run-time performance of the business process.

That the verification of the design of a business process is a fundamental problem that should be undertaken for any process modeling effort, is undisputed. That it has only taken on an increased urgency, given its relevance to current interest in automated control and coordination of business processes, is the motivation for this research.

1.3 The Control Flow Problem

1.3.1 The Control Flow Model

The description of a business process is usually based on a graphical syntax which includes constructs for representing choice and concurrency, along with details of specific operational scenarios, and various perspectives, namely, functional, informational, and organizational [22, 93, 11]. While a process's description must be semantically rich, and a formalized model may be completely context-independent, the underlying process logic can be represented with just a few basic elements that are both amenable to analysis and are also semantically useful. The *control flow model* captures the partial or total ordering among the tasks that constitute the process, and is defined using the following elements [41, 95]:

Task: An abstraction of either a unit activity, or a composite description of a larger sub-process, embedded in the process's definition. It is graphically represented with a rectangular symbol.

AND-Split: A logical operand that models the *concurrent creation* of several parallel threads of control from a single incoming flow.

AND-Join: A logical operand that models the *asynchronous completion* of several parallel sub-threads of execution, to be followed by a common outgoing flow.

XOR-Split: A logical operand that depicts *choice* in the selection of exactly one of several possible outgoing control flows from a single incoming flow.

XOR-Join: A logical operand that merges several *mutually exclusive, multiple sources of control*, to create a common outgoing flow.

Directed Arrow: The flow of control between the various elements is captured graphically with directed arrows leading from the preceding to the succeeding elements (task/logical operand).

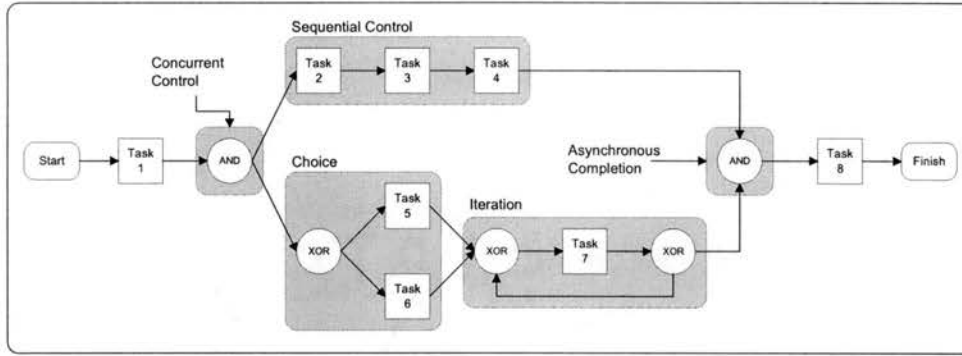


Figure 1.1: A Sample Control Flow Model

Figure 1.1 illustrates all of these basic elements, including iteration, which basically models the recursive nature of the flow of control, as would be required in say, an inspection (reject/accept/re-process) activity. Observe that the control flow model is devoid of the operational details of the tasks, namely, their inputs and outputs, which, while necessary for developing a process description that is conceptually complete, are not essential for capturing the process’s logic. Additionally, the control flow model includes two special constructs – “Start” and “Finish” which indicate that the process has a unique initiation, and a unique termination. There are several advantages to enforcing the unique Start and Finish in the control flow model, namely,

- Sub-processes that have already been verified can be encapsulated and subsumed as *composite* tasks in a larger control flow model to represent, with coarser granularity, the individual task descriptions. Such composite tasks, if needed, could be “blown-up” to reveal their complete detail, by relying on the **S** and **F** nodes of the composite tasks to easily plug into the bigger model.
- It would be possible to achieve an incremental, or piece-wise, verification of the various sections of the process, without requiring that the entire model be specified before any analysis may begin. This would be made possible, by bounding, and isolating with a **Start** and **Finish**, those sections of the model that are logically disjoint and separated from the rest of the model, so far as control flow is concerned. Consequently, this “debugging” at a local level would increase the speed of development of a correct control flow model, which, if need be, could be composed of previously verified, smaller component models.

The four logical operands presented above are adequate for capturing the logic of any process. However, certain process description languages (e.g., IDEF3, EPC) also include constructs for modeling inclusive OR-Splits (-Joins), i.e., activate (merge) at least one of several outgoing (incoming) control flows. This, unfortunately, is very ambiguous in interpretation, and is ill suited for formalization, for it neither suggests which thread(s) to activate, nor how many, except, perhaps those recorded by the domain expert as part of his/her experience. Figure 1.2 illustrates a reformulation of an OR-Split with just XORs and ANDs; the equivalent mapping for an OR-Join would be the dual (i.e., graph with arrows reversed) of the model shown in Figure 1.2(b). Unfortunately, this comes with the penalty that the number of extra XOR or AND operands grows exponentially with the out/in-degree of the OR operand. It is recommended that the use of an OR operand be explicitly discouraged in developing process descriptions, so as to avoid ambiguity and inconsistency of interpretation [2, 53].

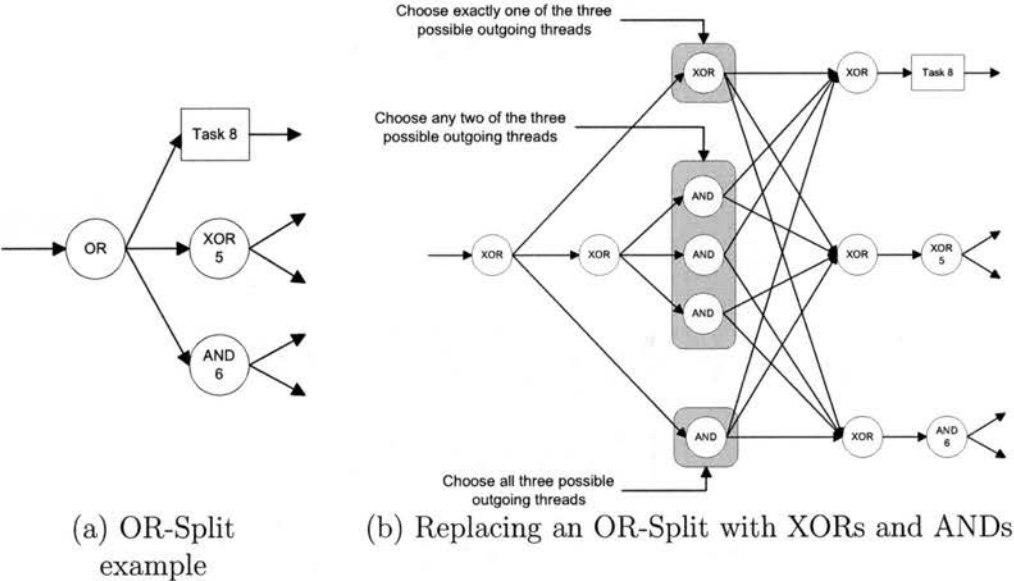


Figure 1.2: Mapping an OR logical operand with XORs and ANDs

1.3.2 Control Flow: Statement of the Problem

The question of verifying the process's design to establish control flow correctness, is simply this – *can it be verified that beginning with Start, the process will always reach Finish?* In order that the definition of control flow correctness may be made more precise, it remains to understand what counts as an incorrect model. Table 1.2 illustrates several examples of incorrect control flow.

There are three points that clarify themselves in all five examples of Table 1.2, namely,

1. The process must terminate exactly once, i.e., *unique termination*.
2. The process must terminate completely, without any residual control flows hanging in the balance, i.e., *proper termination*.
3. The bulk of control flow errors arise from interspersing XOR and AND logical operands.

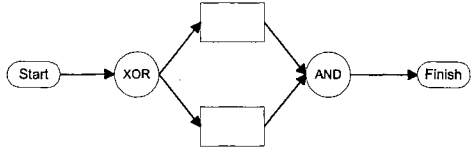
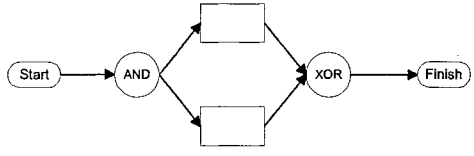
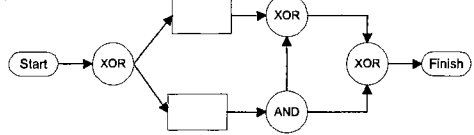
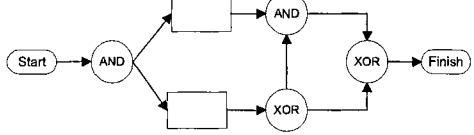
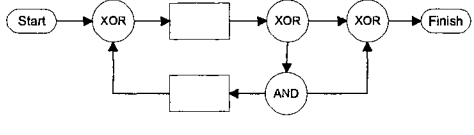
The control flow problem can now be restated as:

The Initiation Problem is to determine if there is a sequence of task executions that will lead to the execution of a particular task – this has been shown to be NP-complete [40].

The Termination Problem is to determine if the control flow specification will lead to a terminal state – this has been shown to require exponential storage requirements [40].

An alternate derivation of the NP-completeness of the control flow problem is presented in Section 3.2.2.

Table 1.2: Incorrect Control Flow Models – Some Illustrations

Incorrect Control Flow Model	Discussion
	<p>An example of deadlock. The process will never terminate, since the AND operand will wait indefinitely for two incoming flows of control, while the XOR creates only one.</p>
	<p>An example of multiple repetitions. The process will “Finish” twice, since the XOR is expecting only one incoming control flow, while the preceding AND activates two parallel paths.</p> <p><u>Note:</u> It may be argued that it does not matter which of the two tasks finishes first, as long as one does, to enable control flow to proceed further. However, in this case, what is required is not an XOR-Join, but an OR-Join, which must be formalized using XORs and ANDs as illustrated in Figure 1.2.</p>
	<p>If the XOR next to “Start” chooses the top branch, the process will terminate properly; if it chooses the lower branch, the process will terminate twice.</p>
	<p>If the XOR on the lower branch chooses the control flow to its top, the process will terminate properly; if it chooses the flow to its right, the process will terminate once, but there will be an AND operand waiting for a control flow that will never arrive.</p>
	<p>A process model with the possibility for infinite repetitions. If the second XOR chooses the branch to its right, the process will terminate properly; if it chooses the lower branch, the process will terminate more than once.</p>

1.3.3 Control Flow: Overview of Research

There are two approaches to establishing the correctness of control flow in business process models, namely, *correctness by construction* (i.e., build it correctly), or *correctness by inspection* (i.e., check it completely). The former relies on strict grammatical rules that govern the composition of the various elements in the model, and this is the basis of the model-theoretic event algebras designed by [81, 30], which, however, do not guarantee that all models can be constructed using the pre-specified composition rules. The latter, on the contrary, is more appealing, in that it does not inhibit the modeler or the analyst. These are the considerations that have prompted the use of graph-theoretic techniques [76, 75, 11, 61], and Petri nets [1, 84, 88] for verification studies.

However, all of these approaches impose restrictions on the form and structure of the control-flow model to render the analysis questions more tractable, namely, that the control-flow model be acyclic (no loops), and that the Petri-net constructions remain *free-choice*.² Moreover, with the exception of [76, 3], all of the other approaches present only theoretical analyses, the implementation of which is left to question.

Additionally, there exists no precise mechanism to isolate and identify the sources of control flow anomalies, since it is not just enough to identify that there is a problem, but, it is more important to identify the *why* and the *where*; these are precisely the incentives for studying the control flow problem.

²A special class of Petri nets wherein all choices within the Petri net are free – this is elaborated further in Section 2.5.

1.4 The Resource-Sharing Problem

1.4.1 Formalism for Specifying Resource Requirements

A process, in the course of its execution, will use some resources – more specifically, tasks in a process will often require the use of resources (e.g., machines, people, instruments) that they *capture* (i.e., access, exclusively use), and which are then *released* by either the tasks that captured them, or by other subsequently executed tasks. The notion of a “resource” as defined here is not to be confused with items like, say, machine-oil or lubricating grease which are consumed, i.e., “depletable resources” exhausted by the process, or with items like, say, scrap and metal-filings which are created by the process. In specifying the resource requirements of a process, the focus is on re-usable, non-perishable, and non-depletable physical or informational entities that are accessed or captured by tasks, and which are then subsequently released wholly, without loss or detriment in their size, quantity, or operational ability [37, 43].

More formally, the (re-usable) resource requirements for the tasks in a process are specified as:

- $\mathbf{R} = \{R_1, R_2, \dots, R_r\}$ is the set of all resources. $\forall R_i \in \mathbf{R}$, $R_i^\# =$ number of units available for resource R_i .
- $\forall R_i \in \mathbf{R}$, $R_i^{Cap} : \mathbf{T} \rightarrow \mathbb{N} = \{0, 1, \dots\}$ is a functional that specifies the number of units of resource R_i *captured* by each task, where \mathbf{T} is the set of all tasks.
- $\forall R_i \in \mathbf{R}$, $R_i^{Rel} : \mathbf{T} \rightarrow \mathbb{N}$ is a functional that specifies the number of units of resource R_i *released* by each task.

1.4.2 Resource-Sharing: Statement of the Problem

In the course of the execution of a business process, deadlock arises when tasks that have captured some resources are blocked indefinitely from access to resources held by other tasks [19, 43, 46]. The following four conditions are necessary for the occurrence of deadlock, namely, [19, 34, 67, 8]

Mutual Exclusion Tasks require exclusive use of resources.

Hold-while-waiting Tasks continue holding onto resources that they have captured, while waiting for other required resources to become available.

No Preemption Tasks holding resources determine when they are released.

Circular-Wait A closed chain of two/more tasks waiting for resources held by one another.

In the context of business process modeling, the first three conditions stated above are unavoidable, i.e., resources are assigned for exclusive use of the tasks that require them, and cannot be preempted without externally aborting the corresponding tasks. To this end, the primary design issue that needs to be addressed in the design of business processes is to alert the designer to deadlock possibilities that may arise from circular-wait conditions that are not immediately evident in the process's design. Clearly, a very elementary check for the correctness of resource allocation is to verify that the number of times a resource R_i is captured is equal to the number of times it is released. However, such a simplistic check is inadequate for guaranteeing the correctness of a process's design. The problems that arise in the consideration of resource-sharing are best motivated with several examples, as presented in Table 1.3. The convention followed for these models is – (i) the capture and release of resources by each task is specified with directed arrows entering and leaving the task symbols, respectively, and (ii) it is assumed that $R_i^\# = 1$ for all the resources cited therein.

Table 1.3: Incorrect Resource Allocation Models – Some Illustrations

Incorrect Resource Model	Discussion
	<p>There are two problems, namely, (i) resource R_5 is released before it is captured, and (ii) a potential circular-wait could arise if T_2 captures R_1 and T_3 captures R_2 and end up waiting indefinitely for each other to release their resources.</p>
	<p>There are two major problems, namely, (i) if T_2 and T_3 are assigned R_1, R_2, respectively, then the process will be deadlocked, since T_1 and T_4 will never be executed for want of resources that will never be released (by T_5 and T_6) before they are completed, and (ii) the number of capture requests for both R_1 and R_2 among the concurrent tasks T_1, T_2, T_3, T_4, exceeds the available number of units, thereby making the design infeasible.</p>
	<p>The process will get deadlocked if T_1 captures R_1 and T_3 captures R_2, whereupon neither T_2 nor T_4 can proceed any further, and the resources will never be released – an example of circular-wait.</p>
	<p>Clearly, the control flow is very straightforward. However, consider an “incident” – T_1 captures R_1 followed by T_2 which captures R_2 and releases R_1, and proceeds to execute T_3; meanwhile, T_1, being enabled, captures R_1 again and starts another <i>instance</i> of the process, thereby resulting in deadlock since the previous instance will not release R_2 without R_1 (task T_4), while the second instance will not release R_1 without R_2 (task T_2)– another circular-wait.</p>

There are several questions that arise naturally from the three examples presented in Table 1.3, namely,

1. **Examples 1 and 3** – what is the order in which resources need to be assigned among concurrently enabled tasks, so as to avoid potential circular-wait?
2. **Example 2** – what is the minimum number of units required of each resource to enable the process’s design to actually succeed? In the case of example 2, for the concurrent tasks T_1, T_2, T_3, T_4 to actually proceed, a minimum of two units is required for resources R_1 and R_2 – this example was solved intuitively; can it be formalized?
3. **Example 4** – the incident described in example 4 merits some more attention. Clearly, the problem cited therein is not immediately evident, and the execution of a *single instance* of the process would proceed perfectly. However, it still remains to alert the designer about the potential for circular-wait that could arise if *multiple instances* of the process become enabled.

The resource-sharing problem can now be restated as:

Single-Instance Verification is to determine if the sharing of resources among tasks within an instance of a process could lead to deadlock.

Multiple-Instance Verification is to determine if the sharing of common resources among various instances of the process could lead to deadlock.

1.4.3 Resource-Sharing: Overview of Research

The study of deadlock has been motivated primarily by problems arising in operating systems, beginning with a problem in concurrent control proposed by Dijkstra [27] and solved by others [54, 24, 28, 56], namely, to develop an algorithm that will guarantee that exactly one among many competing tasks (or programs) will execute their “critical section” (presumably, access to some common computing resource). Subsequently, this

problem has been enlarged thus – “given a set of n tasks and m resources, does there exist an ordered sequence of the tasks that is *safe*, and will enable all the tasks to be completed? [35, 42, 69, 43]. This problem has received enormous attention, given its importance to operating systems, parallel computing, and distributed database systems, an extensive review of which is presented in [46, 39, 82, 31]. These approaches rely primarily on the definition of a resource-allocation graph³ that captures the state of the system at a particular time, namely, the set of unfulfilled resource requests, the set of captured resources, etc., and verifies that the system’s evolution into its next state is also *safe* [19, 43]. The existence of a cycle in these resource allocation graphs is necessary for the occurrence of a deadlock [19], and is sufficient when there is just a single unit of each resource [43, 67, 98, 63]. Unfortunately, these approaches are not applicable to the context of business process modeling, since the process’s logic pre-specifies the order in which the tasks need to be executed. These concerns have also arisen in the study of deadlock as is related to the operation of flexible manufacturing systems [8, 94, 29, 99, 74, 73], a recent review of which is presented in [92].

The most common approaches for handling deadlock are: [43, 46, 94, 57]

Prevention restrains the request structure of processes so that deadlock is impossible, namely, by falsifying any one of the four necessary conditions for deadlock (refer Section 1.4.2).

Detection-Recovery approaches allow deadlock to occur and focus on expedient recovery.

Avoidance uses current state information along with the knowledge of task request and release structures to guide look-ahead policies that control how resources are allocated so that deadlock is avoided.

³also referred to as “wait-for graphs” [67, 58].

The general deadlock avoidance problem has been shown to be NP-complete [90, 34, 32, 67]. In the spirit of the adage “prevention is better than cure,” the focus of the second problem addressed in this research is *deadlock prevention*, namely, to develop techniques for alerting the designer of a business process about potential deadlock possibilities.

1.5 Summary

The purpose of this chapter has been to motivate the need for a formal foundation to verify the correctness of a business process’s design. The design of a business process minimally requires the specification of the process’s logic, and the resource requirements for its constituent tasks. Two major problems have been identified, namely, verifying the correctness of (i) control flow, and (ii) resource-sharing requirements – these are the questions dealt with, in this dissertation.

The control flow problem relates to establishing the correctness of a process’s logic. More specifically, it remains to verify that a single instance of the process will execute correctly from initiation to completion (i.e., *unique termination*), and that there are no incomplete control flows remaining elsewhere upon completion of the process (i.e., *proper termination*) [40]. These two conditions must be satisfied to establish that the control flow model is correct – they are not intended to excessively constrain modeling flexibility; on the contrary, they serve to focus and discipline a modeler’s intuition in developing a more precise design that is logically, and therefore, operationally correct.

The resource-sharing problem relates to establishing the correctness of a process’s (re-usable) resource requirements. More specifically, it remains to establish that the sharing of common resources among different tasks, either within a single-, or across multiple-instances of the process does not lead to situations wherein two or more tasks compete

for resources, without relinquishing control of currently held resources, thereby leading to deadlock. Several interesting challenges emerge from the study of this problem, namely, is deadlock occurring as a result of inadequate resource availability, or is it truly a design error that is not immediately obvious? More particularly, is it necessary to simulate the operation of a process to identify any such design errors? That the answer to this question is NO is a prelude to some of the interesting approaches developed herein.

The remainder of the document is organized as follows. Chapter 2 presents a review of the issues and opportunities in business process modeling, and summarizes all relevant research related to the two problems mentioned above – it should excite the reader to know that the question of verifying the correctness of a process’s resource-sharing requirements has not been previously studied, and that the results presented herein are the first in this regard. Chapter 3 studies the control flow problem, and presents a new algorithm for verifying the correctness of control flow in any control flow model. Chapter 4 extends the results of Chapter 3, and presents a simple Petri net-theoretic approach to studying the correctness of resource-sharing requirements, both in single- and in multiple-instances of a process. Chapter 5 documents the features of a proof-of-concept implementation of the algorithms developed in this work. Chapter 6 concludes this dissertation with a summary of research contributions, and pointers for further research.

Chapter 2

Review of the Literature

Chapter Overview

The purpose of this chapter is to present both an overall appraisal of the issues and opportunities in business process modeling, and a review of the research approaches and results relevant to studying the correctness of control flow and resource-sharing issues.

2.1 Business Process Modeling: Major Issues

The complete conceptual description of a business process requires:

1. specification of the flow of control and the total/partial ordering between the various tasks, including feedback and feedforward modes of action,
2. specification of relevant inputs and outputs, and the flow of data as is dictated by the interconnections between the tasks,
3. assessment of the process's configuration, and a summarization of the process's dynamics (i.e., time duration), and estimates of control-flow transition probabilities (if required), and
4. identification of any hierarchical, or multi-level distinctions in the tasks that constitute the process, i.e., is a task *elemental*, or can the task be expanded to reveal other sub-processes?

These conceptualization requirements have prompted the creation of process *meta-models*, i.e., models about building models, which aim to standardize terminology and suggest an abstraction of how process models must be specified [7, 96]. Research in process meta-models has also been significantly influenced by the need to create process specifications amenable to computer implementation, as is required for workflow automation.

To summarize, the major issues in business process modeling (BPM) can be classified into the following sub-categories [33, 44, 45, 91]:

1. Process meta-models, process definition – language & grammar, and workflow schema representation architectures [60, 17, 59, 96, 18, 15, 55, 66].
2. Analysis of conceptual specifications of processes for syntactic and semantic correctness, and support for performance evaluation & process redesign [3, 1, 11, 76, 88, 84, 77, 79].
3. Implementation and run-time issues related to correctness & failure-handling mechanisms in workflow management systems [18, 50, 51, 33] and adapting to dynamic changes in workflow and process definitions [72].
4. Workflow Management System IT infrastructure & inter-operability standards, spearheaded by the Workflow Management Coalition [41, 95, 96].

This research is related largely to sub-topic 2 above, namely, to develop techniques for verifying the correctness of a process's design. However, it would be very instructive to trace the development of ideas in all the sub-categories above, more so, in the context of today's growing interest in automated solutions and the correspondingly increased demand for improved modeling and analysis techniques.

2.2 Business Processes - General Classification

In addition to the three kinds of processes identified in Table 1.1, business processes can be classified into one of *collaborative*, *production*, *ad hoc*, or *administrative*, depending on their business value and the degree of their repetition [60]. A process of high business value is more of a core competency, i.e., a fundamental process based on which the organization has been established (e.g., loan approval by a bank). The degree of repetition is a measure of how often the process is performed. Figure 2.1 classifies the four different types of business processes, with representative examples.

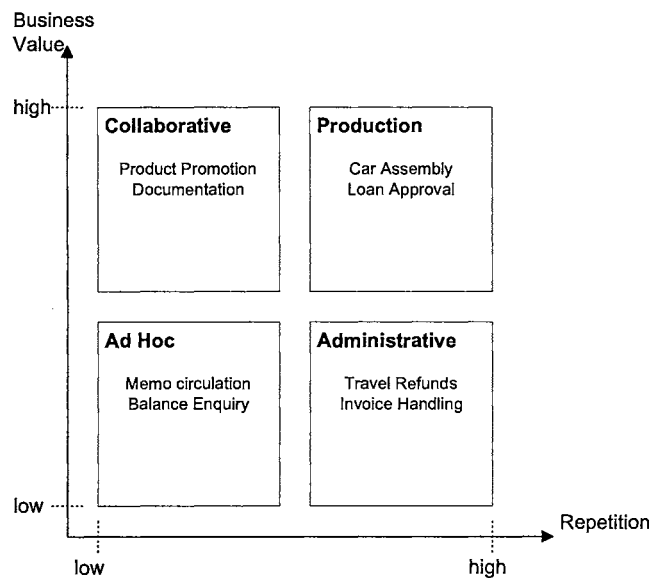


Figure 2.1: Classification of Business Processes

Collaborative processes are characterized by high business values and low repetitions – e.g., building a fighter jet, creating a patch for a Windoze bug, etc. The underlying process is generally unique and specific to the instance of the process. *Ad hoc* processes are characterized by low business values and low repetitiveness, and are created on-the-fly, literally speaking – e.g., enquiring about the number of customers with bank balances in excess of 13.27 dollars, circulating a birthday card for staff signatures, etc. There is

no defined structure or logic for these processes, and it changes from one situation to another. *Administrative* processes are highly repetitive, but of low business value – e.g., processing travel reimbursements, filing plan of study forms, etc. *Production* processes are high-value, high frequency processes that are repeated over and over, and represent the core processes of a company – e.g., approving loans in a bank, sorting mail in the post-office, etc. It is the efficient and effective execution of production processes that define the competitiveness of a company, and consequentially, merit the maximum attention in any BPM effort.

The complete specification of business processes includes (i) the control flow, i.e., the partial and total ordering specifying the sequence of the various tasks, (ii) the input-output requirements (i.e., information, materials) and (iii) the resource (people, machines, etc.) allocations for executing the various tasks. Depending on the type of the process (i.e., material, information, or people), the specification of a process would also include context-specific details like personnel involved, rollback-recovery procedures, exception handling procedures, abort-recovery consistency checks, communication protocols, etc. [33, 18, 21] – this is referred to as the “discovery” of business processes [21]. However, discovering and documenting the sequence of activities¹ within a business process is an iterative and time-consuming process, especially, in stating the flow of control from one activity to the next, specifying the logical transition conditions, etc., all of which reinforce the requirement for a precise modeling methodology.

2.3 Business Process Modeling Methodologies

The purpose of business process modeling is to produce an abstraction of the process that serves as a basis for detailed definition, study, and possibly, re-engineering, to eliminate

¹Both *tasks* and *activities* are used interchangeably in this document.

non-value added activities. To this end, the process model must allow for a clear and transparent understanding of the activities being undertaken, the dependencies among the activities and resources (people, machines, programs, data, etc.) necessary for the process. Process modeling methodologies can be broadly classified into three categories – *communication-based*, *artifact-based*, and *activity-based* [66, 17, 18, 33].

The communication-based methodology represents an action in a process as a communication between a customer and a performer, consisting of four phases – request, negotiation, performance, and acceptance. During any phase of the process, the performer of one process loop can be a customer of another loop, thus presenting any business process as a network of such customer-performer loops. This methodology focuses on communications occurring in the workplace and is geared towards one objective, namely, customer satisfaction, and is not suitable for other goals, especially, process analysis and investigation. The artifact-based approach focuses on the objects (artifacts) that are created, modified, and used in the process, i.e., the modeling of the process is based on the products, and their flow through the various activities; this would be suitable for administrative and ad hoc processes. The activity-based methodology focuses on decomposing the process into tasks that are ordered based on the dependencies (flow of control and data) between them.

The activity-based methodology has a number of distinct advantages [59, 66, 18] – (i) it is easily understood, (ii) it is readily amenable to formalization² and (iii) it is the preferred choice for computerized specification and modeling, as is evident in it being the basis for all the major modeling languages. The reader would no doubt note that the control-flow model of Figure 1.1 is an activity-centered process model.

²Formalizations generally simplify the process model, and replace conceptual descriptions with specialized abstractions, to focus primarily on the problems being studied.

2.4 Workflow Management

Businesses are increasingly relying on enterprise-wide integration of information using technologies like advanced database systems, client-server computing, Web-enabled transactions, etc., to improve the efficiency of their business processes, and to be more competitive in responding to customer needs [18, 50, 79, 66, 77]. These concerns have also stimulated the popularity of *workflow management* as a technique to address the needs for representation, study, and automation of business processes, especially information and people processes.

Workflow management supports both process specification, and automated execution (instantiation, monitoring, and data maintenance) of business processes, and is a next-generation extension to business process modeling that emphasizes the increased role that information systems have come to play in today's businesses [79]. Workflow management facilitates the coordinated execution of the various tasks that comprise a business process; it involves two phases – (i) the *modeling phase* that abstracts from business procedures and defines computer-implementable *workflow* specifications, and (ii) the *execution phase* that executes instances of the workflows to meet business requirements – both these phases are managed and coordinated by a Workflow Management System (WfMS).

2.4.1 The Modeling Phase

A *workflow*, as defined by the Workflow Management Coalition (WfMC)³ [41], is “a procedure where documents, information, or tasks are passed between participants according to a defined set of rules to achieve, or contribute to, an overall business goal.” Or alternatively, based on the activity-centered modeling methodology, a workflow can be viewed as a “collection of steps” that have to be performed in a certain order. A *workflow*

³An international consortium founded in 1993 to standardize terminology and enable inter-operability between different workflow management systems. <http://www.wfmc.org>.

schema specifies the set of steps that comprise a workflow, and the data and control flow between the steps – Figure 2.2 illustrates an example of the process of approving applications in a health insurance firm [50].

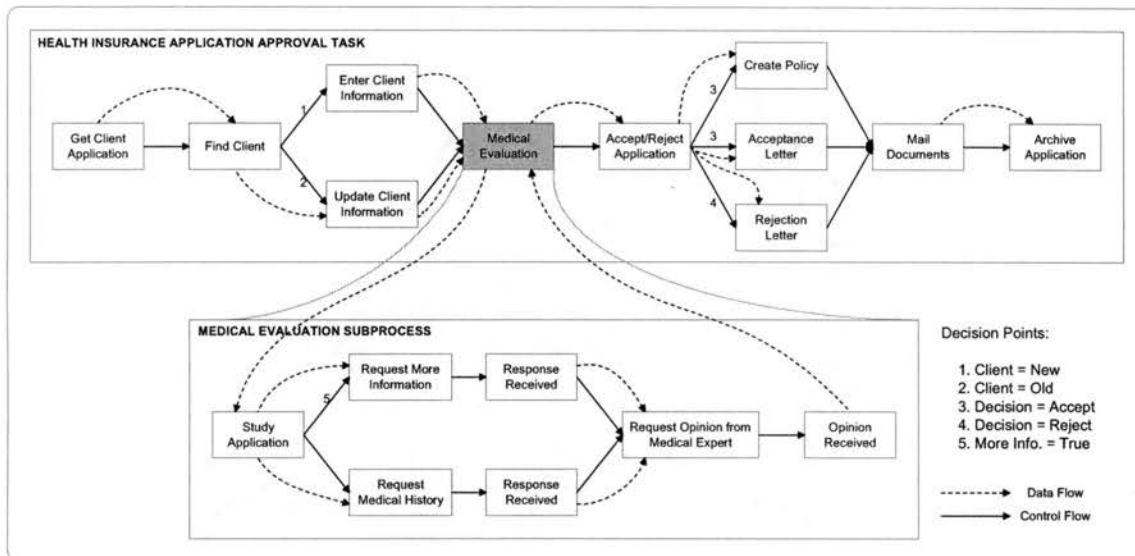


Figure 2.2: Example of a Workflow Specification (adapted from [50])

The data flow specification provides the mapping of data (inputs and outputs) between steps, and the control flow specifies the execution order of the steps. Several types of transition conditions can be specified in the control flow requirements – sequential, conditional branching, concurrent branching, and iteration, all of which were illustrated in Figure 1.1. To perform a business process, a *workflow instance* is initiated. Every workflow instance is associated with a *state* that reflects the values of the various data items associated with the workflow, and the state of the steps in the workflow, i.e., which of the steps have been completed, etc. The WfMC has established standards for process definition, and has developed generic modeling concepts to guide the creation of business process models (respectively, workflow schemas) [95].

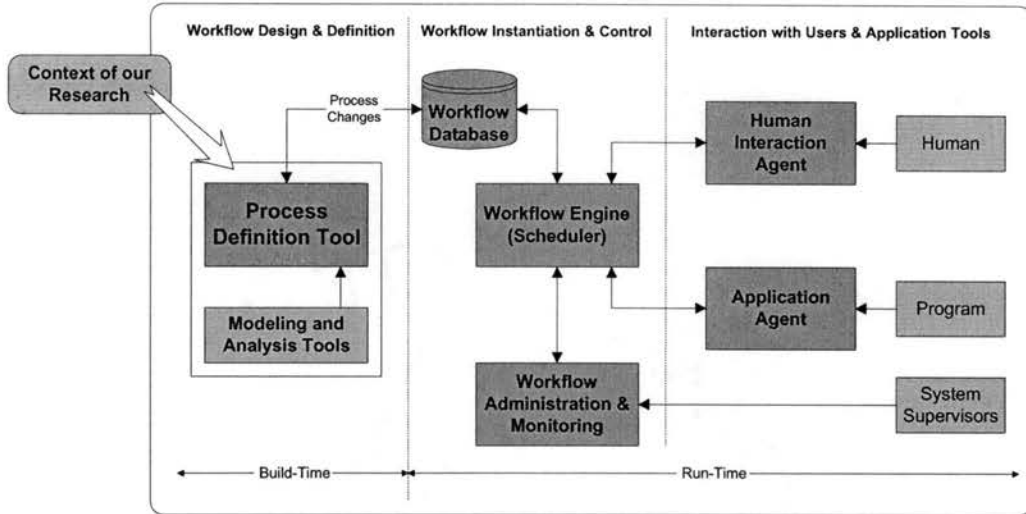


Figure 2.3: Workflow Management System - Reference Model (adapted from [41, 77, 50])

2.4.2 The Execution Phase

Figure 2.3 illustrates the three major functional areas that a WfMS provides support for, namely, workflow design & definition (the context of our research), workflow instantiation and control, and interaction with users & other applications [41, 77, 50]. The definitions of workflows, tasks, staff designations, etc., are all stored in the *workflow database*. This database also stores the states of the workflows that are in progress. Scheduling is usually performed by a *workflow engine*, which refers to the workflow database to determine the state of the various workflows in progress. Staff members interact with the WfMS through a *human interaction agent*, and they are presented with a work-item list that lists all the tasks that have been assigned to them. If a task requires a program or other applications, these are invoked by the *application agent*. The application agents interact with the workflow engine to fetch the data required to execute a particular step, and to communicate back the output (i.e., return status code and data) produced by the step. The workflow database is not accessible to applications external to the WfMS and other external resources (programs, databases, etc.) that are accessed by the applications executed on behalf of the workflow's steps.

2.4.3 Implementation Issues in Workflow Management

Ensuring data consistency and integrity of the workflow database is a problem that is attracting much research – this is largely due to the fact that business procedures are generally of extended duration, and traditional transaction models [18, 50, 51] have proven inadequate. More specifically, the focus has been on resolving correctness issues to ensure data consistency across multiple workflow instances, each of which may be in different states, but could require access to common data. The correctness requirements in workflow implementation can be broadly classified into two categories, namely, execution atomicity, and failure atomicity. Execution atomicity deals with how data is committed and how visibility of data between steps, both within and across workflow instances, is controlled. Failure atomicity determines what is to be done with the data that has already been committed to the steps of a workflow, in the event that a failure disrupts the workflow and affects database management and database integrity. This, and other run-time issues related to database management, data transaction control (check-in, check-out), etc., are beyond the scope of this research.

2.4.4 Process & Workflow Meta-Models – Basic Concepts

The WfMC has established commonly accepted terminology for the various components of a business process model and associated workflow specifications. The meta-model presented in Figure 2.4 is a refinement of that proposed by the WfMC [18, 41, 96].

The interpretation of the meta-model is as follows: the *Business Process* is represented by a *Workflow* that consists of many *Activities* coordinated by this *Workflow*. The *Activity* serves as an abstraction for the *Workflow* and *Task*. The semantics of this approach is that the *Workflow* consists of tasks or sub-workflows, i.e., a hierarchy of nested workflows. The association *cooperates* expresses the possibility of *Workflow* distribution within a

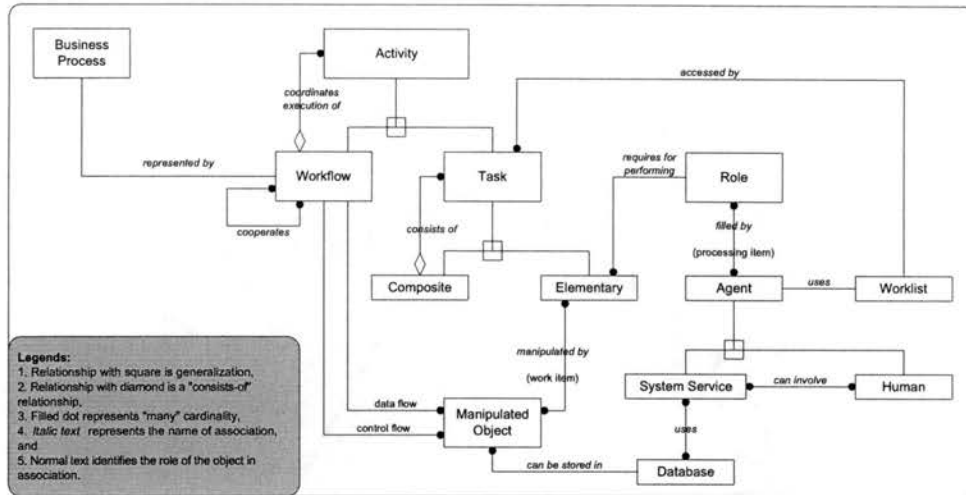


Figure 2.4: Process Meta-Model (adapted from [18])

distributed environment. The *Task* is an abstraction of an *Elementary Task* and a *Composite Task* that actually consists of *Tasks* recursively. The *Elementary Task* requires for its performance a *Role* that can be responsible for many *Elementary Tasks*. The *Role* is fulfilled by an *Agent* that plays the role of a *processing item*. The *Agent* has assigned *Tasks* that it is responsible for, through a *Worklist*. The *Agent* itself can be specialized into a *System Service* (program, application, etc.) or a *Human*. A *System Service* can be associated with many *Databases*. The *data flow* and *control flow* are represented by generic *Manipulated Objects* (any object or piece of information used and manipulated by *Workflow*) that can be stored in the database.

2.4.5 Summary

The discussion has thus far focused on summarizing the major developments in process automation and workflow management. The interested reader is directed to refer additional references, most notably, [60] for Section 2.2, [66] for Section 2.3, [33, 18, 60] for Section 2.4, and [21] for a comprehensive and well-written overview of the issues and opportunities in the discovery, design, deployment, and automation of business processes.

There has also been considerable interest in developing a common specification for documenting and describing business processes to standardize exchange and interaction of business data among companies maintaining different enterprise integration and process management systems – readers are encouraged to refer [48, 96, 7] for additional details. The remainder of this chapter will focus on reviewing current approaches to addressing the control flow and resource-sharing problems.

2.5 Control Flow Verification: Research Review

The control-flow problem was introduced earlier in Section 1.3, and a brief overview of current research presented in Section 1.3.3. To summarize, the current approaches to the control-flow problem may be classified into three main categories:

1. Petri net formalizations [1, 3, 4, 5, 84, 83, 87, 85, 86],
2. Graph-theoretic reductions [76, 75, 61], and
3. Model-theoretic event algebras [81, 30, 88].

2.5.1 Petri-net Formalizations

Petri-nets have emerged as a very popular technique for formalizing business process models for the following reasons [77, 66, 4] – (i) clear and unambiguous description of process logic, (ii) intuitive ease and feel of a self-documenting, graphical formalism that retains complete conceptual clarity, and (iii) extensive qualitative and quantitative analysis capabilities that would vastly extend the power and usefulness of structured process description languages like IDEF3.⁴ The Petri-net equivalent of the control flow model described in Figure 1.1 is illustrated in Figure 2.5.

⁴<http://www.idef3.com>

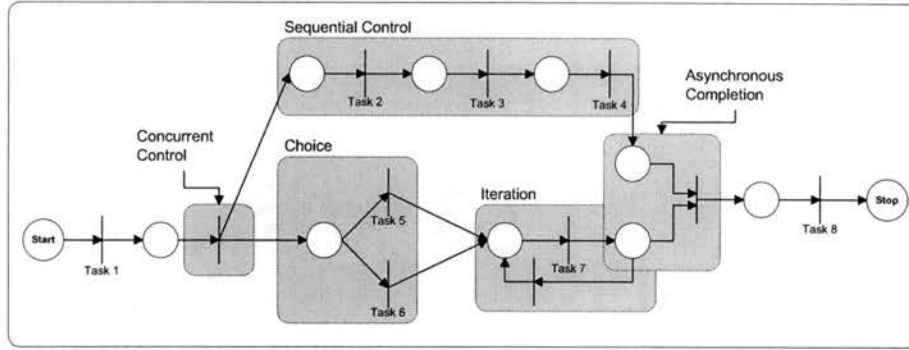


Figure 2.5: Petri Net Representation - An Example (control-flow only)

The standard approach to establishing control-flow correctness in Petri-net formalizations of business process models is to establish the *soundness* property [1, 3], or the *simple-control* property [84, 88], which is the initiation problem, and the termination problem both rolled in one (refer Section 1.3.2). Stated simply, the idea is to put a token in the place labeled **Start** (refer Figure 2.5) and to see if the execution of the Petri net will produce a token in the place labeled **Stop**, without leaving any residual tokens elsewhere in the net. These ideas form the basis of WOFLAN, a modeling and verification tool developed at the EINDHOVEN INSTITUTE OF TECHNOLOGY,⁵ The Netherlands.

The majority of business processes formalized as Petri nets require that the Petri net be *free-choice*, a special class of Petri nets wherein all choices within the net are free[25]. Translated literally, this implies that the choice of which transition to fire, in the presence of conflict, is not influenced by any other place other than the input places of the transitions in contention – Figure 2.6 highlights an example.

The net in Figure 2.6(a) is not free-choice since the resolution of the conflict at place P_1 is not *free*, i.e., the choice on whether T_1 or T_2 will be fired depends on the availability of a token in P_2 , while T_1 may be fired irrespective of the presence/absence of a token

⁵<http://tmitwww.tm.tue.nl/staff/everbeek/projects/woflan/woflan.html>

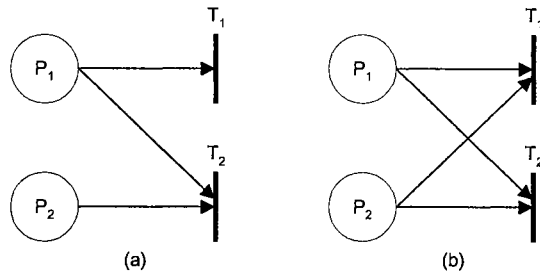


Figure 2.6: Two Examples – (a) Non Free-Choice Net, and (b) Free-Choice Net

in P_2 . Figure 2.6(b), however, describes a free-choice net wherein the choice between T_1 and T_2 is free, since the enabling of either transition requires input tokens only in P_1 and P_2 , and consequently, the choice is equally likely and is unaffected by other elements of the net.

The advantage of requiring that the business process be formalized as a *free-choice* net is that the *soundness* property can be verified in polynomial time [25, 3]. On the downside, restricting the “choice” to free-choice disallows the modeling of all possible business processes. Additionally, current applications of Petri net formalizations require that the business process model be *acyclic*, i.e., without any loops. Both these requirements are easily violated in practical examples, thus reinforcing the need for a generic approach for addressing the control-flow problem without any restrictions on the structure of the control-flow model.

2.5.2 Graph-theoretic Reductions

This is an interesting visual approach to solving the control-flow problem, deriving from the dissertation work of Sadiq [75]; however, Lin *et al.* [61] have established that the algorithm presented in [75] is incomplete, and have proposed extensions to the same. Stated simply, the idea is to remove, from an acyclic control-flow model, all sub-structures of

the graph that are definitely correct, and if possible, reduce the control-flow model to an empty graph. Conversely, if the reduction to an empty graph is not possible, the control flow model is studied further to identify the source of the control-flow anomaly. This has been formalized through five reduction rules – terminal, sequential, adjacent, closed, and overlapped, each of which is applied, in turn, to all vertices of the graph, continuing until no further reductions are possible. These ideas are further illustrated in [76] and form the basis of FLOWMAKE, a modeling and verification tool developed by the DISTRIBUTED SYSTEMS TECHNOLOGY CENTER,⁶ Australia. There are no disadvantages, *per se*, excepting that the approach relies more on the visual nature of the final solution – should the process model be incorrect, then the reduced model (not an empty graph) would have to be “looked-at” to figure out the *where’s* and *why’s* of the control-flow error, since the reduced model loses all resemblance to the original model.

2.5.3 Model-theoretic Event Algebras

The process model constructions developed in [81, 30] specify inter-task dependencies as logical constraints on the occurrence and temporal order of process events. The rigorous grammar underlying these construction techniques necessitates that the process model be correct; however, they lose out significantly on ease of use, and do not guarantee that all models can be specified using the set rules of construction. Additionally, they have not been adopted as the basis for any commercial verification tool. A novel *theory of threads* that attempts to blend the power of process-algebraic reasoning with the modeling ease of Petri nets has been developed by Straub and Hurtado [88] – additional details are still forthcoming.

⁶<http://www.dstc.edu.au/praxis/>

2.5.4 Other Related Network Models

The network-like structure of the control flow model bears similarity to problems commonly addressed in PERT/CPM studies [12], reliability models [80], and network-flow optimization problems [71]. This section briefly summarizes the relevance (or otherwise) of each of these ideas to control flow verification.

PERT/CPM Networks The precedence networks of PERT/CPM cannot model choice, and consequently, cannot be applied to the context of control flow verification.

Reliability Models The primary interest in reliability modeling is in the capture of minimal tie sets, i.e., the path that requires the smallest number of operational links to make the system function, and analogously, minimal cut-sets, i.e., the minimal set of links, the elimination of which will render the system inoperable. Again, as in PERT/CPM, there is no notion of choice or concurrency in reliability networks – parallel links denote redundancy built into the reliability network, and not concurrency as is interpreted in control flow models.

Network-Flow Optimization The correctness of control flow can also be established by representing the control flow model as a 0-1 integer programming formulation (the multi-commodity network flow problem), with nodal flows specified in accordance with their expected behavior (XOR/AND/Task), assigning unit costs to the edges, and solving it to see if a unit flow can be routed from Start to Finish. However, while such a formulation will identify different execution scenarios for a correct control flow model, it does not offer any insight into the source and cause of the control flow error(s), should the IP formulation fail.

2.5.5 Summary

The control-flow problem has been tackled largely by imposing specific restrictions on the underlying representation, namely, that the control-flow model be *acyclic*, or its Petri-net

representation be *free-choice*. Chapter 3 presents an algorithm that is devoid of any such restrictions, and identifies, not just the presence/absence of control-flow anomalies, but also the source of the error(s).

2.6 Resource-Sharing Correctness: Research Review

As identified in Section 1.4.2, there are two main problems to be addressed in verifying the correctness of resource-sharing requirements, namely, the single-instance verification and the multiple-instance verification problem. *There has been no previous research related to studying these problems* – this is perhaps due to the fact that most commercial enterprise automation systems focus primarily on information and people processes and correctness requirements have focused primarily on database consistency checks and transactional guarantees [51, 33, 18, 72]. Material processes are usually addressed by industrial engineers, and the motivation for studying deadlock therein arises largely from considerations of deadlock avoidance in run-time control of flexible manufacturing systems [92].

The design and deployment of business processes begins with *the design* – thus far, there has been some work in developing a formalism for modeling inputs and outputs of a business process, called metagraphs [9, 10, 11], and to use it for understanding connectivity issues. A metagraph is essentially a directed hypergraph, wherein each edge connects a set of *invertices* to a set of *outvertices*. In the context of business processes, an edge in a metagraph would be a task, and its inputs (resp. outputs) would be captured as the edge’s *invertices* (resp. *outvertices*).

Figure 2.7 illustrates a metagraph description of a business process consisting of four tasks $\{e_1, e_2, e_3, e_4\}$, where the ovals represent the inputs and outputs of each task. More specifically, task e_3 requires $\{x_3, x_4, x_5\}$ as its inputs, and produces $\{x_6, x_8\}$ as its outputs.

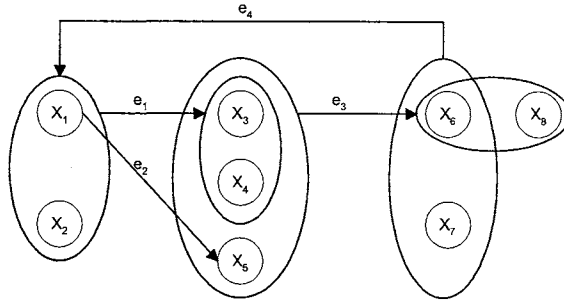


Figure 2.7: Metagraphs - An Example

The general connectivity-related questions addressed using metagraphs are: [11]

1. If a certain input is unavailable, what set of tasks will be affected, and to what extent will the completion of the process be affected?
2. If a certain task is disabled, to what extent will the remainder of the process be affected?

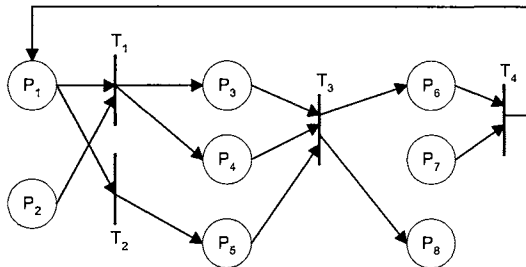


Figure 2.8: Representational Equivalence of Metagraphs & Petri nets - An Example

It is easy to create an equivalent Petri net model for a metagraph, namely, by mapping the vertices and edges in the metagraph to places and transitions in a Petri net. This equivalency is illustrated in Figure 2.8, which presents an equivalent Petri net model for the metagraph of Figure 2.7. However, while a Petri net can model multiplicity in the number of inputs and outputs, the metagraph models only the presence/absence of inputs and outputs, and similarly, while the Petri net specifies the partial/total ordering of the various tasks, the metagraph fails to capture any ordering among the tasks. Moreover,

all similarities between the techniques end here, and each has spawned its own set of distinct analysis techniques. Table 2.1 compares and contrasts the individual strengths and capabilities of both metagraphs and Petri nets.

Table 2.1: A Comparison of Metagraphs and Petri nets

	Metagraphs	Petri nets
Modeling Support		
Control flow	No	Yes
Multiplicity in resource usage	No	Yes
Capturing time-specific information	No	Yes
Support for simulation of process dynamics	No	Yes
Analysis Support		
Foundations	Set-theoretic	Linear-algebraic
Handling cycles	Easy	Hard
Connectivity issues (e.g., critical inputs)	Yes	Not explored

2.6.1 Summary

There has been no previous research related to studying the resource-sharing problem in the design of business processes. More importantly, there are no techniques available to alert the designer about potential deadlock possibilities, save for simulated executions, which include additional overheads in terms of design specification. The techniques presented in Chapter 4 make novel use of the control flow model to alert the designer about potential deadlock possibilities not immediately evident in the process's design.

Chapter 3

The Korreectness Algorithm

Chapter Overview

This chapter presents the KORREECTNESS algorithm, a recursive, backtracking algorithm for verifying the correctness of a control flow model. It does not impose any restriction on the form/structure of the control flow model and its results can also be used to identify the source of control flow error(s), if any. Some interesting results on properties to be expected in random control flow models are also derived.

3.1 Introduction & Background

The basics of the control-flow problem were defined in Sections 1.3. More formally, a control flow model is a directed graph $G = (\mathbf{V}, \mathbf{E})$, where:

- $\mathbf{V} = S \cup \mathbf{T} \cup \mathbf{A}_S \cup \mathbf{A}_J \cup \mathbf{X}_S \cup \mathbf{X}_J \cup F$ is the set of vertices
- $\mathbf{E} = \{(x, y) \mid x, y \in \mathbf{V}; x \neq y\}$ is the set of directed edges leading from x to y
- $\exists! x \in \mathbf{V} \ni \text{indeg}(x) = 0 \Leftrightarrow x$ is the *Start* node, labeled as S
- $\exists! x \in \mathbf{V} \ni \text{outdeg}(x) = 0 \Leftrightarrow x$ is the *Finish* node, labeled as F
- $\mathbf{T} = \{T_i, i = 1, 2, \dots, t\}$ is the set of all *Tasks*

- $\mathbf{A_S} = \{A_i, i = 1, 2, \dots, a_S\}$ is the set of all *AND-Splits*
- $\mathbf{A_J} = \{A_i, i = a_S + 1, a_S + 2, \dots, a_S + a_J\}$ is the set of all *AND-Joins*
- $\mathbf{X_S} = \{X_i, i = 1, 2, \dots, x_S\}$ is the set of all *XOR-Splits*
- $\mathbf{X_J} = \{X_i, i = x_S + 1, x_S + 2, \dots, x_S + x_J\}$ is the set of all *XOR-Joins*

The verification requirements of correct control-flow are [40]:

Unique Termination The process must terminate exactly once.

Proper Termination The process must terminate exactly once, without any residual control flows abandoned elsewhere in the process.

3.2 The Korrektness Algorithm

The KORRECTNESS algorithm is a backtracking algorithm that identifies all valid process execution traces from **Start** to **Finish**. The algorithm may be informally summarized as follows:

- Find all directed acyclic paths from **Start** to **Finish**, ignoring the presence or distinction among the various logical operands and task node – this is readily done with a standard depth-first search.
- Now that all paths from S to F have been found, is it possible to collect or combine them in a manner that represents the execution of the process, while accurately capturing the influence of the various AND (parallelism) and XOR (choice) operands? This is the KORRECTNESS algorithm.
- The paths discovered using the depth-first search would not include cycles in the control flow model. In the event that the model does contain cycles, a few additional rules are applied to eliminate incorrect models before proceeding with the algorithm.

3.2.1 Definitions

A PATH from S to F is an ordered sequence of vertices ($S = v_1, v_2, \dots, v_n = F$) where $v_i \in \mathbf{V}$, and $(v_{i-1}, v_i) \in \mathbf{E}$. A META-PATH is a union of one/more paths, and a VALID META-PATH is a collection of paths, which, taken together, represents one possible process trace, i.e., an instance of the correct execution of the process from *Start* to *Finish*.

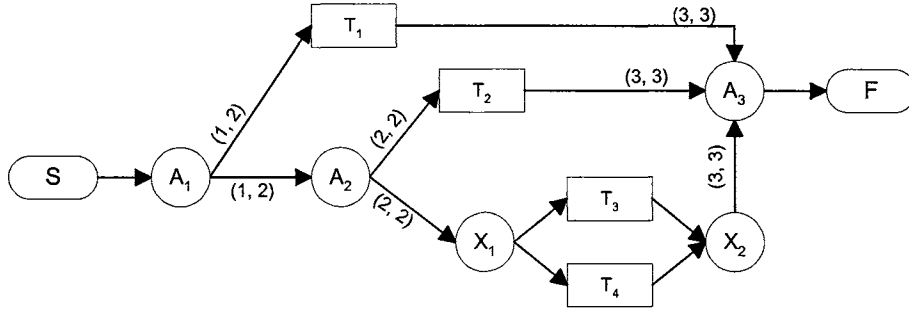


Figure 3.1: An Illustration of Paths and Meta-paths

The set \mathbf{P} of all paths from S to F for the control-flow model in Figure 3.1 is:

- PATH 1 (p_1): (S, A_1, T_1, A_3, F)
- PATH 2 (p_2): $(S, A_1, A_2, T_2, A_3, F)$
- PATH 3 (p_3): $(S, A_1, A_2, X_1, T_3, X_2, A_3, F)$
- PATH 4 (p_4): $(S, A_1, A_2, X_1, T_4, X_2, A_3, F)$

There are two valid meta-paths for the process in Figure 3.1, namely, $\{\text{PATH 1, PATH 2, PATH 3}\}$, and $\{\text{PATH 1, PATH 2, PATH 4}\}$, both representing valid execution traces for the process. Now, how does one go about identifying the various meta-paths in a process model, and how will it help us in verifying if the process model is correct or otherwise? Some more definitions follow.

A COUNTER is a 2-tuple (i, j) that is added as a label by *AND-Split* (or *AND-Join*) A_i to each of its j outgoing (respectively, incoming) edges. Counters are added only to edges originating from (or leading into) AND-Splits (-Joins), and by default, there are no labels on any of the other edges – the counter on edge (x, y) will be identified by the set $c(x, y)$. While identifying the various paths from S to F , the counters that are present in the edges traversed along those paths are also noted. Thus, PATH 3 uses $\{(1, 2), (2, 2), (3, 3)\}$. This is more compactly represented as a 2-tuple (p, C_p) , where p is the name of the path, and C_p is the set of all counters present in path p ; this is stored as the PATH-COUNTER information for the control-flow model of the process. Thus, the path-counter information for the process in Figure 3.1 is $\{(\text{PATH 1}, \{(1, 2), (3, 3)\}), (\text{PATH 2}, \{(1, 2), (2, 2), (3, 3)\}), (\text{PATH 3}, \{(1, 2), (2, 2), (3, 3)\}), (\text{PATH 4}, \{(1, 2), (2, 2), (3, 3)\})\}$.

A few additional concepts are required for identifying the meta-paths. Observe that two paths that share a common XOR-Join (or -Split), namely, by entering (or leaving) through different edges into (or from) a common XOR-Join (or -Split), cannot be in the same meta-path, since it would violate the definition of an XOR operand. To illustrate, PATH 3 and PATH 4, in the example above, cannot occur in the same meta-path, since they share a common XOR-Join, namely, X_2 . However, note that two paths that enter (or leave) a common XOR-Join (or -Split) through the *same edge* do not violate the situation just described, and may likely occur in the same meta-path. More formally, $\forall p_i, p_j \in P, \text{ViolateXOR}(p_i, p_j) = \text{TRUE}$ if both p_i and p_j enter (or leave) a common XOR-operand through different edges.

A few other definitions that would be required are:

- $\forall p \in P$, let $V(p) \subseteq \mathbf{V}$ be the set of vertices covered by path p .
- $\forall x \in \mathbf{V}$, $N^+(x) = \{y \mid (x, y) \in \mathbf{E}\}$ is the set of vertices leading out from x

- $\forall x \in \mathbf{V}, N^-(x) = \{y \mid (y, x) \in \mathbf{E}\}$ is the set of vertices leading into x
- $\forall a \in \mathbf{A}_J, CountersInAND(a) = \bigcup_{x \in N^-(a)} c(x, a)$, i.e., the counters arriving at an AND-Join.
- $\forall a \in \mathbf{A}_S, CountersInAND(a) = \bigcup_{x \in N^+(a)} c(a, x)$, i.e., the counters created at an AND-Split.
- $\forall a \in \mathbf{A}_J, PathsThrough(a) = \{p \in \mathbf{P} \mid a \in V(p)\}$, i.e., the set of all paths through an AND-Join.
- $\forall p \in \mathbf{P}, ANDJoinsInPath(p) = \{a \in \mathbf{A}_J \mid a \in V(p)\}$, i.e., the set of all AND-Joins in a path.

3.2.2 The Control Flow Problem is NP-Complete

Hofstede et al. (1998) use a reduction of the Satisfiability problem to a control flow problem, to prove that the latter is NP-complete. This section presents an alternate derivation that reduces the problem of finding valid meta-paths to that of finding maximal cliques, an approach that offers more insights into the nature of the control flow problem. In addition to the control flow model, three additional graphs, using the set of all paths \mathbf{P} as their vertex set, are constructed.

XOR-Representation Graph $X_G = (\mathbf{P}, E_X)$. The edge set E_X is constructed thus:

$$\forall p_i, p_j \in \mathbf{P}, (p_i, p_j) \in E_X \Leftrightarrow ViolateXOR(p_i, p_j) = TRUE.$$

The edges of this graph identify the paths that share a common XOR-Join or -Split, and thus cannot be in the same meta-path.

AND-Representation Graph $A_G = (\mathbf{P}, E_A)$. The edge set E_A is constructed thus:

$$\forall p_i, p_j, (p_i, p_j) \in E_A \Leftrightarrow ANDJoinsInPath(p_i) \cap ANDJoinsInPath(p_j) \neq \emptyset.$$

This edges of this graph identify the paths that share common AND-Joins, and thus are likely to be included in the same meta-path.

Path-Representation Graph $P_G = (P, E_P)$, where $E_P = E_A \setminus E_X$.

The edges of this graph combine the information contained in A_G and X_G ; the edges identify all pairs of paths that will occur in the same valid meta-path.

The three $\{\cdot\}$ -representation graphs for the control flow model of Figure 3.1 are shown in Figure 3.2.

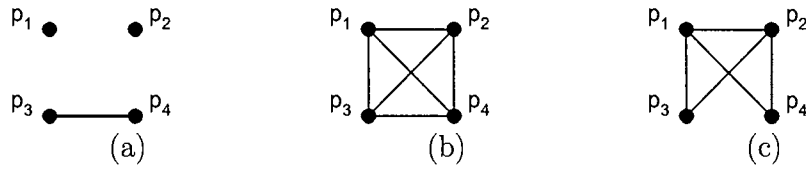


Figure 3.2: (a) XOR-Representation, (b) AND-Representation, and (c) Path-Representation. Graphs

In general, both X_G and A_G will be a collection of disjoint vertices, and other connected components. The XOR-representation graph reveals a few intuitive details about the various meta-paths in a process, namely, that every set of independent vertices (i.e., a set of vertices, no two of which are connected by an edge) in X_G is a potential meta-path, and thus, the number of valid meta-paths cannot exceed the number of independent sets in X_G . Additionally, each complete sub-graph in A_G is a potential valid meta-path, given that all paths in that sub-graph share common AND-Joins, and consequently, common counters. Now, the Path-representation graph essentially combines the information contained in both A_G and X_G , to create a graph consisting of several components, from which all valid meta-paths can be extracted, using the following theorem.

Theorem 3.1 *The control flow model is correct if and only if the vertices in every maximal clique in P_G represent a valid meta-path.*

Proof (\Rightarrow , by contradiction) Suppose $C \subset P$ is a maximal clique in P_G , and the paths in C do not represent a valid meta-path. Now, are all the paths in C required for a valid meta-path? Yes, since they all share common AND-Joins. Since the control flow model is correct, there exists at least one more path $p \notin C$ that is required to complete the valid meta-path. Since p is required, it must merge with a path $p_\alpha \in C$ at some AND-Join, say A_α . Since C is a complete subgraph, any other path $p_\beta \in C$ must also share a common AND-Join, say, A_β , with p_α . Now, since A_α and A_β lie on p_α , there are two possibilities, i.e., there is either (i) a path from A_β to A_α , or (ii) a path from A_α to A_β . In case (i), the path from A_β to A_α lies on both p_β and p_α , which implies that p is incident with p_β as well, since p_β joins p at A_α . In case (ii), the path from A_α to A_β lies on both p and p_α , which implies that p joins p_β as well at A_β . Since this holds for any path p_β , this can be extended to show that p is incident with all other vertices in C , thereby contradicting the maximality of C .

(\Leftarrow , by contrapositive example) Consider the first incorrect model shown in Table 1.2. The Path-representation graph is just the empty graph with two isolated vertices, the vertices corresponding to the two paths from S to F . Each isolated vertex in P_G is a maximal subgraph; however, each vertex by itself is not a valid meta-path, thereby completing the proof. ■

Let \mathbf{M} be the set of all valid meta-paths for the control flow model represented with P_G . An immediate corollary of Theorem 3.1 is:

Corollary 3.1 *The control flow model is correct $\Leftrightarrow \forall p \in \mathbf{P}, \exists m \in \mathbf{M} \ni p \in m$.*

The \Rightarrow part of the proof tells us that for a correct control flow model, every maximal clique in P_G is a valid meta-path, identifying all of which is NP-complete [97]. It also suggests our first bound, namely, that the size of the largest clique in P_G is the size of the largest valid meta-path – can this bound be improved? Suppose, in the extremal case, there is a path p which traverses through all the AND-Joins; clearly, in a correct model this path would require the maximum number of paths to be included along with it, to create a valid meta-path. The maximum number of paths thus required will be equal to the sum of the in-degrees of all the AND-Joins in p , which would be at most $\Delta_{A_J} * a_J$, where Δ_{A_J} is the maximum in-degree of any of the a_J AND-Joins. Thus, the size of the largest valid meta-path in a control flow model (and the size of the largest clique in P_G) is at most $\Delta_{A_J} * a_J$.

The \Leftarrow part of the proof presents some clues on how to identify if the model is incorrect or otherwise, namely, that if a maximal clique is an invalid meta-path, then the control flow model is incorrect. In any case, establishing that the control flow model is correct is equivalent to finding all the maximal cliques in P_G , thereby making it NP-complete – so why study this problem at all?

Let us analyze the steps that would be needed for solving the control flow problem.

Finding all the Paths

The first step is to find the set \mathbf{P} of all directed acyclic paths from S to F – this is readily done with a standard depth-first search routine (Algorithm 1); the special case of handling cycles in the control flow model is discussed in Section 3.2.5. The time taken for finding a single path is $O(|\mathbf{E}|)$, and as we will see in Section 3.2.6, many of our results will depend on $|\mathbf{P}|$, the number of S – F paths. However, there is no closed-form expression for estimating $|\mathbf{P}|$ as a function of the degree distribution $(\Delta_{\mathbf{X}_S}, \Delta_{\mathbf{A}_S})$ and other

parameters, namely, x_s, x_J, a_S, a_J , of the control flow model. Moreover, in the worst-case, the number of S - F paths can be exponential, as is for the extremal example of Figure 3.3.

Algorithm 1: Path Enumeration

Input: The current-node in the search tree, the path traversed thus far, and the goal.

Output: The set \mathbf{P} of all paths from \mathbf{S} to \mathbf{F} .

PATHENUM(*currnode*, *currpath*, *goal*)

- (1) if *currnode* = *goal*
- (2) $\mathbf{PATHS} \leftarrow \mathbf{PATHS} \cup \{\mathit{currpath}\}$
- (3) else
- (4) foreach $x \in N^+(\mathit{currnode})$
- (5) if $x \notin \mathit{currpath}$
- (6) PATHENUM($x, \mathit{currpath} \cup \{x\}, \mathit{goal}$)

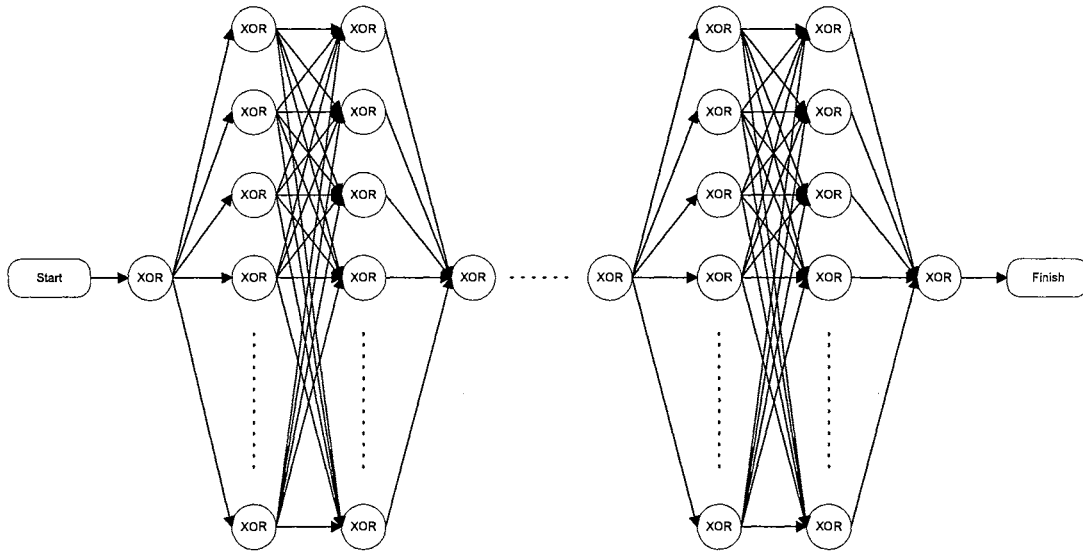


Figure 3.3: A Control Flow Model with the Maximum Number of S - F Paths

Construction of the Meta-Paths

Now that all paths have been found, the maximum number of potentially valid meta-paths would be the power set $\mathcal{P}(\mathbf{P})$ with size $2^{|\mathbf{P}|}$ – all of these possibilities would have to be checked to identify which one is, and which one is not a valid meta-path, which is exponential in the worst case. However, the problem is not entirely without hope.

Observe that if V_1 and V_2 are two valid meta-paths, is it possible that $V_1 \subset V_2$? – No, since that would contradict the maximality of the clique that V_1 represents in P_G . *Thus, there are certain subtleties unique to the character of the control flow problem that can, and have been exploited, in developing MPSEARCH, a recursive backtracking approach for identifying all the valid meta-paths.*

3.2.3 Construction of Meta-Paths

The MPSEARCH procedure is a recursive, backtracking algorithm that identifies all valid meta-paths from a depth-first search tree of all meta-paths. To better motivate the discussion that follows, consider a few questions – what is a valid meta-path, or more simply, why isn't a path sufficient to represent the process's execution? Suppose there is a path p with a non-NULL counter set C_p . The presence of counters in p implies that there is at least one AND-Split (or -Join) occurring in p , which indicates that there are one/more parallel threads created (or merged) along with p in the process's execution. The collective representation of p , along with all of its parallel threads, represents a trace of the sequence of tasks that an instance of the process might progress through – this collective representation is what is defined as a valid meta-path. The idea behind the creation of meta-paths originated from the realization that in a correct process execution, all parallel threads created with an AND-Split are subsequently merged at an AND-Join. More intuitively, the process begins with no counters from S and ends with no counters upon reaching F – this is the principle that has been formalized as the KORRECTNESS algorithm, an informal description of which is presented next.

In the example of Figure 3.1, PATH 1, i.e., $p_1 = (S, A_1, T_1, A_3, F)$, is not a valid meta-path in itself, since it carries the counters $\{(1, 2), (3, 3)\}$, with $(1, 2)$ implying that it is one of the two threads that originate at A_1 , and $(3, 3)$ implying that it is one of the three threads that merge at A_3 . Consequently, some more paths have to be included

along with p_1 to account for the missing threads; let $M = \{p_1\}$ be the current meta-path, and $C_M = \{(1, 2), (3, 3)\}$ be its counter-set. Any other path that is included with p_1 must necessarily pass through some AND-Join present in p_1 . To this end, if there were no AND-Joins in p_1 , and yet if it carries some counter, say (i, j) , then this would be an invalid meta-path, since the sub-thread created at AND-Split A_i could never be merged with any other thread due to the absence of AND-Joins along the path. Clearly, this intuitive approach presents the germ of an idea for diagnostic checking to help identify the source of control flow errors, if any – this is further elaborated in Section 3.2.7.

Is it permissible to include PATH 3 in M ? Yes, since $ViolateXOR(p_1, p_3) = FALSE$, and there is an AND-Join present in p_1 . Is it worthwhile adding p_3 to the current meta-path? Yes, since it carries counters that are present in the counter-set of our meta-path, and thus, the meta-path is now $M = \{p_1, p_3\}$. What is C_M , the current counter-set of M ? Note that when a new path p_i is added to a meta-path M , care must be taken to include only those counters that occur on edges not already traversed by one of the paths included in M , to avoid double-counting of any counter(s) – this is the `GETUSEFULCNTRS` procedure. By this rule, if p_3 is included in M , all of its counters can be appended to C_M , since all of them occur on edges not already covered by the existing path in M , namely, p_1 . Thus, the meta-path is now $M = \{p_1, p_3\}$ and $C_M = \{(1, 2), (1, 2), (2, 2), (3, 3), (3, 3)\}$.

Observe that there are two occurrences of the counter $(1, 2)$ in C_M – this confirms that both the threads that originated at A_1 have been accounted for, and can be removed from all future consideration. Moreover, the `GETUSEFULCNTRS` procedure ensures that these counters will not be included again. Thus, if the number of occurrences of a counter (i, j) in C_M is equal to j , they are removed from the counter-set – this is the `REMOVE-MATCHEDCOUNTERS` procedure. By this rule, both occurrences of $(1, 2)$ in C_M are

removed, whereby, $C_M = \{(2, 2), (3, 3), (3, 3)\}$. The meta-path is not valid yet – it still remains to account for (2, 2) and (3, 3). Consider the most recently added path, namely, p_3 – any other path(s) to be included in the meta-path must also necessarily pass through any AND-Joins in p_3 ; to this end, the reasoning used for including p_3 can be repeated recursively, i.e., by looking at the AND-Joins in the last added path (p_3). There is one AND-Join, namely, A_3 , in p_3 , and $PathsThrough(A_3) = \{p_1, p_2, p_3, p_4\}$. Since p_1 and p_3 are already present in M , the only paths that merit consideration for inclusion in M are p_2 and p_4 .

As before, consider the question – is it permissible to include p_4 in M ? No, since $ViolateXOR(p_3, p_4) = TRUE$, or alternatively, $(p_3, p_4) \in E_X$. More precisely, only paths that do not share a common XOR-Join or -Split, along different edges, with one of the paths already present in the meta-path can be chosen for inclusion. We can however, include p_2 in our meta-path, whereby $M = \{p_1, p_2, p_3\}$. Although $C_{p_2} = \{(1, 2), (2, 2), (3, 3)\}$, the (1, 2) counter cannot be included since it occurs on an edge $A_1 \rightarrow A_2$ that has already been covered by p_3 – this is handled by the `GETUSEFULCNTRS` procedure. Now, $C_M = \{(2, 2), (2, 2), (3, 3), (3, 3), (3, 3)\}$, and by an application of the `REMOVE-MATCHEDCOUNTERS` procedure, it reduces to $C_M = \emptyset$. Thus, one valid meta-path for the control flow model of Figure 3.1 has been identified, namely, by accounting for all sub-threads that were created along the paths from S to F . The intuition described thus far has been formalized into a backtracking procedure, `MPSEARCH` (Algorithm 2), that can be used to generate all the meta-paths that require a particular path. To illustrate, the set of all valid meta-paths that include `PATH 1`, i.e., p_1 , for the control-flow model of Figure 3.1, is generated by `MPSEARCH($p_1, \{p_1\}, C_{p_1}$)`.

Line 1 begins the `MPSEARCH` procedure by removing all matched counters, following which, if the set $Temp$ is empty, it follows that *currmetapath* is a valid meta-path, and

Algorithm 2: Meta-Path Enumeration

Input: The last-added path, the current meta-path, and the current counter-set.

Output: The set of all valid meta-paths that can be generated from the current meta-path.

```

MPSEARCH(lastaddedpath, currmetapath, currcounterset)
(1)   Temp ← REMOVEMATCHEDCOUNTERS(currcounterset)
(2)   if Temp =  $\emptyset$ 
(3)     # we have a valid meta-path – adjoin it to the set of valid meta-paths
(4)     ValidMetaPaths ← ValidMetaPaths  $\cup$  {currmetapath}
(5)     # include the paths in currmetapath in the set of paths covered
(6)     PathsCovered ← PathsCovered  $\cup$  currmetapath
(7)     # mark the last added path
(8)     PathsMarked ← PathsMarked  $\cup$  {lastaddedpath}
(9)   else
(10)    CurrentMetaPathInvalid ← TRUE
(11)    foreach a  $\in$  ANDJoinsInPath(lastaddedpath)
(12)      if Temp  $\cap$  CountersInAND(a)  $\neq$   $\emptyset$ 
(13)        foreach p  $\in$  PathsThrough(a)
(14)          if p  $\notin$  currmetapath and p  $\notin$  PathsMarked
(15)            if NOCOMMONXORJOINORSPLIT(p, currmetapath)
(16)              Temp ← Temp  $\cup$  GETUSEFULCNTRS(p, currmetapath)
(17)              CurrentMetaPathInvalid ← FALSE
(18)              MPSEARCH(p, currmetapath  $\cup$  {p}, Temp)
(19)    if CurrentMetaPathInvalid
(20)      # The current meta-path is an invalid meta-path
(21)      InValMPath ← InValMPath  $\cup$  {{currmetapath}, {currcounterset}}
```

is included with the set of valid meta-paths (line 4). Upon discovering a valid meta-path, the *lastaddedpath* is also *marked* (line 8), as a way of remembering that all valid meta-paths that include it will be discovered in the current call to MPSEARCH, i.e., by exploring all branches leading out from *lastaddedpath*. Additionally, the paths in the set *PathsMarked* serve us well in that, should a *marked* path be encountered in exploring any other branch of the search tree, the search can be terminated, since any/all valid meta-paths in that route would have already been discovered. The procedure NOCOMMONXORJOINORSPLIT(*p*, *currmetapath*) checks to see that *ViolateXOR*(\cdot, \cdot) is *FALSE* for *p* and every other path in *currmetapath*. Additionally, if the search fails to progress along the search tree, *currmetapath* is included in the set of invalid meta-paths (line 21), to be used later for diagnostic checking (Section 3.2.7). The remainder of the procedure is self-explanatory, and is identical to the discussion presented earlier.

3.2.4 The Komplete Korreктness Algorithm

The KORREKTNESS algorithm (Algorithm 3) is a backtracking algorithm for generating all valid meta-paths in the process model. It involves two steps – (i) generate all directed paths from **Start** to **Finish**, and (ii) scan them one by one, calling in turn, the MPSEARCH routine, while taking care not to scan paths that have already been included in other meta-paths. After all the valid meta-paths have been identified, it only remains to check that all paths in **P** appear in at least one valid meta-path to ensure that the control flow model is correct (Corollary 3.1).

Algorithm 3: The Korreктness Algorithm

Input: The control-flow model $G = (\mathbf{V}, \mathbf{E})$

Output: The set *ValidMetaPaths* of all valid meta-paths

KORREKTNESS(G)

- (1) # generate the set **P** of all Paths from S to F
- (2) $\text{PATHENUM}(S, \{S\}, F)$
- (3) # generate the set of all valid meta-paths
- (4) $\text{PathsCovered} \leftarrow \emptyset$
- (5) **foreach** $p \in \text{Paths}$
- (6) **if** $p \notin \text{PathsCovered}$
- (7) $\text{PathsMarked} \leftarrow \emptyset$
- (8) $\text{MPSEARCH}(p, \{p\}, C_p)$

3.2.5 Resolving Cycles in the Control Flow Model

The set **P** includes only directed acyclic paths from S to F ; what if the control flow model was not acyclic? A simple test to check if the model is cyclic or otherwise is to check if $\mathbf{V} \setminus \bigcup_{p \in \mathbf{P}} V(p) = \emptyset$, i.e., do the paths in **P** cover all the vertices of the set **V**? This gives rise to two cases.

Case 1: $\mathbf{V} \setminus \bigcup_{p \in \mathbf{P}} V(p) = \emptyset$

The fact that the paths in **P** include all vertices in **V** does not exclude the presence of cycles – four such possibilities are presented in Table 3.1.

Table 3.1: Control Flow Sub-models with Empty Cycles

Sub-models	Discussion
	(a) This sub-model is correct.
	(b) This sub-model is incorrect, since it violates the requirement of unique termination.
	(c and d) These sub-models are incorrect, since they model concurrency in feedback which makes it impossible for control flow to proceed.

In sub-model (a) of Table 3.1, the XOR-pairs model recursion in the flow of control, which must eventually move ahead from the second XOR – to this end, it is immaterial if control flow is looped more than once within the model, and such a model can be analyzed with the KORRECTNESS algorithm. However, should iteration be misused as in sub-models (b)-(d) of Table 3.1, the algorithm will correctly identify the error, since the counters generated at the AND-Joins/Splits will never be accounted for by any path in \mathbf{P} . Consequently, the KORRECTNESS algorithm can be used without any further checks for the case $\mathbf{V} \setminus \bigcup_{p \in \mathbf{P}} V(p) = \emptyset$.

Case 2: $\mathbf{V} \setminus \bigcup_{p \in \mathbf{P}} V(p) \neq \emptyset$

Let $V^* = \mathbf{V} \setminus \bigcup_{p \in \mathbf{P}} V(p)$. The following two rules can be used to immediately identify if a control flow model with cycles is incorrect.

Rule 1 $\exists a \in \mathbf{A}_S \ni N^+(a) \cap V^* \neq \emptyset \Rightarrow$ The control flow model is incorrect. The fifth example in Table 1.2 illustrates this rule, which essentially means that should an AND-Split lead control flow away into a cycle that does not reach toward F , then

the model is incorrect, since the counter created by the AND-Split will never be accounted for, in any meta-path.

Rule 2 $\exists a \in \mathbf{A}_J \ni N^-(a) \cap V^* \neq \emptyset \Rightarrow$ The control flow model is incorrect. The situation tackled by this rule is similar to that created in sub-models (c) and (d) of Table 3.1, and essentially implies that should an AND-Join receive control flow from an element that is not part of a directed path from S to F , it becomes a situation wherein control flow will not proceed any further.

Now, there are four possibilities, as regards the content of V^* , namely – (i) $V^* \subseteq \mathbf{T}$, i.e., the elements not covered by \mathbf{P} are all tasks with just sequential flow of control among them, (ii) $V^* \subseteq (\mathbf{T} \cup \mathbf{X}_S \cup \mathbf{X}_J)$, i.e., the elements not covered include just choice and/or sequential flow of control among them, (iii) $V^* \cap \mathbf{A}_J \neq \emptyset$, and/or (iv) $V^* \cap \mathbf{A}_S \neq \emptyset$. Both (i) and (ii) can be easily eliminated by applying rules 1 and 2; it is (iii) and (iv) that is more interesting.

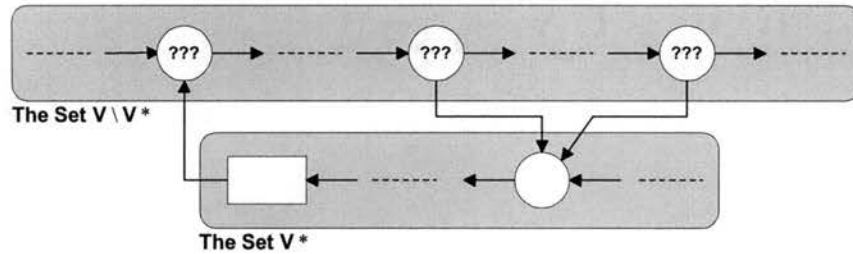


Figure 3.4: Connectivity between Vertices in V^* and $V \setminus V^*$

By definition, the control flow model is a single connected component, and so, the elements in V^* must be connected to the vertices in $V \setminus V^*$ at either XOR/AND logical operands (labeled in Figure 3.4 with ???). If they were connected at an AND-Join/Split, then rules 1 and 2 will eliminate the model as incorrect. Consequently, the only cases that need to be considered are those in which the elements in V^* are connected to the main model via XOR operands – this will be referred to as the *XOR Rule*.

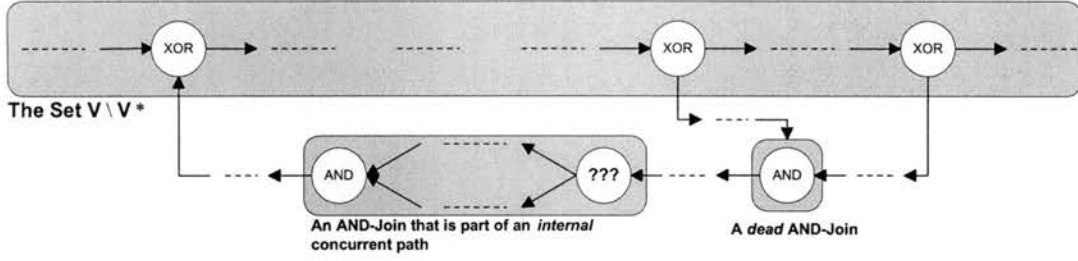


Figure 3.5: Control Flow Sub-model Possibilities for the Case $V^* \cap \mathbf{A}_J \neq \emptyset$

Consider cases (iii) and (iv) presented above. If $V^* \cap \mathbf{A}_J \neq \emptyset$, then there are only two possible ways that an AND-Join may occur in V^* . Figure 3.5 illustrates these two possibilities, namely, an AND-Join is *dead* and will never get activated, or is part of an *internal concurrent path*. If the AND-Join is *dead*, it doesn't affect the KORRECTNESS algorithm, but it still remains to identify that it is *dead*; if it is part of an internal concurrent path, then $V^* \cap \mathbf{A}_S \neq \emptyset$, or else the model would be incorrect, which leads us to (iv) above. Note that the operand labeled ??? in Figure 3.5 may either be an XOR (in which case it would be incorrect), or an AND (which would be (iv) again).

Let $P_{\{x,y\}}$ be the set of all directed, acyclic paths from vertex x to vertex y , and let $V(P_{\{x,y\}})$ be the set of vertices covered by these paths. If $V^* \cap \mathbf{A}_S \neq \emptyset$, then the following rules can be used to (i) identify dead AND-Joins, and (ii) eliminate incorrect models.

Rule 3 $\exists x \in \mathbf{A}_J \cap V^*, \{y, z\} \in \mathbf{X}_S \setminus V^* \ni (|P_{\{y,x\}}| \geq 1) \wedge (|P_{\{z,x\}}| \geq 1) \wedge (|P_{\{y,x\}}| \neq |P_{\{z,x\}}|) \Rightarrow$ the AND-Join x is *dead*. This essentially means that if there are two XOR-Splits $\{y, z\} \in \mathbf{V} \setminus V^*$ which have *different* paths leading into a common AND-Join $x \in V^*$, then the AND-Join is dead, as is illustrated in Figure 3.5.

Consider case (iv): Let $x \in \mathbf{A}_S \cap V^*$ and $y \in \mathbf{X}_J \setminus V^*$. If $|P_{\{x,y\}}| = 0$, then the XOR-Join does not lie on a direct path from x , and can be ignored. If $|P_{\{x,y\}}| = 1$, then the control flow model is incorrect, since there are at least two threads created at AND-Split x , and

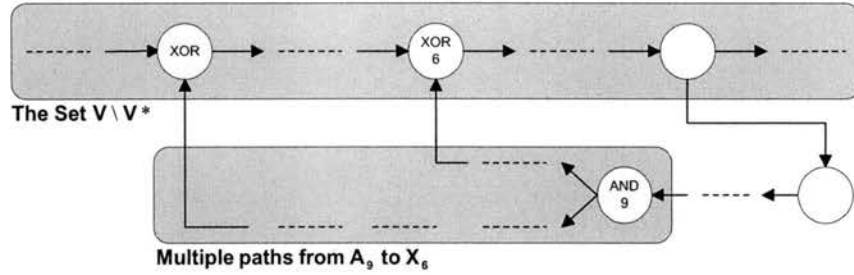


Figure 3.6: Illustration of Multiple Paths from an AND-Split to an XOR-Join

they must all be merged before they reach the main threads in $V \setminus V^*$, or else they will never be merged (by the *XOR Rule*). If the multiple threads created at x were merged at say, node z , then there would be (at least) one path from z to y and two/more paths from x to z , each of which can be combined to create two/more $x - z - y$ paths, thereby contradicting the correctness of a model with $|P_{\{x,y\}}| = 1$. If $|P_{\{x,y\}}| > 1$, then these multiple paths may occur as a result of internal concurrent paths, as is illustrated in Figure 3.5, or multiple distinct paths as is illustrated in Figure 3.6, the latter being incorrect – this can be easily solved by applying the KORRECTNESS algorithm to the sub-model beginning with AND-Split x and ending with XOR-Join y , and using $P_{\{x,y\}}$ as the set of paths from which the valid meta-paths will be constructed. If no valid meta-paths are found, as would be the case for the example in Figure 3.6, then the model is correctly identified as incorrect.

Rules 1 and 2 can both be evaluated in linear time, and Rule 3 can be evaluated in $O(|E|)$, namely, the run-time complexity for generating $P_{\{x,y\}}$. The last rule (case (iv)), however, requires the same run-time complexity as that of the backtracking KORRECTNESS algorithm, which can be exponential in the worst-case.

3.2.6 Complexity Analysis

The KORRECTNESS algorithm identifies all valid (and invalid) meta-paths by selectively enumerating and exploring combinations of the paths from S to F . In the worst case, the algorithm could end up scanning all $2^{|P|}$ possible combinations of paths to identify even one valid meta-path – this could explode further if the number of valid meta-paths is again large. Note, however, that it is not entirely correct to say that all $2^{|P|}$ possible combinations of paths will be explored, without taking into account the structure of the control flow model, namely, the number of AND-Joins, the distribution of the in-degrees of the AND-Joins, the number of AND- and XOR-Splits, etc. – these issues are summarized in the following questions:

- What is the expected number of valid meta-paths in any control flow model?
- What is the expected time it will take to identify a single valid meta-path?

It is clear that the KORRECTNESS algorithm is exponential in the worst case. However, what about on average?

Establishing the Average Number of Valid Meta-Paths

Suppose a control flow model with 20 nodes is randomly drawn – it is possible to estimate the average number of valid meta-paths in such a model? What if the model has 20,000 nodes? It turns out that, *on average, the number of valid meta-paths to be expected in any control flow model is at most 9*. Also, if the number of paths from S to F is say, 50, then, the expected number of valid meta-paths in such a control flow model is only about 4.6.

Let P_n be the maximum number of paths possible between S and F in a control flow model with n vertices, and call this set $P_{max} = \{p_1, p_2, \dots, p_{P_n}\}$. The set of all possible paths that can be discovered for a given control flow model is the power set $\mathcal{P}(P_{max})$ with

size 2^{P_n} . Let $A_v(k)$ be the average number of valid meta-paths that can be discovered among k paths. Consequently, V_n , the average number of valid meta-paths in a control flow model with n vertices, averaged over all possible combinations of paths, is:

$$V_n = \frac{\sum_{k=0}^{P_n} \binom{P_n}{k} A_v(k)}{\sum_{k=0}^{P_n} \binom{P_n}{k}} = 2^{-P_n} \sum_{k=0}^{P_n} \binom{P_n}{k} A_v(k) \quad (3.1)$$

Equation 3.1 essentially sums over the number of ways of choosing k paths from a maximum of P_n , multiplied by $A_v(k)$, the average number of valid meta-paths among k paths ($k = 0, 1, \dots, P_n$), and we have used the identity $\sum_{k=0}^{P_n} \binom{P_n}{k} = 2^{P_n}$. Consider $A_v(k)$ – it was noted earlier in Section 3.2.1 that the maximum number of valid meta-paths cannot exceed the the number of independent sets in X_G , the XOR-representation graph. Consequently, the average number of valid meta-paths among k paths cannot exceed the average number of independent sets in the XOR-representation graph with k vertices. The exact formula for I_k , the average number of independent sets in a graph of order k is ([97], §5.6)

$$I_k = \sum_{r=0}^k \binom{k}{r} 2^{-r(r-1)/2} \quad (3.2)$$

substituting for which, in equation 3.1, and using the relation $A_v(k) \leq I_k$, we get

$$\begin{aligned} V_n &\leq 2^{-P_n} \sum_{k=0}^{P_n} \binom{P_n}{k} \left\{ \sum_{r=0}^k \binom{k}{r} 2^{-r(r-1)/2} \right\} \\ &\leq 2^{-P_n} \sum_{r=0}^{P_n} \left\{ \sum_{k=r}^{P_n} \binom{P_n}{k} \binom{k}{r} 2^{-r(r-1)/2} \right\} \\ &\leq 2^{-P_n} \sum_{r=0}^{P_n} \frac{2^{-r(r-1)/2} (P_n)^r}{r!} \left\{ \sum_{k=r}^{P_n} \binom{P_n-r}{k-r} \right\} \\ &\leq \sum_{r=0}^{P_n} \frac{(P_n)^r}{r!} 2^{-r(r+1)/2} \end{aligned} \quad (3.3)$$

where the last inequality has been obtained by just manipulating the binomial coefficients in the inner sum using the identities $\binom{P_n}{k} \binom{k}{r} = \frac{P_n!}{(P_n-k)!(k-r)!r!} \leq \frac{(P_n)^r}{r!} \binom{P_n-r}{k-r}$, and $\sum_{k=r}^{P_n} \binom{P_n-r}{k-r} = 2^{P_n-r}$.

Now, the series represented in equation 3.3 is a *unimodal* sequence, i.e., the terms t_r increase up to a certain value r_m and then decrease thereon, whereby it follows that the sum of the sequence is at most $(P_n + 1)$ times the largest element in the sequence. To show that the sequence is unimodal, consider the ratio of two successive terms, t_k/t_{k-1} , which evaluates to $P_n/(k2^k)$. It is immediately obvious that for $k \geq \log_2(P_n)$, the ratio is less than 1, assuming $P_n \geq 2$. To better estimate r_m , it remains to solve for k in $k2^k = P_n$, which can be rewritten as

$$(k \cdot \ln(2)) \cdot e^{k \cdot \ln 2} = P_n \cdot \ln(2) \quad (3.4)$$

which, incidentally, is in the standard form for the *Lambert's W* function. The Lambert's W function [14, 20] is evaluated as the value of $W(x)$ that satisfies the equation $W(x)e^{W(x)} = x$, and is usually approximated as:¹

$$W(x) \approx \begin{cases} 0.665 \cdot (1 + 0.0195 \cdot \ln(x + 1)) \cdot \ln(x + 1) + 0.04 & : 0 \leq x \leq 500 \\ \ln(x - 4) - (1 - \frac{1}{\ln(x)}) \cdot \ln(\ln(x)) & : x > 500 \end{cases}$$

Additionally, symbolic processing packages like Maple readily compute the values of the Lambert's function using in-built recurrence relations [13, 20]. Comparing with the standard form of the Lambert's W function, it follows that the value of $k \cdot \ln(2)$ that satisfies equation 3.4 is $W(P_n \cdot \ln(2))$. Thus, the value at which the sequence in equation 3.3 attains its maximum is $r_m = W(P_n \cdot \ln(2))/\ln(2)$. Since the sum of the series in equation 3.3 cannot exceed $(P_n + 1)$ times the value at t_{r_m} , it follows that

$$V_n \leq (P_n + 1) \cdot \frac{(P_n)^{r_m}}{r_m!} 2^{-(r_m(r_m+1)/2)} \quad (3.5)$$

The final result is obtained by using Sterling's approximation [97], i.e., $n! \geq (e/n)^n$, in equation 3.5 to prove that, on average, the number of valid meta-paths in a control flow model on n vertices is

$$V_n \leq \frac{(P_n + 1) \cdot e^{r_m}}{2^{(r_m(r_m+1)/2)}} \quad (3.6)$$

¹<http://www.desy.de/~t00fri/qcdins/texhtml/lambertw/>

Now, equation 3.6 is interesting in its own right. A plot of the value of the estimated upper bound for V_n for $P_n = \{1, 2, \dots, 300\}$ is shown in Figure 3.7. Figure 3.7 offers some interesting insights, namely, that in any control flow model, drawn at random, the average number of valid meta-paths *does not exceed 9*, which shall be referred to as the *threshold of valid meta-paths*, and moreover, as the number of paths from S to F increases, the value of V_n converges to zero!!

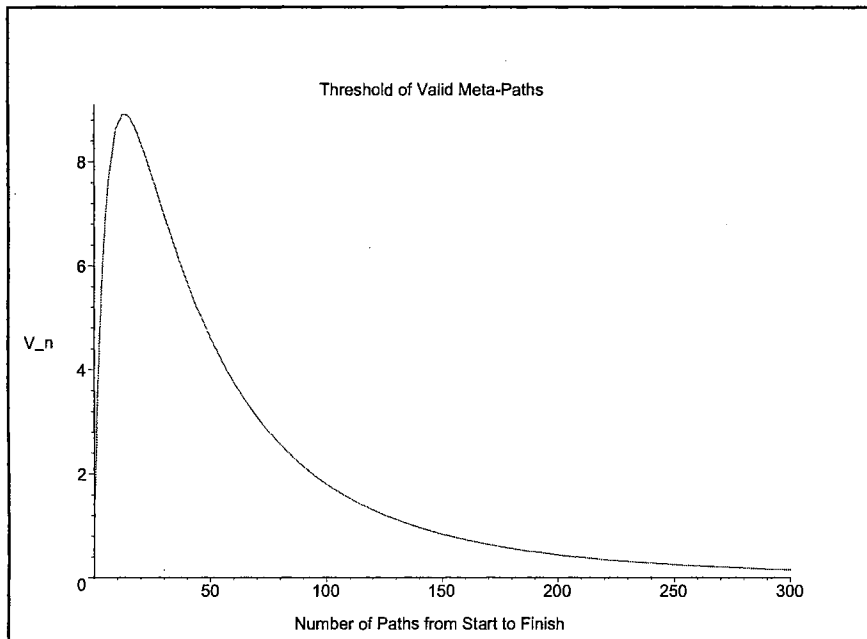


Figure 3.7: Expected Number of Valid Meta-Paths in a Random Control Flow Model

Consider equation 3.6 – it reveals that, irrespective of the order of the control flow model n , the expected number of valid meta-paths is bounded, and is a function only of the number of paths from S to F . Suppose there are 237 paths from S to F . Equation 3.6 and Figure 3.7 confirm our intuition that in a model of such complexity, it is very likely that there are only a limited number of valid meta-paths. Now, what about the run-time complexity? More specifically, what is the expected time it takes to identify one valid meta-path?

Estimating the Run-time for the Korrektness Algorithm

The main sets of notation that would be required are briefly summarized – \mathbf{P} is the set of all S - F paths found in a control flow model on $|\mathbf{V}|$ vertices, with $|\mathbf{E}|$ edges, x_S XOR-Splits, x_J XOR-Joins, a_S AND-Splits, and a_J AND-Joins. The maximum and minimum in-degree (resp., out-degree) of the XOR/AND-Joins (resp., -Splits) will be denoted by Δ_* and δ_* , where $\{*\}$ represents the logical operand under consideration, and $\Delta_A = \max(\Delta_{\mathbf{A}_S}, \Delta_{\mathbf{A}_J})$. Note that the maximum size of a valid meta-path is $\Delta_{A_J} * a_J$ (refer Section 3.2.2). It is also assumed that the three representation graphs X_G , A_G , and P_G have been constructed – this would require a run time of $\Theta(|\mathbf{P}|^2)$, to identify all the $\binom{|\mathbf{P}|}{2}$ possible edges in the graphs. Consequently, checking conditions like $\text{NOCOMMONXORJOINORSPLIT}(p_i, p_j)$ is just equivalent to checking if the corresponding edge is present (or absent) in X_G . A step-by-step analysis of Algorithm 2 is presented next.

Line 1: $Temp \leftarrow \text{REMOVEMATCHEDCOUNTERS}(currcounterset)$

Since counters are created only on edges incident with AND-Joins and -Splits, the maximum number of counters possible is $\Delta_{\mathbf{A}_S} * a_S + \Delta_{\mathbf{A}_J} * a_J = O(\Delta_A)$. Thus the maximum time required for this step of the algorithm is $O(\Delta_A)$ – run through $currcounterset$ once tracking the occurrence count of each counter, and run through it again, removing the matched counters.

Line 11: **foreach** $a \in \text{ANDJoinsInPath}(lastaddedpath)$

Let us condition on the number of AND-Joins in the $lastaddedpath$, which corresponds to the number of times this loop will be executed. The probability that the $lastaddedpath$ contains any AND-Join is the probability that it is one of the at least $\delta_{\mathbf{A}_J}$ paths through the AND-Join, which is $\delta_{\mathbf{A}_J}/|\mathbf{P}|$. Thus, the probability that the $lastaddedpath$ contains k AND-Joins is $\geq \binom{a_J}{k} \left(\frac{k\delta_{\mathbf{A}_J}}{|\mathbf{P}|} \right)$.

Line 12: if $Temp \cap CountersInAND(a) \neq \emptyset$

Suppose the number of paths in $currmetapath$ is l . Consider any path in $currmetapath$. The probability that it contains AND-Join a is the probability that it is one of the at least δ_{A_J} paths through a . Since there are l paths in $currmetapath$, the probability that at least one of them includes AND-Join a is $\geq \frac{l * \delta_{A_J}}{|P|}$.

Line 13: foreach $p \in PathsThrough(a)$

This loop will be executed at least δ_{A_J} times.

Line 14: if $p \notin currmetapath$ and $p \notin PathsMarked$

This probability shall be approximated by q_l , where l is the number of paths in $currmetapath$.

Line 15: if NOCOMMONXORJOINORSPLIT($p, currmetapath$)

The probability that p and some path $p^* \in currmetapath$ share a common XOR-Join or -Split is at most $x_S / \binom{\delta_{X_S}}{2} + x_J / \binom{\delta_{X_J}}{2}$, where $1 / \binom{\delta_{X_J}}{2}$ is the maximum probability that p and p^* share two of the at least δ_{X_J} paths through a particular XOR-Join, and there are x_J choices for the XOR-Join (the argument is similar for XOR-Splits). Since there are l choices for p^* , the probability that p does not share a common XOR-Join or -Split with any one of the l paths in $currmetapath$ is $\geq l * \left(1 - \frac{x_S}{\binom{\delta_{X_S}}{2}} - \frac{x_J}{\binom{\delta_{X_J}}{2}} \right)$.

Line 16: $Temp \leftarrow Temp \cup GETUSEFULCNTRS(p, currmetapath)$.

This step would require a run time of $O(|E|)$ for checking the counters in each path, and consequently a total of $l * O(|E|)$ for all paths in $currmetapath$, which is again $O(|E|)$.

The calculations for the running time of the MPSEARCH procedure can now be summarized. Let $f(l)$ be the running time required by the procedure when the size of *currmetapath* is l . Based on the observations above, the running time can be written as

$$f(l) \geq O(\Delta_A) + \sum_{k=0}^{a_J} \binom{a_J}{k} \frac{k \delta_{A_J}}{|\mathbf{P}|} \cdot \frac{l \delta_{A_J}}{|\mathbf{P}|} \delta_{A_J} q_l l \left(1 - \frac{x_S}{\binom{\delta_{x_S}}{2}} - \frac{x_J}{\binom{\delta_{x_J}}{2}} \right) [O(|E|) + f(l+1)] \quad (3.7)$$

Equation 3.7 is quite unwieldy. To simplify it further – in a worst case scenario – let us suppose that the probability that all the **if** conditions are satisfied is 1. In such a case, equation 3.7 can be rewritten into a more comforting first-order recurrence as follows.

$$\left(a_J \cdot 2^{a_J-1} \cdot \frac{\delta_{a_J}^2}{|\mathbf{P}|} \right) f(l+1) \leq f(l) - O(\Delta_A) \quad (3.8)$$

where we have used the identity $\sum_{k=0}^n k \binom{n}{k} = n * 2^{n-1}$. The solution for the first-order recurrence of equation 3.8 is $f(l) = O(c + \epsilon)^l$, where $c = \frac{|\mathbf{P}|}{a_J \cdot 2^{a_J-1} \cdot \delta_{a_J}^2}$, for every $\epsilon > 0$ ([97], Theorem 1.4.1).

Now, $f(l)$ is the time spent in evaluating all branches in the search tree at a level where the number of paths in *currmetapath* is l . In the worst case, the MPSEARCH procedure may end up exploring all branches at all levels, namely, for $l = \{1, 2, \dots, |\mathbf{P}| - 1\}$, whereby, the absolute worst case running time for finding even one single valid meta-path is estimated as $\sum_l f(l) = O(c + \epsilon)^{|\mathbf{P}|}$.

Clearly, this is the worst-case scenario for the time it takes to complete one run through the MPSEARCH procedure, and correspondingly, to identify one valid meta-path. Since the expected number of meta-paths is bounded (Section 3.2.6), the expected running time for finding all valid meta-paths is also $O(c + \epsilon)^{|\mathbf{P}|}$. Thus, the total running time for the KORRECTNESS algorithm is $O(|E|) + \Theta(|\mathbf{P}|^2) + O(c + \epsilon)^{|\mathbf{P}|}$.

Scope for Improvement

There are two main avenues for improving both the design and the analysis of the KORRECTNESS algorithm, namely:

- Is it possible to design heuristics to “speed-up” the search process? Will it improve the speed of the MPSEARCH procedure if it is begun with a path that has the maximum number of AND-Joins? Or, is it possible to pre-process and sort the set of paths in P taking into account the specific structure of the control flow model, namely, the in- and out-degree distribution of the logical operands? Also, none of the steps in Algorithm 2 make any choices based on the size of *currmetapath* – can this information be used to influence the search time?
- The worst case analysis presented in Section 3.2.6 is quite crude, and does not exploit attributes specific to the structure of the control flow model which will definitely impact the traversal of the search tree. There is much scope for improving the running-time bound for $f(l)$. More specifically, is it possible to estimate the probability q_i that was assumed for line 14 of Algorithm 2? Is it possible to derive a better estimate of the expected running time of the algorithm to show that, perhaps, it is *usually fast*?

3.2.7 Diagnostic Checking of the Control Flow Model

This section will discuss the use of the KORRECTNESS algorithm for identifying the source of control flow errors in a business process model. Consider the incorrect process model of Figure 3.8.

The results of the KORRECTNESS algorithm (implemented in PYTHON) are as follows:

Paths There are 7 paths from Start to Finish. The paths from Start to Finish are:

```
'path2': ['S', 'A1', 'X1', 'T1', 'X6', 'A2', 'X7', 'F'],  
'path3': ['S', 'A1', 'X1', 'T2', 'X5', 'A2', 'X7', 'F'],  
'path1': ['S', 'A1', 'X2', 'T4', 'X7', 'F'],  
'path6': ['S', 'A1', 'X2', 'T3', 'X4', 'X5', 'A2', 'X7', 'F'],
```

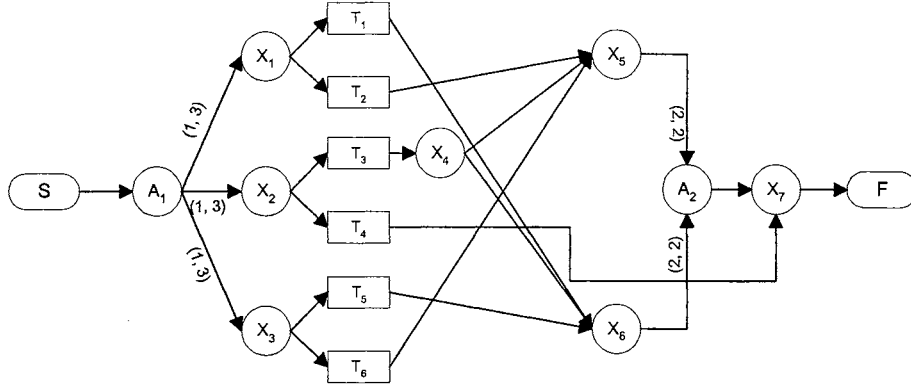


Figure 3.8: Incorrect Process Model - An Example

'path7': ['S', 'A1', 'X2', 'T3', 'X4', 'X6', 'A2', 'X7', 'F'],
 'path4': ['S', 'A1', 'X3', 'T5', 'X6', 'A2', 'X7', 'F'],
 'path5': ['S', 'A1', 'X3', 'T6', 'X5', 'A2', 'X7', 'F']

Counter-Sets The counters in all the paths, including the edge that the counter occurs on, are:

'path2': [[('A1', 'X1'), (1, 3)], [('X6', 'A2'), (2, 2)]],
 'path3': [[('A1', 'X1'), (1, 3)], [('X5', 'A2'), (2, 2)]],
 'path1': [[('A1', 'X2'), (1, 3)]],
 'path6': [[('A1', 'X2'), (1, 3)], [('X5', 'A2'), (2, 2)]],
 'path7': [[('A1', 'X2'), (1, 3)], [('X6', 'A2'), (2, 2)]],
 'path4': [[('A1', 'X3'), (1, 3)], [('X6', 'A2'), (2, 2)]],
 'path5': [[('A1', 'X3'), (1, 3)], [('X5', 'A2'), (2, 2)]]

Predictably, the set of valid meta-paths is empty. The list *InValMPath* of all invalid meta-paths is:

Invalid Meta-paths The invalid meta-path results, i.e., (set of paths, counter-set) are:

- 1: [['path2', 'path6'], [(1, 3), (1, 3)]],
- 2: [['path3', 'path7'], [(1, 3), (1, 3)]],
- 3: [['path1'], [(1, 3)]],
- 4: [['path4', 'path3'], [(1, 3), (1, 3)]],
- 5: [['path5', 'path2'], [(1, 3), (1, 3)]]

None of the paths occur in any valid meta-path – let us investigate further.

- **PATH 1** does not cover any AND-Joins, and so, the (1, 3) counter that it carries can never be removed.
- All the other meta-paths are short of just one more (1, 3) counter – now, why did this occur? Observe that each meta-path includes both the XOR-Joins, namely, X_5 and X_6 . Since each XOR-Join requires only one flow through its incoming arcs, other paths through these XOR-Joins cannot be included. The only other path is **PATH 1**, which, however, cannot be added since it doesn't cover any AND-Joins.

Thus, a wealth of information can be extracted simply by examining the final meta-path results and by tracking the counters that remain unaccounted for, in each incomplete meta-path, to provide precise feedback about the source of the control flow error(s).

3.3 Summary

This chapter presented the **KORRECTNESS** algorithm, a graph-theoretic approach to addressing correctness issues in control-flow models, without any restriction on the form or structure of the business process. The algorithm has been implemented in **MAPS**, a computerised environment for **Modeling and Analysis of Process modelS**, the details of which are presented in Chapter 5.

Additionally, it has been shown that, on average, the number of valid meta-paths in a random control-flow model does not exceed 9. From an implementation standpoint, this is very satisfying, since it would imply that the enumerative approach of the **KORRECTNESS** algorithm does not fall prey to the exponential growth in problem complexity, but instead, is bounded in its average run-time complexity.

Chapter 4

The Resource-Sharing Problem

Chapter Overview

This chapter outlines a collection of novel techniques that exploit the control flow model to gain insight into the structure and behavior of a process. Several simple rules are derived to help the designer compute minimal resource requirements to maximize parallelism within the process, and also to identify design errors that could lead to deadlocks either within a single-instance, or across multiple-instances of a process.

4.1 Introduction & Background

The basics of the resource-sharing problem were introduced in Section 1.4.1 – the resource requirements for a business process may be specified as follows:

- $\mathbf{R} = \{R_1, R_2, \dots, R_r\}$ is the set of all resources. $\forall R_i \in \mathbf{R}$, $R_i^\#$ = number of units available for resource R_i .
- $\forall R_i \in \mathbf{R}$, $R_i^{Cap} : \mathbf{T} \rightarrow \mathbb{N} = \{0, 1, \dots\}$ is a functional that specifies the number of units of resource R_i *captured* by each task, where \mathbf{T} is the set of all tasks.
- $\forall R_i \in \mathbf{R}$, $R_i^{Rel} : \mathbf{T} \rightarrow \mathbb{N}$ is a functional that specifies the number of units of resource R_i *released* by each task.

The verification requirements of correct resource-sharing are:

Single-Instance Verification is to determine if the sharing of common resources among tasks within an instance of a process could lead to deadlock.

Multiple-Instance Verification is to determine if the sharing of common resources among various instances of the process could lead to deadlock.

The principal challenge associated with both problems above is to identify potential circular-wait (CW) conditions that could arise in the process – these are determined primarily by the logic of the process, i.e., the control flow model. It is assumed that the control flow model is correct; consequently, the focus of this chapter is to further study the control flow model, coupled with the additional information contained in R_i^{Cap} , R_i^{Rel} , etc., to identify potential deadlock situations, as were illustrated in Table 1.3.

4.2 The Control Flow Model Revisited

The control flow model was formalized in Chapter 3 – to summarize, it is a directed graph representation of tasks and logical operands (AND, XOR), which taken together, represent the logic and ordering of the process. The main sets of notation that would be required are \mathbf{V} , \mathbf{T} , \mathbf{P} , and \mathbf{M} , representing respectively, the sets of vertices, tasks, the $S - F$ paths, and the set of all valid meta-paths identified by the KORRECTNESS algorithm. Additionally, $\forall m \in \mathbf{M}, V(m) \subseteq \mathbf{V}$ is the set of vertices covered by all the paths included in the meta-path m – the reader is referred to Section 3.1 for notation not covered here.

Consider the control flow model of Figure 3.1, repeated in Figure 4.1. Observe that the control flow model lends itself naturally to being partitioned into sets of concurrent elements (tasks and logical operands), as illustrated in Figure 4.1.

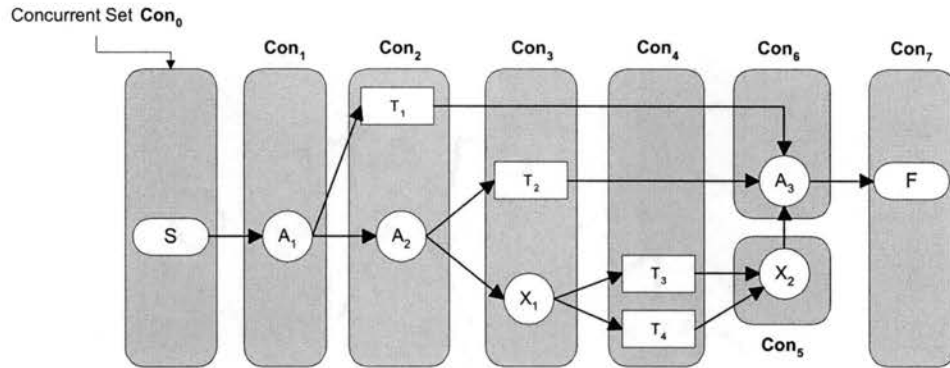


Figure 4.1: Partitioning the Control Flow Model – An Example

Figure 4.1 highlights many interesting ideas – the control flow model has been partitioned into seven disjoint sets of concurrent elements. The concurrent sets are interpreted thus: $Con_2 = \{T_1, A_2\}$ implies that both T_1 and A_2 can be activated simultaneously within a single instance of the process. Additionally, the index 2 in Con_2 suggests the order in which its elements are arrived at, beginning at $Con_0 = \{S\}$. The reader would no doubt wonder, among other questions, why partition the control flow model?

The objective of partitioning the control flow model into sets of concurrent tasks is to explicitly order the tasks, namely, by assigning a functional value $f : \mathbf{T} \rightarrow \mathbb{N}$ such that for any two tasks t_i, t_j , $f(t_i) < (=) f(t_j)$ implies that task t_i precedes (is concurrently enabled with) task t_j in the execution of a process. Such an ordering would aid in immediate investigation of potential circular-wait conditions that could arise between tasks that have the same $f(\cdot)$ value – this is elaborated in Section 4.3. Once the set $\mathbf{Con} = \{Con_i\}$ of concurrent elements have been identified, assigning the functional $f(\cdot)$ is trivial, namely, $\forall t \in Con_i \cap \mathbf{T}, f(t) = i$.

The partitioning of the control flow model into the concurrent sets $\{Con_i\}$ is based on a very simple idea, namely, for an element x to belong to Con_i , all of its incoming elements, i.e., $N^-(x)$ must belong in $\bigcup_{k<i} Con_k$. This is formalized in Algorithm 4.

Algorithm 4: The Partition Algorithm

Input: The control-flow model $G = (\mathbf{V}, E)$

Output: The set $\{Con_i\}$ of all concurrent elements, the function $f : \mathbf{T} \rightarrow \mathbb{N}$

PARTITION(G)

- (1) # Partition the control flow model into concurrent sets
- (2) $Con_0 \leftarrow \{S\}$ # Initialize the algorithm
- (3) $i \leftarrow 1; Temp \leftarrow \emptyset$
- (4) # Repeat until the Finish node is reached
- (5) **while** $Con_{i-1} \neq \{F\}$
- (6) $Con_i \leftarrow \emptyset$ # Initialize the current set
- (7) # Create a set $Temp$ of all elements covered thus far
- (8) $Temp \leftarrow Temp \cup Con_{i-1}$
- (9) # For each element x in the last concurrent set
- (10) **foreach** $x \in Con_{i-1}$
- (11) # For each new vertex y leading out from x
- (12) **foreach** $y \in N^+(x) \setminus Temp$
- (13) # If all of y 's input vertices are present in $Temp$, include in Con_i
- (14) **if** $N^-(y) \subseteq Temp$
- (15) $Con_i \leftarrow Con_i \cup \{y\}$
- (16) # Assign the functional $f : \mathbf{T} \rightarrow \mathbb{N}$
- (17) **foreach** $t \in Con_i \cap \mathbf{T}$
- (18) $f(t) \leftarrow i$
- (19) $i \leftarrow i + 1$ # Increment the set index

The PARTITION procedure (Algorithm 4) is quite self-explanatory. Lines 2 and 3 initialize the algorithm by setting $Con_0 = \{S\}$. The loop in line 5 repeats until the Finish node is reached, and begins by creating a new empty set, i.e., $Con_i = \emptyset$ (line 7). Line 8 appends the previously discovered concurrent set, Con_{i-1} , to the set $Temp$ of all elements covered thus far. Subsequently, for each vertex x in the most recent concurrent set Con_{i-1} (loop in line 10), it remains to check that, for each new (i.e., not included in other concurrent sets) output vertex y of x (loop in line 12), that all of y 's input vertices are present in one of the previously discovered concurrent sets (line 14), for it to merit inclusion in Con_i (line 15). Lines 17 and 18 assign the functional $f : \mathbf{T} \rightarrow \mathbb{N}$ by assigning the index i to all tasks in Con_i , i.e., $\forall t \in Con_i \cap \mathbf{T}, f(t) = i$.

The run-time complexity of the PARTITION procedure is $O(|\mathbf{V}|^2)$ – lines 5 and 10 are executed at most $|\mathbf{V}|$ times, and line 12 is executed at most $\max\{\Delta_{A_S}, \Delta_{X_S}\}$ times. The partitioning of the control flow model was inspired by a similar concept presented in [89].

The $\{Con_i\}$ partitions need to be understood as purely mathematical conveniences obtained from the control flow graph G , minus the actual interpretation of the elements contained within. To see why, consider the set of concurrent elements in Figure 4.1; $Con_4 = \{T_3, T_4\}$ is especially interesting, since it implies that both T_3 and T_4 can be enabled simultaneously, which is impossible since they are activated by a common XOR-Split, X_1 ; this would also be clarified by examining the set of valid meta-paths \mathbf{M} – observe that both T_3 and T_4 do not occur together in any of the valid meta-paths, i.e., $\nexists m \in \mathbf{M} \ni \{T_3, T_4\} \subseteq V(m)$. This is clarified thus: for each valid meta-path $m \in \mathbf{M}$, define $Con_i^m = Con_i \cap V(m)$ to be the concurrent set relevant to m , and $\mathbf{Con}_{|V(m)} = \{Con_i^m\}$ to be the set of concurrent sets with vertices restricted to $V(m)$.

However, note that tasks that have the same $f(\cdot)$ values and occur in the same valid meta-path may not be truly *concurrent* – they may need to be executed in some partial sequence, as constrained by the availability of resources. To illustrate, suppose $R_\alpha^\# = 1$, $f(T_i) = f(T_j)$, and both require resource R_α ; clearly, both T_i and T_j cannot occur concurrently (assuming that they are present in the same valid meta-path) – one must precede the other. But, for each valid meta-path m , $\{Con_i^m\}$ reveals the sets of *truly concurrent elements* that the process designer envisioned for the process; consequently, for the benefits of concurrency to be fully realized, it only remains to ensure that for each valid meta-path m , the number of available resources is adequate to ensure full parallelism for all tasks in any concurrent set Con_i^m . This, and other considerations, are more thoroughly explored in the sections that follow.

4.3 The Resource-Sharing Problem: Some Ideas

The design of a business process minimally requires the specification of the process's logic, and the specifics of resource requirements by the constituent tasks. Additional details like input-output requirements, infrastructure support, activity durations, etc., while necessary for completing the process's description, are not essential to verifying the correctness of a process's design. As illustrated in Table 1.3, the design problems that may arise from resource-allocation can be categorized into one of:

Conservation of Resources To check that the number of times a resource is captured is equal to the number of times it is released. This is enforced by verifying that

$$\forall R_i \in \mathbf{R}, \forall m \in \mathbf{M}$$

$$\sum_{t \in V(m) \cap \mathbf{T}} R_i^{Cap}(t) = \sum_{t \in V(m) \cap \mathbf{T}} R_i^{Rel}(t)$$

This rule essentially checks to see that in each valid meta-path, the number of times each resource is captured is equal to the number of times it is released, i.e., a simple check for conservation of resources.

Release-before-Capture Improperly specified process definitions wherein resource units are released before they are captured. This is enforced by verifying that

$$\forall R_i \in \mathbf{R}, \forall m \in \mathbf{M}, \forall t_\alpha \in V(m) \cap \mathbf{T} \ni R_i^{Rel}(t_\alpha) \neq 0,$$

$$R_i^{Cap}(t_\alpha) + \sum_{\forall t \in V(m) \cap \mathbf{T}: f(t) < f(t_\alpha)} \left[R_i^{Cap}(t) - R_i^{Rel}(t) \right] \geq R_i^{Rel}(t_\alpha)$$

This rule essentially checks to see that in each valid meta-path, the number of units of a particular resource that are being "held" by the process is greater than or equal to the number of units that are being released by a particular task.

Inadequate Capacity The specified resource capacity $\{R_i^\#\}$ is inadequate for satisfying the resource requirements of a single instance of the process.

Circular-Wait The most important and the least evident problem of all, wherein two or more tasks capture a set of resources and end up waiting indefinitely for resources held by one another.

In the context of resource-allocation, there are two main problems that the process designer needs to be alerted to, namely, the obvious and the not-so-obvious. The obvious design errors include “conservation of resources” and “release-before-capture,” both of which have been addressed above. The not-so-obvious problems include “inadequate capacity” and “circular-wait.” Clearly, the circular-waits are the most severe design errors that could escape the attention of a process designer – the intuition derived from the partitions $\{Con_i\}$ will be used extensively in identifying potential circular-waits, especially across multiple instances of the process.

4.4 Single-Instance Verification

With regard to the execution of a single instance of a process, the circular-waits (CWs) are further classified into two categories, namely (i) CWs within a concurrent set, and (ii) CWs across concurrent sets. Both (i) and (ii) are illustrated, based on examples drawn from Table 1.3, in Figure 4.2 – the reader is referred to Table 1.3 for a detailed discussion on both examples.

Identifying CWs within a concurrent set is relatively straightforward, and is discussed in Section 4.4.1. Circular-Waits across concurrent sets are also (easily?) identified by creating an equivalent Petri net representation of the control flow model, and exploring its reachability tree to verify that it is deadlock free, and is discussed in Section 4.4.2.

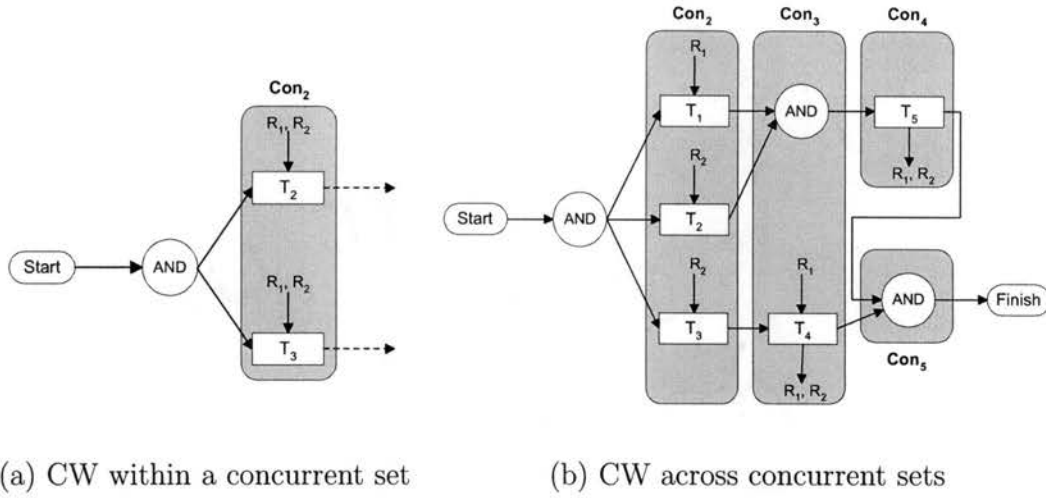


Figure 4.2: Two different classes of Circular-Wait (CW) problems

4.4.1 Identifying Circular-Waits Within Concurrent Sets

It would be useful to begin by summarizing the main question that is being addressed in this section – *are there any potential circular-waits among a set of concurrent activities?* Consider any concurrent set Con_i^m , for some valid meta-path m – if $|Con_i^m \cap \mathbf{T}| \leq 1$, there is no concern about circular-waits within Con_i^m . It is only in the case when there are two/more concurrent tasks that it remains to verify that circular-wait conditions are not existent. A few additional definitions that would be required before continuing the discussion are:

- $\forall t \in \mathbf{T}, t^{Cap} = \{R_i \in \mathbf{R} \mid R_i^{Cap}(t) \neq 0\}$ is the set of resources *captured* by task t .
- $\forall t \in \mathbf{T}, t^{Rel} = \{R_i \in \mathbf{R} \mid R_i^{Rel}(t) \neq 0\}$ is the set of resources *released* by task t .

The general rules presented next will be useful in detecting potential circular-waits within a concurrent set. The special case of when resource capture/release by a task is limited to one unit each is presented first, and is subsequently generalized to allow for multiple units of resource capture/release by a task.

Special Case: $R_i^{\{Cap, Rel\}} : \mathbf{T} \rightarrow \{0, 1\}$ and $R_i^\# \leq 1$. In the special case when the release/capture of resources by a task is limited to one unit each, and unit resource availabilities, the following situation would lead to a circular-wait. If there is a meta-path with two concurrent tasks, say t_i and t_j , each requiring two or more common resources, then a deadlock would arise if both t_i and t_j capture some resources and end up waiting for one another to relinquish their captured resources (refer example 1 in Table 1.3).

More formally, $\forall m \in \mathbf{M}, \forall Con_i^m \in \mathbf{Con}_{|V(m)} \ni |Con_i^m \cap \mathbf{T}| > 1$

$$\exists t_i, t_j \in Con_i^m \cap \mathbf{T} \ni |t_i^{Cap} \cap t_j^{Cap}| \geq 2 \Rightarrow \text{Circular - wait (CW)}$$

General Case: $R_i^{\{Cap, Rel\}} : \mathbf{T} \rightarrow \mathbf{N}$ and $R_i^\# \geq 0$. In the general case when multiple units of a resource may be released/captured by a task, a circular-wait will occur if the combined request for any two common resources R_p and R_q by two concurrently enabled tasks exceeds the resource capacities $R_p^\#$ and $R_q^\#$.

Figure 4.3 illustrates an example, with $R_1^\# = R_2^\# = 2$. A circular-wait will occur if both T_1 and T_2 capture one unit each of R_1 and R_2 – more specifically, the combined request for both R_1 and R_2 by the two concurrently enabled tasks T_1 and T_2 is equal to three, and exceeds the resource capacity of two.

More formally, $\forall m \in \mathbf{M}, \forall Con_i^m \in \mathbf{Con}_{|V(m)} \ni |Con_i^m \cap \mathbf{T}| > 1$

$$\begin{aligned} &\exists R_p, R_q \in \mathbf{R}, t_i, t_j \in Con_i^m \cap \mathbf{T} \ni [\{(R_p, R_q)\} \subseteq t_i^{Cap}] \wedge [\{(R_p, R_q)\} \subseteq t_j^{Cap}] \wedge \\ &[t_i^{Cap}(R_p) + t_j^{Cap}(R_p) \geq R_p^\#] \wedge [t_i^{Cap}(R_q) + t_j^{Cap}(R_q) \geq R_q^\#] \Rightarrow \text{Circular - Wait} \end{aligned}$$

The rules presented above are just elementary count-based checks for identifying potential circular-waits within a concurrent set. Moreover, as will become evident in Section 4.5.2, these checks become redundant with the calculation of minimal resource requirements that guarantee the successful deadlock-free execution of a process, which, by definition, renders void the existence of any circular-wait conditions.

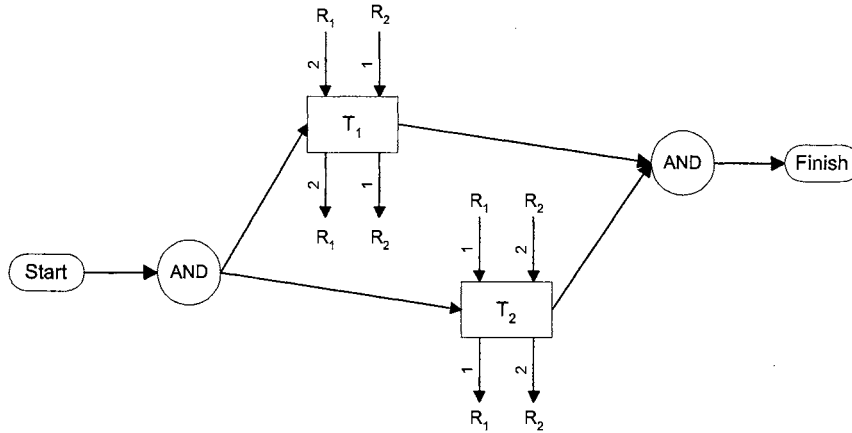


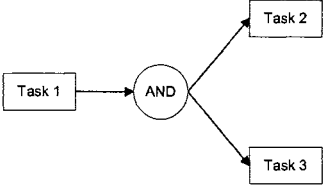
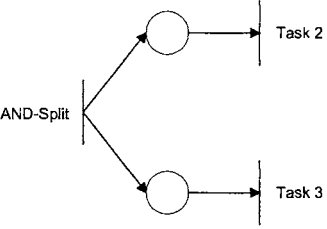
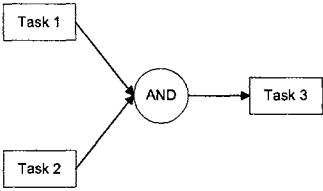
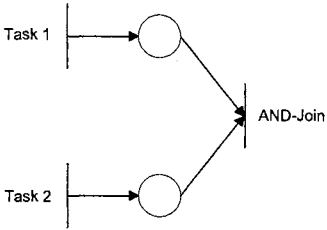
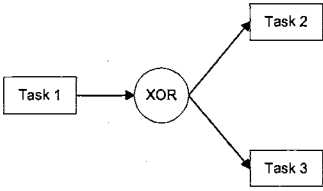
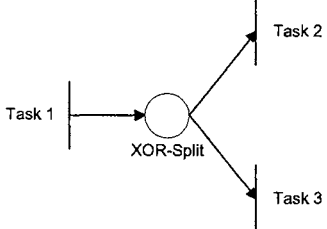
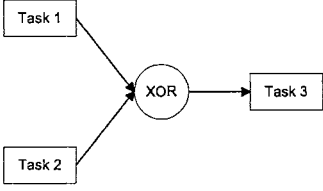
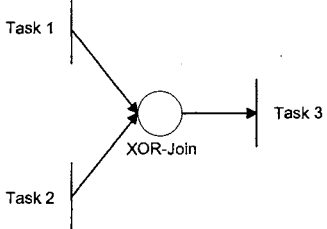
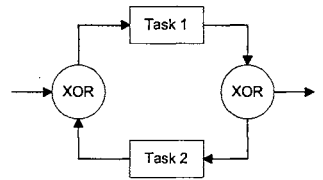
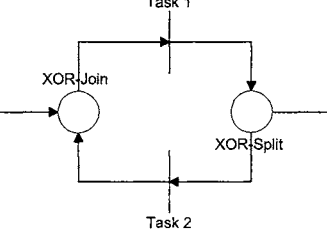
Figure 4.3: Circular-Wait Within A Concurrent Set – Another Example

4.4.2 Identifying Circular-Waits Across Concurrent Sets

The main question addressed in this section is – *is it possible to detect potential circular-waits that may occur among activities that are not necessarily simultaneously enabled?* This question is readily answered by creating a Petri net representation of each valid meta-path of the control flow model, including the resource requirements, and studying its reachability tree to identify potential deadlock possibilities. Table 4.1 illustrates the Petri net mappings used to translate basic control flow elements into net constructions. It is assumed that the reader is familiar with the basics of Petri nets – a brief primer is presented in Appendix A.

The equivalent Petri net construction for the example in Figure 4.2(b) is presented in Figure 4.4 – note that there is only one valid meta-path for this process. The tokens in the net are interpreted thus – (i) a single token in the place “Start” corresponds to the execution of a single instance, and (ii) the tokens in the places corresponding to the resources signify the number of units of the resource that are available, i.e., $R_i^\#$.

Table 4.1: Petri Net Mappings of Basic Routing Constructs

Logical Operand	Graphical Construct	Petri Net Mapping
<p>AND-Split: A point within the process model where a single thread of control splits into two or more threads to be executed simultaneously.</p>		
<p>AND-Join: A point within the process model where two or more different threads of control merge asynchronously.</p>		
<p>XOR-Split: A point within the process model where the thread of control selectively chooses one of several possible paths.</p>		
<p>XOR-Join: A point within the process model where the thread of control from one of several different paths converges.</p>		
<p>Iteration/Feedback Routing: A section within the process model that may require the repetitive execution of one or more activities until certain conditions are satisfied.</p>		

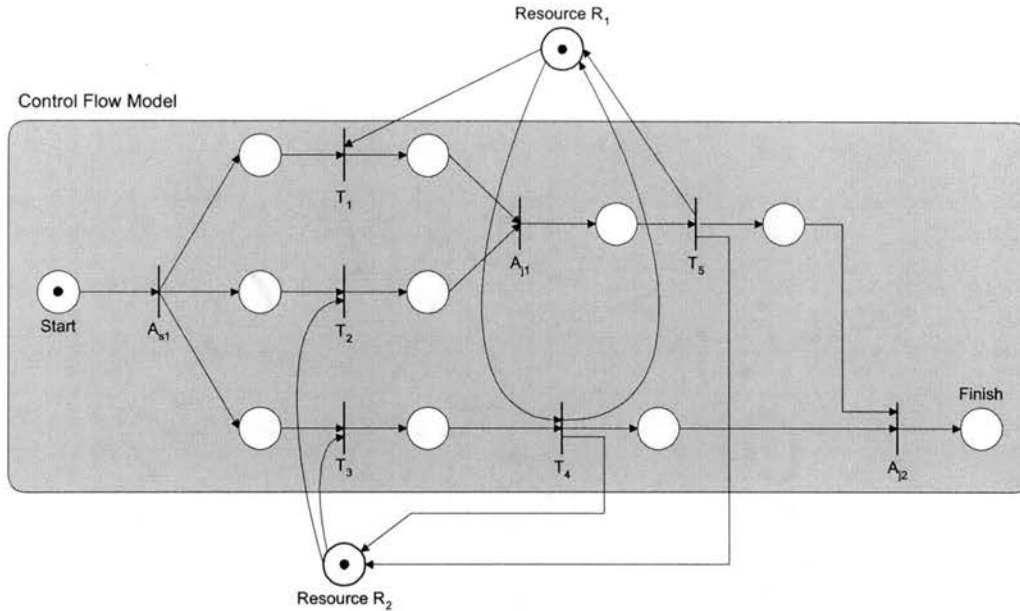


Figure 4.4: Petri Net Representation of the Process Model – An Example

Consider the Petri net illustrated in Figure 4.4 – clearly, the transition sequence A_{s1}, T_1, T_3 leads to deadlock, and will be immediately evident upon studying the net’s reachability tree. This approach can be applied to verifying the correctness of resource-sharing for any process – a total of $|\mathbf{M}|$ (the number of valid meta-paths) different Petri nets will need to be constructed and their reachability trees examined to confirm the presence/absence of deadlocks. Should deadlocks occur, the transition sequence that led to the same can be studied to identify and isolate the reasons for deadlock. Additionally, the Petri nets thus constructed will all be k -bounded, where $k = \max_i(R_i^\#)$, i.e., the maximum number of tokens in the net corresponds to the resource with the highest $R_i^\#$. However, generating the reachability tree is computationally expensive [68, 26] – *is it really necessary to generate the complete reachability tree to even establish that the process’s design is correct?* More specifically, is there a simpler way to just check the process’s design and to answer “YES – the resource-sharing in this process is OK, and will not lead to deadlock,” thereby avoiding the expense of detailed reachability enumeration? That this is so is the impetus for the approaches derived next.

4.5 The Static-Design Net Representation

The purpose of this section is to develop an approach that will identify if the process's resource-sharing requirements will or will not lead to deadlock – should the answer be in the negative, the analysis can be continued with a detailed reachability enumeration of the Petri net construction as presented in Section 4.4.2. It would be useful to begin by summarizing what has been achieved thus far, namely:

1. Given a correct control flow model, partition it into sets of concurrent elements $\mathbf{Con} = \{Con_i\}$, and consequently order the tasks with the functional $f : \mathbf{T} \rightarrow \mathbb{N}$.
2. Confirm that there are no potential circular-waits within each concurrent set in $\mathbf{Con}_{|V(m)}$, for all valid meta-paths $m \in \mathbf{M}$.

Thus far, the control flow model and the resource-allocation requirements have been considered in separate formalisms; Section 4.5.1 presents a Petri-net construction that captures both the ordering suggested by the control flow model and the details of resource allocations.

4.5.1 The Static-Design Net

Consider the following Petri net construction for some valid meta-path m of the process. The concurrent sets $\{Con_i^m\}$ are represented as transitions, and the resources $\{R_i\}$ are represented as places; the places and transitions are connected by weighted arcs, the weights indicating the effective number of units of a particular resource that are captured (or released) by the tasks in the concurrent set, according to as whether the arc is directed from a place to a transition, or vice-versa. Figure 4.5 illustrates this Petri net construction for the example in Figure 4.2(b) – note that $R_1^\# = R_2^\# = 1$ and since there is only one valid meta-path in the process, the superscript m is omitted from Con_i^m .

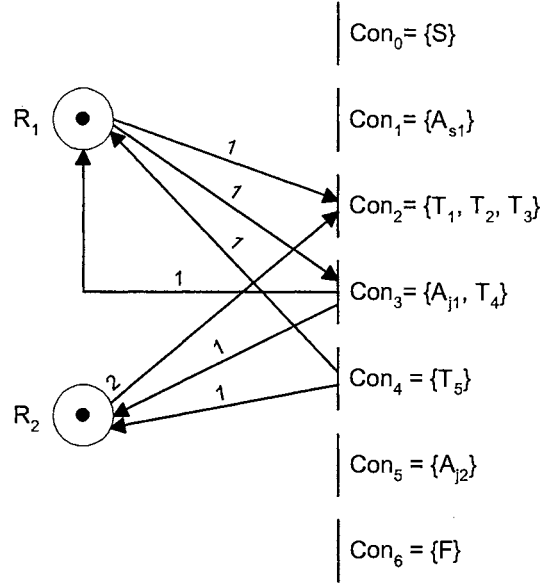


Figure 4.5: The Static-Design Net Representation – An Example

This Petri-net mapping shall be referred to as the **Static-Design Net**, named in part, to reinforce the suggestion that the structure of the net will be used to gain insights about the correctness of the process's design with regard to resource-sharing requirements.

More formally, for each valid meta-path $m \in \mathbf{M}$, its Static-Design net representation is a 4-tuple $SD_{Net}^m = (\mathbf{R}, \mathbf{Con}_{|V(m)}, \mathcal{F}_m^-, \mathcal{F}_m^+)$, where \mathbf{R} , the set of resources, corresponds to the set of places, and $\mathbf{Con}_{|V(m)} = \{Con_i^m\}$, the set of concurrent sets with vertices restricted to $V(m)$, corresponds to the set of transitions. The functionals $\mathcal{F}_m^- : \{\mathbf{R} \times \mathbf{Con}_{|V(m)}\} \rightarrow \mathbb{N}$ and $\mathcal{F}_m^+ : \{\mathbf{Con}_{|V(m)} \times \mathbf{R}\} \rightarrow \mathbb{N}$ specify the weighted arcs connecting places to transitions, and vice-versa. The initial marking of the net is $M_0 = [R_i^\#]$, namely, the capacities of the various resources. The structure of the net SD_{Net}^m is captured with the incidence matrix $\mathbf{C}_m = [c_{ij}]$, where

$$c_{ij} = \sum_{\forall t \in Con_j^m \cap \mathbf{T}} R_i^{Cap}(t) - R_i^{Rel}(t)$$

The Static-Design net of Figure 4.5, and in particular, the example in Figure 4.2(b), reveal several points of interest, namely:

1. Unlike the regular p_i, t_i naming convention for places and transitions, the places and transitions in the static-design net have been named by their actual definitions, namely resources (R_i) \rightarrow places, and concurrent sets (Con_i^m) \rightarrow transitions. This is to avoid any conflict with the definition of a task T_i , and to aid concept clarity.
2. The interpretation of the Static-Design net is as follows:
 - An arc is drawn from a place, R_i (respectively, transition Con_j^m) to a transition Con_j (respectively, place R_i) if there is a task in Con_j^m that captures (respectively, releases) resource R_i .
 - The weight on the arc, and in turn, its direction, is determined by the number of capture and release requests within the concurrent set. To illustrate, observe that both tasks T_2 and T_3 capture a unit of resource R_2 , and so, an arc with weight 2 is drawn from place R_2 to transition Con_2 .
 - The number of tokens in place R_i signifies the number of units $R_i^\#$ available for that resource.
3. There is only valid meta-path in the example of Figure 4.2(b), and the sequence through which the process progresses is strictly $Con_0 \rightarrow Con_1 \rightarrow Con_2 \cdots \rightarrow Con_5$. Thus, the concurrent sets also aid in capturing the process ordering imposed by the control flow model.

In continuation of point (3) above, note that, although Con_0 is, by definition, permanently enabled, and can fire indefinitely, such firing sequences will not be studied – in fact, the control flow model specifies the transition firing sequence, namely, $Con_0 \rightarrow Con_1 \rightarrow Con_2 \cdots \rightarrow Con_5$, i.e., the increasing order of $\{Con_i\}$ suggests the sequence through which the process progresses. Observe that this transition sequence is not possible in the static-design net shown in Figure 4.5 – why so? At the very least, Con_2 cannot be enabled given that there is only one unit of R_2 – this hints at problems either in inadequate resource capacity or potential circular-waits. The intuition just described is formalized in the following theorem.

Theorem 4.1 *Suppose it is given that, for each valid meta-path m in a process, there are no circular-waits within any concurrent set in $\mathbf{Con}_{|V(m)}$. If the transition sequence $Con_0^m = \{S\}, Con_1^m, \dots, Con_q^m = \{F\}$ is enabled¹ for the static-design net of all valid meta-paths m , then there exist no deadlocks within a single instance of the process.*

Proof Consider any valid meta-path m of the process. Since resources cannot be released before they are captured Con_1^m is automatically enabled, as $Con_0^m = \{S\}$ does not capture/release any resource. Consequently, the fact that $Con_i^m, i > 1$ is enabled, and requires, say, resource R_α , implies that there is either another transition $Con_j^m, 1 \leq j < i$ that releases the required number of units of resource R_α , or there are adequate number of units of R_α available to meet the needs of Con_i^m . Since this holds for all Con_i^m , it follows that there cannot be any problems in resource-sharing across concurrent sets in that valid meta-path. Moreover, it is given that there are no circular-waits within any Con_i^m for all valid meta-paths m . Therefore, the transition sequence $Con_0^m, Con_1^m, \dots, Con_q^m$ does indeed correspond to the deadlock-free execution of a single instance of the process for any valid meta-path m , thereby completing the proof. ■

Note, however, that the failure of the conditions of the proof, for some valid meta-path m of a process, does not imply that the process cannot execute correctly. To see why, consider the process represented in Figure 4.4 – while the transition sequence $\sigma_{incorrect} = A_{s1}, T_1, T_3$ leads to deadlock, the reader may argue that the process could still execute correctly with the transition sequence $\sigma_{correct} = A_{s1}, T_1, T_2, A_{j1}, T_5, T_3, T_4, A_{j2}$. Thus, Theorem 4.1 establishes only a *sufficiency* condition – it is not a *necessary* condition for deadlock freedom. To illustrate – the process in Figure 4.6 fails the conditions for Theorem 4.1, but is nevertheless deadlock-free.

¹Refer Definition A.4.

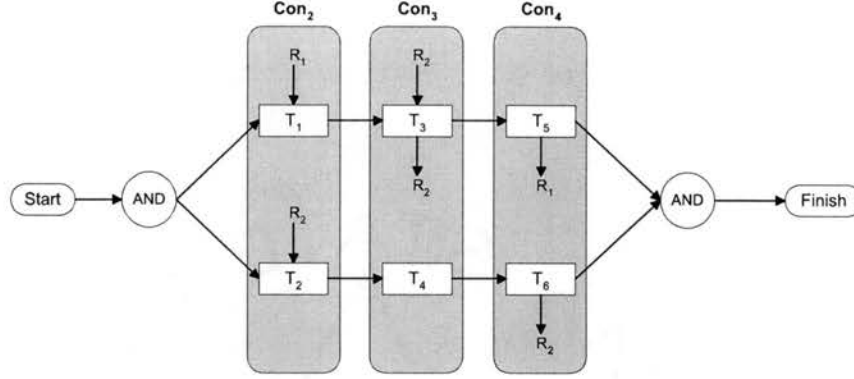


Figure 4.6: Another Example of a Deadlock-Free Process Model

Before continuing with the discussion, it would be worthwhile to pause and reconsider a statement presented in the paragraph above. It is true that, for the process in Figure 4.4, the transition sequence $\sigma_{correct}$ satisfies the requirements of the correct execution of the process. However, observe that $\sigma_{correct}$ breaks the process into two sequential threads $A_{s1}, T_1, T_2, A_{j1}, T_5$ and T_3, T_4, A_{j2} – is this what the process designer envisioned? where is the parallelism? if such an execution was acceptable, then why design the process such that task T_3 is concurrent with tasks T_1 and T_2 ? (refer Figure 4.2(b).)

In summary, what is the contribution of Theorem 4.1? Should the conditions of Theorem 4.1 be satisfied, i.e., the ordered transition sequence $Con_0^m = \{S\}, Con_1^m, \dots, Con_q^m = \{F\}$ is enabled for every valid meta-path m of a process, then, it confirms that a single-instance of the process is deadlock-free without requiring any additional analysis. However, the failure of the conditions for Theorem 4.1 indicates an immediate problem with *inadequate resource capacity* – ideally, the designer envisioned the tasks in each concurrent set Con_i^m , for each valid meta-path m , to be truly concurrent. The parallelism dictated by the design would have been possible had there been adequate number of resource units to meet the resource requirements of each task in Con_i^m , in which case Theorem 4.1 would not have failed; that it failed is the answer to begin answering the question of minimal resource requirements for the process.

4.5.2 Computing Minimum Resource Requirements

The failure of Theorem 4.1 offers an immediate opportunity to compute the minimal resource requirements that will guarantee the successful execution of an instance of the process that also assures the parallelism envisioned by the process's designer. More specifically, for each valid meta-path m , it remains to compute the minimum resource requirement (i.e., the initial marking for the net SD_{Net}^m) that will guarantee that the transition sequence $Con_0 = \{S\}, Con_1, \dots, Con_q^m = \{F\}$ (call it σ^m) will be enabled. Note that upon execution of σ^m , the net will return to its initial marking (by conservation of resources). Consequently, the minimum resource requirements for the process is computed as $R_i^\# = M_0^*(i)$, where, $\forall m \in \mathbf{M}, M_0^* \xrightarrow{\sigma^m} M_0^*$ holds, i.e., $M_0^*(i)$ is the minimum number of units required for resource R_i that will guarantee that the conditions of Theorem 4.1 hold true for all valid meta-paths m in the process.

To illustrate – suppose there are two valid meta-paths m_1 and m_2 in a process, and that two resources R_1 and R_2 are being used by the process. Suppose the smallest initial marking that will satisfy the requirements of Theorem 4.1 for both meta-paths is $[1, 3]$ and $[2, 1]$, respectively. Taken together, the minimum resource requirements that ensure that Theorem 4.1 holds for both meta-paths is $[2, 3]$, i.e., $R_1^\# = 2$, and $R_2^\# = 3$.

It is quite easy to compute M^* . Suppose that $\mathbf{R} = \{R_1, R_2, \dots, R_r\}$ is the set of r resources. Consider some valid meta-path m with $q+1$ concurrent sets, namely, $\{Con_0 = \{S\}, Con_1, \dots, Con_q^m = \{F\}\}$. Let M_0^m be the smallest initial marking for SD_{Net}^m that guarantees that $M_0^m \xrightarrow{\sigma^m} M_0^m$ holds, where the superscript m indicates that the marking corresponds to valid meta-path m . M_0^m is computed thus:

$$\forall i = 1, 2, \dots, r \quad M_0^m(i) = \sum_{j=1}^q \max(\mathcal{F}_m^-(R_i, Con_j^m) - \mathcal{F}_m^+(Con_{j-1}^m, R_i), 0) \quad (4.1)$$

The mechanics of equation 4.1 is quite simple. It essentially keeps track of how many new tokens are required in place R_i , in addition to those returned by Con_{j-1}^m , to enable each Con_j^m . Once M_0^m has been computed, it follows that the minimal requirements for resource R_i is

$$M_0^*(i) = \max_m M_0^m(i) \quad \forall i = 1, 2, \dots, r$$

Interestingly, note that M_0^* also eliminates the possibility for circular-waits within a concurrent set, since the resource requirements for all competing concurrent tasks will be satisfied by M_0^* , thus rendering redundant the count-based checks presented in Section 4.4.1.

Thus, the computations above derive the minimum number of units required for each resource R_i to ensure that the conditions of Theorem 4.1 hold – this will ensure the successful deadlock-free execution of a single instance of the process, with the maximum degree of parallelism as desired by the process’s designer.

4.5.3 Summary

Recall that design errors in resource-sharing are not restricted to single instances (refer example 4 in Table 1.3). Clearly, the basic Petri net approach suggested in Section 4.4.2 can be extended to study multiple-instance verification, simply by increasing the number of tokens in the place corresponding to **Start** and exploring the reachability tree to detect deadlock occurrences. However, can the static-design net be used to derive a quicker answer to help detect the possibility for potential deadlock, without recourse to exhaustive enumeration via reachability analysis?

4.6 Multiple-Instance Verification

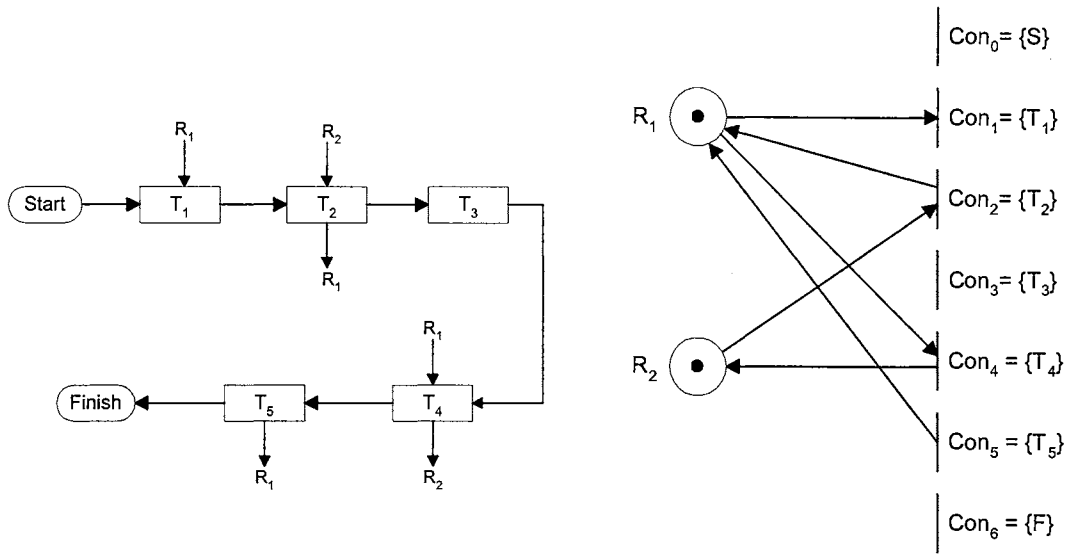
The purpose of this section is to develop an approach that will identify if the sharing of resources across multiple instances of the process will or will not lead to deadlock – the approach will rely on the static-design net representation of the process, and on the notion of transition invariants for a Petri net (refer Section A.4).

Observe that the static-design net will be a collection of isolated vertices and several disjoint, connected components.² There is no sharing of resources across disjoint components (else, they would not be disconnected), and isolated transitions (i.e., concurrent sets) do not capture/release any resources, and hence, are inconsequential. Isolated places correspond to resources that are not being accessed in the valid meta-path that the static-design net represents. Consequently, it is within a connected component of the static-design net, that deadlock possibilities, if any, wait to be unearthed.

Figure 4.7(b) illustrates the static-design representation for example 4 from Table 1.3, also repeated in Figure 4.7(a). Note that the process is completely sequential, with no choice or concurrency; therefore, it readily follows that a single-instance of the process is deadlock free, since the size of each concurrent set is only one.

The weights on the arcs of the static-design net in Figure 4.7(b) have been omitted since they are all equal to one – this net consists of one connected component (consisting of R_1 , R_2 , Con_1 , Con_2 , Con_4 , and Con_5), and other isolated vertices. The connected component has two transition invariants, namely, $Tinv_1 = [Con_1, Con_5]$ and $Tinv_2 = [Con_2, Con_4]$. Does this suggest something? Yes, it does – it suggests that there is a potential problem of circular-wait across two instances with one executing $Tinv_1$ and the other executing

²A *connected* component is a subset of a graph which is disjoint from the remainder of the graph, and within which, every pair of vertices is connected by an undirected path. A *strongly connected* component is a connected component, within which, every pair of vertices is connected by a directed path.



(a) Incorrect Process Design

(b) Static-Design Net

Figure 4.7: Example of a Process with Problems of Deadlock across Multiple Instances

$Tinv_2$, with the latter being denied access to R_1 being held by the former – this is possible since R_1 is released by Con_2 enabling another instance to commence execution. More specifically, the connected component in Figure 4.7(b) runs the risk of deadlock because the resources required by it are not guaranteed to be exclusively available for its execution without the possibility of being captured by other previous/later instances. The intuition underlying this argument is very simple, namely – disjoint components of the static-design net do not have any problems of resource-sharing either within a single- or multiple-instances of the process; consequently, within a connected component, the only way to ensure that it will not get deadlocked is to ensure complete access to all of its required resources before another instance may begin. In short, the transition invariant of a connected component of the static-design net must consist of all the transitions in that component – this is formalized in Theorem 4.2.

Theorem 4.2 *Suppose it is given that there are no deadlocks within a single-instance of the process. Then, there exist no deadlocks arising from resource-sharing across multiple instances of the process executing the same valid meta-path, only if the transition invariant for each connected component of the process’s static-design net consists of all transitions in that component.*

Proof The proof follows immediately from the argument above, and from the definition of a transition invariant. ■

The uniqueness of Theorem 4.2 is that it relies only on the structure of the static-design net, and not on its initial marking $[R_i^\#]$, or the number of instances that are active – it is thus a simple approach to identify problems across multiple instances without the simulated execution of multiple concurrent instances.

Note that Theorem 4.2 also establishes only a *sufficiency* condition, i.e., it is sufficient to show that the transition invariant for each connected component consists of all transitions in that component to confirm that there are no deadlocks when multiple instances of the same valid meta-path are active. To show that it is also a *necessary* condition would be a tremendous achievement, since computing transition-invariants is a polynomial-time operation [64, 26], as opposed to the exponential state-space explosion of reachability enumeration.

There is however, one shortcoming of Theorem 4.2 – it cannot identify deadlock possibilities for multiple instances of the process, each executing different valid meta-paths. The answer to this question would ultimately require the reachability enumeration approach of 4.4.2 with multiple tokens in the place corresponding to **Start**. More specifically, Theorem 4.2 captures the border between that which can be answered in polynomial time, and that which requires exhaustive enumeration.

4.7 Summary

The ideas presented in this chapter form the foundations of the first-ever attempts to study the business process's design in its entirety, including both the process's logic, and its resource requirements – not with simulated executions, but with very elementary graph-theoretic ideas. The most notable achievement is that the simplicity of the approaches presented herein is complemented only by the sheer magnitude and value of the questions addressed and answered. More specifically, the purpose of this chapter has been to study the process's design as the designer envisioned it, and to study its resource allocations with the intent of (i) computing the minimal resource requirements that guarantee the successful execution of the process and exploit the parallelism allowed in its design, and (ii) alerting the designer about potential deadlock possibilities that may arise either within a single-, or across multiple-instances of the process.

An alternate Petri net representation of the process that captures the ordering suggested by the control flow model with the specifics of resource requirements, the STATIC-DESIGN NET, has been developed, and sufficiency conditions have been derived to establish deadlock-freedom both within a single- and across multiple-instances of the process. A simple approach to compute the minimum resource requirements that guarantee the successful deadlock-free execution of the process has also been developed.

The correctness issues studied thus far have focused primarily on design errors arising either from incorrect control flow or improper resource-sharing. What about the *inputs* and *outputs* for each task? Are there any correctness issues that remain to be investigated in the input-output specification for the tasks in a process? Metagraphs [9, 10, 11] have been used to study connectivity issues in the input-output specification of a process. Additionally, an immediate extension of the ideas presented in this chapter would be

to use the concurrent set partitions, $\{Con_i\}$, of the control flow model to study the correctness of a process's input-output specifications. This would require a formalism identical to R_i^{Cap} , and R_i^{Rel} , focusing, in turn, on the *inputs* and *outputs* of each task, and addressing questions similar to those presented in Section 4.3 – these ideas are reserved for future work.

Chapter 5

Modeling and Analysis of Business Process Models

Chapter Overview

This chapter outlines the features of a proof-of-concept implementation of the algorithms developed in this work.

5.1 MAPS: Proof-Of-Concept Implementation

A computerized environment titled **MAPS** – Modeling and Analysis of business Process model**S** has been developed to support the techniques developed in this dissertation. MAPS has been written in PYTHON, an open-source programming language, and its graphical interface coded in Tkinter – it retains the native look and feel of a Windows application, and supports a good graphical editor for the development of process models. MAPS includes the KORRECTNESS algorithm (Chapter 3) for control flow verification, and also provides diagnostic feedback about control flow errors, if any.

The choice of PYTHON as the development language was motivated by two reasons:

1. MAPS is intended to grow as a research test-bed for new ideas and algorithms in business process modeling, and it was essential that the underlying code and program design remain simple, without being excessively clouded with the details of syntax and software-specific overheads.¹
2. The complexity of the algorithms notwithstanding, Python provided for nearly-identical translations of mathematical intuition (especially, set-theoretic formulations) into program syntax, making it ideal as a tool-of-choice that would attract and offer incentives for other researchers to continue experimenting with, and developing MAPS.

This chapter is intended to be a walk-through of MAPS's salient features and the significance of its contribution as a modeling and design verification tool. The main attributes of MAPS are its simple modeling interface, and support for control flow verification.

5.2 Modeling Interface

The development of a process's design in MAPS begins with the specification of its control flow model, followed by the separate specification of its resource requirements. Figure 5.1 presents a screen-shot of MAPS – it illustrates the control flow model of the counter-example presented by Lin *et al.* [61] to show that the algorithms of Sadiq [75] are incomplete.

¹VC++/MFC and JAVA/JFC-Swing applications, to name a few.

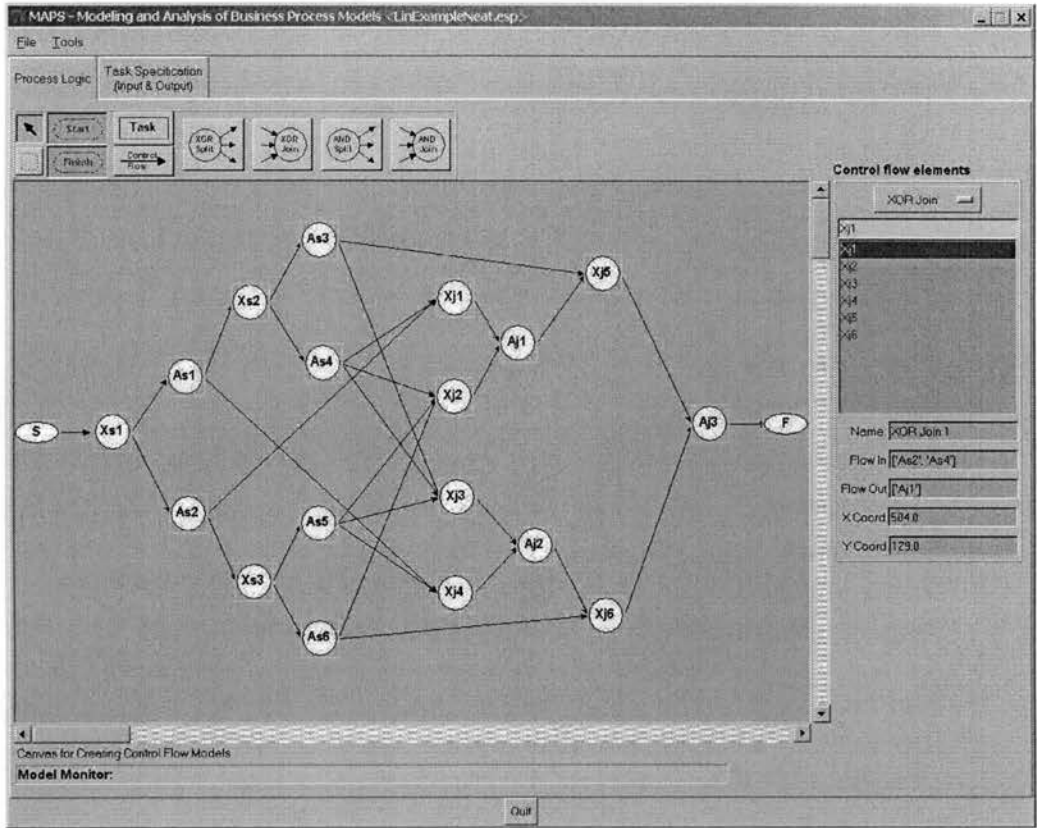
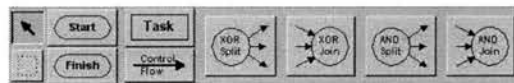



Figure 5.1: A Sample Screen-shot of MAPS

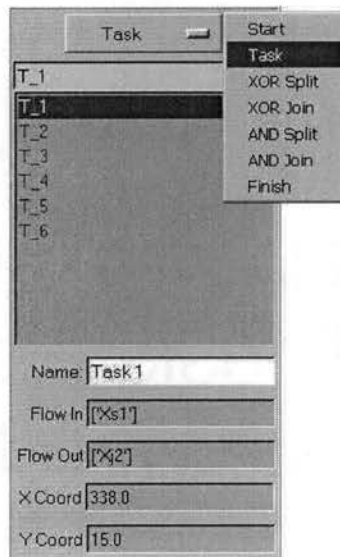
The major components of the graphical editor are:

1. The stencil that the user uses to select the control flow element being drawn.



The stencil offers simple click-to-select functionality – the user selects (left-click) the control flow element that needs to be drawn, and then selects a position in the canvas to place them, or, if a control flow arrow is being drawn, the user selects the “source” (*from*) and “destination” (*to*) of the arrow with consecutive clicks inside the face of two elements on the canvas. The model can contain only one “Start” and one “Finish” – the corresponding stencil elements get disabled thereafter, unless those elements are subsequently deleted from the model.

2. The canvas on which the model is drawn – the canvas supports intuitive operations for both movement and deletion of elements in it. Deletion is enabled only in the “cursor  mode” via right-click mouse operations on the edges of the elements – the mouse cursor will change to a hollow circle to indicate that the element can be deleted with a right-click. Movement of elements is enabled in the “cursor mode” with a left-click hold and release mouse movement, or with a right-click hold and release mouse movement in one of the “non-cursor modes” – in both cases, the mouse click must be inside the face of the element, which will be signalled by the cursor changing to a filled circle with an embedded cross (a *fleur*). The “non-cursor modes” refer to the selection of either a task/logical-operand/arrow on the stencil – a left-click operation either places a new element (task/logical-operand) on the canvas, or is used to consecutively select the source and destination of a control flow arrow.
3. The control flow elements’ tablet (or frame) that provides an easy and accessible summary of each control flow element’s attributes.



This frame is organized as follows – a drop-down options box allows the user to select the type of the control flow element that they need details for, namely, Start,

Finish, Task, or a logical operand. Once the selection is made, say, “Task,” a list of all tasks is generated, ordered by their screen IDs, individual selection of which will populate the basic fields beneath to reveal their names, input and output elements, and on-screen graphical coordinates. The screen IDs of the elements are prefixed with a [“S,” “F,” “T,” “Xs,” Xj,” “As,” “Aj”] to indicate that they are either a “Start,” “Finish,” “Task,” “XOR-Split,” “XOR-Join,” “AND-Split,” or an “AND-Join.” All of the basic fields are fixed and cannot be edited, except for the names of tasks, which can be changed from the program-generated “Task *n*” to something more meaningful, if needed.

4. The model monitor that provides continuous feedback to the modeler about their actions through short messages.

Model Status: You are currently drawing an arrow starting at Task 1 and ending at XOR Split 1

Delete Alert: You have deleted Xs1

Error: You cannot draw more than one edge leading out of an Task. Please restart all over again.

Error: You cannot draw more than one edge coming into an XOR Split. Please restart all over again.

Delete Alert: Control flow arrow from T_2 to T_3 has been deleted.

The model monitor will be indispensable when dealing with complex models – it includes several useful features that will guide the user in the construction of the model, and more so, when the user is contemplating the deletion of some elements. Unlike commercial grade software with the luxury of *undos* and such, the user will have to rely on the model monitor to inject the requisite caution in dealing with mouse-clicks, right or left.

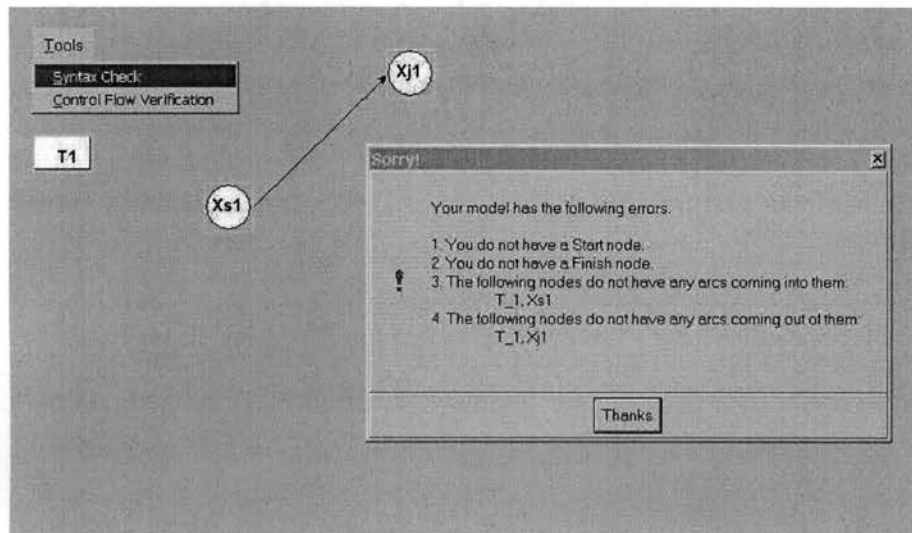
In addition to the features listed above, *help balloons* have been programmed to appear liberally across all aspects of the application to clarify the purpose of all four components above. The models thus created can be saved and retrieved with standard *File* → *Save* operations.

5.3 Analysis and Verification Capabilities of MAPS

The current version of MAPS includes support for verifying the model syntax, and the KORRECTNESS algorithm for control flow verification, both of which are described below.

5.3.1 Syntax Verification

The syntax checks enforced in MAPS ensure that the model does not have any abandoned elements, that it has a “Start” and a “Finish,” and that all other elements have properly defined “from” and “to” elements, as illustrated below.



Additionally, should the user attempt to draw, say, two arcs leading into a task (or an AND/XOR-split), or other such basic modeling errors that violate the definition of the elements, the model monitor will alert the modeler to the same, thereby allowing for immediate model checks as well – the syntax verification capabilities in MAPS are dynamic and work constantly during the development of the model.

5.3.2 Control Flow Verification

The verification of control flow correctness follows the KORRECTNESS algorithm, in that it proceeds by generating the set of $S - F$ paths, the set of valid meta-paths, and the set

of invalid meta-paths, if any. Figure 5.2 shows a snap-shot of the application interface after the KORRECTNESS algorithm has been applied to the control flow model shown in Figure 5.1 – note that the results of the control flow verification procedure will open up on a new page titled “Korrektness Results.”

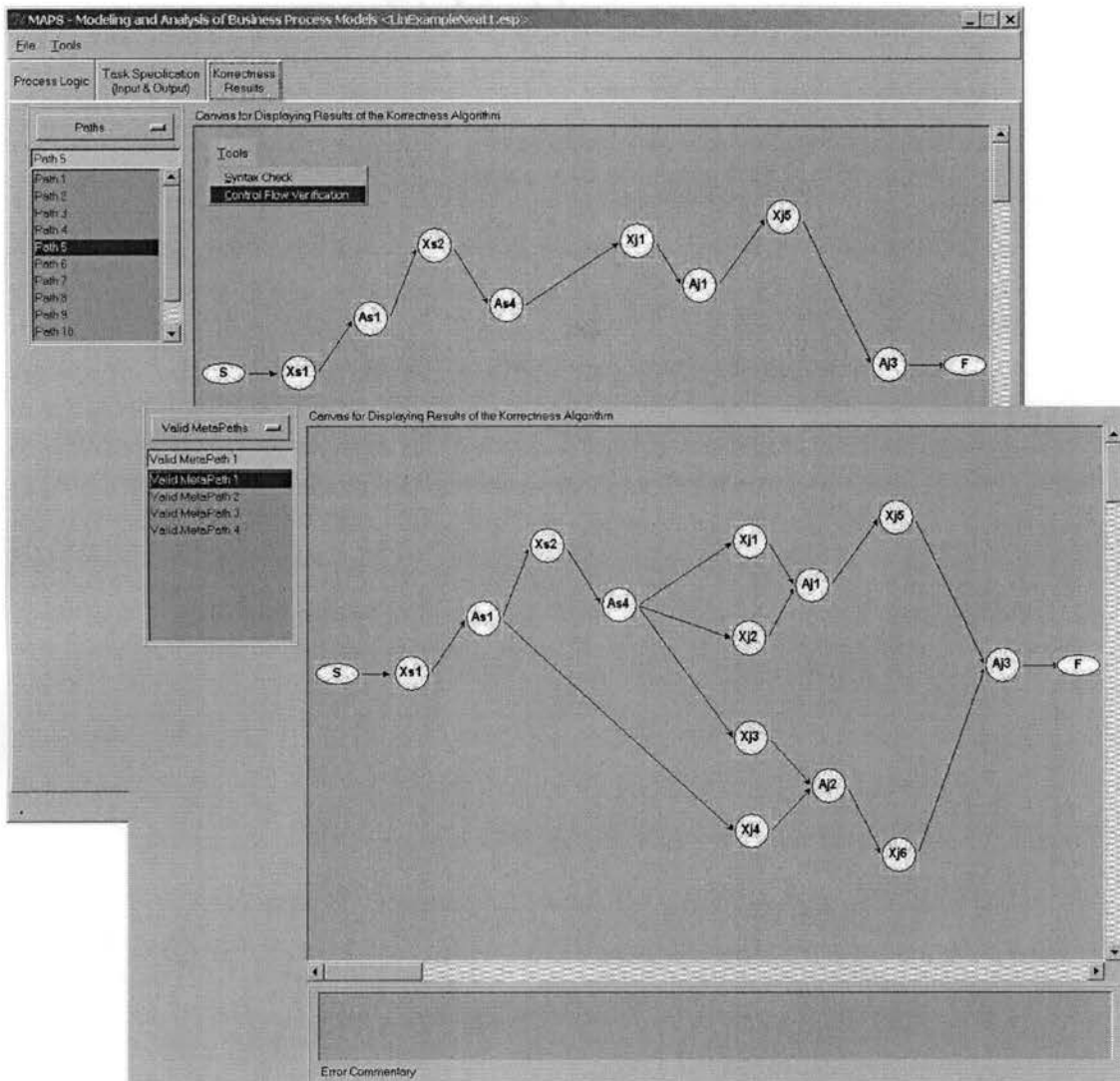


Figure 5.2: Identifying the Set of Valid Meta-paths

The interfaces for browsing through the set of paths, valid, and invalid meta-paths are all designed to be very simple, and follow a layout identical to the control flow elements' frame discussed for the modeling interface.

Figure 5.3 illustrates a snap-shot of the application interface after the KORRECTNESS algorithm has been applied to the incorrect control flow model of Figure 3.8 – it illustrates one of several invalid meta-paths in the process.

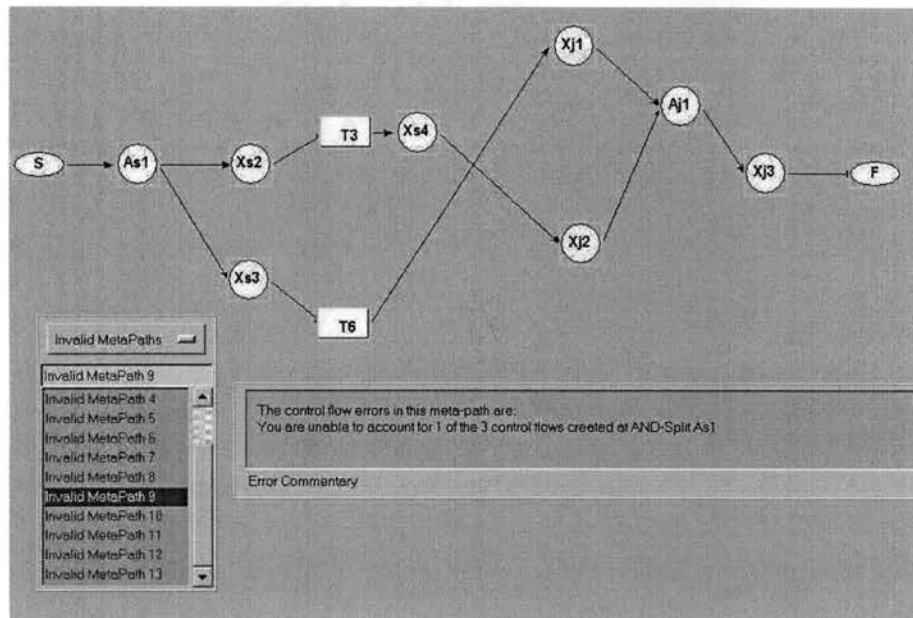


Figure 5.3: Identifying the Set of Valid Meta-paths

Much like the “model monitor” in the modeling interface, the “Error Commentary” provides feedback about the source of the control flow error – the meta-path in Figure 5.3 is invalid because one of the three control flows required at AND-Split As_1 is missing. This “error commentary” feature of MAPS is unique in providing precise reasons as to the failure of a control flow design – what is missing however, is a way to automate the correction of these control flow errors, or at the very least, to give suggestions to fix the same.

5.4 Summary

This chapter outlined the use of **MAPS**, a computerized environment that was developed to support the algorithms developed in this dissertation. MAPS has been designed to be easily extendible with new functionalities, either in graphical modeling, or algorithmic support. Thus far, MAPS includes only qualitative analysis capabilities; it would be a wonderful addition to also incorporate code-support for simulation of a business process, and possibly, detailed Petri net modeling and queuing analysis – these are not careless dreams or whimsical hopes. It is the author’s earnest hope that the simplicity of MAPS’s program design will inspire further work in extending it to make it a useful tool for classroom instruction, and also as a rewarding intellectual exercise for those who choose to relive the joy that was the author’s privilege in creating it.

Currently, MAPS includes an editor for creating, storing, and retrieving control flow models, and analysis support to establish control flow correctness. The following extensions are anticipated for future development of MAPS:

1. Incorporating functionality for verifying the correctness of resource-sharing and computing minimal resource requirements, based on the techniques developed in Chapter 4.
2. Interfaces to (the input and output of) XML descriptions of business process definitions, based on pre-specified process templates as specified by [96, 7], would be a fantastic addition that would impact both the commerce and the care that the BPM software industry extends to design verification.

Chapter 6

Summary & Research Contributions

Chapter Overview

This chapter summarizes the major contributions of this research, and outlines research questions that will expand the reach and value of the design verification techniques developed in this work.

6.1 Summary

The purpose of this dissertation has been to study the *design* of a business process, as determined by its logic, and to give useful feedback to the designer about the correctness of the design. Stated simply, the question being addressed is not “how good is this process’s design,” but is more fundamental, namely, “is the design good at all?” The former question relates to studying the performance of a business process, with regard to its operational efficiency and other metrics summarized through analytical calculations or simulated executions of the process – it necessarily assumes an affirmative answer to the latter question. That the design of a process may not be good to begin with, (and if not, why so?) is the motivation for this dissertation.

This dissertation presents a comprehensive foundation for the formalization and verification of business process designs. A business process could be one of either a material, information, or a people process, or a combination thereof (Table 1.1) – irrespective of the type of the business process, the design of a process minimally requires the specification of the process’s logic, and the resource requirements for its constituent tasks. Business processes arise in numerous contexts; however, the verification issues are the same, namely

1. **FUNCTIONAL ASPECTS** – is the logic of the process correct, i.e., does the flow of control within the process ensure that the process will execute correctly from initiation to completion?
2. **RESOURCE ASPECTS** – is the release and capture of shared resources among different tasks, either in the same or in different instances of a process, well-designed so as to avoid conflict?

This dissertation presents a completely context-independent formalism that bridges the diversity in process types, with the commonality of the questions and correctness issues that arise within – this, above all, is the most significant contribution of this work. Both the control flow and resource-sharing problems have been thoroughly studied, to present, in effect, a solid foundation for the verification of process designs that will significantly change the currently understood interpretation of design verification, which relates primarily to syntactic checks restricted to the graphical modeling formalism.

Chapter 1 presents a short, but precise, introduction that motivates the relevance of, and the challenges associated with, the control flow and the resource-sharing problem. Chapter 2 presents a comprehensive overview of the issues and opportunities in business process modeling, business process automation, and an accurate review of all relevant

research – it is important to note that the resource-sharing problem, as studied in this dissertation has not been previously addressed anywhere else. Chapter 3 presents the **KORRECTNESS** algorithm, a recursive, backtracking algorithm for verifying the correctness of control flow in any process, without any restrictions on the form/structure of its design, and to provide diagnostic feedback about the source of the control flow error (if any). Chapter 4 presents a collection of Petri net-theoretic techniques for studying the correctness of resource-sharing in a process, and to identify potential design errors that could lead to deadlock. Chapter 5 details the success of a proof-of-concept implementation of the algorithms developed in this work.

6.2 Research Contributions

The most significant contribution of this dissertation is a simple formalism for specifying both the control flow (Chapter 3) and resource requirements (Chapter 4) of a process, in a manner that does not diminish the semantic value of the elements being defined, while still retaining sufficient rigor to motivate abstract study of the process’s definition. The specific contributions of this dissertation are listed below.

Control Flow A recursive, backtracking algorithm for verifying control flow correctness has been developed. The algorithm does not impose any restrictions on the form/structure of the control flow model, and some interesting results on properties to be expected in random control flow models have been derived. Additionally, the results of the algorithms also provide precise diagnostic information about the reasons for the control flow error(s), if any.

Resource-Sharing A simple Petri-net theoretic approach for identifying potential deadlocks, especially circular-waits, has been developed. The approach is unique in that it exploits the control flow model to gain intuitions about the structure and behavior of the process, without ever requiring any simulations. More specifically, simple

rules have been developed to (i) compute the minimal resource requirements that guarantee the successful execution of the process, while fully exploiting the parallelism in the process as envisioned by its designer, and (ii) alert the designer about potential deadlock possibilities that may arise either within a single-, or across multiple-instances of the process.

MAPS A computerized environment titled **MAPS – Modeling and Analysis of Process models** (implemented in Tcl/Tk and Python) has been developed to support the algorithms developed in the dissertation. Ideally, MAPS should grow as a research test-bed for new ideas and algorithms in business process modeling. The salient features of MAPS are:

- A good graphical environment for modeling and specifying business processes.
- Algorithms for verifying the correctness of control flow and providing diagnostic feedback about the sources of control flow error(s), if any.

6.3 Future Research Directions

Automatic design verification capabilities are as yet unavailable in current business process modeling softwares. To this end, the ideas developed herein significantly advance the power and potential for the development of good processes, the designs of which are influenced both by the judgment of the domain expert and by the clarity of analysis. In continuance of this work, the following ideas merit inquiry:

1. **Automation of Business Process Redesign** Suppose the process's design, i.e., control flow and/or resource requirements, is incorrect; can the redesign of business processes, to eliminate design errors, be automated? More specifically, can human intuition be replaced with algorithmic deduction and correction? Currently,

diagnostic checking is limited to identification – can it be extended to include elimination?

2. Automatic Reconfiguration of Business Processes Suppose the process’s design, i.e., control flow and resource-sharing, is correct; is it possible to suggest approaches to reconfigure or “optimize” the process’s design, based on the nature of the resource-allocation requirements and the precedence-order relationships that are imposed by the logic of the process? This would require a more precise understanding of the expectations of “optimality” in business process designs – stated simply, the domain expert has identified a particular configuration for the process; can it be improved?

3. Standards for Business Process Specification To develop a formalism for modeling and specification of business processes that blends the ease of modeling intuition with the rigor required for design verification. Ideally, a formalism that capitalizes on the transparency of XML, the ease of graphical modeling, and the support of underlying design verification techniques would greatly enhance the value-addition of enterprise automation.

The research proposed in question (3) above has already been initiated, and the Business Process Management Initiative¹ is spearheading current efforts toward the development of BPML, an open-source standard **Business Process Modeling Language**. There is also a competitive, commercial effort that has been initiated by Microsoft, IBM, and BEA, called BPEL4WS – readers are referred to [48, 78, 36] for a good overview of the issues involved in developing a standard for the design, deployment, execution, control, and optimization of business processes. In conjunction with these efforts, it would be extremely useful to explore the possibilities for abstracting from the XML description of a business process, sufficient detail such as is required for developing an analytical

¹<http://www.bpmi.org>.

or a simulation model, to motivate further analysis of the process's configuration and performance, independent of the context in which the process may arise – this has been partly addressed in IBM's OPS (Operational Specification), an artifact-based approach to business process modeling and enterprise integration [16].

The questions raised in (1) and (2) are more fundamental and as yet unexplored; to allow for a computer to suggest a better process design is very intriguing, and such an ability would lend new meaning to “automation” in business process automation – some preliminary work has been addressed in [38], but, it is still not “automation.” The extent of the second question's appeal is surpassed only by the vagueness of its answer. To answer the same, without requiring context-specific information particular to a process's domain, would be the first steps in establishing a “science-base” for business process modeling. The bridge between the abstract and the real has thus far been absent in business process modeling, so much so, that the question of “deadlock” in business processes is met, not with a !, but with a ? – it is hoped that the techniques developed in this dissertation would contribute to building such a bridge.

Appendix A

Petri Nets: A Primer

Chapter Overview

This appendix presents a quick overview of the basics of Petri nets. Readers are referred to [6, 70, 68, 25, 26] for a more extended discussion on the theory, applications, and analysis of Petri nets.

A.1 What is a Petri Net?

A Petri net is an anagraphical (i.e., analytical and graphical) tool that combines the appeal of graphical description with the rigor of mathematical formalism, making it the preferred tool of choice for modeling discrete event systems. In particular, systems, whose description can be specified as a collection of events, and conditions preceding and succeeding the execution of those events, are well suited to being modeled and studied with Petri nets. Petri nets are especially well-suited to modeling concurrency, asynchronism, and choice. More formally, a Petri net is a 4-tuple, $\mathcal{N} = (P, T, f^+ f^-)$, where $P = \{p_1, p_2, \dots, p_n\}$ and $T = \{t_1, t_2, \dots, t_m\}$ are disjoint, finite sets of *places* and *transitions*, respectively. Additionally, $f^+ : T \times P \rightarrow \mathbb{N} = \{0, 1, \dots\}$ and $f^- : P \times T \rightarrow \mathbb{N}$ are

the incidence functions from transitions to places, and vice-versa. It is common practice to associate places with conditions, and transitions with events in a discrete event system.

The graphical representation of a Petri net is a directed, bi-partite graph with two sets of vertices – places P (represented as circles) and transitions T (represented as lines/bars), and weighted arcs drawn from places to transitions, (respectively, transitions to places), the weights on the arcs corresponding to the values of $f^-(p_i, t_j)$ (respectively, $f^+(t_j, p_i)$), for every pair (p_i, t_j) of places and transitions – Figure A.1 illustrates an example (arcs with unit weights are left unlabeled).

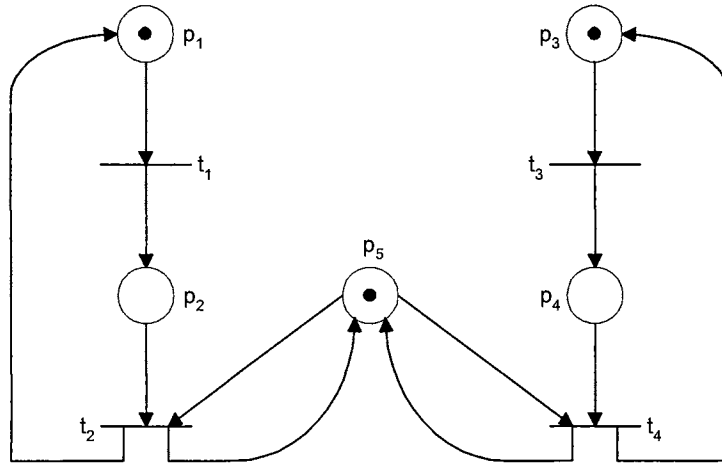


Figure A.1: An Example of a Petri Net Model

For each $t \in T$, $\bullet t = \{p \in P \mid f^-(p, t) > 0\}$ and $t\bullet = \{p \in P \mid f^+(t, p) > 0\}$ represent, respectively, the set of *input* and *output* places of t . The sets $\bullet p$ and $p\bullet$ are defined analogously for each place p . The incidence matrices $\mathbf{C}_{n \times m}^+$ (from transitions to places) and $\mathbf{C}_{n \times m}^-$ (from places to transitions) are defined as

$$\mathbf{C}_{n \times m}^+ = [c_{p,t}^+] = [f^+(t, p)] \quad \mathbf{C}_{n \times m}^- = [c_{p,t}^-] = [f^-(p, t)]$$

The incidence matrix of the net is represented as $\mathbf{C} = \mathbf{C}^+ - \mathbf{C}^-$. A marking is a function $M : P \rightarrow \mathbb{N}$ that assigns a numeric value to each place in the net. Graphically, markings are identified by tokens (black dots) residing in the places of the net, and capture the state of the net. The Petri net \mathcal{N} , together with an initial marking M_0 , is adequate to model the evolution of a system starting at state M_0 . The dynamics of the net's evolution are controlled by firing transitions, the definitions of all of which are presented next. Unless stated otherwise, assume that the initial marking of the net is M_0 .

The initial marking for the Petri net in Figure A.1 is $M_0 = [1, 0, 1, 0, 1]$, and its incidence matrices are

$$\mathbf{C}^+ = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \quad \mathbf{C}^- = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}$$

A.2 Basic Definitions

The vector δ_i will denote the vector whose i^{th} element is 1, and 0 elsewhere, i.e., $\delta_i = [0, 0, \dots, 0, 1, 0, \dots]$. The dimensions of the vector will be obvious from the context in which it appears.

Definition A.1 *A transition t_i is enabled under some marking M_j if $M_j \geq \mathbf{C}^- \delta_i$, i.e., there are enough tokens available in all of t_i 's input places, i.e., $\bullet t_i$, to meet its "input" requirements.*

For the net in Figure A.1, only transitions t_1 and t_3 are enabled under $M_0 = [1, 0, 1, 0, 1]$.

Definition A.2 An transition t_i that is enabled under marking M_j can fire, i.e., it can move tokens from its input places ($\bullet t_i$) to its output places ($t_i \bullet$). More specifically, upon firing an enabled transition t_i , the net evolves from marking M_j to marking M_{j+1} according to the equation

$$M_{j+1} = M_j + \mathbf{C}^+ \delta_i - \mathbf{C}^- \delta_i = M_j + \mathbf{C} \delta_i$$

and is usually notated as $M_j \xrightarrow{t_i} M_{j+1}$, signifying that marking M_{j+1} is immediately reachable from M_j .

Actually, when a transition t fires, it removes $f^-(p, t)$ tokens from each input place $p \in \bullet t$, and deposits $f^+(t, p)$ tokens in each output place $p \in t \bullet$. For the net in Figure A.1, the marking reached from firing transition t_1 is $[0, 1, 1, 0, 1]$.

The function $z : T \rightarrow \mathbb{N}$ will be used to generalize the definition of δ_i presented above, and model multiplicities in the number of times each transition fires – it will be referred to as the *firing-count vector*, with $[z]_i$ indicating the number of times that transition t_i is fired.

Definition A.3 The firing-count vector z is enabled under marking M_j if $M_j \geq \mathbf{C}^- z$. The state that the system evolves into, upon firing z , is $M_{j+1} = M_j + \mathbf{C} \cdot z$ – this is also referred to as the state equation.

For the net in Figure A.1, the marking reached upon executing the firing-count vector $[1, 1, 1, 1]$ is equal to M_0 , i.e., the tokens arrive in their initial places after executing all four transitions.

Note that the firing count vector does not specify the order in which the transitions are fired. Define $\sigma = t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k}$ to be an ordered sequence of transitions, and $\sigma(i)$ to be the i^{th} transition in that sequence.

Definition A.4 A transition sequence σ is enabled under marking M_j if, for all i , transition $t = \sigma(i)$ is enabled under marking $M_{j+(i-1)}$, where $M_{j+(i-1)} = M_j + \mathbf{C} \cdot \sum_{\tau < i} \delta_{\sigma(\tau)}$. More specifically, the transition sequence $\sigma = t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k}$ is enabled under marking M_j if

$$M_j \xrightarrow{t_{i_1}} M_{j+1} \xrightarrow{t_{i_2}} M_{j+2} \cdots \xrightarrow{t_{i_k}} M_{j+k}$$

holds true. This is frequently abbreviated as $M_j \xrightarrow{\sigma} M_{j+k}$.

A marking M_d is said to be reachable from marking M_0 if there exists a transition sequence σ such that $M_0 \xrightarrow{\sigma} M_d$. The *reachability set* $\mathbf{R}(M_0)$ is the set of all markings that can be reached from M_0 . The reachability set for the net in Figure A.1 is $\{[1, 0, 1, 0, 1], [0, 1, 1, 0, 1], [1, 0, 0, 1, 1], [0, 1, 0, 1, 1]\}$.

A.3 Additional Concepts

There are a few additional definitions that merit inclusion for sake of completeness. A Petri net is said to be *acyclic* if it does not contain any cycles, and *pure* if it does not contain any self-loops. A Petri net is said to be *free-choice* if and only if for every two transitions t_1 and t_2 , $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ implies that $\bullet t_1 = \bullet t_2$ – the reader is referred to [25] for an excellent treatise on free-choice Petri nets. Aside from these structural characterizations, there are certain behavioral properties that are defined as follows.

Definition A.5 A Petri net is said to be *proper* if $M_0 \in \mathbf{R}(M_0)$, i.e., if the initial marking is reachable from itself.

Definition A.6 A transition t is said to be *live* if for each marking $M' \in \mathbf{R}(M_0)$, there exists another marking reachable from M' in which t is enabled.

Murata [68] presents several other characterizations of liveness, based on slight modifications of the conditions of Definition A.6 above. A Petri net \mathcal{N} , together with an initial marking M_0 , is said to be *live* if all of its transitions are live.

Definition A.7 A place p is said to be k -bounded if there is an integer k such that for all $M \in \mathbf{R}(M_0)$, $M(p) \leq k$.

The Petri net (N) is said to be k -bounded if all of its places are k -bounded. An 1-bounded Petri net is called *safe*. The net in Figure A.1 is *proper*, *live*, and *safe*.

To summarize, a Petri net captures the structure of a system, and can model its evolution through the movement of tokens. Thus, the net \mathcal{N} , taken together with an initial marking M_0 , and the reachability set $\mathbf{R}(M_0)$, completely describes a system's behavior. Aside from the state evolution rules outlined in Definitions A.3 and A.4, the structure of the net, as represented by its incidence matrix \mathbf{C} , also reveals several clues about the behavior of the system, the study of which is known as *invariant analysis*.

A.4 Invariant Analysis

Recall from Definition A.3 above that the evolution of the system is characterized by the state equation $M_d = M_0 + \mathbf{C} \cdot z$, where $C_{n \times m}$ is the incidence matrix for the net with n places and m transitions.

Definition A.8 A $n \times 1$ vector X is a *place-invariant* if $X^T \cdot \mathbf{C} = 0$.

Suppose X is a place-invariant. Substituting in the state equation, we get $X^T \cdot M_d = X^T \cdot M_0$ (since $X^T \cdot \mathbf{C} = 0$), whereupon it follows that for all markings M reachable from M_0 , the weighted sum of tokens, i.e., $\sum_{i=1}^n X_i \cdot M(p_i)$, is a constant. Consequently, if there exists a place invariant vector all of whose n entries are strictly positive, it follows that the net is bounded.

For the net in Figure A.1, the minimal, linearly independent (i.e., one is not a subset of another) place-invariants are $\{[1, 1, 0, 0, 1], [0, 0, 1, 1, 1]\}$.

Definition A.9 *A $m \times 1$ vector Y is a transition-invariant if $\mathbf{C} \cdot Y = 0$.*

Suppose Y is a place-invariant, all of whose entries are positive. The firing vector corresponding to such a vector returns the net back to its initial marking, i.e., $M_0 + \mathbf{C} \cdot Y = M_0$. Consequently, if there exists a transition-invariant with some positive elements and some zeros, then, it indicates that the net can be returned to its initial marking by firing only a subset of the transitions.

For the net in Figure A.1, the minimal, linearly independent (i.e., one is not a subset of another) transition-invariants are $\{[1, 1, 0, 0], [0, 0, 1, 1]\}$.

The definitions presented in this section are by no means exhaustive – the reader is referred to [26] for a well-written and comprehensive introduction to the theory and applications of Petri nets.

Bibliography

- [1] Aalst, W.M.P. “The Application of Petri Nets to Workflow Management”. *The Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
- [2] Aalst, W.M.P. “Formalization and Verification of Event-Driven Process Chains”. *Information and Software Technology*, 41(10):639–650, 1999.
- [3] Aalst, W.M.P. “Workflow Verification: Finding Control-Flow Errors Using Petri-Net Based Techniques”. In Aalst, W.M.P., Desel, J., and Oberweis, A., editors, *Business Process Management – Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, 2000.
- [4] Aalst, W.M.P., Desel, J., and Oberweis, A.(Eds.). *Business Process Management – Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
- [5] Aalst, W.M.P and Hee, K. *Workflow Management: Models, Methods, and Systems*. MIT Press, Cambridge, MA, 2002.
- [6] Agerwala, T. “Putting Petri Nets To Work”. *Computer*, 12(12):85–94, 1979.
- [7] Arkin, A. “Business Process Modeling Language”. Technical report, Business Process Management Initiative, <http://www.bpmi.org>, 2003.
- [8] Banaszak, Z.A. and Krogh, B.H. “Deadlock Avoidance in Flexible Manufacturing Systems with Concurrently Competing Process Flows”. *IEEE Transactions on Robotics and Automation*, 6:724–734, 1990.
- [9] Basu, A. and Blanning, R.W. “Metagraphs: A Tool for Modeling Decision Support Systems”. *Management Science*, 40(12):1579–1600, 1994.
- [10] Basu, A. and Blanning, R.W. “Metagraphs in Workflow Support Systems”. *Decision Support Systems*, 25:199–208, 1999.
- [11] Basu, A. and Blanning, R.W. “A Formal Approach to Workflow Analysis”. *Information Systems Research*, 11(1):17–36, 2000.
- [12] Bedworth, D.D. and Bailey, J.E. *Integrated Production Control Systems: Management, Analysis and Design*. John Wiley & Sons, Inc., NY, 2 edition, 1987.

- [13] Borwein, J.M. “The Experimental Mathematician: The Pleasure of Discovery and the Role of Proof”. CECM Preprint 02:178, <http://www.cecm.sfu.ca/preprints/2002pp.html>, 2002.
- [14] Borwein, J.M. and Corless, R. “Emerging Methods in Experimental Mathematics”. *American Mathematical Monthly*, 106:181–194, 1999.
- [15] Casati, F., Ceri, S., Pernici, B., and Pozzi, G. “Conceptual Modeling of Workflows”. *Proceedings of the ODER’95, 14th International Object-Oriented and Entity-Relationship Modelling Conference*, Gold Coast, Australia, pages 341–354, 1995.
- [16] Caswell, N.S. and Nigam, A. “Operational Specification: A Technique for Business Process Integration and Analysis”. Invited paper at the INFORMS Fall 2002 Meeting, November 2002.
- [17] Chen, P. *A Use Case Driven Object-Oriented Design Methodology for the Design of Multi-Level Workflow Schemas*. PhD thesis, Department of Computer Science, Illinois Institute of Technology, Chicago, IL, 2000.
- [18] Cichocki, A., Helal, A., Rusinkiewicz, M., and Woelk, D. *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers, MA, 1998.
- [19] Coffman, E.G., Elphick, M.J., and Shoshani, A. “System Deadlocks”. *Computing Surveys*, 3:67–78, 1971.
- [20] Corless, R.M., Gonnet, G.H., Hare, D.E.G., Jeffrey, D.G., and Knuth, D.E. “On the Lambert W Function”. *Advances in Computational Mathematics*, 5:339–359, 1996.
- [21] CSC Corporation. “The Emergence of Business Process Management”. Research report, version 1.0, Computer Sciences Corporation, <http://www.bpmi.org>, January 2002.
- [22] Curtis, B., Kellner, M.I., and Over, J. “Process Modeling”. *Communications of the ACM*, 35(9):75–90, 1992.
- [23] Dalal, N.P., Kamath, M., Kolarik, W.J., and Sivaraman, E. “Toward an Integrated Framework for Modeling Enterprise Processes”. *Communications of the ACM*, 2003 (in print).
- [24] de Bruijn, N.G. “Additional Comments on a Problem in Concurrent Programming Control”. *Communications of the ACM*, 10:137–138, 1967.
- [25] Desel, J. and Esparza, J. *Free Choice Petri Nets*. Cambridge University Press, 1995.
- [26] Desrochers, A.A. and Al-Jaar, R.Y. *Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis*. IEEE Press, NJ., 1995.
- [27] Dijkstra, E.W. “Solution of a Problem in Concurrent Programming Control”. *Communications of the ACM*, 8:569, 1965.

- [28] Eisenberg, M.A. and McGuire, M.R. "Further Comments on Dijkstra's Concurrent Programming Control Problem". *Communications of the ACM*, 15:999–1000, 1972.
- [29] Ezpeleta, J., Colom, J.M., and Martinez, J. "A Petri Net Based Deadlock Prevention Policy for Flexible Manufacturing Systems". *IEEE Transactions on Robotics and Automation*, 11:173–184, 1995.
- [30] Fan, W. and Weinstein, S. "Specifying and Reasoning About Workflows with Path Constraints". In *Proceedings of the 5th International Computer Science Conference (ICSC'99)*, HongKong, China, volume 1749 of *Lecture Notes in Computer Science*. Springer, Dec. 13–15 1999.
- [31] Fujimoto, R.M. "Parallel Discrete Event Simulation". *Communications of the ACM*, 33:30–53, 1990.
- [32] Garey, M.R. and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman Co., San Francisco, 1979.
- [33] Georgakopoulos, D., Hornick, M., and Sheth, A. "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure". *Distributed and Parallel Databases*, 3:119–153, 1995.
- [34] Gold, E.M. "Deadlock Prediction: Easy and Difficult Cases". *SIAM Journal of Computing*, 7:320–336, 1978.
- [35] Habermann A.N. "Prevention of System Deadlocks". *Communications of the ACM*, 12:373–377, 385, 1969.
- [36] Harmon, P. "XML Business Process Languages". The Business Process Trends Newsletter, <http://www.bptrends.com>, March 2003.
- [37] Havender J.W. "Avoiding Deadlock in Multitasking Systems". *IBM Systems Journal*, 7:74–84, 1968.
- [38] Hofacker, I. and Vetschera, R. "Algorithmical Approaches to Business Process Design". *Computers & Operations Research*, 28:1253–1275, 2001.
- [39] Hofri, M. "The Deadlock Problem in Computing and Communication Systems – An Annotated Bibliography". Technical Report 500, Technion, Haifa, Israel, 1988.
- [40] Hofstede, A.H.M., Orłowska, M.E., and Rajapakse, J. "Verification Problems in Conceptual Workflow Specifications". *Data & Knowledge Engineering*, 24(3):239–256, 1998.
- [41] Hollingsworth, D. "The Workflow Reference Model". Technical Report WfMC-TC-1003, The Workflow Management Coalition, <http://www.wfmc.org/standards/docs.htm>, 1995.
- [42] Holt, R.C. "Comments on Prevention of System Deadlocks". *Communications of the ACM*, 14:36–38, 1971.

- [43] Holt, R.C. “Some Deadlock Properties of Computer Systems”. *Computing Surveys*, 4:179–196, 1972.
- [44] Hsu, M. “Special Issue on Workflow Systems”. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 16(2), 1993.
- [45] Hsu, M. “Special Issue on Workflow Systems”. *Bulletin of the Technical Committee on Data Engineering, IEEE*, 18(1), 1995.
- [46] Isloor, S.S. and Marsland, T.A. “The Deadlock Problem: An Overview”. *IEEE Computer*, 9:58–77, 1980.
- [47] Kamath, M., Dalal, N.P., Chaugule, A., Sivaraman, E., and Kolarik, W.J. “A Review of Enterprise Modeling Techniques”. In Prabhu, V., Kumara, S., and Kamath, M., editors, *Scalable Enterprise Systems – Recent Advances*. Kluwer Academic Publishers, 2003.
- [48] Kamath, M., Dalal, N.P., and Chinnanchetty, R. “The Application of XML-Based Markup Languages in Enterprise Process Modeling”. *Proceedings of the 11th Annual Industrial Engineering Research Conference*, Orlando, USA, 2002.
- [49] Kamath, M. and Sivaraman, E. “Analysis of Business Processes”. Invited paper at the INFORMS Fall 2002 Meeting, November 2002.
- [50] Kamath, M.U. *Improving Correctness and Failure Handling in Workflow Management Systems*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, MA, 1998.
- [51] Kamath, M.U. and Ramamritham, K. “Correctness Issues in Workflow Management”. *Distributed Systems Engineering Journal*, 3(4):213–221, December 1996. Special Issue on Workflow Systems.
- [52] Keller, G. and Detering, S. “Process-Oriented Modeling and Analysis of Business Processes using the R/3 Reference Model”. In Bernus, P. and Nemes, L., editors, *Modeling and Methodologies for Enterprise Integration*, pages 183–200. Chapman & Hall, U.K., 1996.
- [53] Kiepuszewski, B., Hofstede, A.H.M., and Aalst, W.M.P. “Fundamentals of Control Flow in Workflows”. Technical Report FIT-TR-2002-02, Queensland Institute of Technology, Brisbane, Australia, 2002.
- [54] Knuth, D.E. “Additional Comments on a Problem in Concurrent Programming Control”. *Communications of the ACM*, 9:321–322, 1966.
- [55] Knutilla, A., Schlenoff, C., Ray, S., Polyak, S.T., Tate, A., Cheah, S.C., and Anderson, R.C. “Process Specification Language: An Analysis of Existing Representations”. NISTIR 6160, National Institute of Standards and Technology, Gaithersburg, MD, 1998. <http://www.mel.nist.gov/psl/>.

- [56] Lamport, L. “A New Solution of Dijkstra’s Concurrent Programming Problem”. *Communications of the ACM*, 17:453–455, 1974.
- [57] Lawley, M.A. and Reveliotis, S.A. “Deadlock Avoidance for Sequential Resource Allocation Systems: Hard and Easy Cases”. *The International Journal of Flexible Manufacturing Systems*, 13:385–404, 2001.
- [58] Lee, D. and Kim, M. “A Distributed Scheme for Dynamic Deadlock Detection and Resolution”. *Information Sciences*, 64:149–164, 1992.
- [59] Lei, Y. and Singh, M.P. “A Comparison of Workflow Metamodels”. *Proceedings of the ER-97 Workshop on Behavioral Modeling & Design Transformation Issues and Opportunities in Conceptual Modeling*, Los Angeles, CA, November 1997. <http://osm7.cs.byu.edu/ER97/workshop4/ls.html>.
- [60] Leymann, F. and Roller, D. *Production Workflow: Concepts and Techniques*. Prentice-Hall, Inc., 2000.
- [61] Lin, H., Zhao, H., Li, H., and Chen, Z. “A Novel Graph Reduction Algorithm to Identify Structural Conflicts”. In *Proceedings of the 35th Annual Hawaii International Conference on System Science (HICSS-35’02)*. IEEE Computer Society Press, 2002.
- [62] Linthicum, D.S. *Enterprise Application Integration*. Addison-Wesley, Boston, USA, 2000.
- [63] Lynch, J.F. “Random Resource Allocation Graphs and The Probability of Deadlock”. *SIAM Journal on Discrete Mathematics*, 7:458–473, 1994.
- [64] Martinez, J. and Silva, M. “A Simple and Fast Algorithm to Obtain all Invariants of a Generalized Petri Net”. In Girault, C. and Reisig, W., editors, *Application and Theory of Petri Nets*, volume 52 of *Lecture Notes in Computer Science*, pages 301–311. Springer-Verlag, 1982.
- [65] Mayer, R.J., Menzel, C.P., Painter, M.K., deWitte, P.S., Blinn, T., and Perakath, B. “Information Integration for Concurrent Engineering (IICE) – IDEF3 Process Description Capture Report”. Technical report, KBSI Systems, Inc., College Station, TX, <http://www.idef.com>, 1995.
- [66] Mentzas, G., Halaris, C., and Kavadias, S. “Modelling Business Processes with Workflow Systems: An Evaluation of Alternative Approaches”. *International Journal of Information Management*, 21:123–135, 2001.
- [67] Minoura, T. “Deadlock Avoidance Revisited”. *Journal of the ACM*, 29:1023–1048, 1982.
- [68] Murata, T. “Petri Nets: Properties, Analysis, and Applications”. *Proceedings of the IEEE*, 77(4):541–580, 1989.

- [69] Parnas, D.L. and Habermann, A.N. “Comment on Deadlock Prevention Method”. *Communications of the ACM*, 15:840–841, 1972.
- [70] Peterson, J.L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., 1981.
- [71] Rardin, R.L. *Optimization in Operations Research*. Prentice-Hall, NJ, 1998.
- [72] Recihert, M. and Dadam, P. “ADEPT_{flex} – Supporting Dynamic Changes of Workflows Without Losing Control”. *Journal of Intelligent Information Systems*, 10:93–129, 1998. Special Issue on Workflow Management Systems.
- [73] Reveliotis, S.A. “Liveness Enforcing Supervision for Sequential Resource Allocation Systems: State of the Art and Open Issues”. In Cailaud, B., Xie, X., Darondeau, P., and Lavagno, L., editors, *Synthesis and Control of Discrete Event Systems*, pages 203–212. Kluwer Academic Publishers, 2002.
- [74] Reveliotis, S.A. and Ferreira, P.M. “Deadlock Avoidance Policies for Automated Manufacturing Systems”. *IEEE Transactions on Robotics and Automation*, 12:845–857, 1996.
- [75] Sadiq, W. *On Verification Issues in Conceptual Modeling of Workflow Processes*. PhD thesis, Department of Computer Science and Electrical Engineering, The University of Queensland, Australia, 2001.
- [76] Sadiq, W. and Orłowska, M.E. “Analyzing Process Models using Graph Reduction Techniques”. *Information Systems*, 25(2):117–134, 2000.
- [77] Salimifard, K. and Wright, M. “Petri-Net based Modeling of Workflow Systems: An Overview”. *European Journal of Operational Research*, 134(3):218–230, 2001.
- [78] Shapiro, R. “A Technical Comparison of XPDL, BPML, and BPEL4WS”. Business Process Trends – White Paper/Technical Brief, <http://www.bptrends.com>, 2002.
- [79] Sheth, A., Aalst, W.M.P., and Arpinar, I. “Processes Driving the Networked Economy”. *IEEE Concurrency*, 7(3):18–31, 1999.
- [80] Shooman, M.L. *Probabilistic Reliability: An Engineering Approach*. McGraw-Hill Book Company, NY, 1968.
- [81] Singh, M.P. “Formal Semantics for Workflow Computations”. Technical Report TR-96-08, Department of Computer Science, North Carolina State University, January 1996.
- [82] Singhal, M. “Deadlock Detection in Distributed Systems”. *IEEE Computer*, 22:37–48, 1989.
- [83] Straub, P. and Hurtado, C.L. “A Theory of Parallel Threads in Process Models”. Technical Report RT-PUC-DCC-95-05, Computer Science Department, Catholic University of Chile, Santiago, Chile, 1995. <ftp://ing.puc.cl/doc/techReports>.

- [84] Straub, P. and Hurtado, C.L. "The Simple Control Property of Business Process Models". *Proceedings of the XV Conference of the Chilean Computer Society*, Arica, Chile, Oct. 30-Nov. 3, 1995. <ftp://ing.puc.cl/doc/techReports>.
- [85] Straub, P. and Hurtado, C.L. "Avoiding Useless Work in Workflow Systems". *Proceedings of the International Conference on Information Systems Analysis and Synthesis*, Orlando, USA, July 22-26, 1996.
- [86] Straub, P. and Hurtado, C.L. "Business Process Behavior is (almost) Free-choice". *Computational Engineering in Systems Applications, Session on Petri Nets for Multi-Agent Systems and Groupware*, Lille, France, July 9-12, 1996. <ftp://ing.puc.cl/doc/techReports>.
- [87] Straub, P. and Hurtado, C.L. "Understanding Behavior of Business Process Models". *Proceedings of the First International Conference on Coordination Languages and Models*, Cesena, Italy, April 15-17, 1996. <ftp://ing.puc.cl/doc/techReports>.
- [88] Straub, P. and Hurtado, C.L. "Control in Multi-Threaded Information Systems". Unpublished working report, Computer Science Department, Catholic University of Chile, Santiago, Chile, 1997. <ftp://ing.puc.cl/doc/techReports>.
- [89] Stremersch, G. and Boel, R.K. "Structuring Acyclic Petri Nets for Reachability Analysis and Control". *Discrete Event Dynamic Systems*, 12:7-41, 2002.
- [90] Sugiyama, Y., Araki, T., Kasami, T., and Okui, J. "Complexity of the Deadlock Avoidance Problem". *Systems, Computers, Controls*, 8:44-51, 1977.
- [91] Veijalainen, J., Lehtola, A., and Pihlajamaa, O. "Research Issues in Workflow Systems". *Proceedings of the 8th ERCIM Database Research Group Workshop on Database Issues and Infrastructure in Cooperative Information Systems* Trondheim, Norway, August 1995. <http://www.ercim.org/publication/ws-proceedings/8th-EDRG/8th-EDRG-contents.html>.
- [92] Venkatesh, S. *Deadlock Detection and Resolution in Discrete Event Simulation and Shop Floor Control*. PhD thesis, Department of Industrial Engineering, Texas A&M University, College Station, TX, 1996.
- [93] Vernadat, F.B. "CIM Business Process and Enterprise Activity Modeling". In Bernus, P. and Nemes, L., editors, *Modeling and Methodologies for Enterprise Integration*, pages 171-182. Chapman & Hall, U.K., 1996.
- [94] Viswanadham, N., Narahari, Y., and Johnson, T.J. "Deadlock Prevention and Deadlock Avoidance in Flexible Manufacturing Systems Using Petri Net Models". *IEEE Transactions on Robotics and Automation*, 6:713-723, 1990.
- [95] WfMC. "Terminology & Glossary". Technical Report WfMC-TC-1011, The Workflow Management Coalition, <http://www.wfmc.org/standards/docs.htm>, 1999.

- [96] WfMC Work Group 1. "Interface 1: Process Definition Interchange Process Model". Technical Report WfMC-TC-1016-P, The Workflow Management Coalition, <http://www.wfmc.org/standards/docs.htm>, 1999.
- [97] Wilf, H. *Algorithms and Complexity*. Prentice-Hall, Inc., NY., 1986.
- [98] Wojcik, B.E. and Wojcik, Z.M. "Sufficient Condition for Communication Deadlock and Distributed Deadlock Detection". *IEEE Transactions on Software Engineering*, 15:1587–1595, 1989.
- [99] Xing, K.Y., Hu, B.S., and Chen, H.X. "Deadlock Avoidance Policy for Petri-Net Modeling of Flexible Manufacturing Systems with Shared Resources". *IEEE Transactions on Software Engineering*, 15:1587–1595, 1989.

VITA²

Eswar Sivaraman

Candidate for the Degree of
Doctor of Philosophy

THESIS: FORMAL TECHNIQUES FOR ANALYZING BUSINESS PROCESS MODELS

MAJOR FIELD: Industrial Engineering & Management

BIOGRAPHICAL:

Personal Data : Born in Kumbakonam, Tamil Nadu, India, September 15, 1975, son of Mrs. K. Vasantha and Mr. E. Sivaraman.

Education : Graduated from Padma Seshadri Bala Bhavan Senior Secondary School, Madras, India, in May 1992; received the Bachelor of Technology degree in Manufacturing Engineering from National Institute of Foundry & Forge Technology, Ranchi, India, in May 1996; received the Master of Science degree in Industrial Engineering & Management from Oklahoma State University in May 1998; completed requirements for the Doctor of Philosophy degree at Oklahoma State University in May 2003.

Experience : Graduate Teaching Assistant, School of Industrial Engineering and Management, Oklahoma State University, Fall 1996 – Spring 2003; Graduate Research Associate, Center for Computer Integrated Manufacturing, Oklahoma State University, Fall 1998 – Spring 2003; Summer Intern (Mathematical Programming), CIENA Corporation, Cupertino, CA, June 2000 – July 2000.

Honors : OSU Graduate Research Excellence Award for outstanding Ph.D. Dissertation, Spring 2003; OSU Graduate Research Excellence Award for outstanding M.S. Thesis, Spring 1998; Phi Kappa Phi, Spring 2003; Pi Mu Epsilon, Fall 1998; Alpha Pi Mu, Fall 1997.

Professional Membership : INFORMS, IIE, and AMA.