The Pennsylvania State University

The Graduate School

The Mary Jean and Frank P. Smeal College of Business Administration

ESSAYS IN COLLABORATIVE SUPPLY CHAINS: INFORMATION SHARING, EVENT MANAGEMENT AND PROCESS VERIFICATION

A Thesis in

Business Administration

by

Rong Liu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2006

The thesis of Rong Liu was reviewed and approved* by the following:

Akhil Kumar
Professor of Information Systems
Thesis Advisor
Chair of Committee

Dawn M. Russell
Assistant Professor of Supply Chain and Information Systems

Alan J. Stenger
Professor of Supply Chain Management

John Yen
University Professor of Information Sciences and Technology

John E. Tyworth
Professor of Supply Chain Management
Chair of the Department of Supply Chain and Information Systems

*Signatures are on file in the Graduate School

ABSTRACT

With the advent of information technology in a highly interconnected economy, supply chains are required to be agile and adaptive. An agile and adaptive supply chain can sense changes or events in real time and respond to them quickly by self adjustment. This adjustment involves changes in information sharing and supply chain processes. Therefore, *information sharing*, *event management*, and *process modeling and verification* play an important role. In this dissertation, each of these issues is addressed in a separate essay. Therefore, this dissertation consists of three interrelated essays.

The first essay describes a new methodology for achieving supply chain configurations by dynamic information sharing. We propose a parameterized model to analyze information sharing. By changing the parameters of this model, we actually adjust information sharing needs and achieve different supply chain configurations, which are further evaluated in terms of supply chain performance. When events or changes bring new information sharing needs, we can respond to the needs with suitable configurations.

The second essay focuses on supply chain events. A formal approach based on colored time Petri nets is developed to model and analyze events. Based on Petri net models, we derive dependency graphs to analyze causal relationships between events. We also perform sensitivity analysis to show the impact of different event resolution strategies on supply chain performance.

The third essay models and verifies supply chain processes based on workflow technologies. Inter-organizational processes are usually complicated and unstructured.

We describe a taxonomy that serves as a framework for analyzing unstructured workflows. The taxonomy characterizes unstructured workflows in terms of two considerations: improper nesting and mismatched pairs. Based on this taxonomy, we develop a diagnosis algorithm that can detect structural flaws, show causes of flaws and draw conclusions on workflow correctness.

Based on the results of these three essays, we are able to develop a technical architecture for agile and adaptive supply chains. This architecture contains an event management engine to detect and analyze events. Suitable configurations are selected in response to events. Finally, processes are carefully verified to ensure accurate information sharing and supply chain configurations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my gratitude to all people who have given me great support during my five-year journey. It would be impossible for me to complete this thesis alone without their generosity and kindness.

The first person I would like to thank is my advisor, Dr. Akhil Kumar. I deeply thank him not only for gently, patiently, yet firmly guiding me towards the completion of this thesis from the very beginning, but also for his care of my professional development and valuable training in pursuing my personal and professional goals in my life. His generous support and warm encouragement has always been a great source of strength and confidence for me over the years. I also would like to thank my committee members, Dr. Dawn Russell, Dr. Alan Stenger and Dr. John Yen for being supportive and providing constructive comments on my thesis. Special thanks to Dr. Dawn Russell for her weariless counseling and emotional support.

I also extend my gratitude to Dr. W.M.P. van der Aalst of Eindhoven University of Technology, Netherlands, whom I never met but worked with on some portion of this thesis and patiently answered my questions on Petri Nets. In addition, grateful thanks to IBM for providing generous financial support to this research.

A hearty thanks to the department of Supply Chain and Information Systems for the generous financial support during my graduate study. Special thanks to Beth Bower, Alice Young, Teresa Lehman and LuAnn Jaworski for their superb administrative assistance and convenience provided to make my life at the office enjoyable and hassle free.

I am very grateful to my officemate, Dr. Kusumal Ramsook, for her years of warm-hearted support and friendship ever since we became to know each other.

Finally, I would like to thank my big family that stand behind me back in China. My deep and everlasting thanks go to my father, who never had a chance to see me reach this point, but I know he has always been watching me there ever since I was born. Always thank my husband for his unconditional and endless love. He is my outstanding proofreader and also the one who sat close to me and accompanied me in writing this thesis through numerous long nights.

Chapter 1

Introduction

Supply chains have been transformed completely with the advent of sophisticated information technology in highly interconnected event-driven economy [56]. In the event-driven economy, supply chains should be able to react responsively to internal or external *events* in order to meet consumers' demand. This reflects a need for sense-and-respond capability [32]. For example, in an integrated supply chain with tight delivery times and low tolerances, unexpected events or exceptions occur almost regularly because of gaps between planning and actual execution in a dynamic environment [11]. Even a minor unexpected event such as a late arrival of a shipment or a machine breakdown can propagate in such a supply chain and has far-reaching effects, such as the well-known bullwhip effect [48]. Therefore, a supply chain must be able to handle such events responsively and appropriately [42]. It is well known that *information sharing* plays a key role in achieving this capability [30, 52, 56]. Supply chain partners can share relevant information in a wide range of areas, such as demand, inventory and shipment status as well as strategic information, including market trends and production capabilities, in order to respond to inevitable unanticipated events.

Moreover, to facilitate prompt and accurate information sharing, business processes are integrated across supply chain partners [56], resulting in complicated inter-organizational processes. It is essential that a process should not only precisely capture business requirements but also ensure successful execution. A problematic process could

fail to execute properly and cause considerable loss to a business. As integrated processes become more complex, *process verification* assumes greater importance.

1.1 Problems to Be Addressed

In this research, we attempt to address the three related issues in collaborative supply chains: *information sharing*, *supply chain event management*, and *process modeling and verification*. As supply chains evolve beyond the confines of individual organizations, information sharing has become the Holy Grail in supply chain technology. Although the value of information sharing is well recognized, there is little research on how to achieve dynamic information sharing. Dynamic information sharing requires that when events or changes in supply chains raise new information sharing needs, information sharing is adjusted in a timely manner and supply chain processes are also adjusted accordingly in support of the needs. We investigate how to leverage information sharing dynamically in response to events or changes in supply chains.

Second, to respond to events quickly and appropriately, we need to understand events, especially, their causes and consequences. A supply chain can generate a large number of events. Some events have significant consequences while others may be trivial. A methodology is needed to extract significant events and suggest effective solutions to them promptly. We design a formal approach to modeling supply chain events. This approach can facilitate cause-effect analysis to achieve real-time visibility about the implications of events and also traceability to the root causes of events.

Moreover, this approach can help decision makers to evaluate event resolution strategies in terms of supply chain performance through simulation.

Finally, integrated supply chain processes must be carefully verified to ensure that information is exchanged among supply chain partners as required. Workflow technology has emerged as an important tool for businesses to integrate processes across supply chains. A correct workflow represents a process which can be executed properly without any deadlocks or other structural flaws. One accepted notion of correctness is structuredness. Structured workflows are always correct but they are restrictive because strict rules must be followed during the workflow design phase. Inter-organizational processes, which are typically integrated from individually designed processes, may not follow these rules and tend to be unstructured. Therefore a systematic approach is needed to analyze and verify unstructured workflows. We develop a taxonomy that serves as a framework for analyzing unstructured workflows. The taxonomy characterizes unstructured workflows in terms of two considerations: improper nesting and mismatched pairs. This taxonomy can be used to detect structural flows, give correction suggestion on structural flaws, and analyze the possibility of transforming an unstructured workflow to a structured one. An analysis algorithm for unstructured workflows is developed based on this taxonomy.

Figure 1-1 shows the three problems to be addressed. As this figure shows, supply chain events are analyzed and significant events are extracted. To respond to these events, information sharing needs may be changed. Therefore, information sharing is adjusted in terms of relevancy, timeliness, format, etc. As a result, processes, especially, inter-organizational processes, may be changed accordingly in order to meet these

information sharing needs. These processes should be verified carefully to ensure their correct execution.

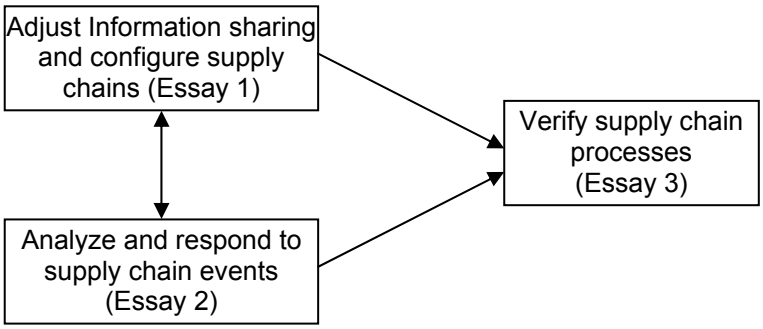


Figure 1-1: Problems to Be Addressed

1.2 Thesis Structure

This dissertation consists of three separated, but interrelated essays, as shown in Figure 1-1. These three essays are organized into three chapters (Chapters 2-4). Each chapter contains a separate research paper with its own introduction, main body and a conclusion.

The first essay (Chapter 2) studies dynamic information sharing. It proposes a parameterized model to analyze information sharing and develops a methodology for designing supply chain configurations. By changing the parameters of this model, we actually adjust information sharing needs, and as a result, achieve different supply chain configurations. A supply chain configuration refers to a set of supply chain activities, the specific pattern of inter-organizational linkages and information sharing between them. Supply chain configurations are the means of responding to events or changes in supply chains quickly and appropriately. A complete example is used to demonstrate this

methodology. We also give results of simulation experiments to compare various supply chain configurations and to understand the effect of dynamic information sharing on the performance of a supply chain. Thus, we show how to achieve supply chain configurability by leveraging information sharing.

The second essay (Chapter 3) studies supply chain events systematically. A formal approach based on Petri net technologies is developed to model and analyze events. Moreover, seven basic patterns are used to capture modeling concepts that arise commonly in supply chains. A detailed example is provided to show how to combine these patterns to build a complete Petri net and analyze events using dependency graphs and simulation. Dependency graphs can be used to analyze the various events and their causes. This study shows that through this approach, supply chains are able to track the causes of events and forecast events with precise time information, then making a complex problem tractable. Simulation is, in addition, used to analyze various supply chain performance metrics (e.g., fill rates, replenishment times, and lead times) under different event resolution strategies. Using sensitivity analysis, we can show the effect of changing parameter values of events on supply chain performance metrics. Therefore, we can actually manage supply chain performance by managing supply chain events.

The third essay (Chapter 4) models and verifies supply chain processes based on workflow technologies. Most workflow tools support structured workflows despite the fact that unstructured workflows can be more expressive. The reason for this is that unstructured workflows are more prone to errors. In this essay, we describe a taxonomy that serves as a framework for analyzing unstructured workflows. The taxonomy characterizes unstructured workflows in terms of two considerations: improper nesting

and mismatched split-join pairs. Based on this taxonomy we categorize workflows that are correct and others that are not. We also introduce a relaxed notion of correctness called weak correctness, as opposed to the conventional notion of strict correctness. Then, we develop a framework for analyzing unstructured workflows in terms of weak correctness and give an algorithm for diagnosing workflows. The diagnosis algorithm detects structural flaws and provides a detailed report giving causes for deadlocks and other structural flaws. In addition, this algorithm gives suggestions on how to correct those structural problems. The results of our research will be useful for researchers investigating expressiveness and correctness issues in unstructured workflows.

Finally, Chapter 5 concludes this dissertation and summarizes the contributions. Moreover, the implications of this essay to supply chain management practices are discussed. The limitations of this thesis and our future work are also described in this chapter.

Chapter 2

Information Sharing in Supply Chains: Modeling, Configurations, and Results

Abstract: As supply chains evolve beyond the confines of individual organizations, information sharing has become the Holy Grail in supply chain technology. Although the value of information sharing is well recognized, there is little research on how to use it to configure supply chains. This essay proposes a parameterized model to capture information sharing in a supply chain. By changing the parameters of this model, we actually adjust information sharing and create supply chain configurations. Configurations are the means of responding to events or changes in supply chains in a timely manner. A complete example is used to demonstrate this methodology. We also perform simulation experiments to compare configurations and to understand the effect of information sharing on supply chain performance.

2.1 Introduction

In recent years, the competitive business environment, marked by the acceleration of globalization and increasing customer demand for ever higher level of service, has forced companies to reduce costs while still providing high quality products and services in ever greater variety and customizability. This challenge has compelled companies to improve their supply chains, not only to optimize the internal logistic functions, but also to build real-time *collaboration* across organizations. Supply chain collaboration can be defined as a means by which all companies in a supply chain work together for mutual gains and is characterized by information sharing [9, 26, 72]**.** Research has shown that through information sharing, companies can establish strategic partnerships, develop

supply chain plans jointly, coordinate their processes, and create efficiencies and cost savings in the entire supply chain [37, 72].

Information sharing is also closely related to *coordination*. Coordination is defined as managing dependencies between activities [62]. Malone et al. [63] give three basic types of dependencies: *flow*, *sharing*, and *fit*. A supply chain involves all these three types of dependencies. Information sharing plays a major role in improving supply chain coordination. With *flow dependencies*, one supply chain activity produces resources, say raw materials, which are used by another activity. Sharing schedules and status information can make these activities better coordinated. *Sharing dependencies* occur when multiple supply chain activities all use the same resource, such as common parts or shipping capabilities. To manage sharing dependencies, the resource provider can share its supply condition or it can have knowledge about the incoming demand from these activities. Finally, multiple activities have a *fit dependency* if they collectively produce a single resource. For example, collaborative design involves such dependencies. As a supply chain practice, some manufacturers invite important suppliers to work on collaborative designs in order to come up with solutions which match supply more closely with demand [52, 74]. Typically, joint plans can be a coordination mechanism for activities with this type of dependencies. As a result, information sharing leads to improved supply chain coordination, better-aligned activities, and then streamlined supply chain processes.

Also, it is obvious that *supply chain visibility* is not possible without information sharing. Supply chain visibility means the ability to see from one end of a supply chain to the other [23]. It implies a clear view of upstream and downstream inventory, demand

and supply conditions, and schedules and status of different supply chain activities. Supply chain visibility is viewed as an effective way to reduce uncertainties in supply chains [50, 96]. In recent years, uncertainties have become a major concern in supply chains. The direct consequences of uncertainties are increased inventory and information distortion (such as poor demand forecasts). Moreover, the distortion propagates through a supply chain and is amplified at each stage – the well-known *bullwhip effect* [48], which has been identified as one of the biggest causes of inefficiencies in a supply chain. Through information sharing, the demand information flows upstream from the points of sale, while product availability information flows downstream in a systematic manner [50, 96]. Moreover, Gosain et al. [30] showed that information sharing can increase supply chain flexibility, the extent to which supply chain linkages are able adapt to changing business conditions. In addition, information sharing can lead to new knowledge creation in supply chains [61].

However, as the level of collaboration increases, shared information tends to be richer and more diverse. A critical issue is how to manage information sharing so that companies have enough visibility about the status of the whole supply chain, and yet the volume of shared information should not be overwhelming [61]. More importantly, shared information is "relevant enough and generated frequently enough so that partners can make decisions that compensate for the inevitable unplanned occurrences" [26]. This requires companies to adjust their information sharing (e.g., by relevancy, frequency, accuracy, aggregation level, etc.) in a timely manner in response to various events or exceptions in supply chains. Such an adjustment may result in changes of a supply chain process, such as changes of constituent activities, changes of activity execution

sequences, and new exception handling processes. Malhotra et al. [61] also pointed out that supply chains need to architect inter-organizational processes to coordinate information exchange. We view such changes as new *supply chain configurations* (or simply *configurations*).

A supply chain configuration refers to a set of supply chain activities, the specific pattern of inter-organizational linkages and the coordination mechanisms, especially information sharing among those activities. In general, supply chain configurations reflect a supply chain's experience of reacting to events or changes and inferences can be derived from them in response to similar events or changes in the future. In that sense, supply chain configurations can be referred to as a part of "organizational memory" [30, 61]. More precisely, they are "memory" acquired by a supply chain.

In this essay, we approach this goal of designing supply chain configurations by leveraging information sharing. There is growing interest in infrastructures and frameworks for information sharing in supply chains [13**,** 36]. We use a parameterized information sharing model to describe information sharing involved in an inter-organizational process and show how to modify parameters to adjust information sharing and then achieve new supply chain configurations. In particular, different configurations can be evaluated by simulation in terms of supply chain performance. When events or changes in information sharing needs are sensed, we find the appropriate configurations or create new configurations by modifying inter-organizational processes or the information sharing model in response to the changes. Figure 2-1 gives an overview of this methodology. We will illustrate this methodology with a complete example and the implementation of this methodology is also discussed.

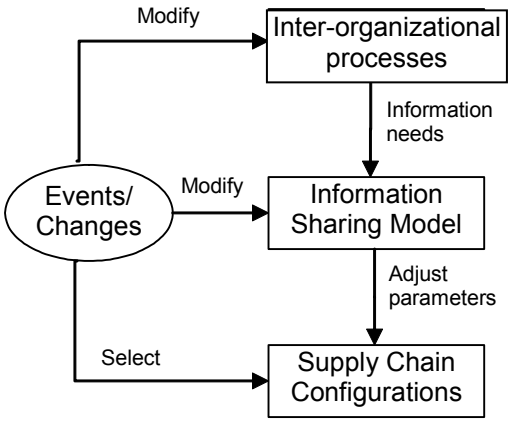


Figure 2-1: Overview of Supply Chain Configurations

The remainder of this essay is organized as follows. The next section introduces the concept of supply chain configurability in SCOR model and other related work. The information sharing model is described in Section 2.3. In Section 2.4, our methodology for information sharing is illustrated by a complete example. Section 2.5 describes and analyzes the results of our simulation in detail. The implementation of this methodology is briefly discussed in Section 2.6. Section 2.7 concludes the essay and briefly describes our planned future work.

2.2 SCOR Model and Other Related Work

The concept of *supply chain configurability* is introduced in the Supply Chain Operations Reference (SCOR) model [87]. SCOR is a business process reference model that provides a framework for configuring supply chains to achieve a set of specific goals in terms of performance metrics. SCOR contains four main levels of process details. It consists of five core management processes at the top level (i.e., process type level): plan,

source, make, deliver and return. The second level of SCOR, the configuration level (i.e., process category level) contains 30 core process categories that can be used to configure a supply chain. At this level, organizations can configure their ideal or actual operations by combining the generic SCOR process categories such as *make-to-stock, make-to-order, deliver stocked products, deliver made-to-order products*, etc. The third SCOR level (i.e., process element level) describes the processes in more detail using process elements. It can be used to fine tune the operations of a company by associating performance metrics for processes and process elements, and recording a set of best practices for each process. Actual implementation of supply chain management practices occurs at Level 4 (i.e., implementation level) of the SCOR model, but the details are not specified.

According to the SCOR model, supply chain configurability means companies in a supply chain are able to select proper *core process categories*, *set up collaboration*, and *coordinate their inter-organizational processes* mainly by *plans*, when new trigger events call for new configurations. For example, in a supply chain with three companies, a supplier, a manufacturer, and a distributor, each company may select different process categories as shown in Table 2-1 [12]. This configuration may need to be updated when events, such as new product releases, new facility deployment, or channel design changes, occur. For instance, if the distributor requires the manufacturer to do customization (say, install customized software), the delivery process at the manufacturer side is changed from *D2* (Delivery make-to-order product) to *D3* (Delivery engineer-to-order product).

Table 2-1: A Sample Supply Chain Configuration

Supplier	Manufacturer	Distributor
S1 (Source stocked product) *M1* (Make stocked product) *D1* (Deliver stocked product)	*S2* (Source make-to-order (MTO) product) *M2* (Make MTO product) *D2* (Deliver MTO product)	*S1* (Source stocked product) *D3* (Deliver stocked product)

Thus, the SCOR model emphasizes supply chain configurations in terms of process categories. It allows companies to implement their operation strategies through the configurations they choose for their supply chains at Level 2**.** In addition, the supply chain processes are coordinated mainly by *joint supply chain plans*. Obviously, this configuration is at a higher level than that at the detailed process element level, or even the implementation level. Our goal is to address configuration issues at Levels 3 and 4. Moreover, we analyze the needs of information sharing and use it to fine tune supply chain processes in response to external or internal events.

Therefore, we extend the concept of supply chain configurability defined in the SCOR model. We define *supply chain configurability* as: *the extent to which a supply chain can change process categories, fine tune process elements, dynamically adjust process implementation, and coordinate inter-organizational processes, in response to internal and external events, by leveraging information sharing between organizations.*

Our definition not only includes configurability at a higher process category level, but also emphasizes it at lower levels, including the process element and implementation levels. Configurability at all the three levels is necessary for greater responsiveness and flexibility of a supply chain. Because of the challenging business environment, supply chains are constantly facing a variety of changes, both expected and unexpected.

Moreover, not every change has strategic implications that would lead to a new supply chain configuration at the process category level. Most changes are short-term, have an impact on tactical or operational decisions, and need a timely response. Rather, a new configuration at the process category level is usually achieved in a relatively long time and at a great cost, while on the contrary, a new configuration at the lower levels only involves tuning specific process elements, adjusting implementation practices, reconfiguring information sharing, and making other changes that can be accomplished quickly and flexibly. Therefore, configurability at the lower levels provides supply chains the capabilities of responding to short-term changes and handling external disruptions smoothly, fostering *agility* [52].

Some other related work includes a conceptual sense-and-adapt framework for dynamic adjustment with organizational memory [30]. Research shows that organizational memory allows organizations to recognize types of adjustments needed in response to events or changes [30, 61]. Still, we lack a detailed methodology for storing and utilizing organizational memory for supply chains. In this essay, we develop such a methodology based on information sharing. Haeckel [32] suggested a framework for adaptive enterprises which emphasizes that a company needs to continuously reengineer itself in response to customers' changing needs. Also, a technical framework for sense-and-respond business management is proposed in [42].

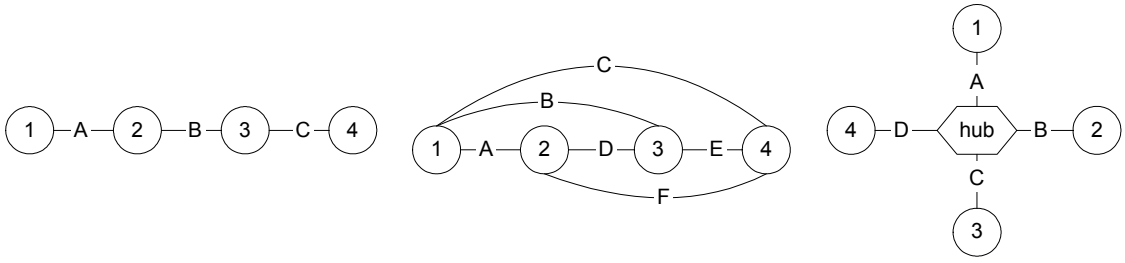
Next, we will introduce an information sharing model and show how to derive various supply chain configurations from this model.

2.3 Information Sharing Model

In this section, we describe a modeling approach for information sharing. It is primarily based on events, conditions and information flows. But first we introduce several information sharing structures and data objects.

2.3.1 Information Sharing Structures

Information sharing simply means sending *data* from one partner to another or joint creation of data objects by two or more partners. Consequently, *data objects* are exchanged between partners by means of *information flows*. In a supply chain consisting of multiple partners, typically, information sharing can be conducted in three different structures: *sequential*, *reciprocal*, and *hub-and-spoke*, as shown in Figure 2-2. More sophisticated or hybrid structures can be obtained by combining these basic types.



(a) Sequential (b) Reciprocal (c) Hub-and-spoke

Figure 2-2: Information Sharing Structure

(1) *Sequential Information Sharing*: In this structure, information is only shared between two neighboring partners. For example, in Figure 2-2(a), Partner 1 and Partner 2 share data item *A*, and Partners 2 and 3 share data item *B*. The information flows link the

processes of different companies together into a sequential chain. The traditional arm's length or transactional partnerships [72] can fit into this structure. For example, retailers send replenishment orders to distributors and distributors send aggregated replenishment orders to suppliers, but there is no direct information sharing between retailers and suppliers. This structure is the simplest arrangement to implement. Each pair of partners can establish their own protocols for exchange without the need for any universal standard. They could rely on EDI or some other communication mechanisms.

(2) *Reciprocal Information Sharing*: This is a more complex information sharing structure. Each partner may communicate with several others. For example, as shown in Figure 2-2(b), Partner 1 can share data *A* with Partner 2, *B* with Partner 3, and *C* with Partner 4. There are multiple data objects shared in the whole supply chain. For example, to reduce the bullwhip effect, retailers can share their actual usage with wholesalers, distributors, manufacturers, and any other upstream partners. Another example is that all partners in a supply chain can jointly develop supply chain plans to optimize the performance of the whole supply chain. Difficulties of this structure lie in different formats of shared data objects and possible inconsistencies between the information of different partners. For example, in Figure 2-2 (b), if data objects *B* and *D* are related, it is possible that *B* from Partner 1 and *D* from Partner 2 are inconsistent. The next structure, hub-and-spoke, which uses a hub as an intermediate to consolidate data objects, is able to deal with such inconsistencies.

(3) *Hub-and-spoke Information Sharing*: This arrangement is based on a central hub which communicates with all partners, as shown in Figure 2-2(c). In general, an

Internet-based e-hub in this architecture serves as a virtual marketplace, thus facilitating a full range of business processes and interactions between trading partners. The hub coordinates, stores, aggregates and maintains information about each partner, makes decisions, and then communicates them to all partners. Similarly, the hub has the ability to manage events and respond to them dynamically. For example, Covisint [24] launched by DaimlerChrysler, Ford, General Motors and other automakers, is based on the idea of such a centralized hub.

Information sharing structures determine information flows between partners. Any change to data objects likely causes new information flows. For example, in Figure 2-2(a), if data object *A* is generated by Partner 1, when *A* is changed, Partner 1 sends the update to Partner 2 as an information flow. Because of this change, Partner 2 may generate another information flow that sends the update of *B* to Partner 3. Therefore, the dependencies between shared data objects decide the sequences of information flows. In addition, these dependencies also determine the complexity of information flows. For example, in a reciprocal structure, if all data objects are interrelated, the information flows could be very complicated.

2.3.2 Information Sharing Objects

It has been recognized that supply chain partners need to share information in a broad range of situations and areas, including both operational information such as inventory status and shipment status, and strategic information such as market trends, production capabilities and exceptions in order to respond to unanticipated changes in

supply chains promptly and appropriately [30, 61]. However, in general, the types of data objects shared between partners depend in large part on the nature of the relationship that might range from a traditional arm's-length relationship to one in which the partners are tightly integrated. In an arm's-length relationship, information is transmitted only for executing transactions, while information sharing in tightly integrated collaborative arrangements aims at improving supply chain synchronization and optimizing the whole supply chain instead of an individual partner [12, 25, 72]. For example, single orders placed by a buyer with a seller cause transactions to happen just between these two partners, but the aggregated orders, if shared with any upstream partners, can increase supply chain visibility. Therefore, it is important to determine what data objects are shared (in terms of relevance, accuracy, completeness, etc), how they are shared (in terms of frequency, timeliness, etc.), and the impact of the sharing of these data objects on supply chain performance (in terms of feedback and evaluation). These requirements can be generalized as the dimensions of information quality [69]. In general, the quality of shared information can be a major concern in supply chain collaboration. We will propose a model which can capture these dimensions explicitly.

Lee and Whang [49] summarize six types of shared information: *inventory level*, *sales data*, *order status for tracking/tracing*, *sales forecast*, *production schedule*, and other information such as *performance metrics* and *capacity*. Other types could be *information about the occurrence of events or changes* in supply chain environment, *joint supply chain plans* or *business plans*, and *product information* such as *catalog* and *product design* which is used for collaborative design [58].

Since shared data objects range from structured data like orders to unstructured data like ad hoc product designs, standardization of data format is important. XML is becoming a de facto standard for exchanging messages. Moreover, XML-based data standards are preferable because of their flexibility [8]. Partners can define their own XML data standards and conveniently convert data between XML and relational databases. Later on, we will show how XML-based data templates can facilitate information flows conveniently between partners.

2.3.3 Information Sharing Modeling

Next, we will introduce a modeling approach for information sharing. We extend Event-Condition-Action (ECA) rules [65] to information sharing. ECA rules provide a formalism for active database capabilities. *An ECA rule specifies that when an event occurs and if certain conditions hold, a specific action is executed.* Actions are typically the operations of databases, such as query, update, insertion, and deletion. In our context, actions mean sending information flows. Moreover, an information flow can be decomposed into a set of parameters, such as sender, receiver and shared data objects. Therefore, information sharing between partners can be described in terms of the following parameters: *events*, *conditions*, *information flows* (*senders*, *receivers*, *data objects*, *data templates*, *requested recipient actions*, *frequency*, *batch/real-time*, *aggregation levels*). The main advantage of this parameterized approach is that information sharing can be leveraged by adjusting these parameters, as we will show in the next section. Next, we describe different parameters with details.

(1) Events

Events are signals for information flows to occur. Typically, there are three types of primitive events. Composite events can be obtained by combining primitive events using disjunction and sequence operators. Events can be assigned different priorities. In addition, arguments may be attached to events. Three primitive events are described as follows.

- *Data change event*. Changes may involve adding or updating shared data objects. For example, if two partners have a joint supply chain plan, any partner who updates this plan must notify the other by sending an information flow. In addition, receiving a shared data object (e.g., a ship notice) is also an event. The receiver may respond to it by sending an information flow (e.g., a goods receipt). The arguments for such events can be shared data objects. For example, the event “ship notice received” can be formulated as "Data_received('ship notice') ".
- *Temporal events*. Some information flows may be triggered by temporal events. For example, a retailer may agree to send its weekly usage to a supplier at 5 PM on every Friday. Further, temporal events can be *absolute, relative,* or *periodic*.
- *Exceptions*. Exceptions are unexpected supply chain events. A supply chain event is "any individual outcome (or non-outcome) of a supply chain cycle, (sub) process, activity, or task" [6]. In this essay, we will focus on exceptions related to supply chain performance. Such an exception could also cause the adjustment of information sharing between partners. For example, in a Vendor Managed Inventory (VMI) arrangement, if sharing usage weekly leads to high forecast errors, the vendor may

ask the retailer for real-time usage in order to improve its forecast precision. Exception events normally receive higher priority than normal events.

(2) Conditions

Conditions are a collection of queries on shared data objects. If all shared data objects are XML documents, those queries can be defined using XPATH or XQUERY [95]. For example, the condition "*inventory level of an item*<3500" can be evaluated using an XPATH statement like "*Inventory*/*item*[*quantity*<3500] ".

(3) Information Flows

When an event occurs and specific conditions are satisfied, an associated information flow is sent out. Information flows can be further decomposed into a set of parameters described as follows. In general, these parameters reflect the requirements for the quality of shared information. Some parameters are mandatory and the others are optional. Optional parameters describe some properties of information sharing. More parameters pertaining to describe information flows can also be added when necessary.

Mandatory parameters:

- *Sender and Receiver*(*s*): are the communicating partners. In general, an information flow can have more than one receiver. The sender and receiver parameters show the structure of information sharing.
- *Data objects* (*Data_Obj*): contain the data to be shared. Shared data objects should be relevant to a specific collaborative scenario. In addition, the data should be *accurate* and *complete*. In a dynamic supply chain, information *relevant* to one situation may

be irrelevant to another. Therefore, information sharing needs should be analyzed and adjusted in a timely manner.

- *Templates*: give the formats of data objects, such as EDIFACT, XML and other data standards. Since a variety of data objects is shared and those data objects may be in different formats, to make receivers understand them, a possible solution is to exchange their data templates along with the data. Therefore, in this model, data templates are shared by senders and receivers as a parameter of information flows. Also, the *compatibility* of templates should be considered. For example, XML is a well-known standards and it could be a good option for providing data compatibility.

Optional parameters:

- *Requested recipient action* (*Req_action*): the actions taken by the recipient after the flow is received.
- *Frequency*: the frequency of updating shared data objects, e.g. daily, weekly, monthly, etc.
- *Batch/Real-time*: this is the mode in which the information is transferred. Along with the frequency parameter, this parameter specifies the requirement for the *timeliness* of shared information.
- *Level of aggregation*: the aggregation level may be transactional (each POS transaction), per item or per brand etc. This parameter further specifies the *relevance* requirement of shared information.

Information flows can be linked together by means of events and associated conditions. When a flow occurs, it can generate an event indicating some changes to shared data objects or prompt the recipient to take action on it, and perhaps another flow

is then generated if the corresponding conditions are satisfied. Table 2-2 shows two example information flows. The first flow is triggered by a temporal event. The second flow is initiated when a data object is received and an associated condition holds at that time. The details of these two flows will be described later. Next, we will describe our methodology for generating an information sharing model and using this model to achieve new supply chain configurations.

Table 2-2: Examples of Information Flows

1. sendUsage		**2. proposeOrder**	
Event:	Friday, 5 pm (Temporal event)	***Event:***	Weekly Usage received
Information flow		***Condition:***	Item inventory < 3500
Sender:	Customer	***Information flow***	
Receiver:	Vendor	*Sender:*	Vendor
Data_Obj:	Weekly Usage	*Receiver:*	Customer
Template:	EDI #852	*Data_Obj:*	Replenishment order
Req_Action:	Propose order	*Template:*	EDI #855
Batch/ Real-time:	Batch	*Req_Action:*	Accept/Modify order
Level of aggregation	Aggregated by item	*Batch/ Real-time:*	Real-time

2.4 Configuring Supply Chain Processes

2.4.1 Methodology

In this section, we will discuss how to apply the parameterized approach to supply chain processes and the kinds of problems that arise in doing so. First, we need to describe information sharing between partners. Such sharing takes place by exchange of data objects. Moreover, the data objects have dependencies between them. Thus, typically, it is not very straightforward to capture information sharing structures and dependencies between shared data objects directly. A better approach is to derive an information sharing model from supply chain processes. A supply chain process contains *intra-organizational* sub-processes that are internal to a particular partner, and *inter-organizational* sub-processes that span multiple partners. Those inter-organizational processes directly involve information sharing. Therefore, we focus on inter-organizational supply chain processes. In general, our methodology is to describe an inter-organizational process and capture it by an information sharing model, modify parameters of this model to get different supply chain configurations, and then compare the performance of these configurations.

To describe an inter-organizational process precisely, we use UML activity diagrams [73]. UML is becoming a well-known standard for describing processes. For example, it is a modeling standard for RosettaNet [78] to illustrate the PIP business process flows. An activity diagram can define a process in terms of the control flows and object flows among its constituent actions. A control flow describes the sequence in which actions are executed in a process. An object flow specifies an object that is either

responsible for initiating an action or used by an action. Both control flows and object flows capture the dependencies between actions and reflect the need for coordination. However, when a process crosses the confines of an organization, it is difficult for this organization to impose direct constraints on its partners' actions through control flows unless a high level of collaboration has been established. Therefore, in general, object flows serve as an indirect coordination mechanism. Thus, in this essay, we focus on object flows. We model supply chain activities as actions, and data inputs or outputs of supply chain activities as objects. In addition, we show how *swimlanes* can be used to distinguish different partners. A detailed example will be provided later. With such an activity diagram, we can immediately recognize information flows and shared data objects involved in this process. Specifically, any object flow from one partner to another can be considered as an information flow, and the object of this flow can be treated as a shared data object. In addition, this diagram also shows the structure of information sharing and also the dependencies of shared data objects. Therefore, using activity diagrams, we can precisely describe the information sharing between partners and then we are able to represent it using a parameterized information sharing model.

Next, we propose a general methodology that involves the following steps:

1. Describe/modify a process as a UML activity diagram and check if the UML diagram is correct;
2. Extract cross-swimlane object flows from the UML diagram and save them as parameterized information flows in a table. Adjust parameter values to get different configurations;

3. Check if the new configurations are correct (in terms of parameter values, conditions, etc.);
4. Simulate different configurations and compare their performance;
5. Store the configurations in a standard form such as XML and exchange them with business partners.

2.4.2 Example: Vendor Managed Inventory (Step 1)

Here, we turn to illustrating this methodology with a detailed example: *Vendor Managed Inventory* (VMI). VMI is a collaborative arrangement typically between a vendor and its customers, such as retailers. In this arrangement, the vendor takes over the replenishment planning task for its partners. The main purpose is to reduce the safety stock as a buffer on both the vendor side and the customer side because of demand and supply uncertainties. Figure 2-3 shows that the main steps in VMI are as follows:

(1) Customers share their actual demand or usage with the vendor;

(2) The vendor generates the demand forecast and places a replenishment order for customers accordingly;

(3) Customers review the replenishment order and confirm it;

(4) The vendor then sends a ship notice and this is followed by physical goods transfer;

(5) Customers acknowledge the actual receipt;

(6) There may also be need for exception handling through the Supply Chain Event Management (SCEM) [59, 70] mechanism.

In VMI, trading partners establish some metrics to evaluate the performance of their collaboration such as *fill rate, inventory turns*, cost and other criteria. Therefore, it is necessary to trigger a special exception handling process in case the expected performance is not achieved. The term, SCEM is a subsystem for exception handling. Ideally, SCEM must detect and report exception events, and analyze them in real-time. However, exception handling is usually not a fully automated process and manual intervention is often necessary. An exception in VMI might occur, for example, when the fill rate drops below the required level.

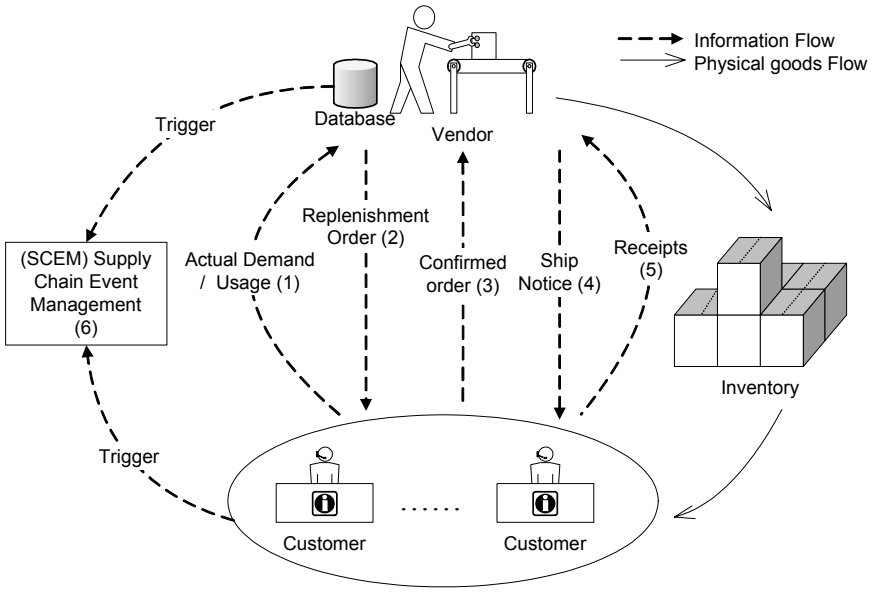


Figure 2-3: Vendor Managed Inventory (VMI)

Figure 2-4 shows the UML activity diagram for this VMI process. This UML model offers many advantages. First, a UML diagram clearly shows trading partners and their activities in the collaboration, since each swimlane represents a partner and its activities are located in this swimlane. Second, a UML diagram can be used to identify shared data objects. In Figure 2-4 , the objects associated with object flows which cross

swimlanes are shared data objects. Third, a UML diagram can be used to determine information flows, the dependencies between information flows, and the dependencies between shared data objects. Finally, this diagram can be shared by multiple partners easily, either as a drawing or after converting it into XML. In Figure 2-4, the seven information flows are denoted by numbers in the sequence in which they occur.

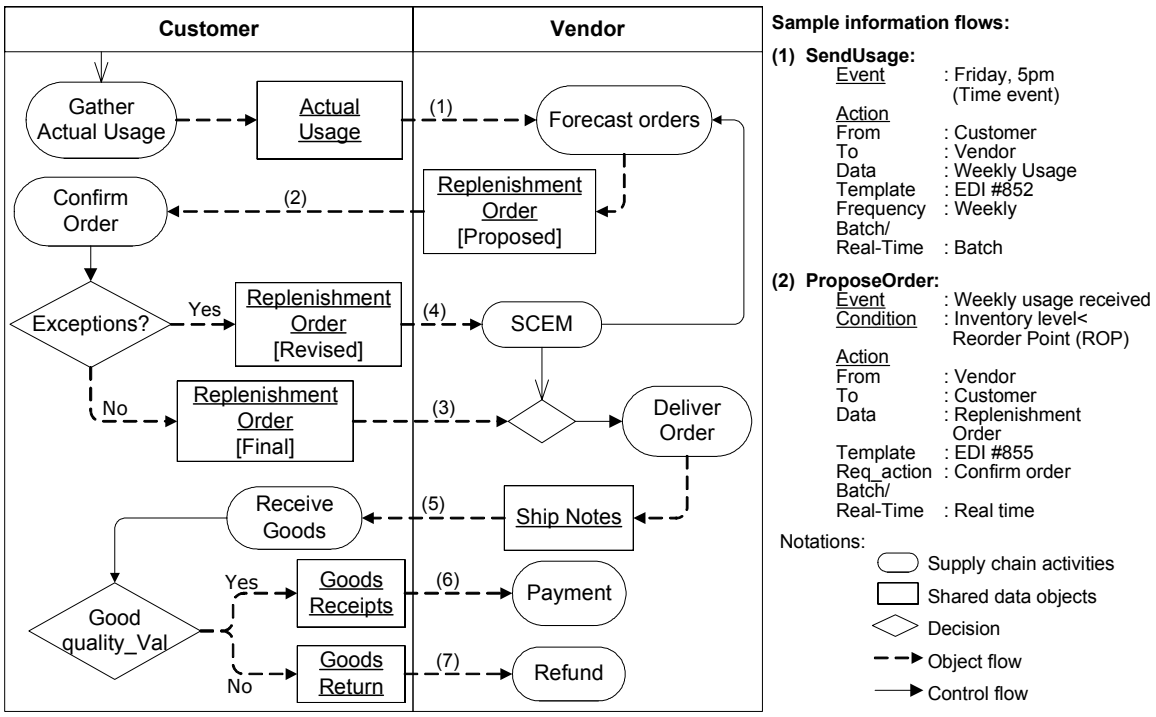


Figure 2-4: Modeling VMI Process with UML Activity Diagram

Later, we will give a UML data model and show that the flows must conform to this model.

2.4.3 Supply Chain Configurations (Step 2)

Next, we can extract the information flows from the UML activity diagram and store them in a table. This step can be facilitated with automated tools. For example,

first, the pictorial UML diagram can be converted into a textual XML description using conversion tools [17]. In particular, we have converted a UML model of a VMI supply chain into XML using tools such as Visual UML [90] and Rational XDE [39]. Once the process description is available in XML, it can be parsed to extract each individual flow by writing an XML Stylesheet Transformations (XSLT) script and storing it in a *configuration table* which maintains configuration information about a process. Additional information, not captured by a UML diagram, such as template numbers for each document exchange, and transfer mode, can be added to the rows of the table pertaining to conditions and other parameter values of those information flows. In addition, one could add the expected delay for each flow, so the actual throughput time could be compared against the expected value. The table can also be queried for such information. Thus, *the configuration table gives the rules of interaction between partners*, and any changes made to it must be propagated to all the partners immediately (preferably in real-time). It may be either replicated at each partner's location or stored centrally in a hub. Table 2-3 shows a configuration table for information flows extracted from the UML diagram of VMI. This table can capture the information sharing involved in the VMI process, and is called configuration *C1*.

In a configuration table, every information flow is initiated by an event, and takes place upon checking an (optional) associated condition; if the condition is true, then the flow takes place. The information flow involves a sender, receiver(s), a data object(s) and a requested action from the receiving partner. The data in row 1 of Table 2-3 corresponds to the *sendUsage* information flow. This flow occurs at 5 PM every Friday (a temporal event) and there is no associated condition with it. Thus, the usage

information is sent in a weekly usage form (i.e., a standard template) from the customer to the vendor. The second row describes the action taken by the vendor on receiving the usage. If the inventory value falls below the reorder point, then a new information flow called *proposeOrder* is sent from the vendor to the customer. The customer either accepts the proposed replenishment order (row 3), or rejects it and sends a modified order to the vendor (row 4). The vendor then processes the order and generates a ship notice (row 5) according to the shipping instructions. Finally, upon receipt of the product, the customer inspects the quality of the product and sends an acknowledgment or a rejection notice.

Table 2-3: Sample Data in Configuration Table for VMI (Configuration *C1*)

Information Flow	Event/ time	Condition	Action (Send Information Flow)				Requested Recipient Action	Batch/ Real-time
			Sender	Receiver	Data Objects	Template		
(1) sendUsage	Monday, 5 PM	–	Customer	Vendor	Weekly Usage	#852	Propose Order	Batch
(2) proposeOrder	Usage Received	Inventory <*ROP* (Reorder Point)	Vendor	Customer	Repl. Order [Proposed]	#855	Confirm Order (Accept or Reject)	Batch
(3) acceptOrder	Proposed order received	–	Customer	Vendor	Repl. Order [No change]	#855	Generate Ship Notice	Real-time
(4) modifyOrder	Proposed order received & Exception (fill rate<*ft*)	–	Customer	Vendor/ SCEM	Repl. Order [Revised]	#855	Generate Ship Notice	Real-time
(5) ShipNotice	Confirmed Order received	If Shipday = Sat; ship_gnd else ship_air	Vendor	Shipper/ Customer	Ship Notice	#857	Receive goods	Real-time
(6) GoodsReceipt	Goods received	Quality_ val >= *q*	Customer	Vendor	Goods Receipts ACK	#861	NONE	Real-time
(7) GoodsReject	Goods received	Quality_ val < *q*	Customer	Vendor	Goods Return	#862	Refund	Real-time

In this framework, there are several avenues for configuration. First, changes may be made to the frequencies of flows (daily instead of weekly), usage information from multiple customer sites, etc. For example, say the "event/time" of the first row of Table 2-3 is changed from "*Friday, 5 PM*" to "*Daily*, *5 PM*". This change leads to a new configuration, called configuration *C2*, shown in Table 2-4.

Further, configuration *C2* not only causes more frequent flows, but also results in a more responsive supply chain, since the vendor can track the customer's inventory on a daily basis (instead of weekly) and replenish the inventory more responsively. Moreover, this change also allows management to make improvements by tuning other parameters. For instance, since inventory is replenished more responsively, and the uncertainty in inventory supply is also reduced, the safety stock can be consequently decreased in order to improve the rate of inventory turns. This improvement is reflected in the decrease in the reorder point in the condition of flow "*proposeOrder*" (in Table 2-3). Similarly, responsive replenishment also results in a higher fill rate that is included as a parameter in the event associated with the "*modifyOrder*" flow.

Table 2-4: Configuration for Real-Time Information Sharing (*C2*)

Information Flow	Event/ time	Condition	Action (Send Information Flow)				Requested Recipient Action	Batch/ Real-time
			Sender	Receiver	Data Objects	Template		
(1) sendUsage	Every day, 5 PM		Customer	Vendor	Daily Usage	#852	Propose Order	real-time
Rows (2) – (7) of Table 2-3								

Although real-time information sharing reduces forecast errors and improves fill rate, it may increase costs because of more frequent order replenishment in a VMI arrangement. A conjecture is that if the fill rate already reaches a satisfactory level, say

95%, real-time information sharing may not be necessary; real-time information sharing is required only when the fill rate is below a predefined level. For example, when the fill rate is below 95%, we say an exception occurs. Therefore, we can create a new configuration, *C3* that mixes weekly with daily information sharing:

Configuration C3: IF *Exception occurs, i.e., fill rate< ft, Configuration 2* ELSE *Configuration 1*.

Still, many other adjustments may be made to the parameter values. In Table 2-3, the reorder point (row 2) or the target level for the fill rate (row 4) may be changed to a different value by the customer. The condition in row 5 allows the customer to specify that the shipment mode depends on the ship day, and this condition can be configured too. In addition, in row 6, a quality threshold can be specified and varied. Finally, the formats of documents can also be easily changed by specifying a new template name, if, say one partner modifies its documents. All of the above changes can be made "on-the-fly, " while other flows remain unchanged.

For example, a VMI arrangement typically uses a static reorder point. However, if the actual daily usage in this arrangement is affected by seasonal factors, a dynamic reorder point may make inventory replenishment more responsive. Therefore, the "condition" of the second row of Table 2-3 can be changed to "Inventory <*ROP*sf*," where *sf* is a seasonal factor. Seasonal factors can be calculated, for example, by dividing the average usage of one season by the average of a whole year. More sophisticated methods for computing seasonal factors can also be implemented. The change in the condition leads to another configuration called configuration *C4* (see Table 2-5). Note

C4 assumes daily information sharing. Later on, we will compare it with *C2* (see Table 2-4) , which also assumes daily information sharing, but does not consider seasonality.

Another aspect of configurability relates to the process itself. This may involve modifying an existing flow (i.e. change in a parameter value), adding a new flow, or deleting an existing flow. For example, suppose the *Order Delivery* activity is outsourced to a third-party shipper. The vendor shares the quantities and shipping profiles of replenishment orders with the 3rd party and the 3rd party arranges shipment automatically. This change will require a revised UML diagram such as Figure 2-5. Following our method, the new UML diagram will eventually lead to a modified configuration table. In this case, an additional flow, say, *sendDeliveryNotes*, from the vendor to the 3rd party may be inserted above row 5 in Table 2-3 to inform the 3rd party about the anticipated delivery for a new order.

Table 2-5: Configuration with Dynamic Reorder Point (*C4*)

Information Flow	Event/ time	Condition	Action (Send Information Flow)				Requested Recipient Action	Batch/ Real-time
			Sender	Receiver	Data Objects	Template		
(1) sendUsage	Everyday, 5 PM	–	Customer	Vendor	Daily Usage	#852	Propose Order	Batch
(2) proposeOrder	Usage Received	Inventory < ROP (reorder point) **sf* (seasonal factor)	Vendor	Customer	Repl. Order [Proposed]	#855	Confirm Order (Accept or Reject)	Batch
Rows (3) – (7) of Table 2-3								

With the help of the UML model, we can determine the affected information flows (modified, new added, or deleted). In Figure 2-5, the shaded part highlights the changes: (1) a new swimlane is added for the 3rd party shipper, which is responsible for

the activity *Deliver Order*; (2) there is a new flow, say *sendDeliveryNotes*, carrying *DeliveryNotes* from the vendor to the 3rd party; and (3) subsequently, the existing flow for *ShipNotice* should be modified since now it is sent from the 3rd party to the vendor and the customer. Table 2-6 shows the flows of this new configuration called *C5*.

Parameters of configuration *C5* can also be adjusted. For example, suppose the vendor can make small deliveries by itself at a low cost. Only when the order quantity reaches a high level, say 5000, 3rd party delivery is required. This new configuration can be described as follows:

Configuration 6: IF *(Order quantity>5000) C5* ELSE *C1*.

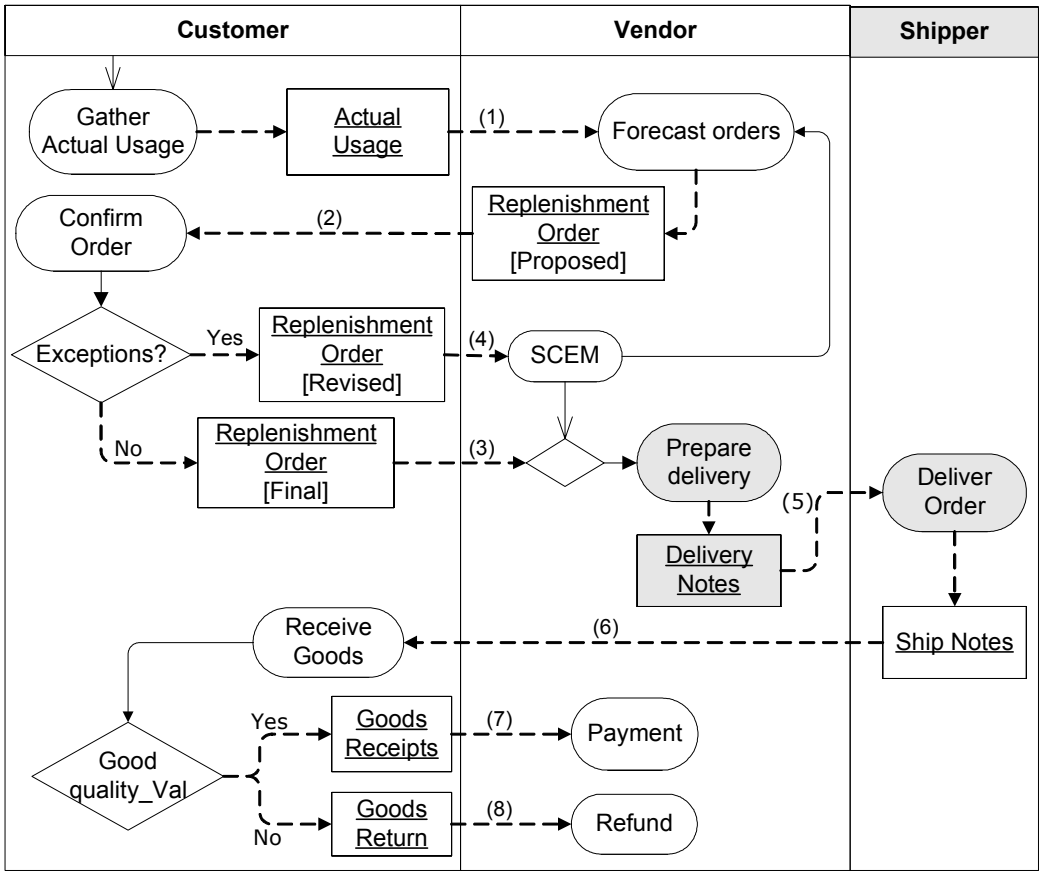


Figure 2-5: Modified VMI Process Shown in a Revised UML Diagram

Table 2-6: Supply Chain Configuration with 3rd Party Delivery (*C5*)

Information Flow	Event/ time	Condition	Action (Send Information Flow)				Requested Recipient Action	Batch/ Real-time
			Sender	Receiver	Data Objects	Template		
Rows (1)-(2) of Table 2-3								
(3) acceptOrder	Proposed order received	–	Customer	Vendor	Repl. Order [No change]	#855	Generate delivery notes	Real-time
(4) modifyOrder	Proposed order received & Exception (fill rate<ft)	–	Customer	Vendor/ SCEM	Repl. Order [Revised]	#855	Generate delivery notes	Real-time
(5) send-Delivery-notes	Confirmed Order received	If Shipday = Sat; ship_gnd else ship_air	Vendor	3rd Party deliver	Delivery notes	#857	Ship notice	Real-time
(6) ShipNotice	Delivery notes received	–	3rd party deliver	Vendor / Customer	Ship Notice	#857	Receive goods	Real-time
Rows (6) – (7) of Table 2-3								

For the above six configurations, all information flows are initiated by shared demand. Moreover, sharing information about the occurrences of important events, especially exceptions, makes a supply chain agile and able to recover quickly from sudden setbacks [52]. For example, after the earthquake in 1999 which caused serious supply delay of PC components from Taiwan, with real-time shared information about the extent of the suppliers' problem, Dell quickly implemented a contingency plan which steered demand away from products built from those components [52]. This contingency plan can certainly be treated as a supply chain configuration. Similarly, in this VMI arrangement, if the vendor experiences serious machine breakdowns, and, as a result, replenishment orders are delayed, the vendor can notify the customer of the occurrences of such events so that the customer can take backup plans, for instance, turning to

alternative vendors for replenishment. Therefore, we can have a new configuration *C7* as described in Table 2-7.

Table 2-7: Configuration with Shared Information about Event Occurrences (*C7*)

Information Flow	Event/ time	Condition	Action (Send Information Flow)				Requested Recipient Action	Batch/ Real-time
			Sender	Receiver	Data Objects	Template		
(0)	Machine breakdown notified		Customer	Alt. vendor	Rush Repl. order	#855	Confirm Order	Real-time
(1) sendUsage	Monday, 5 PM	–	Customer	Vendor	Weekly Usage	#852	Propose Order	Batch
(2) proposeOrder	Usage Received and without machine breakdown	Inventory <*ROP* (Reorder Point)	Vendor	Customer	Repl. Order [Proposed]	#855	Confirm Order (Accept or Reject)	Batch
Rows (3) – (7) of Table 2-3								

2.4.4 Verification of Supply Chain Configurations (Step 3)

So far we have discussed examples of various configurations for information flows through Table 2-3 to Table 2-7. Configurations can be stored in a standard form such as relational database tables or XML files, and exchanged with business partners. Moreover, it is important that these configuration tables be constructed according to a data model. Such a data model is shown in Figure 2-6. This data model can be used to verify that each configuration is correct. In particular, the flow dependencies must be well-maintained, the syntax of the conditions must be correct, and other values in the table such as template ids must be valid. For example, an information flow, say *X*, generates an event, say "data object *A* received", and this event triggers another flow, say *Y*. Thus, this induces a dependency between flows *X* and *Y*. If the configurations are

stored in a relational database, we can easily verify this type of dependency using an SQL statement as follows:

SELECT Flow. FROM Flow, Event, Data*

WHERE Flow.Event=Event.Event_ID AND Event.Data=Data.Data_ID AND

Flow.sender NOT IN (SELECT Receiver FROM Flow WHERE Data=Event.Data)

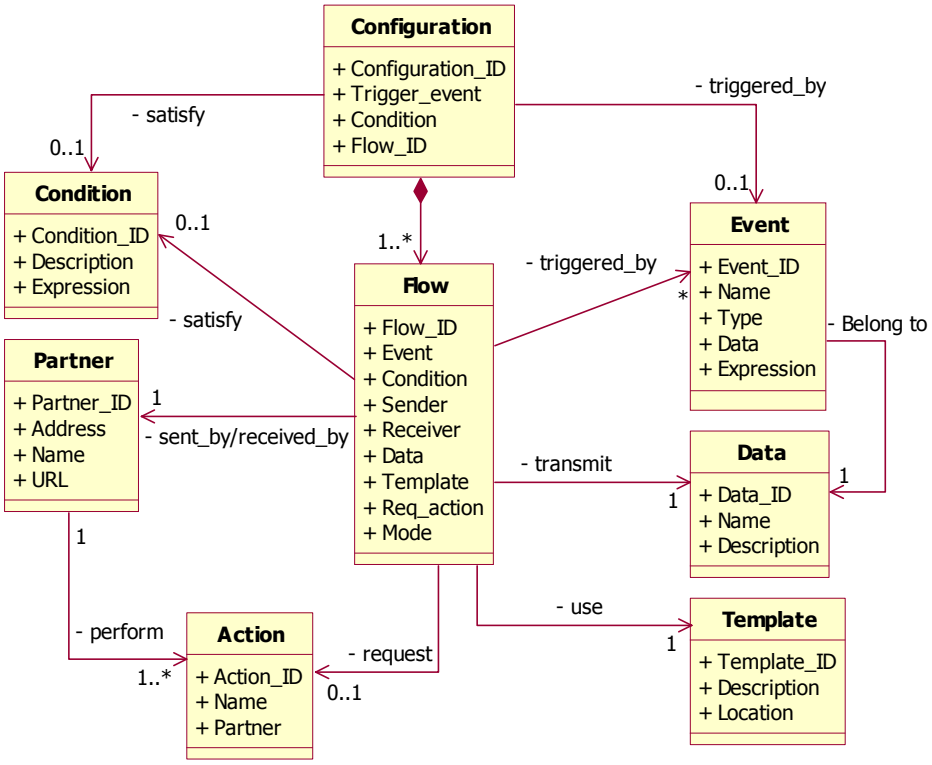


Figure 2-6: Data Model of Configurations

If this query returns a non-empty answer, none of the flows in this answer can be initiated because their prerequisite events never happened, leading to a broken sequence of information flows; otherwise, the sequence is well-maintained. Similarly, if configurations are stored in XML, one can use the XQuery language [95] to verify that the dependencies are correct. With regard to the conditions in supply chain

configurations, for now, we assume a simple, intuitive language for describing conditions, and omit the details of the syntax for it.

2.5 Simulation (Step 4)

We saw above that the information sharing model can lead to different configurations of a supply chain process. Next, we need to evaluate each configuration and determine which configuration is most suitable under some particular circumstance. A simulation prototype of this model is developed to simulate each configuration and measure its performance.

To evaluate a configuration, we need to select appropriate performance metrics for it. For example, for the VMI process, the appropriate performance metrics could be *fill rate*, *inventory turns*, and cost. In SCOR model, there are five categories of performance metrics: *reliability*, *responsiveness*, *flexibility*, *cost*, and *assets*. Each process element is associated with specific performance metrics. In general, we can refer to the SCOR toolkit [87] to select proper performance metrics for a process.

A preliminary prototype was built with the CSIM 19 simulation tool [68]. This prototype contains the following modules: *Configuration Setting*, *Data Initialization*, *Flow Processing Engine*, *Data Processing*, and *Performance Calculation*. The *Configuration Setting* module contains the configuration tables to be used in the simulation. In the *Data Initialization* module, process-specific data, such as order size and lead time, are stored. The *Flow Processing Engine* controls the flows based on the configuration setting. This module can be designed with reference to the architecture of

ECA applications [65]. The *Data Processing* module will generate required shared data objects based on templates and evaluate the conditions in flows. The *Performance Calculation* module is used to calculate the performance metrics of interest. In addition, this module can generate exceptions if a particular performance metric is below its predefined level. Figure 2-7 shows the design of this prototype. Among these modules, the flow processing engine can be used in any scenarios, while others may need some customization. This is a generic prototype which can be used to evaluate any configuration. Note that a particular configuration may also be evaluated using other simulation software. For example, the configurations of this VMI arrangement can also be simulated using Arena Simulation Software [44]. Angulo et al. [10] also used Arena to simulate information sharing in a VMI arrangement. Certainly, Arena has limitation in evaluating supply chain configurations. It may be difficult to process complicated events using Arena, since Arena is not based on ECA rules. We tested our configurations using both the prototype and Arena.

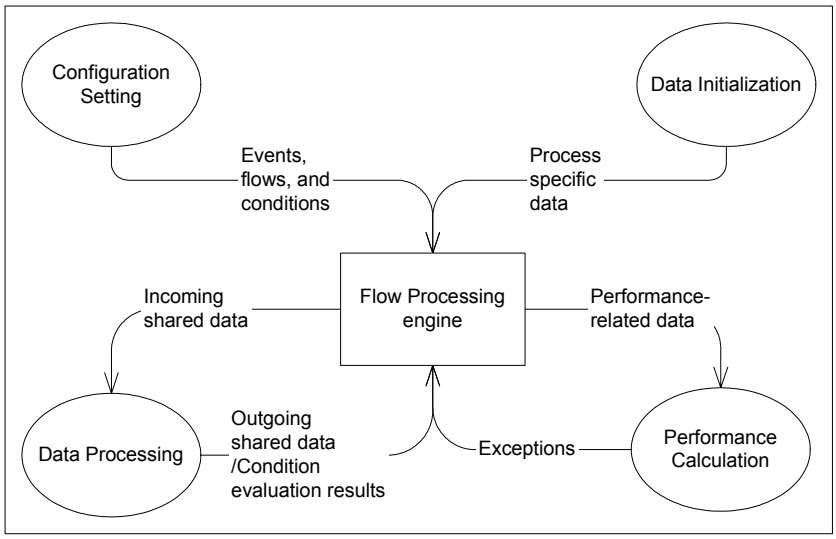


Figure 2-7: Design of Simulation Prototype

We first simulate and compare configurations *C1*, *C2* and *C3*, and then we test the performance differences between configurations *C2* and *C4* when seasonal factors are introduced to the daily usage. Finally, if the vendor experiences machine breakdowns, we test configuration *C7* where the information about the occurrences of those events is shared, and compare it with *C2*.

2.5.1 Simulation Setting

The initial setting of the simulated supply chain process is shown in Table 2-8. We assume that there is only one product involved in this VMI arrangement. The daily usage at the customer site follows a Gamma distribution with α=1.25 and β=400 (i.e, Gamma(1.25, 400), mean=$\alpha\beta$=500, variance $\alpha\beta^2$=200000). Research shows that if the lead time for an item and the demand per unit of time are both stochastic, Gamma distribution is a good choice for the resulting demand during the lead time [43, 88]. In addition, Gamma distribution has non-negative values. Moreover, since demand variability [92] may have impact on information sharing, we will also test the performance of configurations when demand variability changes. Demand variability is measured by the coefficient of variation (CV), the standard deviation of daily demand divided by the mean. Figure 2-8 shows the probability density function of the daily demand which follows different Gamma distributions. These distributions have the same mean, i.e. 500, but different standard deviation and therefore different demand variability. Obviously, from this figure we can see that when α is large, Gamma distribution closely approximates a normal distribution.

Table 2-8: Simulation Setting

Simulation setting	Values
Daily usage	Gamma distribution, Gamma(400, 1.25)
Reorder point	3500
Replenishment order size	6000
Lead time of replenishment orders	5 days

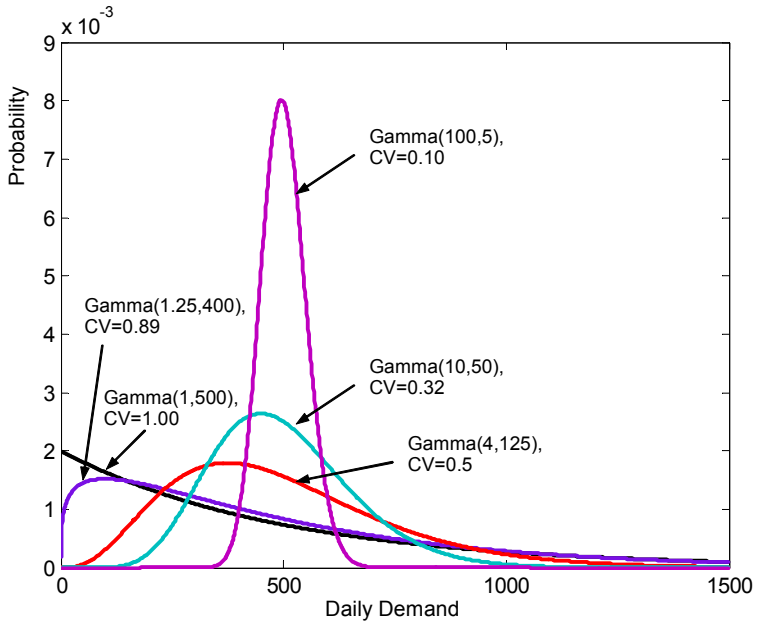


Figure 2-8: Probability Density Function of Gamma Distributions

The lead time for replenishment order is 5 days. A (*ROP*, *Q*) inventory policy is used, i.e., whenever the vendor knows that the inventory at the customer site is below reorder point (*ROP*), a replenishment order with order size *Q* is proposed. We assume *ROP*=3500 (i.e., actual usage during the lead time + 2 days' safety stock = 500 x 5 + 2 x 500 = 3500) and *Q*=6000. Table 2-9 shows Configuration 1 (*C1*) with specific parameter values and delay times.

To evaluate a configuration, appropriate performance metrics are chosen [20]. These are *average flow time* (or *inventory turns*), *order fill rate* and *annual total cost*. *Average flow time* is the time in days it takes to consume the average inventory (i.e., average inventory / average daily sales) and accordingly, *inventory turns* = the number of days in a year / average flow time. We assume 250 business days in a year. *Order fill rate* is defined as the percentage of demand fulfilled by the customer from available inventory. Initially, the required fill rate is set to 90%.

Table 2-9: Flow Table for Configuration (*C1*) with Specific Parameters

Information Flow	Event/ time	Condition	Action (Send Information Flow)			Delay time (days)
			Sender	Receiver	Data Objects	
sendUsage	Monday, 5 PM		Customer	Vendor	Weekly Usage	0.5
proposeOrder	Usage Received	Inventory < 3500 (ROP)	Vendor	Customer	Repl. Order [Proposed] (qty = 6000)	0.5
acceptOrder	Proposed order received		Customer	Vendor	Repl. Order [No change]	0.5
modifyOrder	Proposed order received and fill rate < 90%		Customer	Vendor/ SCEM	Repl. Order [Revised qty = qty*1.1]	0.5
ShipNotice	Confirmed Order received		Vendor	Shipper	Ship Notice	0.5
GoodsReceipt	Goods received		Customer	Vendor	Goods Receipts ACK	3.0

In addition, we calculate all costs incurred in the whole supply chain. To evaluate average cost, we make a number of assumptions. These assumptions do not affect the estimates of the other two performance indexes, order fill rate and average flow time. A simple but realistic cost structure is chosen based on a sale price of each item at $1.00 per unit at the customer side (the other costs can be considered to be proportional to this sales price). Partial fulfillment is allowed, whereas back orders are counted as lost orders.

Average shortage cost per lost item is 20% of the sales price, which reflects the cost of lost potential sales opportunities. In addition, average carrying cost per item per year is 20% of the sales price, which reflects the cost of storing and handling the product. Average transportation cost per item is $0.10. Average manufacturing cost per item from the vendor is $0.20, but it is 50% higher if the item is purchased from an alternative vendor. Setup cost for every replenishment order is $100 incurred by order handling and setting up a production run. When a replenishment order is proposed, if the accumulated order fill rate is below 90%, a penalty of $1000 is applied because the performance fails to reach the required level (see "modifyOrder" row in Table 2-9). This penalty reflects the sales loss as a result of customers switching to competitive brands since their needs cannot be satisfied. Table 2-10 shows the cost structure. The total cost per year is calculated as follows:

Total cost per year = setup cost of replenishment orders+ manufacturing cost + transportation cost + carrying cost + shortage cost + penalty

Table 2-10: Cost Structure of VMI Arrangement

Cost components	**Price($)**
Average carrying cost per item per year	0.20
Average transportation cost per item	0.10
Setup cost for every replenishment order	100
Shortage cost per lost item	0.20
Average manufacturing cost per item	0.20
Average manufacturing cost per item (alternative sourcing)	0.30
Penalty if fill rate < 90% on receiving a proposed order	1000

Angulo et al. [10] used a similar cost structure to test the impact of information accuracy and information delay on supply chain performance in a VMI arrangement. Of course, different supply chain scenarios may have different cost structures. To further

demonstrate the impact of the cost structure on configuration selection, we will provide sensitivity analysis for the key cost components later.

2.5.2 Simulation 1 – Comparing Weekly Sharing, Daily Sharing, and Mixed Sharing

The following three configurations are simulated:

C1: Weekly information sharing (see Table 2-9)

C2: Daily information sharing (in Table 2-9, change the event/time of the first flow to "everyday, 5 PM")

C3: Dynamic information sharing (Suppose the required fill rate is 95%): IF Exception occurs (i.e., *fill rate* < 95%) THEN *C2* ELSE *C1*

We simulated these three configurations for 15 replications each for a period of 1000 days. Table 2-11 shows the numbers of each type of flows and the total number of flows in different configurations during the simulation. Table 2-12 shows the performance results of each configuration.

First, the fill rate of *C1* is the lowest among the three configurations. Compared with *C1*, *C2* has a much higher fill rate, close to 100%. Real-time information sharing contributes to this high fill rate. However, in *C2*, the average flow time is also increased by 1.5 days. In other words, on average, more inventory is kept in the customer's warehouse. Obviously, more frequent replenishment orders lead to this increase of average flow time. As shown in Table 2-11, approximately 21 replenishment orders are issued in *C2*, compared to only 19 orders in *C1*. Therefore, one can see that the fill rate increases, but more replenishment orders are placed and a larger inventory is kept.

Table 2-11: Number of Each Type of Simulated Flow

Information Flows	Number of flows (Per Year)					
	C1 (Weekly sharing)		C2 (Daily sharing)		C3 (Mixed sharing)	
	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.
SendUsage (Weekly)	50	0	-	-	23.69	6.18
SendUsage (Daily)	-	-	250	0	131.57	30.9
ProposeOrder	18.98	0.68	20.60	0.54	19.82	0.51
AcceptOrder	16.45	4.74	20.58	0.54	19.52	0.67
ModifyOrder	2.53	4.19	0.02	0.07	0.30	0.47
ShipNotice	23.17	0.69	20.60	0.54	19.82	0.51
GoodsReceipt	23.17	0.69	20.60	0.54	19.82	0.51
Total number of flows	156.85	2.06	332.40	2.16	230.45	23.11

Table 2-12: Performance Comparison of *C1*, *C2* and *C3*

Configuration	Fill rate (%)		Average flow time (days)	
	Mean	Std. deviation	Mean	Std. deviation
C1 (Weekly sharing)	92.28	1.39	6.54	0.26
C2 (Daily sharing)	98.49	0.61	8.07	0.16
C3 (Mixed sharing)	95.01	0.26	7.34	0.18

Although there is naturally a trade-off between fill rate and inventory turns, it would be interesting to explore whether it can be finetuned to achieve a satisfactory fill rate while keeping the inventory turns as high as possible. We believe information sharing is the answer here, and test this belief in configuration *C3*. Recall that in *C3*, weekly sharing and daily sharing are mixed. Daily sharing is used when the fill rate drops below 95% (if the required order fill rate is 95%, an exception is triggered whenever the order fill rate is below this level). In other words, information sharing is adjusted dynamically when exceptions occur. The last column of Table 2-11 gives the numbers of

each type of flow for *C3*, while Table 2-12 compares the fill rate and average flow time for *C1*, *C2* and *C3*. As Table 2-12 shows, *C3* realizes not only a satisfactory fill rate, 95%, but also less average flow time, about 0.73 day less than that in *C2*. In *C3*, information is not always shared in real time, but is shared whenever necessary or in "quasi-real time" [26].

Table 2-13 compares the total cost per year incurred by each configuration. The calculation of total cost is based on the cost structure in Table 2-10. As Table 2-13 shows, the total cost of *C2* is lower than that of *C1* because *C2* has much higher fill rate than *C1*, and, as a result, *C2* incurs significantly lower shortage cost and fewer penalties than *C1*. This saving can balance the extra setup, manufacturing, shipping and carrying costs resulting from more inventory required by *C2*. However, although the shortage cost of *C3* is higher than that of *C2*, *C3* still incurs slightly lower total cost than *C2*. Because *C3* keeps less inventory than *C2*, the cost reduction in setup, manufacturing, shipping and carrying inventory can effectively compensate for the extra shortage cost and penalties resulting from lower order fill rate in *C3* (3.48% lower than that in *C2*).

Table 2-13: Cost Comparison of *C1*, *C2* and *C3*

Configuration	Cost Components ($)						Total Cost Per Year ($)	
	Setup	Manufacturing	Shipping	Carrying	Shortage	Penalty	Mean	Std. Dev.
C1 (Weekly Sharing)	**1,898**	**23,084**	**11,542**	**654**	1,947	2,533	41,659	4,236
C2 (Daily Sharing)	2,060	24,722	12,361	807	**382**	**17**	40,349	1,056
C3 (Mixed Sharing)	1,981	23,816	11,908	736	1,254	300	**39,997**	1,039

Configuration C3 shows that the desired supply chain performance (order fill rate, cost etc.) can also be achieved through flexible information sharing. The frequency of

information sharing should be determined in a dynamic environment by the performance requirements of supply chain processes. Moreover, the simulation also shows that information sharing can be a tool for adjusting supply chain processes dynamically in response to exceptions in supply chains.

2.5.3 Simulation 2 – Comparing Static vs. Dynamic Reorder Point

Next, we simulate the impact of a dynamic reorder point on supply chain performance. Christiaanse [22] hypothesized that a dynamic reorder point may be used to reduce the intensity of the bullwhip effect. Here we verify this conjecture through simulation and show how a dynamic reorder point can be used to deal with seasonality in demand. Suppose the actual daily usage has a seasonal feature and a period of, say 1 year, has four seasons. Each season is of equal length but the mean of the daily usage in each season varies. Table 2-14 shows the distributions used to generate the daily usage in each season.

Table 2-14: Gamma(α, β) Distributions of the Actual Usage in Four Seasons

Season	Distribution of the actual daily usage
1	Gamma(1.25, 250)
2	Gamma(1.25, 350)
3	Gamma(1.25, 450)
4	Gamma(1.25, 550)

Next, we investigate the impact of a reorder point on supply chain performance. The following two configurations are compared:

C2: Daily information sharing with a static reorder point, i.e., 3500 (from Table 2-9)

C4: Daily information sharing with a dynamic reorder point, i.e., 3500**sf* , where *sf* is a seasonality factor (see Table 2-5)

To get the value of *sf* above, we compute the moving average over a small period, say 10 days, and divide this average by 5000. Of course, other more complex calculations for *sf* are also possible. Thus, this seasonal factor changes over time. Next, we can show that even with such a rough estimation of the seasonal factor, the order fill rate of the supply chain improves.

Table 2-15 shows the performance indexes of *C2* and *C4*. When seasonal daily usage is introduced, the fill rate for *C2* is about 4% lower than that without seasonal factors (see Table 2-12). In terms of fill rate, *C4* clearly outperforms *C2*, but the average flow time also increases by 1 day. This simulation showed that a dynamic reorder point can be an effective way to counter fluctuation of the actual usage and raise order fill rate.

Table 2-15: Performance Comparison of *C2* and *C4* with Seasonal Actual Usage

Configurations	Fill rate (%)		Average flow time (days)		Replenishment orders (per year)	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
C2 (static reorder point)	94.82	0.56	7.65	0.18	22.13	0.49
C4 (dynamic reorder point)	96.69	0.54	8.67	0.21	22.57	0.56

In addition, using the cost structure in Table 2-10, we find the total cost per year only slightly increases when a dynamic reorder point is introduced. Table 2-16 compares the cost incurred by *C2* and *C4*. Since *C4* keeps one more day inventory than *C2*, the setup, manufacturing, shipping and carrying costs incurred by *C4* are higher than those in *C2*. As a result, the total cost of *C4* is higher than that of *C2*. For a cost-sensitive supply

chain, *C4* does not outperform *C2*. However, if the shortage cost per item increases to, say \$0.4, *C4* will incur lower total cost and therefore become a better choice than *C2*.

Table 2-16: Cost Comparison of *C2* and *C4*

Configuration	Cost Components ($)						Total Cost Per Year ($)	
	Setup	Manufacturing	Shipping	Carrying	Shortage	Penalty	Mean	Std. Dev.
C2(Static Reorder Point)	2,213	26,560	13,280	765	1,454	0	44,273	1,038
C2 (Dynamic Reorder Point)	2,257	27,088	13,544	867	928	67	44,750	1,104

2.5.4 Simulation 3 – Sharing Information about Event Occurrences

Next, we simulate the impact of sharing information about the occurrence of machine breakdown events on supply chain performance. Fox et al. [27] showed that sharing information about unexpected disruptions in supply chains can enhance the coordination of supply chain partners and reduce the negative consequences of those disruptions.

Suppose that the machines at the vendor side break down sometimes. During the breakdown, all replenishment orders are delayed until the problem is fixed. The up time of these machines follows an exponential distribution with a mean of 90 days, i.e. EXP(90), and the down time follows EXP(5). With configuration *C2*, the vendor does not notify the customer of the occurrences of breakdown events, so replenishment orders could be delayed. With configuration *C7*, the customer is notified when breakdown events occur, and then it turns to alternative vendors for replenishment. The manufacturing cost of alternative vendors is 50% higher than that of the main vendor. The lead time of alternative sourcing follows a uniform distribution between 3 and 5

days, i.e., U(3,5). After the machines are fixed, the customer resumes the replenishment activities with the VMI vendor as before. The following two configurations are compared:

C2: Daily usage is shared, but machine breakdown information is not shared; no alternative sourcing.

C7: Daily usage and breakdown information is shared; alternative sourcing is used during breakdowns (see Table 2-7).

Next, we can show that with sharing of breakdown event information, the performance of the supply chain improves in terms of order fill rate and total cost. Table 2-17 shows order fill rate and average flow time of *C2* and *C7*. With occasional machine breakdowns, the fill rate of *C2* is 5% lower than that without breakdown events (see Table 2-12). Moreover, compared with *C2*, *C7* has only slightly increased average flow time.

Table 2-17: Performance Comparison of *C2* and *C7*

Configurations	Fill rate (%)		Average flow time (days)		Replenishment orders (Per Year)			
					VMI Vendor		Alt. Sourcing	
	Mean	Std. dev.	Mean	Std. dev.	Mean	Std. dev.	Mean	std. dev.
C2 (Without sharing breakdown information)	93.84	2.05	6.98	0.25	19.35	0.48	-	-
C7 (Sharing breakdown information)	95.92	0.81	7.15	0.17	18.25	0.78	1.78	0.74

In terms of fill rate, *C7* clearly outperforms *C2*. Also, as Table 2-18 shows, the total cost per year decreases when the customer is notified of the machine breakdowns and alternative sourcing is introduced. Although an extra cost is incurred by the alternative sourcing, *C7* leads to lower shortage cost and fewer penalties than *C2*.

Table 2-18: Cost Comparison of *C2* and *C7*

Configuration	Cost Components ($)							Total Cost Per Year ($)	
	Setup	Manufacturing	Alt. Vendor	Shipping	Carrying	Shortage	Penalty	Mean	Std. Dev.
C2 (Without sharing breakdown information)	1,935	23,456	--	11,728	698	1,557	1,967	41,341	3,278
C7 (Sharing breakdown information)	2,003	21,926	3,210	12,033	715	1,031	217	41,135	1,411

2.5.5 Configuration Sensitivity Analysis

From Table 2-13, we can see some cost components varies significantly among these three configurations. Next we do the analysis of configuration sensitivity to these components: carrying cost, shortage cost and the penalty. Finally, we will analyze the sensitivity of configurations to demand variability.

2.5.5.1 Sensitivity to Carrying Cost and Shortage Cost

Figure 2-9(a) shows the sensitivity analysis of carrying cost. If we change the carrying cost per item per year from $0.10 to $0.40 (but keep other cost components unchanged), we can see *C3* always outperforms the other two. This result can be explained by Table 2-13, which clearly shows that the carrying cost only amounts to 2% of the total cost. The change on carrying cost makes no significant impact on the total cost of the supply chain.

On the other hand, if the shortage cost per item varies from $0.00 to $0.85, the configuration with the lowest total cost moves from *C3* to *C2* as shown in Figure 2-9 (b).

Clearly, when the shortage cost per item increases, the lower the order fill rate, the faster the total cost per year increases. For example, when shortage cost per item is \$0.50, the shortage cost per year amounts to 11% of the total cost in *C1*, but only around 2% of the total cost in *C2*.

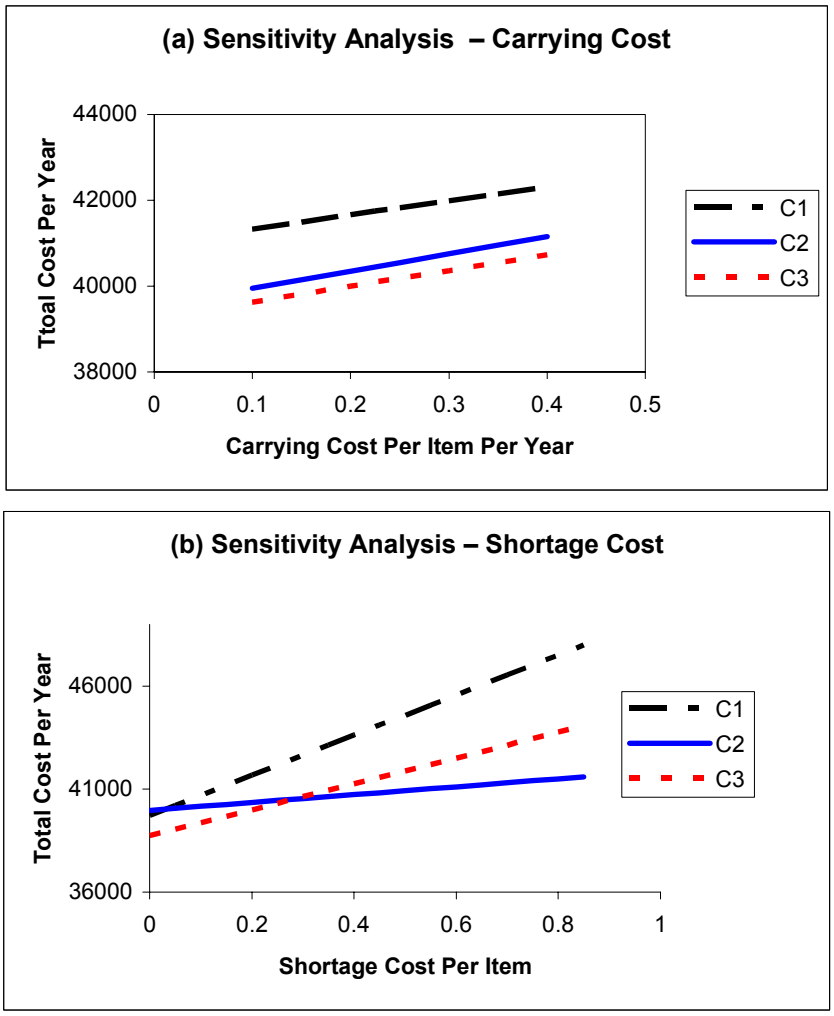


Figure 2-9: Sensitivity Analysis of Cost

Figure 2-10 shows which configuration can achieve the lowest average cost when both carrying cost and shortage cost vary. Obviously, when shortage cost per item

increases, *C2* will achieve the lowest total cost. Moreover, the cost difference among these three configurations increases as shortage cost per item increases.

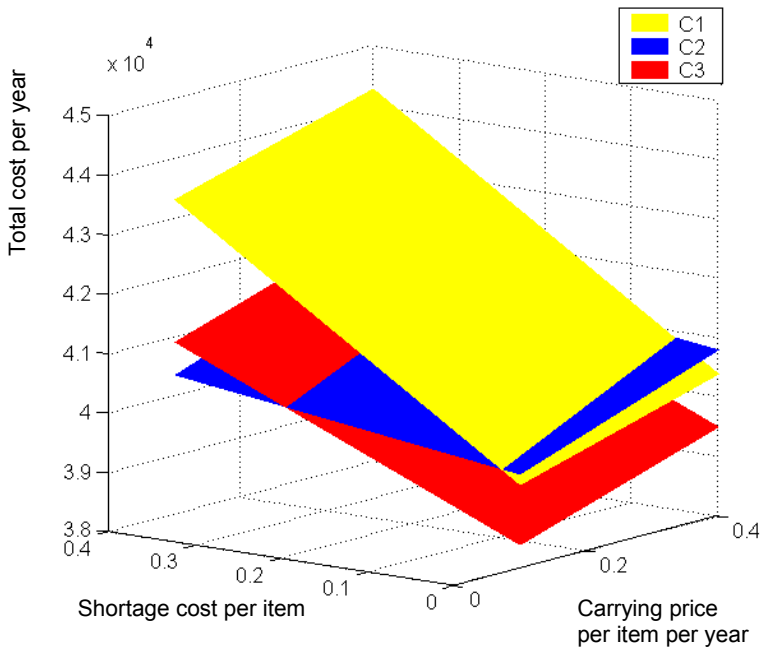


Figure 2-10: Total Cost Sensitivity to Shortage and Carrying Costs

2.5.5.2 Sensitivity to Penalty

Figure 2-11 shows the sensitivity of configurations to the penalty imposed when order fill rate is below 90% upon receiving a proposed replenishment order. This figure shows that when the penalty is very small (less than $300), *C1* incurs the lowest total cost. When the penalty increases, *C3* has the lowest total cost. When the penalty is very high (more than 2300), *C2* incurs the lowest cost since its order fill rate rarely falls below 90%. In general, the penalty represents the cost of losing potential market share because

of failures in order fulfillment. In a market with many competitive products, such a cost could be very high. Therefore, real-time information sharing is especially important, as this analysis result shows.

Note that in our simulation experiments, we use accumulated order fill rate. If the order fill rate is calculated on a monthly basis, we find that in *C1*, the order fill rate falls below 90% more frequently (around 150 days out of 1000 days). The punishment for a low order fill rate could be very heavy. In addition, in our experiments, a penalty is imposed only when order fill rate drops below 90% upon the retailer receiving a proposed replenishment order. In general, a supply chain can design an appropriate penalty mechanism based on its competitiveness in the market. This mechanism will have impact on selecting the most suitable configurations, as this simulation experiment shows.

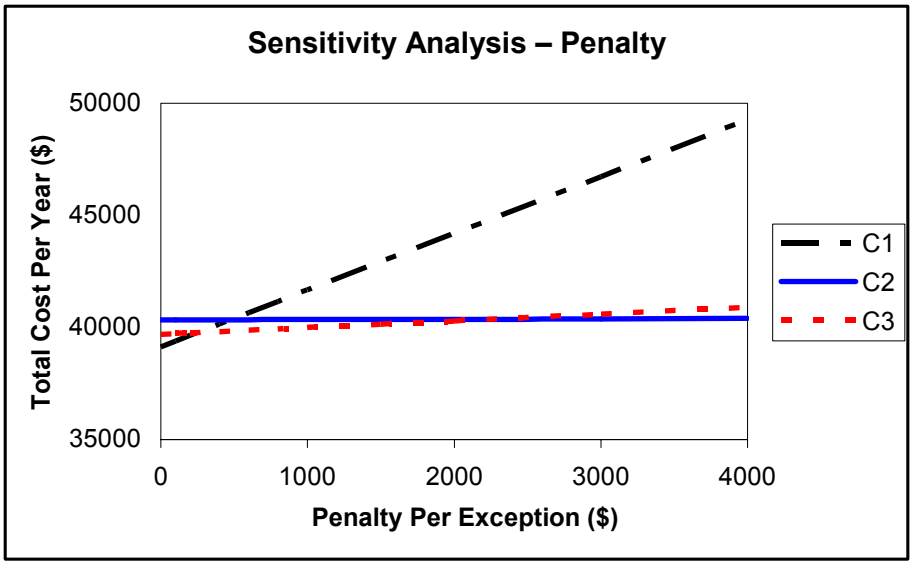


Figure 2-11: Sensitivity to Penalty

2.5.5.3 Sensitivity to Demand Variability

Waller et al. [92] mentioned that daily demand variability varies widely in different industries. For example, the demand variability is in general lower (around 0.10~0.30) in consumer products and significant higher (perhaps greater than 1.00) in electronics. Waller also showed that when the variability increases, more inventory is required to ensure a certain level of order fill rate. Next, we test the impact of demand variability on the selection of supply chain configurations.

Table 2-19 shows five daily demand distributions, which have the same mean but different standard deviation. Therefore, the demand variability ranges from high to low. The probability density functions of these five distributions are shown in Figure 2-8. The performance of configurations *C1*, *C2* and *C3* under different demand distribution is shown in Figure 2-12.

Table 2-19: Distributions and Variability of Daily Demand

Distribution	Gamma(100, 5)	Gamma(10, 50)	Gamma (4, 125)	Gamma(1.25, 400)	Gamma (1, 500)
Mean	500	500	500	500	500
Standard dev.	50	158	250	447	500
Demand Variability	0.10	0.32	0.50	0.89	1.00

As Figure 2-12 (a) shows, when the demand variability increases, the order fill rate in each configuration drops because the variation in demand leads to more lost orders. However, order fill rate in *C3* only changes slightly. Recall that *C3* is a mix of weekly and daily information sharing and the portion of weekly or daily information sharing is adjusted by the order fill rate. When the demand variability increases, the portion of weekly information sharing is reduced but that of daily information sharing is

increased. In other words, for *C3*, when the variability is low, information is mainly shared on a weekly basis and this is frequent enough to maintain 95% order fill rate. However, when the demand variability is as high as 1.00, *C2* (daily sharing), but not *C1* (weekly sharing), is still able to achieve a 95% order fill rate. Therefore, when $CV = 1.0$, *C3* adjusts to almost daily information sharing. Thus, the order fill rate of *C3* rarely changes because of such a self adjustment.

Figure 2-12 (b) shows that when demand variability increases, the inventory turns of *C1* slightly decrease, because it becomes more likely that the order fill rate drops below 90% and supply order size is increased by 10% (see Table 2-9) at this time (more average inventory will be kept). Inventory turns of *C2* barely change because the order fill rate is always above 90%. However, as the variability increases, inventory turns of *C3* decrease because as information is shared more frequently (almost daily), inventory is replenished more frequently, and as a result, more inventory is kept.

Figure 2-12 (c) compares the total cost incurred by each of these three configurations. Clearly, when the demand variability is low, weekly information sharing (*C1*) can achieve a high order fill rate (above 95%), and it requires relatively lower level inventory than daily information sharing (*C2*). As a result, *C1* incurs lower total cost than *C2*. When the demand variability is high, daily information sharing is required to ensure a high order fill rate in order to keep the cost low. Since *C3* is a combination of *C1* and *C2*, it approximates to *C1* when the variability is low but moves close to *C2* when the variability is high. Therefore, *C3* can balance the shortage cost and the cost of replenishing inventory and always achieve the lowest cost, as the demand variability changes.

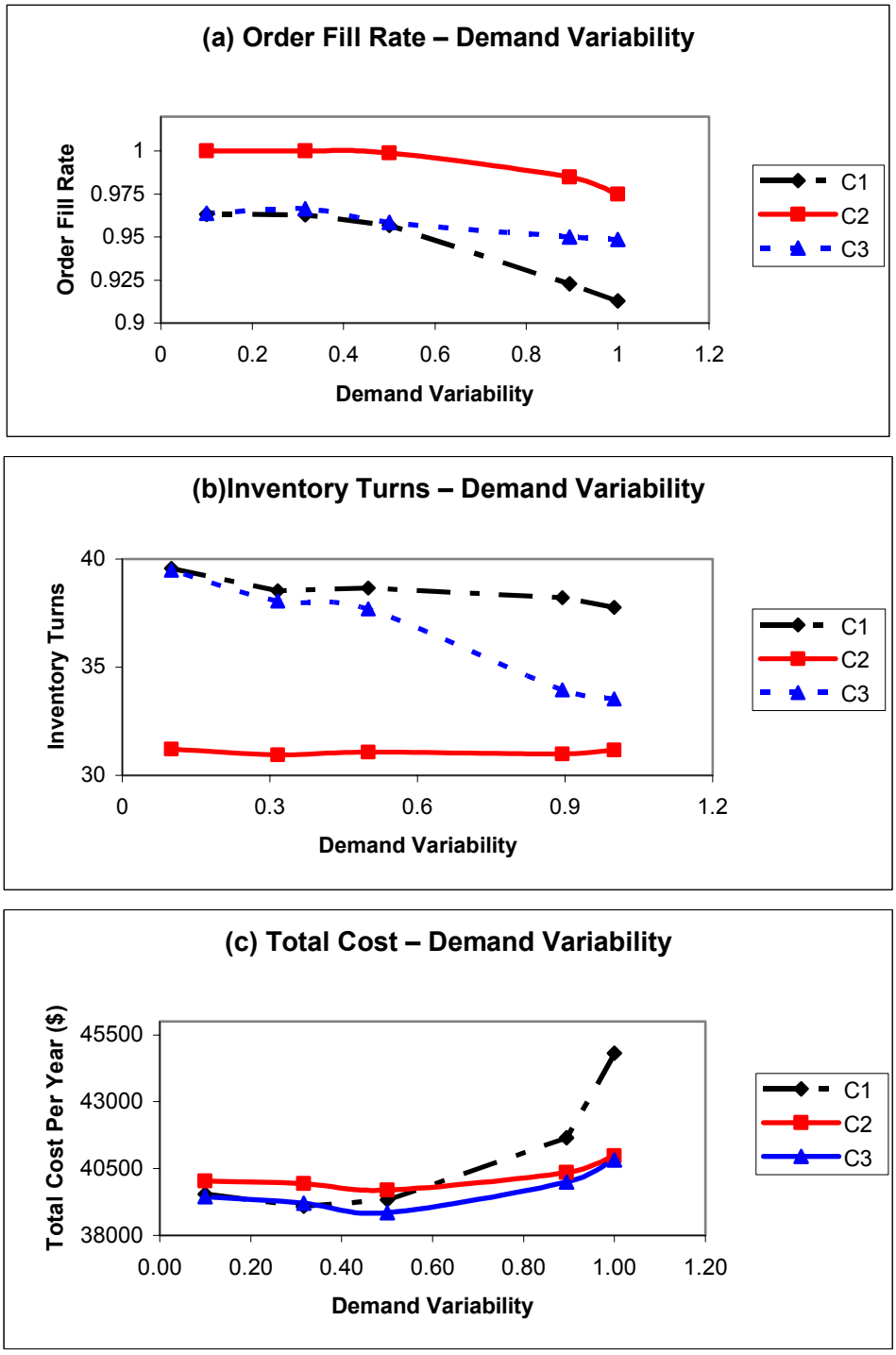


Figure 2-12: Sensitivity of Configurations to Demand Variability

The sensitivity analysis further suggests that in order to achieve the lowest cost in a supply chain, the configurations should be carefully evaluated. *Changes in supply chain environment could make a previously optimal configuration no longer optimal*. For example, if the shortage cost per item is increased to above $0.50 (say, because of shortage, ultimate customers lose goodwill and potential sales are lost), clearly, a higher fill rate is preferred. In addition, the changes in penalty mechanisms (e.g., for unsatisfactory order fill rate) and demand variability can also affect the performance of a configuration. In general, changes in supply chains can result in different information sharing needs and suitable configurations should be used accordingly. Therefore, it is evident that effective sharing of information is possible only through a systematic approach to modeling information sharing, simulating the effect of information sharing, and directly associating information sharing with the performance metrics of supply chain processes. This is exactly the goal of our research.

2.5.6 Discussion of Supply Chain Configurations

So far, we demonstrated how to analyze information sharing and design supply chain configurations. We also showed that configurations should be evaluated in terms of supply chain performance. We should point out that the simulation here is used to demonstrate how to evaluate different configurations and select the most suitable one that can satisfy the information sharing needs for a particular situation. It is not to show that one configuration is absolutely better than others. Using different parameters or inventory policies, we may get different results. A supply chain should test the performance of

configurations with parameters pertaining to it. In addition, the simulation here is to demonstrate that each configuration should be evaluated in some way. One may also use some other approaches to evaluate configurations. For example, for this VMI arrangement, optimization techniques of inventory models [55] may be another evaluation approach.

Note that in our VMI example, although most shared information, such as demand and order status, is related to operational processes, this does not mean that this methodology is only applicable to processes at the operational level. Actually, this example also illustrates how information sharing can affect decisions at tactical or strategic level. For example, when machine breakdowns happen, the vendor may or may not share information about the occurrence of the events. Sharing such information could bring alternative vendors to this supply chain and thus more competition. However, without sharing such information, exceptions, such as unsatisfactory fill rate, could happen and then impinge on the effectiveness of this collaborative arrangement. Certainly, this dilemma is more than an operational decision. Moreover, another advantage of this methodology is that it can show the detailed impact of information sharing on processes on different levels and also the potential benefits (e.g., improved fill rate and cost reductions). This clear view can encourage supply chain partners to share information willingly or reduce resistance to necessary information sharing. In addition, all configurations are shared among supply chain partners and per se are new knowledge or "organizational memory" created from information sharing [61].

2.6 Implementation

The methodology presented in this essay can be implemented to support dynamic information sharing. For example, if the various supply chain partners agree to set up a hub to coordinate supply chain processes (see Figure 2-2 (c) in Section 2.3.1), this methodology can be used to develop the architecture of such a hub as follows. First, the hub stores all configurations, data templates and data, and up-to-date performance metrics. Second, an information flow engine based on ECA rules [65] is the core of this hub and it is used to facilitate information flows exchanged between partners. We have discussed a prototype of this engine in Section 2.5. In addition, this hub can directly communicate with each partner's enterprise information systems to process shared data objects. Finally, to build "sense-and-respond" capabilities [32] in a supply chain, an event engine [59] can be included to detect, analyze, and respond to events in real time. Fed with events from an information flow engine, the event engine can filter and process primitive events and generate alerts or notifications for only the significant ones. Based on these events the supply chain configuration may be modified suitably. A preliminary architecture of the hub is shown in Figure 2-13. The details of this architecture need to be further developed.

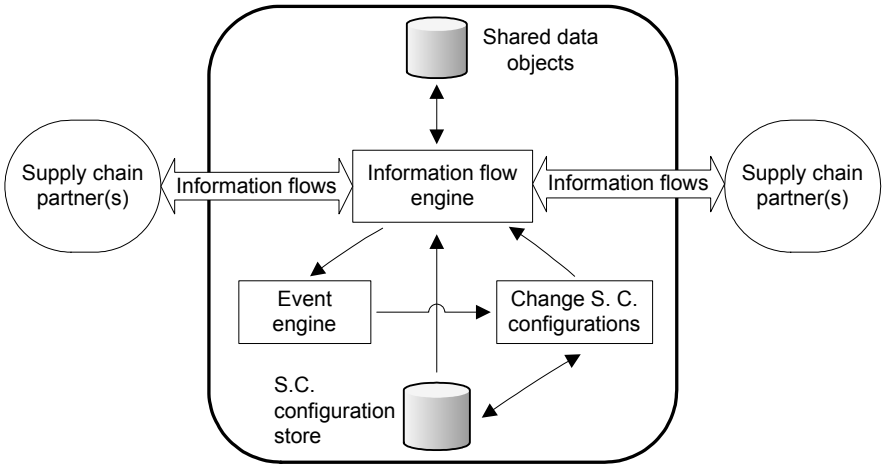


Figure 2-13: Architecture of an Information Sharing and Event Management Hub

2.7 Conclusion and Future Work

Information sharing plays a key role in supply chain collaboration, which requires timely information about suppliers, manufacturing, distribution, retailing, and demand. In this essay, we introduced a methodology which leverages information sharing to configure supply chains based on well-known technologies including UML, XML and ECA rules. This methodology consists of several steps, many of which can be automated (or partially automated) using existing tools. Through this methodology, we are able to analyze information sharing, create supply chain configurations, evaluate configurations and use them suitably in response to supply chain changes.

We showed that supply chain changes (e.g., changes in cost structures, market competitiveness and demand variability etc.) and exceptions can lead to different information sharing requirements and then suitable configurations should be selected to meet the requirements. The results of the simulation show that a well-designed

configuration can lead to improved performance of a supply chain. In addition, all configurations are shared among supply chain partners and, per se, create new knowledge or "organizational memory" (Malhotra et al. 2005).

We expect our future work to extend this methodology to the strategic level in designing supply chains. For example, we would like to investigate what strategic configurations can enable a supply chain to successfully switch from make-to-stock (MTS) operations to make-to-order (MTO) mode, which matches supply more closely to demand. Moreover, we plan to focus on the detailed procedure for verifying ECA rules in supply chain configurations and on also the implementation of our methodology.

Chapter 3

A Formal Modeling Approach for Supply Chain Event Management

Abstract: As supply chains become more dynamic, there is a need for a sense-and-respond capability to react to events in a timely manner. In this essay, we propose Petri nets extended with time and color as a formalism for event management. We describe seven basic event patterns that capture common modeling concepts in supply chains and also show how to use the patterns as building blocks to model a complete supply chain. Based on Petri net models, events and their causes are analyzed using dependency graphs. We also use simulation to analyze supply chain performance under different event resolution strategies. In addition, we perform sensitivity analysis to study the effect of changing event parameters on performance indicators. We show that by modeling timing and causality issues accurately it is possible to improve supply chain performance.

3.1 Introduction

The pressures of global competition and the need for extensive inter-organizational collaboration are forcing companies to streamline their supply chains and make them agile, flexible and responsive. Consequently, a supply chain must be able to handle large numbers of events, both expected and unexpected. The unexpected events, also called *exceptions*, typically arise because there is usually a gap between supply chain planning and execution [11]. Supply chain planning sets a target that can be achieved based on a given set of constraints at a given time. In a dynamic supply chain environment, the constraints are always changing, so exceptions or deviations from plans occur almost regularly. Examples of exceptions are inaccurate forecast, product out-of-stock, shipment delay, etc., and they are costly. Moreover, events tend to propagate in collaborative supply chains across partners, resulting in the well-known bullwhip effect

[48]. Such risks have given rise to a new research area of Supply Chain Event Management (SCEM). The goal of SCEM is to introduce a control mechanism for managing events, in particular, exceptions, and responding to them in a timely manner.

A supply chain event is "any individual outcome (or non-outcome) of a supply chain cycle, (sub) process, activity, or task" [6]. Events are correlated with each other to form a "cloud" of events; some events have significant consequences and therefore they must be monitored closely, while others are of lesser importance. The critical problem lies in extracting the significant events and responding to them in real-time. Doing so requires an ability to monitor them proactively, simulate them to help decision-making, and use them to control and measure business processes [70, 85]. In this essay, we present a methodology that uses a Petri net approach to formulating supply chain event rules and analyzing the cause-effect relationships between events.

Petri nets are a powerful modeling technique for problems involving coordination in a variety of domains. A variant of Petri-nets called *time Petri-nets* allows us to model time intervals also. Considering the dynamic characteristic of supply chain events, such Petri nets are useful for describing the time constraints associated with events. Examples of time constraints are: "*event* e_1 *follows event* e_2 *after time T* " and "*N occurrences of event* e_1 *within time T lead to event* e_2". These temporal constraints are important for proper correlation between events; otherwise, the management could be unable to anticipate events or track causes of events. To deal with variety in case data (e.g., order ids, order quantities, rush orders versus normal orders, etc.) we extend the model with "token colors", i.e., we use *time colored Petri-nets*.

Using time colored Petri-nets we can model event patterns common in Supply Chain Management (SCM). These patterns can be composed as will be demonstrated using a Vendor Managed Inventory (VMI) example. To demonstrate that the Petri-net basis allows for different types of analysis we used CPN Tools [75] to simulate different scenarios for the VMI example. The mapping onto CPN Tools allows us to investigate the performance of event resolution strategies. In addition, dependency graphs are used to analyze cause-and-effect relationships of events.

There are a variety of SCEM systems from companies, such as SAP, i2, and Manugistics [85]. Most systems mainly perform monitoring and provide "early warning" rather than analyzing events and suggesting solutions [11, 16, 64, 66, 70, 85]. Actually, the more powerful part of SCEM would be the capability of "aggregating data from key business systems at a high level and presenting the ramifications of exceptions and the possibilities of solutions" [64]. Therefore, this research can contribute to the research area of SCEM in three ways. First, this work introduces a formal and general approach to modeling events and event rules, and the approach provides flexibility in associating occurrence counts and temporal constraints with events, avoiding the customization problem which often poses as an obstacle to the implementation of the existing SCEM systems [16]. Second, this approach allows excellent event analysis, including event forecasting with time information and causality analysis, which provide real-time visibility about the implications of events and traceability to the root causes for events. Third, it offers a way to track supply chain performance metrics by events, and shows how through simulation decision makers can compare different strategy alternatives or fine-tune a solution in terms of key performance indicators.

This essay is structured as follows. Section 3.2 gives an overview of events, event rules, event aggregation, event causality, and our notion of a dynamic supply chain. Section 3 describes Petri nets briefly. Section 3.4 introduces event semantics and seven event patterns or building blocks of event rules. It shows that a complete event Petri net can be constructed by using these blocks. Section 3.5 presents a detailed example to illustrate how to use Petri nets to examine event causality and forecast subsequent events. Section 3.6 gives simulation results of the example to illustrate the practical value of our approach. Section 3.7 describes related work and compares our approach with others, while Section 3.8 concludes the essay with a brief description of future work.

3.2 Overview of Supply Chain Events

When supply chain partners are integrated, events at one partner may have impact on other partners, and their responses to these events may cause a storm of events. Therefore, causality analysis is the key to controlling such a storm. Our analysis begins with events and event rules.

In general, events in an organization occur in the following three types: (1) *task status related events,* such as the end of a task or the beginning of a task, which are usually regular; (2) *events produced by a task*, such as events "stock partially available" and "out of stock", which are the possible results of the "check availability" task; and, (3) e*xternal events* which may arrive from other supply chain partners or from the external environment, e.g., *new order arrival, inbound shipment delay, import policy change* etc.

These types of events are captured directly during a process, and called *simple or primitive* (as opposed to *composite*) *events*. Composite events are *derived* from simple events by *event aggregation*. A composite event is deduced when a group of simple events occurs [60]. A group of simple events may together reveal potential problems. For example, if a product is out of stock once in a month, perhaps it is quite normal and an alarm should not be generated, but *if this stock out happens two times in a week, then it may reflect some underlying problems in the supply chain and this should be recognized by generating an event.* As another example, a group of stock trading events, related by accounts, timing and other data, taken together, may constitute a violation of a policy or a regulation [60]. *Event aggregation* is a mechanism to filter simple events and extract meaningful information from them by setting up alarms in advance.

Thus, *event aggregation* extracts value from a management point of view out of trivial and unorganized simple events. In order to achieve this objective, it is important to recognize event patterns and set up *aggregation rules*. Besides aggregation rules, *business rules* must also be considered. Business rules capture the causal relationships between events. For example, if an order is delayed for more than time *T*, then it is automatically cancelled. Therefore, a rule is needed to express that the event "order delayed by *T* " is a cause of event "order cancelled".

Moreover, a supply chain is viewed as a series of synchronous and asynchronous interactions among trading partners. Usually, when an event, particularly an exception, happens, the trading partner responsible for it may react to this event within a reasonable *resolution time* to resolve it. For instance, suppose an order is delayed for delivery. If the delay is within an acceptable range specified by the customer, the customer is notified

of the delay and the order is processed. However, if the delay exceeds the acceptable tolerance (also called *expiration time*), the order should be automatically cancelled, and hence, the event "order delay" is not relevant in this case. On the other hand, a series of new actions arise because of this new event, such as canceling the order, removing any reservations made, refunding any payments, etc. Therefore, to model events and event rules precisely, our modeling approach should be able to capture such temporal constraints correctly. In our analysis, each event is associated with two time values: *resolution time* and *expiration time*. In most cases, event resolution takes an unpredictable amount of time because of complexities of various business situations and it is more realistic to set up a resolution time interval. We will show how to capture the dynamic aspect of events in the later sections.

3.3 Petri Net Preliminaries

A Petri net is a directed graph consisting of two kinds of nodes called *places* and *transitions*. In general, places are drawn as circles and transitions as boxes or bars. Directed arcs connect transitions and places either from a transition to a place or from a place to a transition. Arcs are labeled with positive integers as their weight (the default weight is 1). Places may contain tokens. In Figure 3-1, one token is represented by a black dot in place *p1*. A marking is denoted by a vector M, where its p^{th} element $M(p)$ is the number of tokens in place p. The firing rules of Petri nets are [71]:

(1) A transition t is *enabled* if each input place of t contains at least $w(p,t)$ tokens, where $w(p,t)$ is the weight of the arc from p to t. (By default, $w(p,t)$ is 1.)

(2) The *firing* of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t,p)$ tokens to each output place p of t, where $w(t,p)$ is the weight on the arc from t to p.

There is another special type of arc called an inhibitor arc with a small circle rather than arrow at the end. An inhibitor from a place to a transition prohibits the transition from being enabled, and thus firing, if there is a token in the place. An example of an inhibitor arc is given later.

In this essay, we use *Time Colored Petri Nets* (TCPN), i.e., Petri nets extended with *time intervals* and *token values*. First of all, the above classical Petri nets can be extended by associating a time interval $[I_1, I_2]$ with each transition, where I_1 (I_2) is the *minimum* (*maximum*) time the transition must wait for before firing *after* it is enabled. Such a Petri net is known as Time Petri net (TPN) [91]. If $I_1 = I_2$, we just associate one time value with each transition[1], while if the interval is not specified, then $I_1 = I_2 = 0$. Analysis techniques for TPNs are discussed in [14, 91]. Second, tokens can be tagged with data values (or a color) to create a colored Petri net (CPN) [40, 41]. For example, we use tokens of different colors (or values) for each order or product. For a given place, all tokens must be from one color set.

---

[1] The *Time* Petri Nets discussed in this essay should not be confused with *Timed* Petri Nets. A Petri net is called *Timed* Petri net [91, 99] if each transition is associated with a fixed time instead of a time interval. The two types of Petri nets have very different semantics. As discussed in [14], *Time* Petri Nets are more general than *Timed* Petri Nets.

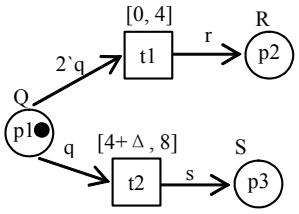


Figure 3-1: Colored Time Petri Net

In Figure 3-1, *Q, R* and *S* represent different color sets. *q*, *r*, and *s* are variables, such that $q \in Q$, $r \in R$, and $s \in S$. In a TCPN the arcs are also labeled with colors. For example, in Figure 3-1, two tokens colored "*q*" are consumed if transition *t1* fires. The fired transition *t1* will put one token colored "*r*" in place *p2*. Moreover, if there are two tokens colored "*q*" continuously existing in place *p1*, transition *t1* will fire no later than time 4. If there is still a token colored "*q*" remaining in place *p1* after time 4 (relative to the arrival of this token), transition *t2* will fire shortly after time 4 (denoted as 4+Δ, where Δ is a very short time period, close to 0) and before or at time 8.

3.4 Event Formulation and Event Patterns

3.4.1 Event Semantics

Having given a preliminary introduction to Petri nets, now we turn to developing the techniques to formulate event rules as Petri net structures. In most cases, events are not only the triggers but also consequences of supply chain tasks, i.e. one event causes another. Therefore, it is quite natural to model events as places that represent pre-conditions or post-conditions of transitions. Thus, events and places will be used interchangeably while modeling events. Moreover, time Petri nets offer an attractive

choice for modeling the dynamic aspect in supply chains. To make such models, we first formulate events and event rules as follows:

Event rule R: $e_1\ (n_1x_1,\ I_0) \xrightarrow{[I_1, I_2]} e_2(x_2)$, where

e_1 : input event class

n_1 : number of event instances (for simplicity, we just say events), i.e., number of tokens (by default, $n_1 = 1$).

x_i : data value of event i for $i = 1,2$. In other words, the color of tokens, $x_i \in$ color set X_i.

I_0 : expiration time of e_1.

$\rightarrow$: "imply" or "lead to", which establishes a cause-effect relationship between the left side and right side of the rule.

$[I_1, I_2]$: an optional time interval which corresponds to the event resolution time. In order not to make the problem trivial, we require $I_2 < I_0$. If this interval is not specified, we assume $I_1 = I_2 = 0$.

e_2 : output event class. For every rule, only *one* instance of e_2 is generated because it is not necessary to repeat supply chain events.

This event rule shows the semantics of event e_1 succinctly. Suppose e_1 continues to arrive at a system. If the number of its occurrences reaches a threshold, say n_1, and these events persist in the system long enough, event rule R can be triggered during interval $[I_1, I_2]$, and e_2 is then generated. I_0 is the expiration time of e_1. If rule R does not fire within the $[I_1, I_2]$ interval, then e_1 expires. After e_2 occurs, e_1 may normally be

consumed by rule *R*. However, if e_1 is required by another rule, then a token should be returned to e_1. Hence, two representations are possible for event rules:

Representation 1 (**consumption case -** e_1 is consumed): This case can be modeled as a Petri net shown in Figure 3-2. This representation is useful when an event is not required by multiple rules.

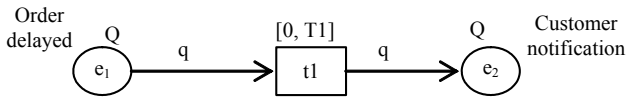


Figure 3-2: Petri Net of Example 1 Showing a Rule *R*

Representation 2 (**non-consumption case -** e_1 is not consumed): Event e_1 is not consumed because it may be required by another rule. Nevertheless, event e_2 must not be generated multiple times from these occurrences of e_1. This case can be accurately modeled as a Petri net as shown in Figure 3-3.

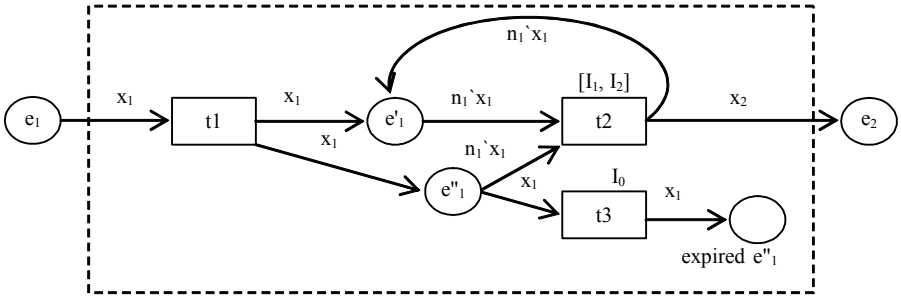


Figure 3-3: Petri Net Representation for Non-Consumption Case

When comparing Figure 3-2 and Figure 3-3, one can note several differences. First of all, the representation chosen in Figure 3-3 abstracts from color sets, and focuses on timing issues and causalities. Second, events are not consumed. Third, we consider the situation where n_1 events need to occur to trigger another event. Since event e_1 is not

consumed by rule *R*, we need a special mechanism to prevent event e_2 from being generated repeatedly. Therefore, as Figure 3-3 shows, Place e_1 is first *transformed* into two places, e'_1 and e''_1, through a transition *t1*. Tokens in e''_1 are consumed if transition *t2* fires, while n_1 x_1 tokens (denoted as $n_1`x_1$) are brought back to place e'_1[2]. (Note that n_1 events are needed to enable transition *t2*.) Therefore, after the first firing, although there are n_1x_1 tokens in place e'_1, transition *t2* cannot fire, and thus, at most one e_2 event is generated (with respect to n_1x_1 tokens). If transition *t2* does not fire (because of insufficient tokens in e'_1), e''_1 expires at the end of expiration time by firing transition *t3*. The notion of expiration time will be discussed further in the next section.

These two representations are employed in our patterns in the next section.

3.4.2 Event Patterns to Model Supply Chain Rules

Next we will develop several patterns for constructing complex temporal event relationships and also give equivalent logical expressions for these patterns. In general, three logic connectives, OR ($\vee$), AND ($\wedge$), Negation ($\neg$), can be used on either the left or the right side of an event rule. Since modeling of time is crucial in understanding the behavior of our Petri net models, we call these patterns *temporal event patterns*.

We will show later that these patterns can be used as building blocks to create event networks in supply chains. We will demonstrate that these patterns allow us to

---

[2] This will create cycles in this Petri net model. In general, we allow cycles in Petri net models. However, Petri net models should not create "cycles" in event dependencies (i.e., event *A* leads to event *B* and *B* leads back to *A*). Such "cycles" in event dependencies will cause unnecessary repetition of events. In this essay, all Petri net models are carefully designed to avoid such "cycles". Still, an approach to verifying Petri net models is necessary and this remains as part of our future work.

capture sophisticated relationships involving multiple event instances, event expiration times and resolution times. Thus, this modeling approach can be used to model large varieties of typical supply chain events. We will also illustrate the patterns by examples.

Pattern 1 (**simple cause-result pattern)**: A *cause-result pattern* is the most basic pattern for describing event relationships. It shows that event e_1 can cause event e_2 within a time period $[I_1, I_2]$. More formally, this relationship is expressed as: $e_1 (x_1) \xrightarrow{[I_1, I_2]} e_2 (x_2)$.

Example 1: If an order is delayed (e_1), contact customer (e_2) before time *T1*, i.e., $e_1 (q) \xrightarrow{[0, T1]} e_2 (q)$ (Note: q is order numbers).

Figure 3-2 (in the previous section) shows the time Petri net model of this example. Note that *order numbers* can be considered as a color set here, i.e., each order has a different color. We use Q to denote this color set, and q is a variable for any order in Q. Transition *t1* must fire within time *T1* after it is enabled. Transition *t1* corresponds to the action "notify customer".

Pattern 2 (**Repeat_cause-one_effect pattern):** This pattern concerns the case where multiple occurrences of one event within a certain time period cause another single event to occur. Formally, this relationship can be described as:

$e_1 (n_1 x_1, I_0) \xrightarrow{[I_1, I_2]} e_2 (x_2)$, where n_1 occurrences of event e_1 for instance x_1 cause event e_2 to occur. There are numerous situations where this pattern is useful.

Example 2: If product s is out of stock (e_1) more than once within period *T2*, contact the supply chain manager (e_2). (Note, s is the product ID). Formally, this rule can be represented as $e_1 (2s, T2) \xrightarrow{[0, \Delta]} e_2 (s)$.

This example introduces the notion of *expiration time* of events. If an event is not consumed (in this case, event e_1) by a rule, it may expire after a time interval. The Petri net model in Figure 3-4 represents the time constraints pertaining to these events. Whenever tokens arrive at place e'_1 and e''_1 (as a result of event e_1) transition *t2* and *t3* are enabled, but they cannot fire immediately. When there are two tokens arriving in place e'_1 and e''_1, transition *t1* fires immediately and produces the event e_2, "Notify SC Manager". After transition *t1* fires, two tokens are returned to place e'_1, because event e'_1 may be used by other rules. However, tokens in place e''_1 are consumed, so transition *t1* cannot fire repeatedly. Since transition firing takes no time, *t3* is still continuously enabled. If a token stays in place e'_1 for time *T2* after its arrival, *t3* fires and event e_1 expires. Thus, it is possible that event e'_1 expires without *t1* firing, if there is only one token arriving within interval *T2*. Simultaneously, transition *t2* fires so that e''_1 expires.

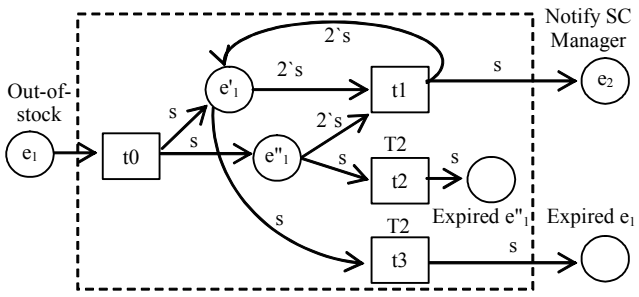


Figure 3-4: Petri Net of Example 2 (Pattern 2)

Pattern 3 (Inclusive choice): The need for this construct arises when multiple, alternative events can occur based on temporal conditions. Formally, this rule can be expressed as:

$$e_0\,(n_0x_0, I_{00})\ \{[\xrightarrow{[I_{11}, I_{12}]}\ e_1\,(x_1)]\ \vee\ [\xrightarrow{[I_{21}, I_{22}]}\ e_2\,(x_2)]\ \vee\ \ldots\ \vee\ [\xrightarrow{[I_{m1}, I_{m2}]}\ e_m\,(x_m)]\}$$

Thus, in general, an event e_0 could produce one of many events ranging from e_1 to e_m based on the time intervals associated with these events. In general, these time intervals could overlap; however, by ensuring the intervals are non-overlapping it would be possible to make a deterministic choice based on time. The following example illustrates this pattern.

Example 3: If an order, with lead time *L2*, has not been shipped (i.e., not consumed by some other rule) within time *L2* after it is confirmed (e_0), the order is treated as delayed (e_1) (but e_0 is not consumed yet); however, if an order is delayed by more than time *T3*, it is treated as undeliverable and cancelled (e_2). (Perhaps the customer does not want it if the delay is more than *T3*. So e_0 is consumed at this time.) This example can be formulated as:

$$e_0\,(q, L2+T3+2\Delta)\ \{[\xrightarrow{[L2,L2+T3]} e_1\,(q)\ ] \vee [\xrightarrow{L2+T3+\Delta} e_2\,(q)\ ]\ \}$$

This event rule can be represented as a Petri net model in Figure 3-5. As the Petri net model shows, when an order is confirmed, a token is placed in place e'_0 and e''_0 as well. Transitions *t1*, *t2*, *t3*, and *t4* are enabled but do not fire at that moment. If this token is consumed by the shipment transition *t5* before time *L2* (relative to its arrival), transitions *t1*, *t3*, and *t4* are disabled, but transition *t2* will fire at time *L2+T3+2Δ* after the token arrival. Otherwise, if during the time interval [*L2, L2+T3*] this token remains in place e'_0, transition *t1* will fire. After transition *t1* fires, this token is immediately brought back to e'_0 because some other rules (like *t4*) may use it later. If there is still a token in e'_0 after *L2+T3*, transition *t4* fires and produces event "order cancelled". Thus, the token in e'_0 is consumed. In general, if this rule is triggered, it can produce two possible results: order delayed and cancelled, or only order delayed, depending upon the

temporal relationships. One can see this rule actually has complex semantics, yet its Petri net model can precisely describe such temporal relationships. Note that in Figure 3-5, transition *t3* never fires and it can be removed. We keep this transition in the figure for consistency with event semantics.

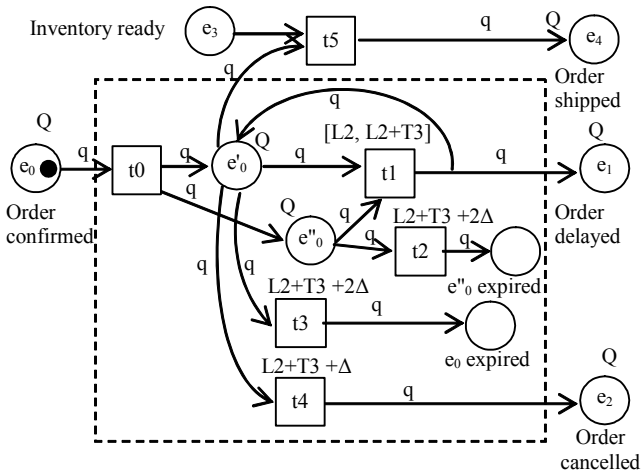


Figure 3-5: Petri Net Model of an Order Process (Pattern 3)

Pattern 4 (1 of N causes – single result Pattern): A result can have multiple alternative (combination of one or more) causes. Hence, there is a need for this pattern, and its formal logical expression is as follows:

$$\{[e_1(n_1x_1,I_{10}) \xrightarrow{[I_{11},I_{12}]}] \vee [e_2(n_2x_2,I_{20}) \xrightarrow{[I_{21},I_{22}]}] \vee \ldots \vee [e_m(n_mx_m,I_{m0}) \xrightarrow{[I_{m1},I_{m2}]}]\}$$
$$e_0(x_0)$$

In this expression, the cause of event e_0 may be any one of e_1, e_2, ..., e_m. Typically, this structure could be used to indicate the cause for an event. Figure 3-6 is the Petri net presentation of this structure, if every source event e_1, e_2, …, e_m is consumed by this rule. Note that the notion of expiration times can be applied to this pattern. For simplicity, the expiration times and expiration transitions are not shown in Figure 3-6. For example, if transition *t1* does not fire (because of insufficient tokens), e_1 will expire

at time I_{10} by firing an expiration transition. The same standard simplification is applied to the next three patterns. A specific example of Pattern 4 is given as below.

Example 4: When a rush replenishment order is rejected (e_1), or delayed (e_2) by more than time *T4* (if the delay is less than *T4*, the delayed time can be compensated by faster shipment), contact alternative vendors (e_0). Logical form:

$$\{[e_1(q) \xrightarrow{[0,T5)}] \vee [e_2(q) \xrightarrow{T4}]\}\, e_0$$

As the Petri net model in Figure 3-7 shows, in this example, if an order is rejected by a vendor, an alternative vendor must be contacted in a short interval, say [0, *T5*]. If the order is delayed, a token is put into place e_2 immediately. How long this token remains in place e_2 is exactly the order delay time. If the order is delayed for time *T4*, then transition *t2* has been continuously enabled for the same time, so transition *t2* fires immediately. The fired transition means that, in order to replenish inventory in time, alternative sourcing is required. If the delay does not exceed time *T4* and then the inventory is ready, a fast shipment (e_4) is used to compensate for this delay. Therefore, through these examples, we see that colored time Petri nets not only present the transformation of events, but also simulate the underlying business activities. The latter advantage cannot be achieved by its logical formulations.

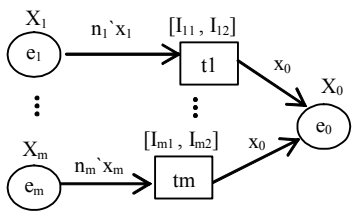


Figure 3-6: Petri Net for *1 of N Causes – Single Result* (Pattern 4)

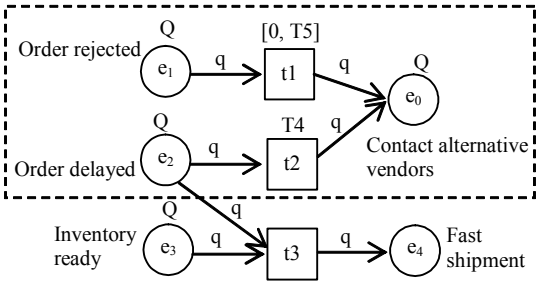


Figure 3-7: Petri Net of Example 4 (Pattern 4)

Pattern 5 (1 cause – N results Pattern): This pattern recognizes that a cause may have multiple consequences and captures all *concurrent* consequences of a particular event. Logically, this is expressed as:

$$e_0\,(n_0x_0,\, I_{00}) \xrightarrow{[I_{01},\, I_{02}]} \{e_1\,(x_1) \wedge e_2\,(x_2) \wedge \ldots \wedge e_m\,(x_m)\}$$

In this expression, n_0 occurrences of event e_0 generate m different events concurrently. Figure 3-8 shows the Petri net representation of this rule. As Figure 3-8 shows, the m events are represented as output places of transition *t1*, which is enabled by n_0 occurrences of input event e_0. Transition *t1* fires within an interval $[I_{01}, I_{02}]$ after n_0 tokens are placed in e_0. In this case, this Petri net representation shows a *1 cause – N results* pattern.

Example 5: If an order is delayed (e_0), notify customer (e_1) *and* reschedule the shipment (e_2) immediately, i.e. $e_0\,(q) \xrightarrow{[0,\, T6]} [e_1\,(q) \wedge e_2\,(q)]$.

Figure 3-9 is the Petri net model of Example 5. If *T6* approaches 0, *t1* fires instantaneously. This simple example illustrates that this structure can be used to present the concurrent events that originate from the same cause.

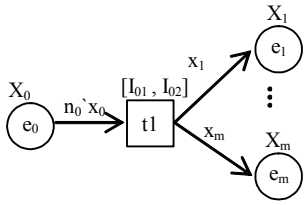


Figure 3-8: Petri Net for 1 Causes – N Results (Pattern 5)

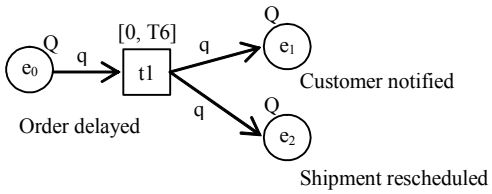


Figure 3-9: Petri Net of Example 5 (Pattern 5)

Pattern 6 (**N causes – 1 result Pattern):** This pattern is the reverse of the above pattern, and it is used to model the concurrent causes of a particular event. The following formulation shows that, there are *m* preconditions, e_1, e_2, ..., e_m, which occur simultaneously to arrive at event e_0:

$$\{e_1 (n_1x_1, I_{10}) \wedge e_2 (n_2x_2, I_{20}) \wedge \ldots \wedge e_m (n_mx_m, I_{m0}) \} \xrightarrow{[I_{01}, I_{02}]} e_0 (x_0)$$

Similarly, assuming every sourcing event is consumed by this rule, this structure can be transformed into a Petri net as shown in Figure 3-10. As the Petri net model shows, each precondition can be modeled as an input place of transition *t0*, and the result e_0 is the output place of this transition. This Petri net exhibits an *N causes – 1 result* pattern; so does the Petri net representation of Example 6.

Example 6: When the shipper of a confirmed order (e_2) is not available (e_1), find another shipper (e_3) in a short time *T7*. This rule can be logically formulated as: $[\, e_1 (q) \wedge e_2 (q) \,] \xrightarrow{[0, T7]} e_3 (q)$.

The Petri net of Figure 3-11 shows the precise representation of this rule. In Figure 3-11, two events e_1 and e_2 in conjunction produce one result, event e_3.

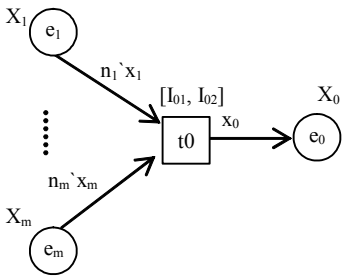


Figure 3-10: Petri Net for *N Causes – 1 Result* Pattern

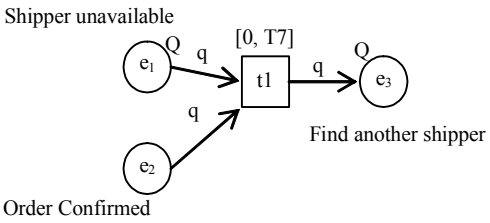


Figure 3-11: Petri Net of Example 6 (Pattern 6)

Pattern 7 (non-occurrence of an event pattern): The above patterns were all based on the occurrence of events. However, non-occurrence of an event can also signal valuable information and hence we need a pattern for that. Negation is usually used to express the non-occurrence of a particular event. Typically, non-occurrence of an event and occurrence of some other events may, in conjunction, cause some other significant events to happen. The following logical formula describes this situation:

$$\{e_1 (n_1x_1, I_{10}) \wedge [\neg e_2 (n_2x_2, I_{20})]\} \xrightarrow{[I_{31}, I_{32}]} e_3 (x_3)$$

Event e_1 and non-outcome of e_2 cause e_3. Actually, if event e_1 is consumed by this rule, this formulation can be transformed into a Petri net similar to Figure 3-11, except the arc from place e_2 to transition *t1* replaced by an inhibitor arc, as Figure 3-12 shows.

(See [21] for the semantics of inhibitor arcs in colored Petri nets.) In general, Figure 3-12 and the Petri net representation of Example 7 have a *non-occurrence* pattern.

Example 7: When an order arrives (e_1), if there is *no* out-of-stock (e_2) situation, the order is confirmed (e_3). Logically, this rule can be formulated as:

$$\{e_1(q) \wedge [\neg e_2(s, T2)]\} \xrightarrow{[0, T8]} e_3(q).$$

Figure 3-13 is the Petri net model of Example 7. After the order arrives, a token is put into place e_1. At that time, if there is no token in place e_2 it means there is no out-of-stock event, and transition *t1* fires in a short time, say [0, *T8*], and puts a token in place e_3 to indicate the order is confirmed. Otherwise, if there is a token in place e_2, it inhibits the firing of transition *t1*.

However, some Petri net tools may not support inhibitor arcs. We can substitute inhibitor arcs with an equivalent structure that works for CPN tools [75] and is described in Appendix A.

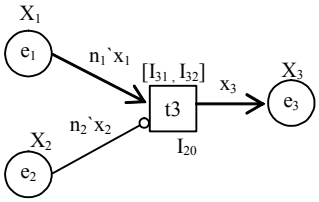


Figure 3-12: Non-Occurrence Pattern (Pattern 7)

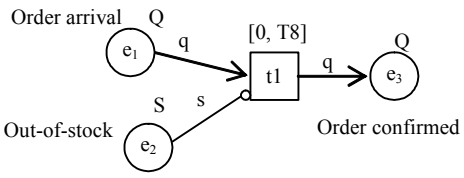


Figure 3-13: Petri Net Example 7 (Pattern 7)

In this section, we have developed 7 basic patterns that capture cause-effect relationships in Petri-nets. Next, we will use an example to show that these patterns can be combined together as building blocks to create more complex Petri-nets.

3.4.3 Composing New Patterns and Creating User-Defined Patterns

Above we discussed 7 basic patterns to capture complex cause effect relationships. Now we demonstrate how they can be combined to create new user-defined patterns. In general, patterns can be combined (or composed) if they have common input or output events (i.e., places that have the same label). By superimposing common places shared by existing patterns, new patterns can be created. This approach has been used in modeling logic programs as Petri nets [83]. Obviously, if two patterns do not share any events then they cannot be directly composed. The possible scenarios for pattern combination are as follows:

(1) The output place of one pattern is the input place of another pattern (sequential)

(2) The two patterns have one or more common input places (Parallel 1)

(3) The two patterns have one or more common output places (Parallel 2)

(4) The two patterns have common input and output places (Parallel 3)

When two patterns are composed in sequence, they form more complex cause-effect chains. On the other hand, if they share common inputs, the patterns will compete for firing by taking tokens from the common event places and have exhibit more complex interactions. A detailed analysis of the various possible interactions for the different

combinations is beyond the scope of this essay; however, we will illustrate our approach by showing how two new, non-trivial and useful, user-defined patterns can be created.

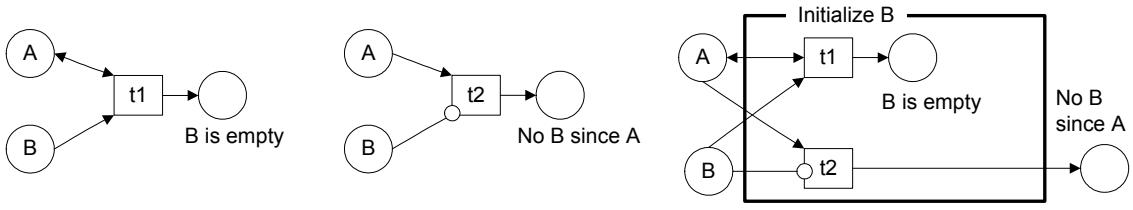
First, we create a new pattern to "initialize *B* when *A* occurs", as shown in Figure 3-14. Thus, when event *A* occurs, already existing occurrences of event *B* must be cleared. This *event initialization* pattern can be created by composing existing patterns as follows:

(1) Using Pattern 6, when event *A* happens, if prior *B* events exist, they are cleared (Figure 3-14(a)).

(2) Then, Pattern 7 is used such that transition *t2* fires when the place for event *B* is empty and puts a token in the place marked "No *B* since *A*" (Figure 3-14(b)).

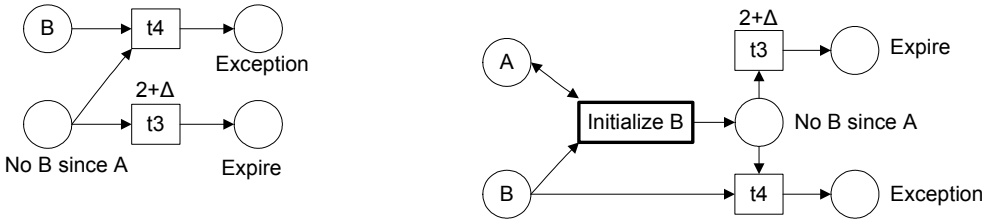
(3) Combine Pattern 6 and Pattern 7 by superimposing common places for events *A* and *B* (Figure 3-14 (c))

Similarly, in Figure 3-15 we show how another new pattern called *consecutive events* can be created using this new *event initialization* pattern as a building block. The *consecutive events* pattern generates an exception event when events *A* and *B* (initialized after *A*) happen within a time interval *T*. To create this pattern, we first apply Pattern 6 to places "No *B* since *A*" and *B*, as shown in Figure 3-15(a). If tokens do not arrive in *B* within the required interval, then tokens in places "No *B* since *A*" are said to expire. Later, we combine the new *event initialization* pattern with Pattern 6, as shown in Figure 3-15(b). We can foresee various applications for these new patterns. For example, the consecutive events pattern could be used in a supply chain to notify the manager if a "machine breakdown" and "no shipment arrival" events occurred within 1 day. Similarly, in telecommunication applications also such consecutive events would be useful [28].



(a) Apply Pattern 6 (b) Apply Pattern 7 (c) Combine Pattern 6 and 7

Figure 3-14: Composing *Event Initialization* Pattern by Combining Patterns 6 and 7



(a) Apply Pattern 6 (b) Combine event initialization and Pattern 6

Figure 3-15: Composing *Consecutive Events Pattern* from *Event Initialization* Pattern and Pattern 6

Clearly, although the seven basic patterns are not exhaustive, the ability to compose them and create new patterns is a powerful feature that allows us to model most realistic situations. Moreover, if necessary, new primitive patterns can also be created from scratch by giving their Petri net description.

3.5 An Example of Event Causality Analysis Using Petri Nets

In this section, we will first show a Petri net that is built using the above seven patterns in the context of a realistic supply chain scenario. Subsequently, we will analyze event causality by simulation and dependency graphs.

3.5.1 Scenario of Events and Rules for a Complete Petri Net

First, we will give an example scenario description. Suppose there is a Vendor Managed Inventory (VMI) arrangement between a distributor and a vendor. In this arrangement, the vendor manages the inventory level for the distributor, proposes new supply orders to the distributor, and ships them after the distributor's approval. The distributor sells products to its customers, and normally ships its customers' orders (for simplicity, we just call them orders) from stock, but whenever there is an out-of-stock situation, a rush supply order is placed with the vendor. When there is more than one out-of-stock event in a week at the distributor, this situation should be considered as a supply chain exception and reported to the supply chain manager immediately. The vendor would usually respond to rush supply orders as soon as possible, but they may be rejected if there is a serious production delay. Moreover, in case of production delay, all supply orders may be delayed. The distributor can contact an alternative vendor for replenishment in case that its normal or rush supply order is delayed or rejected. Figure 3-16 shows all the trading partners in such a supply chain. *For simplicity, in this figure we assume there is one product, but one can similarly model multiple products also as we show in Section 6.* First, we need to identify the events and then write the rules that connect them together. The events of interest are summarized in Figure 3-17 and each event corresponds to a place in the Petri net.

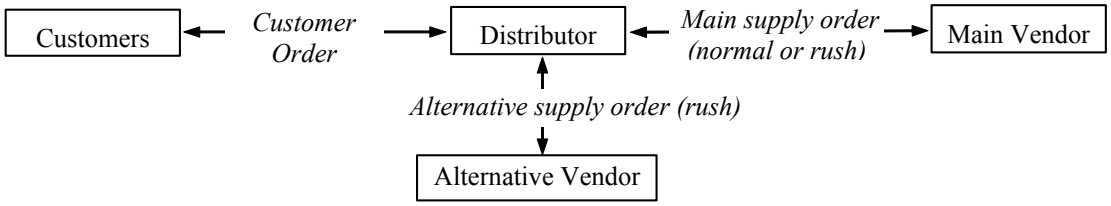


Figure 3-16: Interactions between Trading Partners in the Example Supply Chain

Place (or event) description		
p1: Customer order arrival		p2: Out-of-Stock
p3: Back order		p4: Rush supply order
p5: Rush supply order confirmed		p6: Customer order confirmed
p7: Customer order delayed		p8: Notify customer of order delay
p9: Customer order cancelled		p10: Customer order shipped
p11: Out-of-Stock event expires		p12: Notify supply chain manager
p13: Rush supply order rejected		p14: Production delay
p15: Supply order delayed		p16: Contact alternative vendors
p17: Stock unavailable when delivery is due		p18: Rush supply order shipped
p19: Alternative sourcing failed		p20: Customer order rejected
p21: Production delay (p14) resolved		p25: Back order cancelled
p22: p2′ expired	p23: p3′ expired	p24: p5′ expired
p26: p5″ expired	p27: p6′ expired	p28: p2″ expired

Figure 3-17: Possible Events in the Supply Chain

Next, we consider the rules that relate these events to one another, and also refer to the corresponding patterns used for modeling these rules (in parentheses).

1. When a customer order arrives (*p1*) and there is no out of stock (not *p2*), the order is confirmed (*p6*). (Pattern 7: *Non-occurrence*)
2. When a customer order arrives (*p1*) but there is an out-of-stock (*p2*), a back order (*p3*) is generated. (Pattern 6: *N causes – 1 result*)
3. When a back order occurs, a rush supply order with lead time *L1* is sent to the vendor (*p4*). (Pattern 1: *Simple cause-effect*)

4. When the rush supply order is confirmed by the vendor (*p5*), the back order is also confirmed to the customer (*p6*). A back order must be confirmed within *L2+T3*, where *L2* is the lead time of the back order, and *T3* is the maximum allowed delay time; otherwise, it expires and is cancelled (*p25*) (Pattern 6: *N causes –1 result*)
5. If there is a production delay (*p14*), any incoming rush supply order is rejected (*p13*), because there is no production capacity left to fulfill any rush supply order in a short time. Otherwise, the rush supply order is confirmed. A production delay can be resolved in time interval [*a*, *b*]. (Pattern 6: *N causes – 1 result*; Pattern 7: *Non-occurrence*)
6. A rush supply order is shipped during time [0, *L1*] if there is no production delay (not *p14*). (Pattern 7: *Non-occurrence*)
7. A production delay (*p14*) can cause a supply order delay for more than time *T4* (*p15*) and can also lead to unavailable inventory when customer order delivery is due (*p17*). (Pattern 5: *1 cause – N results*)
8. If a rush supply order is rejected (*p13*) or delayed (*p15*) for more than time *T4*, contact alternative vendors for alternative sourcing (*p16*). (Pattern 4: *1 of N causes-single result*)
9. When a rush supply order is shipped (*p18*) by one of alternative vendors, the corresponding back order can be confirmed (*p6*) and shipped (*p10*); otherwise, the customer order can be rejected (*p20*). (Pattern 6: *N causes – 1 results;* Pattern 1: *Simple cause-effect*)
10. When a supply order is shipped from a vendor (*p18*), inventory is available for delivery (so if there is a token in *p17*, it is removed). (Pattern 6: *N causes – 1 result*)

11. When delivery is due, if inventory is available (not *p17*), the order is shipped (*p10*). (Pattern 7: *Non-occurrence*)

12. a. If an order (with lead time *L2*) has not been shipped in time *L2* after it is confirmed (p6), the order is delayed (*p7*).

    b. If an order is delayed (*p7*) more than time *T3*, then the order is cancelled (*p9*). (Pattern 3: *Inclusive choice*)

13. If there are two unresolved out-of-stock events (*p2*) during time *T2*, the supply chain manager is contacted immediately (*p12*). (Pattern 2: *Repeat_cause-one_effect*)

14. If the order is delayed (*p7*), notify the customer at time *T1* (*p8*). (Pattern 1: *Simple cause-effect*)

The above 14 rules can be formulated in terms of colored time Petri nets as shown in Figure 3-18. The darkened places in the figure are input events of this net. Place *p1* contains two different tokens representing the two order arrivals. Events which are not consumed by event rules are transformed into multiple places, such as *p2*, *p2'*, and *p2''*, where *p2* holds tokens for events, and the others are special mechanism for preventing repetitive firing of transitions.[3]

This Petri net was implemented using CPN Tools [75]. CPN Tools is a graphical computer tool supporting Colored Petri nets (CPN). The details of the CPN implementation are given in Appendix A. In addition, since CPN Tools does not explicitly support time, a workaround is introduced to add temporal constraints to a

[3] This point was explained in Section 3.4.1in the non-consumption case

transition.[4] Figure 3-19 (b) shows an implementation of Rule 1 and Rule 2 from the detailed supply chain example above (See Figure 3-19(a)) in CPN Tools. In Figure 3-19(b), "ORDER", "STOCKOUT", and "BACKORDER" are color sets of different places. Place *p1* contains two tokens. For example, 1`(2, [a, c])@10 means there is one token (1`) with color (2, [a, c]) and timestamp 10 (@10). Note that "2" is the order number (of integer data type) and [a,c] is a list of products (of list data type), which here denotes two products, "a" and "c" for order 2. In addition, Figure 3-19 (b) is a hierarchical Petri net. The tag "rule1&2" attached to a transition shows this transition can be substituted with the sub-Petri net of Figure 3-19(a). It should also be noted that an inhibitor arc is substituted by an equivalent structure with normal arcs, so inhibitor arcs also reflect causal relationships between events. For example, in Figure 3-19(b), *p6* (confirmed order) depends on both *p1* (order arrival event) and *p2_1* (out-of-stock). More precisely, an occurrence of *p1* and non-occurrence of *p2_1* lead to *p6*.

[4] CPN Tools supports a notion of time. However, since it is an executable language allowing for automatic simulation it requires deterministic or stochastic time. Hence, the interval times are translated into guards based on an explicit clock.

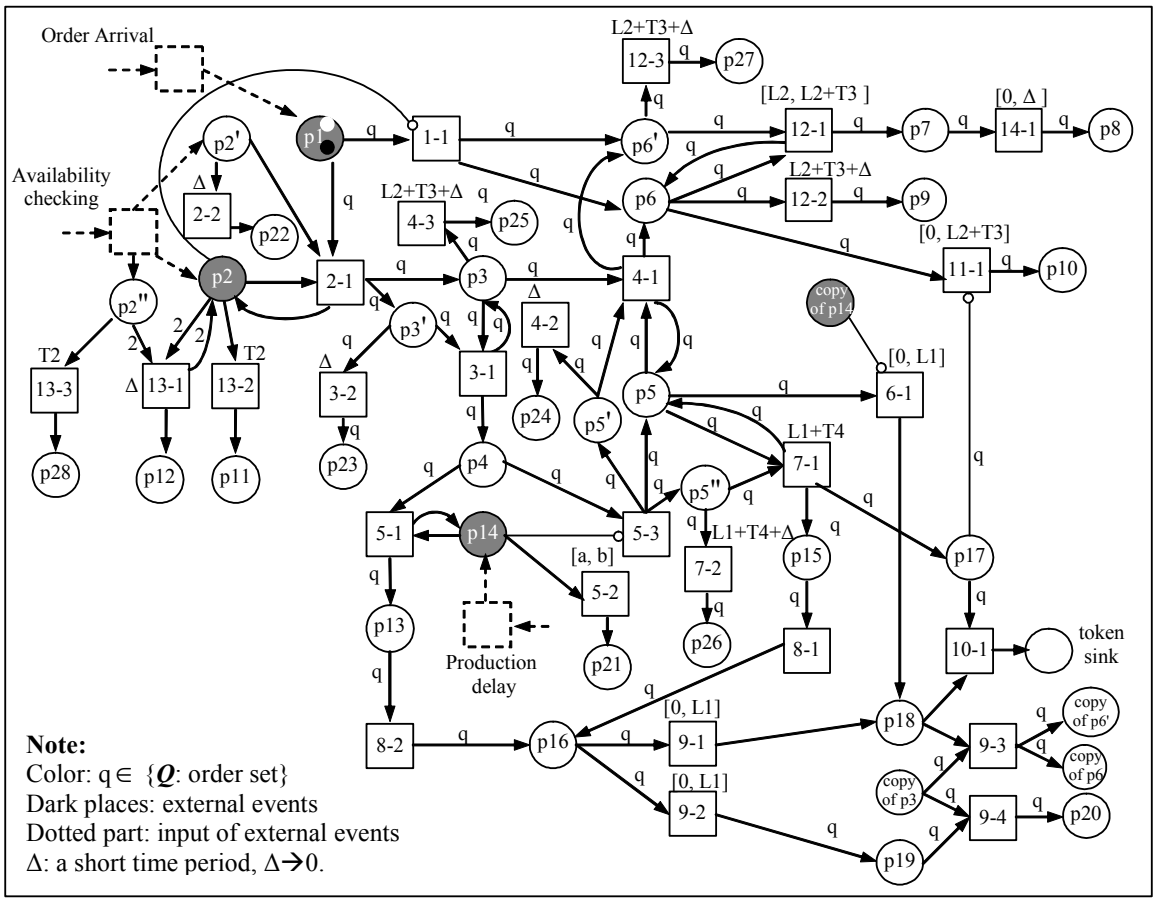
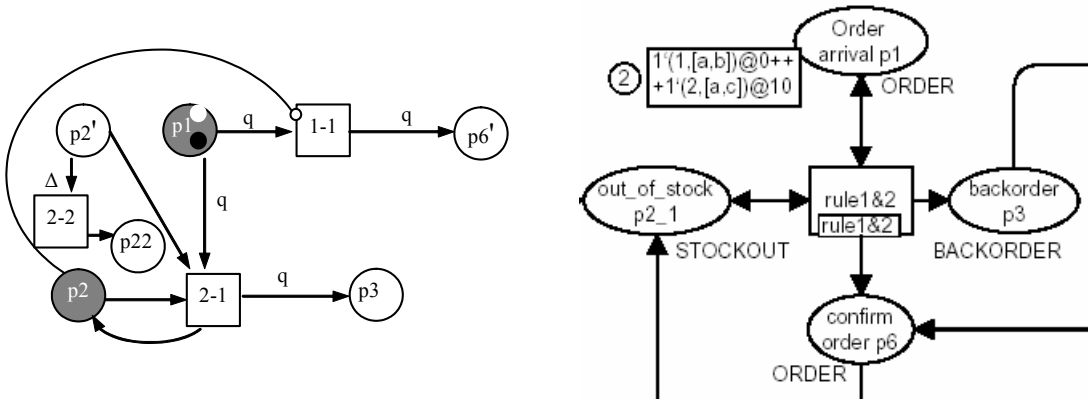


Figure 3-18: A Supply Chain Event Petri Net



(a) Rules 1 & 2 (b) CPN implementation

Figure 3-19: Mapping Rules 1 and 2 (in Figure 3-18) to CPN Tools

Details of CPN Tools and hierarchical Petri nets can be found in [40]. Next, we will analyze events using *dependency graphs* based on running this model.

3.5.2 Dependency Graph Analysis

The Petri net shown in Figure 3-18 (and its CPN Tool representation shown in Figure A-1, Appendix A) can be considered as an "event machine," i.e. when fed with input events, it will generate a set of composite events (both intermediate and final), and show the causal relationships between them. The behavior of this "machine" for the life of a particular instance or for a given time period can be represented by a simple dependency graph [31]. A *dependency graph* is a cause-effect graph of events produced from one or more Petri net instances (say, one or more orders) over a time period. The dependency graph is created from the Petri net by using the rule that *the output event*(*s*) *of a transition depends upon its input event*(*s*).

By executing the Petri net model with actual case data, we can create dependency graphs to show causal relationships that actually transpired between events. Table 3-1 describes the sequence of event occurrences and the transitions that fire when the events take place. The relationships are reflected in Figure 3-20 that shows an event dependency graph generated based on the Petri net of Figure 3-18. Moreover, it also gives the correspondence between place numbers and event numbers. (Note that the events and the corresponding place numbers are not always the same.) The table also gives time values in the last column. These times are based on assigning suitable values for a hypothetical case to the parameters of Figure 3-18 as follows in time units (say, days):

L1 = 20, *L2* = 50, *T1* = 1, *T2* = 50, *T3* = 20, *T4* = 10, *a* = 60, *b* = 80.

Table 3-1: A Trace of Possible Event Sequence Generated from Figure 3-18

Event	Description	Trans. fired	Place	Time
E1	Order *O1* arrival	-	p1	0
E2	Out-of-Stock of product *A* (forO1)	-	p2	0
E3	*O1* is on back order	1-1	p3	0
E4	Rush supply order *R1* is placed for *O1*	3-1	p4	0
E5	Supply order *R1* is confirmed to customer	5-3	p5	0
E6	Order *O1* is confirmed	4-1	p6	0
E7	Order *O2* is received	-	p1	10
E8	Product *A* is out-of-stock (forO2)	-	p2	10
E9	*O2* is placed on back order	1-1	p3	10
E10	Rush supply order *R2* is placed for *O2*	3-1	p4	10
E11	Contact supply chain manager	13-1	p12	10
E12	Product *A* production is delayed	-	p14	10
E13	Rush supply order *R2* is rejected	5-1	p13	10
E14	Alternative vendor is contacted for *R2*	8-2	p16	10
E15	Rush supply order *R1* is delayed for time *T4*	7-1	p15	30
E16	Product A is unavailable when *O1* is due	7-1	p17	30
E17	Alternative vendor is contacted for *R1*	8-1	p16	30
E18	Rush supply order *R2* is shipped from the alternative vendor (i.e., non-occurrence of event "product unavailable when *O2* due")	9-1	p18	30
E19	Order *O2* is confirmed	9-3	p6	30
E20	Order *O2* is shipped	11-1	p10	31
E21	Order *O1* is delayed	12-1	p7	50
E22	Alternative sourcing attempt for *R1* failed	9-2	p19	50
E23	Notify customer about order *O1* delay	14-1	p8	50
E24	Order *O1* is cancelled	12-1	p9	71

Figure 3-20 enables us to analyze the various events and their causes. The events that represent *exceptions* are shaded in this figure. The consequences of a particular event can be traced forward along this directed graph, while the causes of it should be traced backwards until one or more root nodes are reached. For example, it is not difficult to see that *E8* and *E12* are the main causes of exception *E13*, i.e., product *A* was out of stock with the distributor and a rush supply order *R2* was issued, but this rush supply order was

rejected by the vendor because of a production delay. Similarly, the graph shows that the ultimate exceptions resulting from *E12* are *E21*, *E22* and *E24*. The sequence of main events is as follows:

Production is delayed (*E12*)→*rush supply order R1 is also delayed* (*E15*)→*another vendor is contacted* (*E17*)→*alternative sourcing failed* (*E22*)→*Order O1 cancelled* (*E24*)

Moreover, notice that the exception *E11* (notify supply chain manager) happens because of two stock-out events of product *A* within 50 time units as denoted by events *E2* and *E8*. Thus:

Stock out of A for order O1 (E2) & Stock out of A for order O2 (E8) → Notify supply manager (E11)

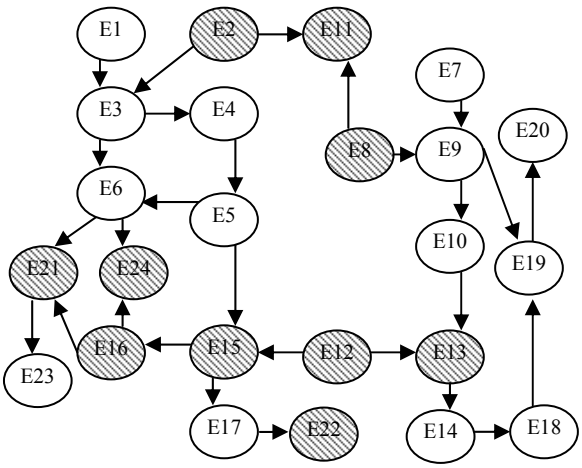


Figure 3-20: Dependency Graph of Table 3-1 (Exceptions Shaded)

Actually, Figure 3-20 only shows one possible scenario and gives the ultimate disposition of orders *O1* and *O2* (*O1* was cancelled, while *O2* was fulfilled). Figure 3-21 shows another out-of-stock situation during order fulfillment; however, now the outcome is different. Here, *E16* (Product *A* unavailable when *O1* is due) is resolved by *E18* (Rush supply order for *R2* shipped from the alternative vendor). Therefore, order *O1* is shipped

(*E20*) within its lead-time. Later on, in spite of *E21* (alternative sourcing for *R1* fails), rush supply order *R1* is shipped (*E22*) from the main vendor after some delay. Eventually, order *O2* is also fulfilled (*E24*) by the incoming inventory from rush order *R1*. The modified events for this scenario are shown in Table 3-2 (events *E1* through *E17* are the same as in Table 3-1). Figure 3-21 shows the new dependency graph for these events. Nevertheless, *E11* (notify supply chain manager) still happens as before.

Table 3-2: An Alternative Scenario of Events Generated from Figure 3-18

Event	Description	Trans. fired	Place	Time
… Events *E1* thru E*17* are same as in Table 1 …				
E18	Rush supply order for *R2* shipped from the alternative vendor.	9-1	p18	30
E19	Product available for *O1* (token in p17 removed) (i.e., non-occurrence of event "product unavailable when *O1* due")	10-1	p17	30
E20	Order *O1* shipped	11-1	p10	30
E21	Alternative sourcing for *R1* fails	9-2	p19	50
E22	*R1* Shipped from the main vendor	6-1	p18	80
E23	Order *O2* confirmed	9-3	p6	80
E24	Order *O2* shipped	11-1	p10	80

These two dependency graphs show only two of many possible scenarios and serve to illustrate our approach. The advantage of this approach is that using a Petri net model as an event machine we can generate dependency graphs to predict and analyze different "interesting" scenarios. Moreover, by playing "token games", supply chain managers can explore a large number of possible event dependency graphs which lead to desirable results (e.g. order fulfilled successfully) or significant exceptions (e.g., order cancellation). The design of an algorithm or heuristic that can automatically generate dependency graphs containing such events of interest is left as a future exercise. In this

context it should be noted that it is possible to analyze all possible dependency graphs using reachability analysis techniques [14] for time colored Petri nets. However, as discussed there this is not very feasible for large problems for complexity reasons and heuristic techniques are required. Next, we provide a summary of simulation results and analyze their implications for supply chain management.

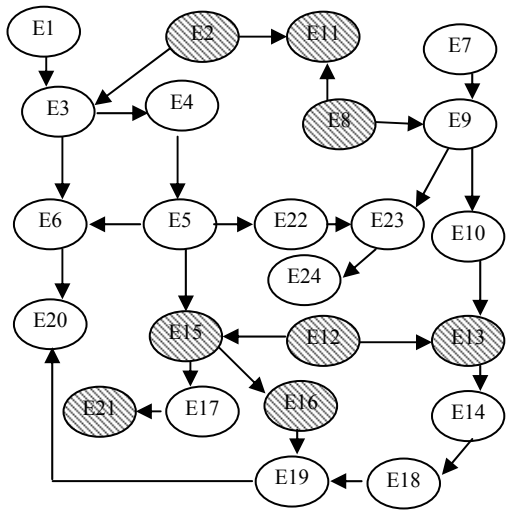


Figure 3-21: Dependency Graph of Table 3-2

3.6 Simulation Results

To demonstrate the practical value of our approach, a detailed simulation experiment was conducted. In this simulation, we generated a large number of customer order arrival events and traced the order fulfillment process in terms of times of occurrence of each event. To make the simulation realistic, we assume there are three products, say *A*, *B*, and *C*. In general, more products can also be supported. Table 3-3 shows the parameters of our simulation experiment.

Table 3-3: Simulation Parameter Settings

Parameter Name	Value or Distribution
Set of items in a customer order	Random selection from three products: A, B, and C
Customer order inter-arrival time	Exponential distribution with mean of 7 time units
Prob. of successful alternative sourcing (PSAS)	0.5
Inter-arrival time between production delayed events	Exponential distribution with mean of 100
Resolution time for production delay (RT)	Uniform distribution range [60, 80]
Normal supply arrival schedule	2 arrivals for each item every 30 time units

The simulation runs for a period from 0 to 3500 time units. 500 customer orders are generated and processed. Among them, 445 orders were successfully shipped, and the other 55 orders were cancelled or rejected because of out-of-stock events and failures to find alternative sourcing. In Table 3-4, the "baseline case" column summarizes the number of main events generated during the simulation interval. Table 3-4 also shows that although about one quarter of customer orders (135 out of 500) occur in stock out situations, yet most of them (84 out of 135) can still be successfully fulfilled through rush supply orders. In addition, about 10% of customer orders (48 out of 500) are fulfilled by alternative sourcing, which shows that alternative sourcing is important.

Table 3-5 shows the detailed distribution of out-of-stock events by product. Each product accounts for about one third of these 182 out-of-stock events. In practice, it may be difficult for a supply chain manager to trace each one of these 182 events individually. Using Rule 13 (see Section 3.5.1), we can filter these events and reduce the number of events sent to the manager. Thus, the supply chain manager may be notified only when there are *two out-of-stock events* within a 50 time unit interval. Therefore, the number of events which needs management attention is reduced to 80, about 40% of the original

number of events. Moreover, the manager can adjust Rule 13 to further reduce this number suitably.

Table 3-4: Comparing Different Strategies in Terms of Events

Events	Baseline case	Strategy 1	Strategy 2
	RT = [60, 80], PSAS = 0.5	RT = [30, 50]	PSAS = 0.7
Order arrivals (p1)*	500	500	500
-- Customer order shipped (p10)	445	473	475
-- Customer order cancelled (p9)	4	3	1
-- Customer order rejected (p20)	47	21	20
-- Back order cancelled (p25)	4	3	4
Out-of-stock events (p2) *	182	182	182
Production delay (p14) *	35	35	35
Customer order delayed (p7)	4	4	1
Back order (p3)	135	135	135
rush supply order (p4)	135	135	135
rush supply order fulfilled	84	111	111
-- by main vendor	36	77	33
-- by alternative vendors	48	34	78
Rush supply order rejected by main vendor (p13)	102	60	102
Supply order delayed (p15)	8	16	6
Contact alternative vendors (p16)	110	76	108
Alternative sourcing failed (p19)	62	42	30
Performance Indexes			
Customer order fill rate	89%	95%	95%
Average customer order fulfillment time	28	27	28
Average replenishment time of rush supply orders (main vendor)	54	18	27
Average replenishment time of supply orders (alternative vendors)	10	10	11

*: These are input events. The three strategies have the same input events.

In addition, the events in Table 3-4 can be used to calculate key performance indexes of the supply chain. As Table 3-4 shows, the fill rate of customer orders is 89% and the average time between an order arrival and the shipment of the order is 28 time units. In addition, on average, it takes 54 time units for the main vendor to replenish rush

supply orders, because production delays occur frequently (35 delay events) and they last a while before being resolved. In contrast, it takes a shorter average time (10 time units) to get supplies from alternative vendors. In general, since the customer order fill rate is somewhat low, the performance of this system may need to be improved. We show next how this can be done with our approach.

Table 3-5: Numbers of Out-of-Stock Events

Products	A	B	C	Total
Out-of-stock events	53	66	63	182
Notify supply chain manager of out-of-stock events	24	29	27	80

An important aspect of our approach is the ability to do sensitivity analysis. To show how such analysis can help to improve the performance of this supply chain, we alternately considered the effect on performance of changing two parameters: reducing the resolution time of production delays (Strategy 1), and increasing the probability of finding alternative sourcing (Strategy 2). Strategy 1 considers the possibility that a production delay can be resolved in a time interval [30, 50] instead of [60, 80]. For Strategy 2, another alternative vendor is introduced into the supply chain so that the probability of finding alternative sourcing is increased to 0.7. The simulation results of these two strategies are also shown in Table 3-4. Using Strategy 1, although there is a large number of back orders, more than a half of them (77 out of 135) are still delivered through successful rush supply orders from the main vendor (only 34 back orders are replenished by alternative vendors). For the second strategy, 70% of back orders (78 out of 111) are fulfilled by alternative vendors. Both strategies lead to an increase in the fill

rate of customer orders. Thus, compared with the baseline strategy, Strategy 1 and Strategy 2 can increase the fill rate to 95%. Similarly, other scenarios can be explored and analyzed in detail with this technique.

3.7 Comparison with Related Work

Related research for detailed modeling of supply chains is still limited. Active databases rely on event-condition-action (ECA) rules [65]. Such rules make databases "active" by allowing them to react to events, i.e., when an event occurs, if some conditions hold, an action (such as database update, insert, query) is taken. However, the drawback of ECA rules is that they cannot do event chaining in a natural way, and hence cannot easily facilitate the analysis of cause-effect relationships between events. Moreover, they are also unable to trace back the causes of events, or forecast future events. Finally, temporal attributes cannot be modeled explicitly.

One domain in which event management has been studied with considerable interest and success is the area of network management. Here the objective is to manage large number of low-level events that may be related and to extract high-level events that require management attention while ignoring the unimportant ones. Hasan et al. [34] provide a conceptual framework for describing causal and temporal relationships between network events. In [31], Gruschke give a dependency graph based algorithm for event correlation in networks. This algorithm is used to map raw events in the network to faulty objects based on the links in the graph. These approaches are relevant in supply chains also, but they lack a precise representation of temporal constraints. In [18], Time Petri

nets are integrated into databases and used for semantic mapping of events in computer networks. The transitions are associated with guard conditions expressed as database constraints. It is an interesting approach with possible applications in supply chains, but harder to implement and verify. In particular, there is no standard approach to transform an event rule to a Petri net and time constraints are captured in an ad-hoc way. Case-based approaches for event correlation in networks are given in [54]. These methods compare a new case against a database of cases and look for stored solutions; however, they require an application-specific model and are computationally complex. Rule-based or knowledge-based approaches are discussed in [28, 94]. Here the knowledge of the expert is described in rules and the rules are applied to diagnose a new problem. However, formal representations of rules are not provided, and hence it is difficult to extend those approaches to other domains.

Other related work includes a proactive SCEM system with agent technology discussed in [16]. While it focuses on event monitoring and alert generation, this system lacks the capability of analyzing events and suggesting solutions. Patterns have been studied in many domains, but the ones developed in the context of workflow management [5, 80] are the closest to our work; however, they do not address the complex temporal constraints. Classical Petri nets have been used to model rules in knowledge bases [57, 67, 97]. However, high-level time colored Petri nets are naturally more expressive because, besides capturing temporal constraints elegantly, they can support a rich vocabulary of event rules, such as sequence operators (*and*, *or*), modifiers (*last*, *nth*, *any*, *none*), and predicates [28]. We have modeled the part of this vocabulary that is relevant in supply chains, such as *and*, *or*, *any*, and *none*, in these seven patterns because our

focus is on the most common patterns that arise in supply chains. The other part of the vocabulary consisting of modifiers can also be modeled as a further extension using the concepts of guards or multi-set colors in Petri nets (see [40, 41]).

3.8 Conclusions

We developed an approach for modeling event relationships in a supply chain through Petri-nets. The formalism consists of seven basic patterns that capture cause-effect relationships in Petri-nets. These patterns can be combined together as building blocks to create other patterns and also more complex Petri-nets. We used a very extensive example to illustrate this approach and showed in detail how dependency graph analysis can be used to determine causal relationships between events in a dynamic supply chain. It should be noted that these relationships are complex and depend upon the exact timing of events. We demonstrated that slight changes in temporal relationships can result in a very different dependency graph and also final outcome.

Petri net simulation offers a mature technique for analyzing the Petri net models, and the easy availability of many Petri net software packages is an asset. We implemented Petri net models using CPN tools and performed sensitivity analysis by simulation. By changing a specific event parameter, such as event resolution time, we can show how supply chain performance is affected. Such scenario analysis can suggest solutions to improve supply chain performance. Therefore, by managing events, we can actually manage supply chain performance. We ran comprehensive simulation

experiments illustrated how this approach can help decision makers to improve supply chain performance.

In summary, as supply chains become more tightly integrated across partners, it is becoming increasingly important to respond in real-time to events (also called sense-and-respond capability). We described a novel approach to model event relationships in a supply chain using Petri-net patterns that can be combined to create realistic Petri-net models of supply chains. We further implemented a model in a Petri-net modeling and simulation tool, and ran simulation experiments with it. A unique feature of the approach is that the Petri-nets are constructed from patterns or building blocks which can be composed together and extended to create new user-defined patterns.

In future work, we would like to develop more formal verification techniques for the supply chain models, and also develop heuristics for reachability analysis of dependency graphs to predict "interesting" events.

Chapter 4

An Analysis, Taxonomy and Correctness Algorithms for Unstructured Workflows

Abstract: Most workflow tools support structured workflows despite the fact that unstructured workflows can be more expressive. This is because unstructured workflows are more complex, and thus also more prone to errors. In this essay, we describe a taxonomy that serves as a framework for analyzing unstructured workflows consisting of both acyclic patterns and loops. The taxonomy characterizes unstructured workflows in terms of two considerations: improper nesting and mismatched pairs. We also introduce a relaxed notion of correctness called weak correctness, as opposed to the conventional notion of strict correctness. Then, we develop a framework for analyzing unstructured workflows in terms of weak correctness and give an algorithm for diagnosing workflows. The diagnosis algorithm detects structural flaws and reports causes for blocked nodes, deadlocks and multiple instances. The results of this study will be useful for researchers investigating expressiveness and correctness issues in unstructured workflows.

4.1 Introduction

Workflow technology has emerged as an important tool for businesses to integrate and automate business processes, not only within a company, but also throughout the entire supply chain, giving rise to complex inter-organizational processes. Process modeling involves methodologies for designing business process models properly before they are implemented as workflows [29]. It is essential that process models not only precisely capture business requirements but also ensure successful workflow execution. If a process is put into production before being carefully checked and verified, it could fail to execute properly and cause considerable loss to a business. A correct process model is one without structural flaws, such as deadlocks, livelocks, and lack of synchronization [4,

81]. Therefore, it is very important that the correctness of workflows be verified before hand, i.e. before the process models are implemented. This requires a systematic way to detect and fix structural flaws at design time. Moreover, as interest in web services grows [7], it will be increasingly important to use process modeling methods to create composite web services that can combine several individual services in a meaningful way.

Traditionally, structured workflows have been widely used to model business processes because of their guaranteed correctness. However, as workflow processes become more complex, those processes cannot always be represented in a structured way. Moreover, structured workflows are not able to offer the desired flexibility and expressive power. Hence, unstructured representations that offer greater expressive power are required. Also, a lot of business processes are designed by non-technical end-users. From the point of view of such a user, there is more flexibility to be able to design a workflow with both structured and unstructured patterns and then have a diagnosis algorithm to analyze it. Based on the analysis, the algorithm may either convert the workflow design into an equivalent structured representation, or may point out any errors for the user to correct and then rerun the algorithm.

To illustrate an unstructured workflow, consider Figure 4-1 that shows a simple workflow for handling orders in a just-in-time supply chain. In this scenario, after an order is received (*A1*), the available supply is checked (*A2*), and accordingly, two orders are placed for standard items (*A3*) and special items (*A4*) in parallel with separate suppliers. If special items are immediately available, the supply schedule is updated (*A5*). After both standard and special items are received, the order is assembled and

delivered (*A6*). However, if special items are not available, an alternative supplier must be found (*A7*) and a new schedule for delivery is proposed to the customer (*A8*). Unfortunately, the scenario described in Figure 4-1 cannot be precisely modeled as a structured workflow. One approximate way to model it in a structured way is by placing *A3* and *A4* in a sequence. However, this would not capture the desired semantics.

In addition, Figure 4-2 shows another unstructured workflow which models a paper review process. In [76], this workflow is called *parallel branching with final selection*. In this process, a manuscript is reviewed by two people. As soon as one review is received, the decision is made and the other reviewer is notified not to send a review. Again, this realistic process cannot be modeled as a structured workflow, and is characterized later as an *AND-OR mismatched pair*. However, this workflow may raise the problem that *A2* and *A3* are not synchronized properly, even though it works well in practice.

These workflows also raise several questions. For instance, returning to the workflow in Figure 4-1, is the workflow correct? If the standard and special items are immediately available, then the workflow finishes normally. However, if the special items are not available, then an alternative path is taken via *A7* and *A8*. In this case, the workflow gets blocked at an AND-join node *C1J*, and, one path of the workflow reaches the end, while another path remains unfinished. We treat such a workflow as *weakly correct* (as opposed to *strictly correct*) because it did finish, even though one path (containing *A3*) did not synchronize properly. Later we will characterize this example as one of *AND-OR improper nesting*. Such situations arise only in unstructured workflows and therefore, new relaxed notions of correctness are required. With regards to the

blocked And-Join at *C1J*, it is possible to add another step *A9* that would return the standard items purchased and, in effect, undo *A3*. Thus, this workflow will not deadlock, even though there is redundant work in the form of *A3*. Also, if the standard items can be used later, there is no need to return them at all.

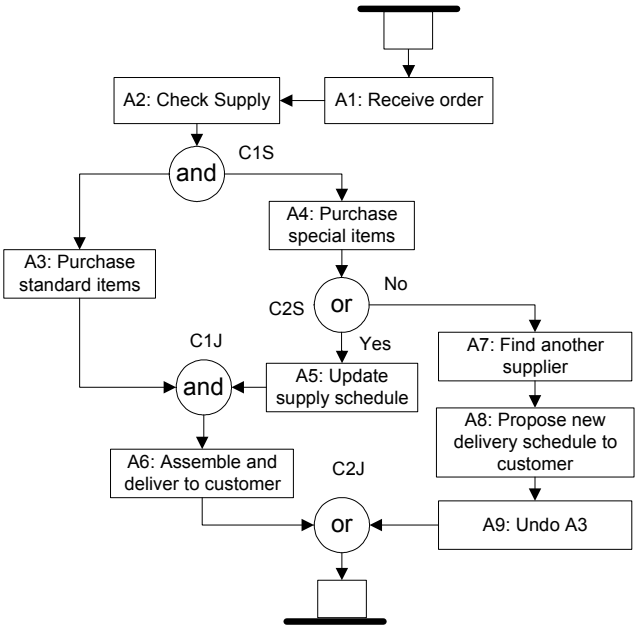


Figure 4-1: Order Handling in a JIT Supply Chain (AND-OR Improper Nesting)

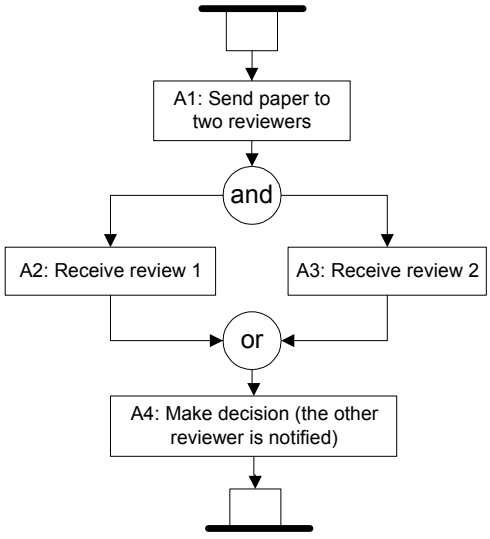


Figure 4-2: A Paper Review Process (AND-OR Mismatched Pair)

We argue in this essay that workflows shown in Figure 4-1 and Figure 4-2 should be designable and also verifiable. In the case of Figure 4-1, if an improper exit occurred, causing *A3* to remain as an unfinished path, possible solutions are to undo *A3* automatically or by a user-defined step shown as *A9* in Figure 4-1, or inform the user about it during verification and let the user decide. Similarly, for the workflow in Figure 4-2, possible ways of handling multiple instances are to ignore the second instance at the Or-Join node, and provide support for the additional instance (such as generating an exception or a notification). However, a critical challenge lies in how to verify such workflows and show that they are weakly correct. Therefore, verification assumes great importance.

The second question relates to determining and analyzing the structural anomalies in a workflow. If execution problems exist, what structural flaws cause them? How can those execution problems be detected based on the workflow structure and how can structural flaws be corrected? These questions will be answered in this essay.

Workflows allow coordination of various activities in a process through control elements such as *AND-Split*, *AND-Join*, *OR-Split* and *OR-Join*. One accepted notion of correctness is structuredness. A *structured workflow* is one in which each split control element (e.g., AND, OR) is matched with a join control element of the same type, and they are also properly nested. However, not all workflows are structured; some unstructured workflows give more expressive power than structured ones, and are also correct. Thus, the requirement of structuredness is restrictive. In this essay, we introduce notions of *improper nesting* and *mismatched pairs* as a means to organize our taxonomy of unstructured workflows. The taxonomy serves as a means of analyzing the main

building blocks that constitute a given workflow model and helps us in determining the correctness of this workflow model. Also, following the pioneering work of Kiepuszewski et al. [45], we categorize unstructured workflows that have equivalent structured mappings and those that have what we call *quasi equivalent* structured mappings. Quasi-equivalence is a relaxed notion of equivalence based on uni-directional bisimulation [45], and it allows multiple instances of an activity to exist concurrently. We will show later that in certain situations, multiple instances do not cause correctness problems. Finally, we develop a diagnosis algorithm that can identify structural flaws of a workflow and point out the causes of such flaws, derive any structured mappings if they exist, and draw conclusions on the correctness of this workflow. In particular, we use a notion of weak correctness, which relaxes the traditional concept of strict workflow correctness and emphasizes proper termination [35] of a workflow. This concept can also be used to include contingency plans in workflow models and to handle exceptions.

Related work in this area is still limited, some notable studies being [3, 15, 19, 45, 76, 82, 89]. A graph reduction technique is proposed in [82]. Although this technique can detect some structural conflicts through a reduction process, it gives no details about the causes of these conflicts, and, therefore, provides no help for further improvement. A verification approach based on workflow decomposition is given in [19]. This approach may not be able to verify unstructured workflows that are hardly decomposed. [45] addresses the possibility that an unstructured workflow can be mapped to a structured one through equivalence preserving transformations, but the discussion is mainly through examples, and lacks generality. The importance of structural consistency and correctness is also addressed in the context of the ADEPTflex model [76]. Logic-based approaches

for workflow verification are discussed by Bi and Zhao [15]. The approach of van der Aalst and Verbeek is based on converting a workflow into a Petri net and then checking correctness using a tool like Woflan [3, 89]. The drawback with this approach is that after the workflow is converted into a Petri-net, it loses its natural structure and is less understandable by a non-expert end-user. In contrast to those approaches, our diagnosis algorithm not only provides a Yes/No answer on the correctness, but also gives structural evidence of execution problems as well as suggestions for corrections.

This essay is organized as follows. Section 4.2 gives workflow definitions and introduces our taxonomy. Structural flaws are discussed and verified under the framework in Section 4.3. Section 4.4 discusses the possibilities of transforming an unstructured workflow to structured one. Section 4.5 considers situations where loops are present. The algorithm as well as diagnosis experiments are described in Section 4.6. Section 4.7 summarizes this essay and concludes with a brief description of future work.

4.2 Workflows Definitions and Taxonomy

4.2.1 Workflow definition

Definition 1 (**Workflows**) A workflow is a directed graph consisting of activities, arcs, and control elements. The control elements are of the following types: *start, end, AND-Split*, *AND-Join*, *OR-Split*, and *OR-Join* where**:**

(1) Arcs are used to connect activities and control elements.

(2) *In-degree* $d^-(n)$ and *out-degree* $d^+(n)$ indicate the number of arcs entering and leaving a workflow node n respectively.

(3) Each workflow has only *one* start node and *one* end node. For a start node s, $d^-(s) = 0$ and $d^+(s) = 1$; for an end node e, $d^-(e) = 1$ and $d^+(e) = 0$.

(4) For any activity a, $d^-(a) = 1$ and $d^+(a) = 1$.

(5) For any join element j, we require $d^-(j) = 2$ and $d^+(j) = 1$. Similarly, for any split element s, $d^-(s) = 1$ and $d^+(s) = 2$. The in-degree (d^-) and out-degree (d^+) of an element can determine whether it is a split or a join.

(6) Every activity or control element is in at least one path from the start node to the end node.■

Figure 4-3 shows the graphical notation for workflow nodes. For simplicity, and without loss of generality, we assume that each control element has only two incoming (or outgoing, as the case may be) branches. A join with $d^-(j) > 2$ can be represented by a combination of multiple join elements. Similarly, a split with $d^+(s) > 2$ can be achieved by a combination of multiple split elements. In addition, Definition 1 also implies that, in a workflow, there is at least one path between any two nodes. This rigorous workflow definition simplifies our verification approach, but it by no means limits the application of this approach. Other relaxed workflow models [15, 82] can also be verified by this approach after simple conversions in accordance with Definition 1.

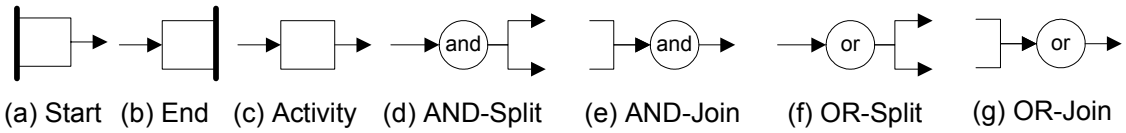


Figure 4-3: Graphical Representations of Workflow Control Elements (or Nodes)

Definition 2 (**Workflow paths**): A workflow path p is a sequence of nodes n_1, n_2,…, n_t (t >1) along with arcs in a directed workflow graph. Path p is denoted as $p = (n_1, n_2, \ldots, n_t)$. n_i is a *upstream* node of n_j and n_j is the *downstream* node of n_i, for $1 \leq i < j \leq t$. ■

Definition 3 (**Workflow instance**): A workflow instance is simply an execution log of the tasks and control nodes actually executed after the start node is initiated, and before either the end node is reached or none of the other nodes can be executed ■

4.2.2 Semantics of Control Elements

Next, we discuss the semantics of the workflow control elements. Semantically, after an AND-Split element, both of its branches can be executed concurrently. Moreover, an AND-Join element can be executed only after both of its incoming branches have been executed. We also assume an OR-Split element has the semantics of *exclusive choice*, i.e., both branches of an OR-Split are exclusive and only one branch can be executed at one time. An *inclusive choice* can be achieved by a combination of And-Split and exclusive choice [5]. The *semantics of an OR-Join element* can be one of the followings:

(1) *Single execution*: The OR-Join element is executed only once after whichever branch is done first. The other branch is discarded when they finish.

(2) *Multiple executions*: The OR-Join element is executed every time an incoming branch is done. However, in a situation where an AND-split node is paired with an OR-join node as in Figure 4-2, the OR-join node can be activated twice, once after each incoming branch. This situation leads to a problem called *multiple instances*, as we will discuss later.

Different workflow tools handle OR-Join nodes differently. In [46], Kiepuszewski compared different behaviors of OR-Join nodes observed with workflow products. For example, IBM MQSeries [38] assumes the semantics of "single execution", while multiple executions can happen with Forte (now Sun eInsight Business Process Manager [86]). With the semantics of "single execution", there will be no execution problem of multiple instances. In this essay, since we focus on verification, we assume each OR-Join node has the semantics of "multiple executions", which could cause correctness issues. We will identify the workflow situations where multiple instances might be created.

4.2.3 Structured Workflows

Structured workflows impose certain restrictions on the relationships between control elements. In particular, in a structured workflow, each AND-Split element must have a corresponding AND-Join element, and each OR-Split element has a corresponding OR-Join element. There are four *basic types of structured workflows* [45]: *sequence*, *decision structure*, *parallel structure*, and *structured loop*, as shown in Figure 4-4. A complex structured workflow can be composed inductively based on these four types or patterns [45]. In addition, any structured workflow (or sub-workflow) can be treated as a

single composite activity [82]. Therefore, we will also use the symbol of activity (see Figure 4-3(c)), to denote any structured workflow or sub-workflow. However, if those restrictions are not followed during workflow composition, the resulting workflow is unstructured. Next, we introduce the taxonomy of unstructured workflows.

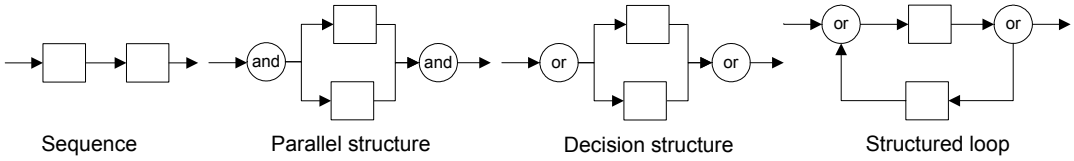


Figure 4-4: Basic Types (or Patterns) of Structured Workflows

4.2.4 Unstructured Workflows – Taxonomy

We categorize unstructured workflows in terms of two considerations, *mismatched pairs* and *improper nesting*. We start with the concept of corresponding control elements.

Definition 4 (**Corresponding control elements**): A split element *s* corresponds to a join element *j*, if two *minimal* paths, starting along two different outgoing arcs of *s,* first join at *j*. This corresponding pair *s* and *j* is denoted by (*s*, *j*). ■

By *minimal* we mean that any node in any path from *s* to *j* does not have the correspondence property with *s*. For example, in Figure 4-5, the corresponding pairs are (*C1S*, *C1J*), (*C2S*, *C2J*), (*C3S*, *C3J*), (*C4S*, *C4J*), but not (*C1S*, *C2J*), since a path from *C1S* to *C2J*, i.e., {*C1S*, *A1*, *C2S*, *A4*, *C1J*, *A7*, *C2J*} is not minimal as it already contains *C1J*, a corresponding element of *C1S*. Similarly, we do not have a correspondence (*C1S*,

C3J) as *C1J* is in a path from *C1S* to *C3J*. Next, we state some simple lemmas without proof.

Lemma 1: Every split control element must have at least one corresponding join control element.

Lemma 2: In general (unstructured) workflows, the split and join control elements need not be of the same type.

Lemma 3: A split control element may have multiple unique corresponding join control elements. A join control element may correspond to more than one split control elements.

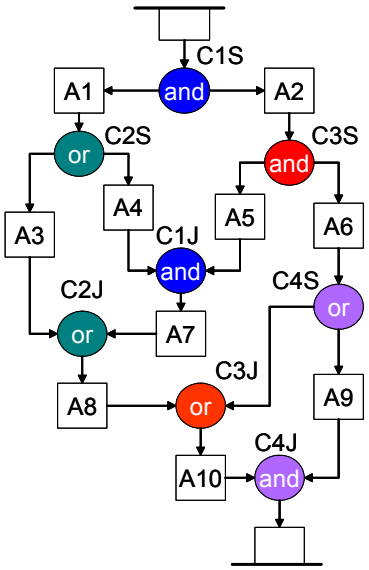


Figure 4-5: An Workflow with Mismatched Pairs and Improper Nesting

Definition 5 (**Mismatched pair**): A pair of corresponding control elements (s, j) is mismatched if s is an OR-Split and j is an AND-Join element, or s is an AND-Split and j is an OR-Join element. (s, j) is called a *mismatched pair*. ■

There are two types of mismatched pairs: (AND, OR) (e.g., (*C3S*, *C3J*) in Figure 4-5) and (OR, AND) (e.g., (*C4S*, *C4J*) in Figure 4-5).

Definition 6 (**Improper nesting**): A pair of control elements (*s*, *j*) is *improperly nested* with another pair of control elements (*u*, *v*), if *s* (or *j*) is in a path from *u* to *v*, but *j* (or *s*) is not in this path. This is denoted as $(u\ ^{s}\ v)\ ^{[}_{]}\ (s, j)$ (or $(u\ ^{j}\ v)\ ^{[}_{]}\ (s, j)$). We also write $(u, v)\ ^{[}_{]}\ (s, j)$ to denote the general situation where (*u*, *v*) and (*s*, *j*) are improperly nested but the locations of nodes are not specified. ■

In Figure 4-5, there are four improper nestings denoted as: $(C1S\ ^{C2S}\ C1J)\ ^{[}_{]}\ (C2S\ ^{C1J}\ C2J)$, $(C1S\ ^{C3S}\ C1J)\ ^{[}_{]}\ (C3S\ ^{C1J}\ C3J)$, $(C2S, C2J)\ ^{[}_{]}\ (C3S\ ^{C2J}\ C3J)$ and $(C3S\ ^{C4S}\ C3J)\ ^{[}_{]}\ (C4S\ ^{C3J}\ C4J)$. Note that once both *s* and *j* are in one path between *u* and *v*, we say (*s*, *j*) is *properly nested* with (*u*, *v*), although *s* and *j* may not both be in another path between *u* and *v*.

Definition 7 (Order of Improper nesting) $(u, v)\ ^{[}_{]}\ (s, j)$ is called *first-order improper nesting*, if there is no (*x*, *y*), such that $(u, v)\ ^{[}_{]}\ (x, y)$, where $(x, y) \neq (s, j)$. $(u, v)\ ^{[}_{]}\ (s, j)$ is called *nth-order improper nesting* if there exist $(u, v)\ ^{[}_{]}\ (x_i, y_i)$, where $(x_i, y_i) \neq (s, j)$ and i=1, 2, …, n-1. We use $(u, v)\ ^{[}_{]}\ \{(x_1, y_1), (x_2, y_2),\ldots, (x_n, y_n)\}$ to denote n pairs of corresponding elements nested into (*u*, *v*).■

For example, in Figure 4-5, there is second-order improper nesting (*C1S*, *C1J*)$^{[}_{]}${(*C2S*, *C2J*), (*C3S*, *C3J*)} and third-order improper nesting (*C3S*, *C3J*)$^{[}_{]}${(*C1S*, *C1J*), (*C2S*, *C2J*) , (*C4S*, *C4J*)}.

Obviously, a structured workflow is one that does not have any mismatched pairs or improper nesting. On the other hand, unstructured workflows, in general, have

mismatched pairs and/or improper nesting. This lack of structuredness may cause structural flaws and give rise to correctness issues, discussed as follows.

4.2.5 Workflow Correctness

In general, there are two typical structural flaws in workflows: *multiple active instances of the same activity* (or in short *multiple instances*) and *deadlocks* [45, 82]. Next, we give precise definitions for these two structural flaws.

Definition 8 (Multiple Instances) can arise at an OR-Join node when it is paired with an AND-split node (see Definition 4), and thus *both* its incoming paths can potentially be taken separately, and, therefore, the OR-Join and subsequent activities are activated twice.■

Situations shown in Figure 4-2 and Figure 4-5 are examples of *multiple instances*. For example, in Figure 4-5, multiple instances could occur at *A10* as a result of the two paths {*C3S*, *A5*, *C1J*, *A7*, *C2J*, *A8*, *C3J*} and {*C3S*, *A6*, *C4S*, *C3J*} being taken separately after *C3S*. In general, multiple instances arise because of a mismatched (AND, OR) pair, for instance, (*C3S*, *C3J*) in Figure 4-5. Multiple instances can lead to some undesirable results, such as redundant activities and competition for resources.

However, although the workflow shown in Figure 4-2 also has a mismatched (AND, OR) pair, such a pair only does not cause any deadlocks. We classify this kind of workflows as weakly correct. The advantage of allowing this kind of structure is that it gives greater flexibility. However, it is the responsibility of the application to make sure that either all the activity instances are handled properly, or just ignoring the subsequent

ones if doing so produces no harmful effects in the process execution. Thus, we take the view that a diagnosis algorithm should detect and report such cases and then let the process designer decide whether they are acceptable.

Definition 9 (Deadlock) A workflow is *deadlocked* when an AND-Join node is blocked (i.e. exactly one of its incoming paths can be taken) and any node downstream of this node cannot be executed. ■

Definition 10 (Deadlock-free Workflows) A workflow is *deadlock-free* if it never leads to any deadlocks during execution for all possible execution paths. ■

If a workflow is deadlock-free, it can always terminate properly. This is a fundamental requirement for workflow execution; nonetheless, many workflows suffer from this problem. For example, in Figure 4-5, there might be a deadlock at *C4J* if path {*C3J*, *A10*, *C4J*} is taken after *C4S*. Hence, this workflow could never end.

In addition, an unstructured workflow may have another problem, *blocked nodes*, such as *C1J* in Figure 4-1. Moreover, a blocked node could cause a workflow to deadlock.

Definition 11 (Blocked Nodes) An AND-Join node *b* is *blocked* if there exists at least one workflow instance where *exactly* one of its incoming paths can be taken. ■

It should be noted that *only one incoming path* can always be taken for a blocked node. Let *b.in* be a two-character string that records which incoming paths of *b* is taken. If *b.in* is:

"L-": the left incoming path of *b* is taken but the right one cannot;

"R-": the right incoming path of *b* is taken but the left one cannot;

"LR": either incoming path of *b* might be taken but not both.

For example, in Figure 4-5, *C4J* is a blocked node and *C1J*.in = "L-". Note that, if *b.in* = Ø, neither incoming path of *b* can be taken. This situation is not problematic because it just means that the AND-Join will not be activated at all and there is no correctness issue then. The correctness issues only arise when one incoming branch of an AND-join node is activated.

Blocked nodes indicate that a workflow has structural flaws. *A strictly correct workflow should not contain any blocked nodes*. However, a blocked node is only a necessary condition for a workflow to deadlock. For a deadlock, there must be a blocked node, but a workflow with blocked nodes could still be deadlock-free and therefore it is *weakly correct*. Thus, we can define two levels of workflow correctness.

Definition 12 (Strict Correctness) A workflow is *strictly correct* if it leads to neither *blocked nodes* nor *multiple instances*. ■

Definition 13 (Weak Correctness) A workflow is *weakly correct* if it is deadlock-free. ■

Obviously, structured workflows are strictly correct. Workflows shown in Figure 4-1 and Figure 4-2 are weakly correct. Strictly correct workflows have been well studied in [1, 82]. However, [82] cannot be used to check weak correctness of a workflow. Actually, in [82], the difference between deadlocks and blocked nodes is not clearly identified. Sadiq and Orlowska defined "deadlocks" as the situations where paths from an OR-Split node join at an AND-Join node and therefore a workflow path cannot continue any further. This definition is equivalent to our notion of blocked nodes. As a result, the algorithm proposed in [82] cannot tell if a workflow is truly deadlocked or if it only has blocked nodes. For instance, it will conclude that the workflow shown in

Figure 4-1 is "deadlocked". We feel that it is important to clearly make this distinction between blocked nodes and deadlocks since our notion of weak correctness depends on it. In addition, although [1] can show that a weakly correct workflow could have extra tokens remaining in the WF-net after it terminates, the causes of those extra tokens are not clearly identified. In this paper, we focus on weakly correct workflows by analyzing blocked nodes.

4.2.6 Workflow Correctness Taxonomy

Thus, a workflow can be classified into the following categories:

(1) Strictly correct

(2) Multiple instances

(3) Deadlock

a. *Always deadlocked*: all workflow instances are deadlocked.

b. *Some deadlocks*: at least one workflow instance has a blocked node and this node is a deadlock causing node because it cannot be bypassed.

c. *Deadlock-free*: none of the workflow instances is deadlocked. However, some instances may have blocked nodes, but those nodes are not deadlock causing.

Figure 4-6 shows a workflow with first-order improper nesting, $(C1S \overset{C2J}{[} C1J)$ $(C2S \overset{C1S}{[} C2J)$. Table 4-1 shows all combinations of first-order improper nesting situations, and characterizes them in terms of correctness according to the above classification. This allows us to analyze all first-order improper nesting scenarios;

however, note that the results may be different in the case of higher-order improper nesting. We will analyze these later in the essay.

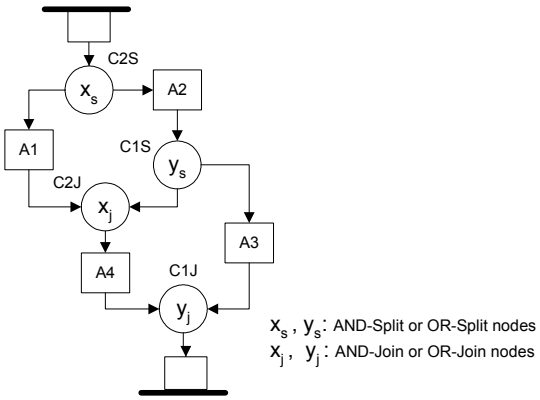


Figure 4-6: First-Order Improper Nesting

Table 4-1: Behavior of First-Order Improper Nesting and Mismatched Pair Types

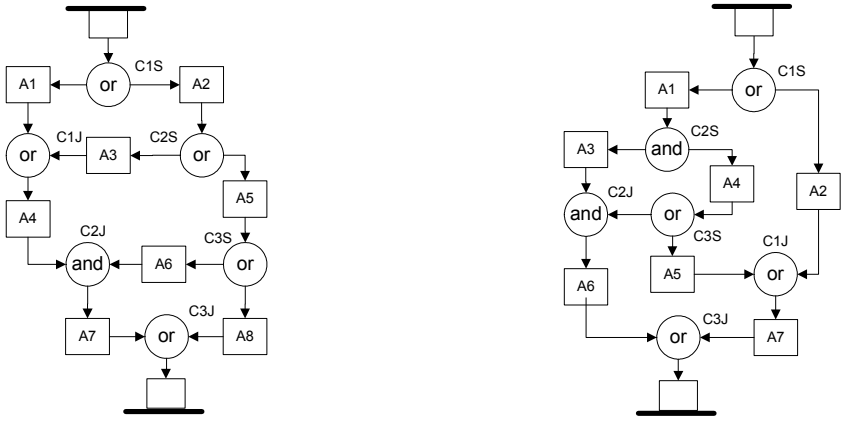
Type	(C1S	C1J)	(C2S	C2J)	Correctness issues	Structured transformation
1	OR	OR	OR	OR	strictly correct	Yes
2	OR	OR	OR	AND	some deadlocks	No
3	OR	OR	AND	OR	multiple instances	q-equivalent mapping
4	OR	OR	AND	AND	deadlock-free	No
5	AND	AND	OR	OR	some deadlocks	No
6	AND	AND	OR	AND	always deadlocked	No
7	AND	AND	AND	OR	multiple instances	q-equivalent mapping
8	AND	AND	AND	AND	strictly correct	No
9	OR	AND	OR	OR	always deadlocked	No
10	OR	AND	OR	AND	always deadlocked	No
11	OR	AND	AND	OR	some deadlocks	No
12	OR	AND	AND	AND	always deadlocked	No
13	AND	OR	OR	OR	multiple instances	q-equivalent mapping
14	AND	OR	OR	AND	some deadlocks	No
15	AND	OR	AND	OR	multiple instances	q-equivalent mapping
16	AND	OR	AND	AND	multiple instances	q-equivalent mapping

4.3 Analysis of Structural Flaws

Mismatched pairs and improper nesting are the main causes for these two structural flaws. In this section, we show analytical results pertaining to those structural flaws. We start with finding blocked nodes and deadlocks.

4.3.1 Blocked Nodes and Deadlocks

In general, *if a workflow is deadlocked*, it implies that an *AND-Join node is blocked and it cannot be bypassed*. We will discuss the idea of bypassing a blocked node shortly. In the two workflows shown in Figure 4-7, both *C2J*s are blocked nodes. In Figure 4-7(a), *C2J.in* = "LR". In Figure 4-7(b), *C2J.in* = "L-". However, in Figure 4-7(a), if path {*C1S*, *A1*, *C1J*, *A4*, *C2J*} is taken, this workflow cannot proceed any further and is deadlocked. However, in Figure 4-7(b), if the path {*C1S*, *A1*, *C2S*, *A4*, *C3S*, *A5*} is taken, even though *C2J* is blocked, yet this workflow can continue to the end and is not deadlocked. We need a way to find blocked nodes. Next, Theorem 1 allows us to narrow down the set of nodes that may be blocked, by identifying *potentially blocked nodes*.



(a) A deadlocked workflow (b) A deadlocked free workflow

Figure 4-7: Two Workflows with Blocked Node *C2J*

Theorem 1: Only the AND-Join nodes either in mismatched (OR, AND) pairs or in improper nestings of the form (AND ^{OR} AND) $^{[}_{]}$ (OR, OR/AND) can be *potentially blocked nodes*.

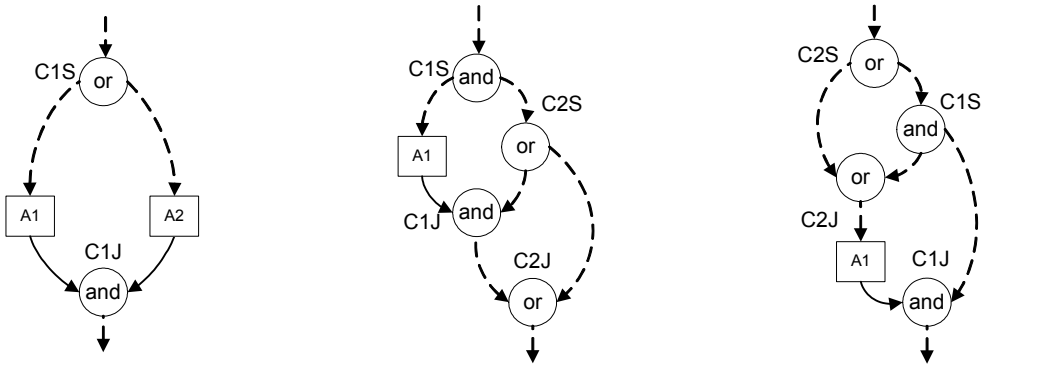
Proof: Since only AND-Join nodes can be blocked, (OR, AND) pairs or (AND, AND) pairs must be present in a workflow with blocked nodes. Thus, there are two cases:

(i) (*OR, AND*) *pair* (see Figure 4-8 (a)). Here only one outgoing path after the OR-Split can be taken, and this path might be the only incoming path of the AND-Join node. Hence, this AND-Join node at *C1J* is a potentially blocked node.

(ii) (*AND, AND*) *pair is blocked*. Here, there are two possibilities: (a) an OR-Split node in the path from the And-Split to the And-Join node takes a new path (away from the AND-Join node), as shown in Figure 4-8 (b); (b) an OR-Join node makes a new path to the AND-Join node without passing the AND-Split node, as shown in the workflow in Figure 4-8 (c). Either possibility is caused by improper nesting (AND ^{OR}

AND)$^{[}_{]}$(OR, OR/AND) where the OR-Split node (or OR-Join node) is in a path between the AND pair, but its corresponding node is not in this path.

Hence, this covers (by enumeration) all the cases where potentially blocked nodes may arise. ■



(a) (OR, AND) pair (*C1S*, *C1J*) *C1J*. in = "LR"

(b) (*C1S* ^{C2S} *C1J*) $^{[}_{]}$(*C2S*, *C2J*) *C1J*. *in* = "L-"

(c) (*C1S* ^{C2J} *C1J*) $^{[}_{]}$(*C2S*, *C2J*) *C1J*. *in* = "L-"

Figure 4-8: Examples with Potentially Blocked Node *C1J*

From Theorem 1, we can immediately conclude that an unstructured workflow that only has improper nesting in the form of (AND, AND)$^{[}_{]}$(AND, AND) or (OR, OR)$^{[}_{]}$(OR, OR) never results in any potentially blocked nodes. Moreover, it never leads to multiple instances since it has no mismatched (AND, OR) pairs. Therefore, such a workflow is strictly correct. An example is shown in Figure 4-9. This workflow contains improper nesting (*C1S*, *C1J*) $^{[}_{]}${(*C2S*, *C2J*), (*C3S*, *C3J*)} in the form of (OR, OR)$^{[}_{]}$(OR, OR), and (*C4S*, *C4J*)$^{[}_{]}$(*C5S*, *C5J*) in the form of (AND, AND)$^{[}_{]}$(AND, AND). Thus, we state Lemma 4 without proof as follows.

Lemma 4: An unstructured workflow with only improper nesting in the form of (AND, AND)$^{[}_{]}$(AND, AND) or (OR, OR)$^{[}_{]}$(OR, OR) is strictly correct. ■

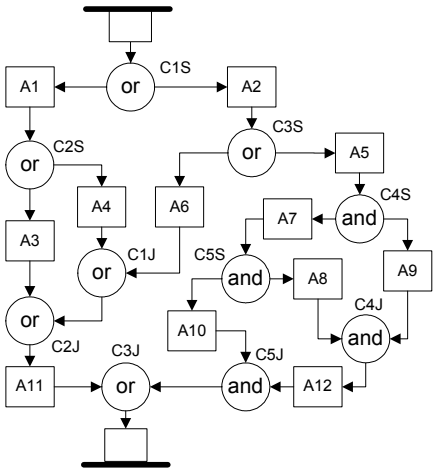


Figure 4-9: A Strictly Correct Workflow (but with Improper Nesting)

Also, by Theorem 1, for a potentially blocked node *pb*, *pb.in* can be determined as follows:

(1) If *pb* is the AND-Join node of an (OR, AND) pair, *pb.in* ="LR" (see Figure 4-8 (a));

(2) If *pb* is the AND-Join node in the AND pair of (AND $^{\text{OR-Split}}$ AND)$^{\lceil}_{\rfloor}$(OR, OR/AND), *pb.in* = {"L-" if the OR-Split node is in the *right* incoming path; otherwise "R-"} (see Figure 4-8 (b));

(3) If *pb* is the AND-Join node of (AND $^{\text{OR-Join}}$ AND)$^{\lceil}_{\rfloor}$(OR-Split, OR-Join), *pb.in* = { "L-" if the OR-Join node is in the *left* incoming path; otherwise "R-"}(see Figure 4-8 (c));

Figure 4-8 also shows the *pb.in* of each potentially blocked node. Note that in Figure 4-8, dotted lines are used to indicate the paths that may contain more activities or control nodes but those nodes are not shown here for simplicity. It should be clearly noted that the AND-Join nodes discussed in Figure 4-8 are only *potentially blocked*. We can show that (OR, AND) or (AND $^{\text{OR}}$ AND) $^{\lceil}_{\rfloor}$ (OR, OR/AND) may not always lead to

blocked nodes. An example is given in Figure 4-10. In this workflow, an AND pair is improperly nested with an OR pair in a $(C1S\ ^{C3S}\ C1J)^{\lceil}_{\rfloor}(C3S, C3J)$ pattern, but *C1J* is not blocked because of the mismatched (AND, OR) pair (*C2S*, *C2J*).

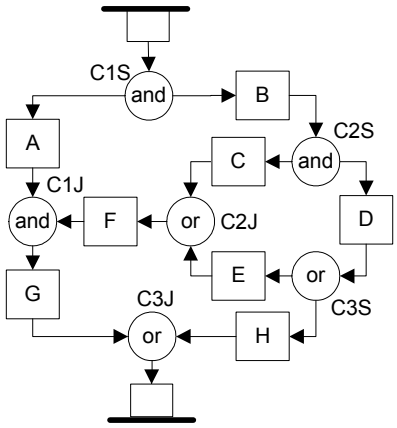


Figure 4-10: A Workflow with $(\text{AND}\ ^{\text{OR}}\ \text{AND})^{\lceil}_{\rfloor}(\text{OR, OR})$; No blocked nodes

Therefore, we need to test any potentially blocked node, say *pb*, to determine if it is actually blocked. Our test can lead to three possible results: (1) *pb* is actually not blocked as shown in Figure 4-10; (2) *pb* is indeed blocked, but the workflow is *not* deadlocked because some nodes downstream of *pb* can still be executed; and (3) *pb* is indeed blocked, and the workflow is deadlocked.

If *pb* is in fact not blocked, we should be able to find an AND-Split node upstream of *pb*, from which two paths join at *pb* without any OR-Split nodes in them. For example, in Figure 4-10, *C1J* is actually not blocked because there exists two paths {*C1S*, *A*, *C1J*} and {*C1S*, *B*, *C2S*, *C*, *C2J*, *F*, *C1J*} which contains no OR-Split nodes. However, if *pb* is indeed a blocked node, then to make a workflow terminate properly (i.e. reach the end node), a *bypass* must be found. A *bypass* is an alternative path that allows a workflow to complete despite one or more nodes being blocked.

Theorem 2: If a blocked node *b* is not a *deadlock causing node*, there exists a bypass from an AND-Split node upstream of *b* to an OR-Join node downstream of *b*.

Proof: We first show that there must be an OR-Join node downstream of *b* using an argument based on contradiction. Suppose there are no OR-Join nodes downstream of *b*. Then, the downstream nodes of *b* are activities, split nodes, AND-Join nodes, or the end node n_e. Let (b, n_1, ..., n_i, …, n_e) be any path after *b*. n_1 cannot be executed since one required input is from *b*, which is blocked. In general, n_i cannot be executed since its required input n_{i-1} is never executed, for $1< i \leq e$. Therefore, node *b* becomes a deadlock causing node, which contradicts the assumption of the theorem.

Next, we show that there must be an AND-Split upstream of *b*. Since *b* is blocked by definition, and has two incoming paths (*p1* and *p2*), out of which, say *p1*, is taken. Further, because *b* is not a deadlock causing node, there is at least one path, say *pass1*, going through an OR-Join node downstream of *b*. Both paths *p1* and *pass1* can be taken, but they are not in a sequence (path *pass1* cannot be taken after *p1*, since *b* is the dead end of *p1* and, also, *p1* cannot be taken after *pass1*, since *pass1* contains a downstream node of *b*). Therefore, paths *p1* and *pass1* must be reached from an AND-Split node, which is obviously upstream of *b*. Finally, *pass1* does not contain *b* and is, therefore, a bypass for *b*. ■

Based on Theorem 2, clearly, if *b* is a deadlock causing node, either there does not exist an AND-Split node upstream of *b*, or there is no bypass for *b*. Therefore, we can design an algorithm to test a potentially blocked node *pb* based on Theorem 2 as discussed next.

4.3.2 Testing Potentially Blocked Nodes

4.3.2.1 Algorithm – Checking a Potentially Blocked Node

We first develop an algorithm, say `CHK_BLK_NODE`, which can check one potentially blocked node, as shown in Figure 4-11. The algorithm for checking multiple potentially blocked nodes will be shown shortly.

The basic idea of algorithm `CHK_BLK_NODE` is that we try to find an AND-Split node upstream of *pb*. If we do not succeed, then we can conclude *pb* is a deadlock causing node; otherwise, we further test whether there are two paths starting from this AND-Split node joining at *pb* without passing any OR-Split nodes. If we can find two such paths, then *pb* is actually not blocked. Otherwise, we try to find a bypass starting from this AND-Split for node *pb*. If such a bypass exists, then *pb* is not a deadlock causing node; otherwise, it is. The steps of the algorithm are discussed in more detail below.

An AND-Join node *pb* has two predecessors, *pb.pred_left* and *pb.pred_right*. Assuming *pb* is blocked, the path from only one predecessor can be taken at one time into *pb*. Suppose this predecessor is called *pred1*, and the other *pred2*. The algorithm calls the Procedure `Remove_paths` alternately to remove the paths leading to the blocked node from each predecessor. In Step *RP1*, we start from *pred1* and trace backward paths to remove all paths that have been taken in order to make this incoming path execute, until we reach either the start node or an AND-Split in each backward path.

```
                    Algorithm CHK_BLK_NODE

Input: workflow wf, potentially blocked node pb
Output:result∈{1,2,3} /* 1:"pb is not blocked",2:"pb is blocked but
                  not deadlock causing", 3:"pb is deadlock causing"*/
if (pb.in[0]=="L") {pred1 = pb.pred_left; pred2 = pb.pred_right;}
else               {pred1 = pb.pred_right; pred2 = pb.pred_left;}
flag = 0; result = 0; wf' = wf   /*make a copy of the original workflow */
for (i=1,2) {    /* loop two times */
Remove_paths(wf, pb);
(RES1) if (flag == 1) {result = 1; return(result); exit();}/Report Result*/
       else {
         if(only start and end nodes are left and are disconnected)
            {result = 3; return(result); exit();}  /* deadlock causing node */
             else {result = 2;                     /* a bypass is found */
                   if(i == 2) {return(result); exit(); }    }}
(RES2) if (pb.in[1] ≠ '-') { flag = 0; wf = wf';
         if (pb.in[1] == "L")
              {pred1 = pb.pred_left; pred2 = pb.pred_right;}
         else {pred1 = pb.pred_right; pred2 = pb.pred_left;}}
       else { return(result); exit() ;}
}    /*outer for loop*/
Procedure Remove_paths(wf, pb) {
/* RP1: Remove the paths taken to reach node pb via   pred1 */
(RP1.1) Starting from pred1 (if pred1 exists) remove any activity node, join
node, or OR-Split node in the backward path.
(RP1.2) Stop when either the start node or an AND-Split node is reached. Call
the AND-Split node AS1.
(RP1.3) If the AND-Split node is "disconnected", i.e., its predecessor or both
of its successors have been removed, it should be removed.
/*RP2: Remove all paths that are disabled from reaching node pb*/
(RP2.1) Starting from pred2 (if pred2 exists), remove any activity node, join
node or AND-Split node (AS2 ≠ AS1) in the backward path. If AS2 = AS1, then
set flag = 1 and skip to REP
(RP2.2) Stop on reaching the start node or an OR-Split node.
(RP2.3) If the OR-Split node is "disconnected", i.e., its predecessor or both
of its successors have been removed, it should be removed.
/*RP3: Remove "disconnected" nodes*/
(RP3.1) Recursively remove any disconnected node (except start and end nodes),
i.e., one with its successors(s) or predecessor(s) completely missing.
(RP3.2) Remove any AND-Join node (except potentially blocked nodes) with only
one predecessor and one successor, since it needs two predecessors to make it
executable.
(RP3.3) Repeat (1) & (2) until no further removal is possible.
(RP3.4) Change any AND-Split node, OR-Split node, or OR-Join node that has
only one predecessor and one successor to a "dummy" activity node, say
Dummy_n.
} /*end of procedure Remove_paths*/
```

Figure 4-11: Algorithm — Checking a Potentially Blocked Node

Moreover, for *pb* if the incoming path from *pred1* is taken, it means the other incoming path from *pred2* is disabled. The disabled path is removed. In Step *RP2*, we

remove any nodes along the backward paths starting from *pred2*, until either the start node or an OR-Split node is reached in each of those paths. If an OR-Split node is reached, this node may lead a path away from *pb* and thus make *pb* blocked. Nevertheless, this path could become a bypass for *pb*, so we keep this node and stop removing more nodes in this path. However, if this backward path passes through an And-split node that was already encountered in step *RP1*, then we can conclude *pb* is not blocked (see Lemma 6 below).

Then, in step *RP3* we reorganize the truncated workflow by removing disconnected nodes and simplifying nodes with one missing branch. However, in Step *RP3.2* (see Figure 4-11), if an AND-Join node with only one predecessor was identified as a potentially blocked node, this node should be kept for further testing. Moreover, after *RP1-3*, if the workflow has multiple potentially blocked nodes, the truncated workflow may still contain some of them. We will continue testing these nodes in the same approach, as we will discuss shortly. Note that in the truncated workflow, a potentially blocked node may have only one predecessor because the other has been removed in the test. During the continuing test, we apply Steps *RP1* (or *RP2*) only when the corresponding predecessor exists (i.e., *pred1* exists for *RP1* and *pred2* exists for *RP2*, as described in Figure 4-11).

Note that, if *pb.in* = "LR", the outer loop is normally repeated twice – we test the workflow instance with the left and right branches of *pb* alternately. However, if *pb.in* = "L-" or "R-", then the loop is executed only once and the results are produced in step *RES1*. If, in fact, *pb* is not blocked (indicated by *flag* = 1), the algorithm shows the final result and stops. Here there is no need for the second iteration even if *pb.in* = "LR". If

after *RP3*, the start and the end nodes are disconnected, we can conclude *pb* is a deadlock causing node and the algorithm can stop without the need for a second iteration (even if *pb.in* = "LR"). Otherwise, Step *RES2* initiates a second iteration of the algorithm to check the case where the right branch of *pb* is taken and the left one is not. Note that *pb* is not deadlock causing only if both iterations return a result "blocked but not deadlock causing" (i.e., *result*=2). This situation will be illustrated by an example shortly.

4.3.2.2 Algorithm – Checking Multiple Potentially Blocked Nodes

So far we only considered workflows with one potentially blocked node. In general, workflows may have multiple potentially blocked nodes. In order to handle multiple such nodes, we can start with one node, and apply Algorithm `CHK_BLK_NODE` to simplify the workflow by removing various paths leading up to it. Now, in the simplified workflow (if not disconnected) we can again repeat the same procedure, if there are still blocked nodes remaining in it. In this way, the workflow is successively simplified until either a deadlock causing node is found, or the simplified workflow contains no potentially blocked nodes. If two nodes, say pb_1 and pb_2, have no path between them, we can choose to test either one first. We can show that the sequence of testing them will not affect the final result (see Lemma 12 in Appendix B). To make the test more efficient, we can arrange potentially blocked nodes such that upstream nodes are tested first. If an upstream node cannot be bypassed, obviously, there is no need to test downstream nodes. However, we do not include this optimization in our algorithm.

The algorithm `CHK_MULT_BLK_NODES` is an extension of Algorithm `CHK_BLK_NODE` and is shown in Figure 4-12. This is a recursive algorithm and it uses the same procedure `Remove_paths` used in `CHK_BLK_NODE`. After the first blocked node is diagnosed, and a deadlock is not found, it updates the list of the remaining potentially blocked nodes and calls itself again to check them. If at any stage a potentially blocked node leads to deadlock, then the algorithm can give a result value of '3' to indicate a deadlock and terminate. Otherwise, the algorithm loops through all the potentially blocked nodes, and returns a final result value of '1' or '2'. Note that the call to procedure `Remove_paths` is made by reference so it returns a truncated workflow after removing certain paths from it.

Based on our discussion, the complexity of this algorithm can be estimated as follows. Since in the worst case, each iteration needs to go through all nodes in the workflow and two iterations are needed for *pb.in*='LR', the complexity of this algorithm is $O(N^{n1} + (2N)^{n2})$, where N is number of nodes in the workflow, *n1* is the number of potentially blocked nodes with *pb.in*='L-' or *pb.in*='R-', and *n2* is the number of potentially blocked nodes with *pb.in*='LR'. However, in average, the computation is much less because (1) if a node is found to be deadlock causing or actually non-blocked, there is no need for the second iteration; and (2) in case of multiple potentially blocked nodes, after testing one node, some workflow nodes are removed and the workflow is simplified.

```
                          Algorithm CHK_MULT_BLK_NODES
Input: workflow wf,  /* wf has an array of potentially blocked nodes wf.PB[] */
       previous test result  /* the default value of result = 1 */
Output: result∈{1,2,3} /* 1:"no blocked nodes", 2:"blocked but not deadlock
                                causing nodes", 3:"deadlock causing nodes" */
pb = PB[0];
if (pb.in[0]=="L") {pred1 = pb.pred_left; pred2 = pb.pred_right;} else
                   {pred1 = pb.pred_right; pred2 = pb.pred_left;}
flag = 0;     wf' = wf;
for (i=1,2) {    /* loop two times */
Remove_paths(wf, pb)        /* see Procedure Remove_paths in Figure 11*/
/* Result */
(RES1) if (flag == 1) {
          wf = wf'; remove pb from PB[]; /*use original workflow, update PB[]*/
          if (wf.PB[] ≠ Ø) {result = CHK_MULT_BLK_NODES(wf, result);}
         return(result); exit();}
        else {
          if (only start and end nodes are left and are disconnected)
              {result = 3; return(result); exit();}
          else{ result = 2; remove pb from PB[];
              if (wf.PB[] ≠ Ø) {result = CHK_MULT_BLK_NODES(wf, result);}
              if (i == 2) {return(result); exit();}}
(RES2) if (pb.in[1] ≠ '-')   {
          if (pb.in[1] == "L") {pred1 = pb.pred_left; pred2 = pb.pred_right;}
          else                 {pred1 = pb.pred_right; pred2 = pb.pred_left;}
          flag = 0; wf = wf';
       else { return(result); exit(); }
      }    /*outer for loop*/
```

Figure 4-12: Algorithm – Checking Multiple Potentially Blocked Nodes

Lemma 5: Algorithm CHK _BLK_NODE is correct.

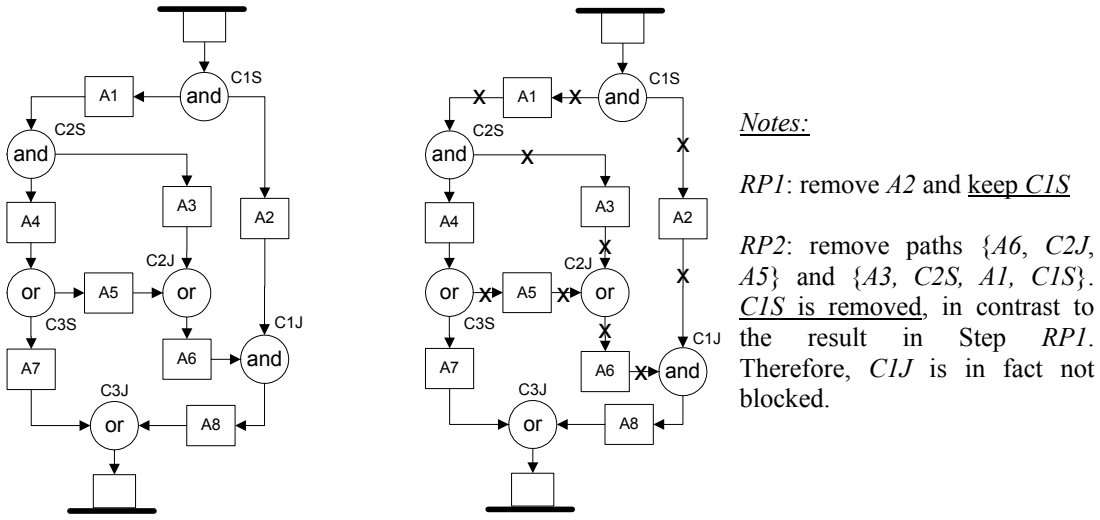
Proof sketch: The correctness argument follows from Theorem 2. Suppose a potentially blocked node, say *pb* is tested. Through Step *RP1*, we try to find an AND-Split node, say *AS1*, upstream of *pb*. There are three possible cases. (1) If *AS1* does not exist, *pb* cannot be bypassed. In this case, the start node is reached in Step *RP1* and after Steps *RP2-3*, only the start and the end nodes are left. Then in Step *RES1*, the correct conclusion is drawn. (2) If *AS1* is found, one possibility is that *AS1* node is removed in Step *RP2.1*. Step *RES1* shows *pb* is actually not blocked, according to Lemma 6 (see below). (3) If *AS1* is found and Step *RP2* finds an OR-Split node, say *OS1*, Step *RP3*

searches for a bypass. According to Theorem 2, such a bypass should be from *AS1* to an OR-Join node downstream of *pb*, say *OJ1*. If *OJ1* does not exist, Step *RP3* will reach the end node and return a disconnected workflow. Therefore, Step *RES1* gives the correct conclusion that *pb* is a deadlock causing node. If *OJ1* exists, a bypass is found and Step *RES1* concludes correctly. Also, if *pb.in*="LR", we repeat Steps *RP1-4* to test the other situation where the right incoming path of *pb* is taken but the left is not. ■

The proof for `CHK_MULT_BLK_NODES` is similar to the above proof for `CHK_BLK_NODE` algorithm, but is omitted for brevity. Next we illustrate these steps with examples.

4.3.2.3 Examples

Example 1: Figure 4-13(a) shows a workflow where *C1J* is a potentially blocked because of (*C1S* ^{C3S} *C1J*) $^{[}_{]}$ (*C3S* ^{C1J} *C3J*). After Step *RP1*, we remove *A2* and then reach an AND-Split node *C1S*. We stop removing nodes and *C1S* is kept. When we continue to execute Step *RP2*, we remove the path {*A6*, *C2J*, *A5*} and keep *C3S*. In another path, we remove {*A3, C2S, A1*, *C1S*}. Then we find that we need to remove the AND-Split node *C1S* that was kept after Step *RP1*, as shown in 4-13 (b). For this situation, we can prove that *C1J* is actually not blocked.



(a) *pb* = *C1J*, *pb.in* = "R-" (b) Applying Steps *RP1-2*

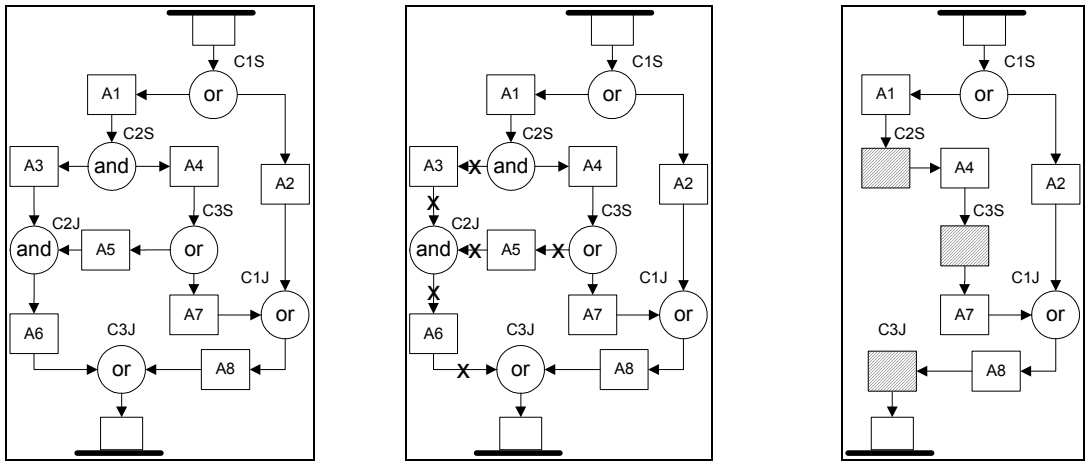
Figure 4-13: Example 1: Non-blocked Node *C1J*

Lemma 6: A *potentially* blocked node *pb* is actually not blocked if an AND-Split node reached in Step *RP1* (as a stopping point) is also reached in Step *RP2*.

Proof: Say, we reach an AND-Split node in Step *RP1* while tracing backwards from *pred1*. The algorithm does not trace the backward path any further. Let us call this node *AS1*. Then, in step *RP2*, the algorithm again traces the backward path from *pred2*. Let us say it reaches another AND-Split node, called *AS2*. Normally, the algorithm requires this node to be removed and the backward tracing to continue. However, if *AS1=AS2*, it means that there are two clear paths from *AS1* to node *pb*, both these paths can be taken. Hence, node *pb* is, in fact, not blocked. ■

As an intuitive explanation, in the workflow shown in 4-13(a), actually we can always find two concurrent paths {*C1S*, *A2*, *C1J*} and {*C1S*, *A1*, *C2S*, *A3*, *C2J*, *A6*, *C1J*} between *C1S* and *C1J* and these two paths contain no OR-Split nodes.

Example 2: Figure 4-14 (a) shows a workflow that is similar to the one in our motivating example of Figure 4-1, but slightly larger. Here *C2J* is potentially blocked. After Step *RP1*, we remove *A3* and stop at an AND-Split node *C2S*. Next, we need to test whether there is a bypass originating from this AND-Split node or the potentially blocked nodes are actually not blocked. This is done by Step *RP2*. During Step *RP2*, we remove *A5* and then reach an OR-Split node *C3S*. In Step *RP3*, we remove *C2J* and *A6*. The result is shown in 4-14 (b). After reorganizing the remaining nodes, we get a workflow as shown in 4-14 (c), where *C2S*, *C3S* and *C3J* are highlighted as dummy activities nodes. Clearly, this workflow is correct and is a bypass of node *C2J* since it can proceed to node *C3J*, a downstream node of *C2J*.



(a) *pb* = *C2J*, *pb.in* = "L-"

(b) Step *RP1*: remove *A3* and keep *C2S*
Step *RP2*: remove *A5* and keep *C3S*
Step *RP3*: remove *C2J* and *A6*

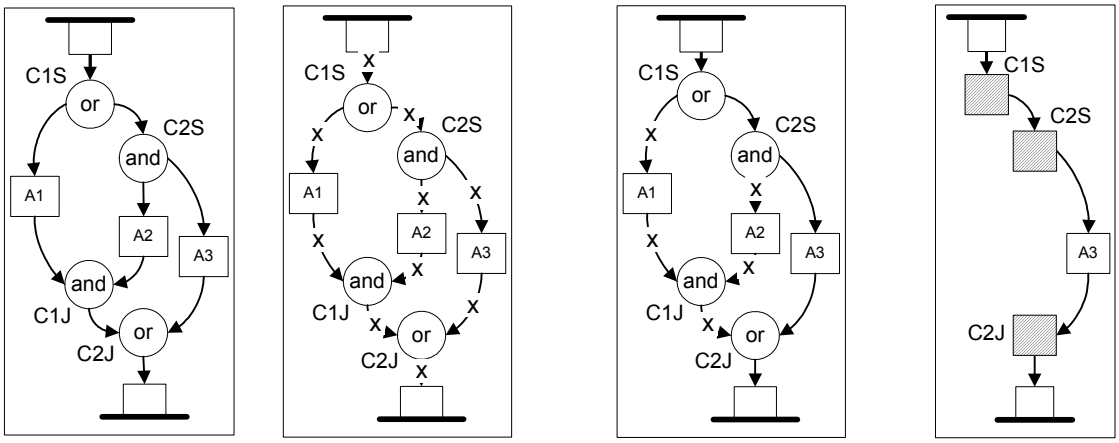
(c) Find a bypass. *C2J* is not a deadlock causing node

Figure 4-14: Example 2: A Deadlock-free Workflow with Blocked Node *C2J*

Example 3: In Figure 4-15 (a), *C1J* is potentially blocked and *C1J.in* = "LR". At one time, only one incoming path is taken and therefore *C1J* is potentially blocked. To

test whether this workflow is deadlock-free, we need to perform two tests by alternately setting *pred1* to *C1J.pred_left* and *C1J.pred_right*.

In Test 1 as shown in Figure 4-15 (b), only the incoming path from *A1* is taken. After Step *RP1*, we remove *A1* and *C1S*, and then reach the start node. In this case, the workflow is deadlocked. In Test 2 as shown in Figure 4-15 (c), the incoming path from *A2* is chosen. After Steps *RP1-3*, we remove nodes as shown in Figure 4-15 (c). Therefore, we find a bypass as shown in Figure 4-15 (d) and conclude that this workflow is not deadlocked in this case. Only if both tests show that *C1J* is not a deadlock causing node, we can conclude this workflow is deadlock-free; otherwise, the workflow can be deadlocked in some case.



(a) *pb* = *C1J*
pb.in = "LR"

(b) Test 1: *pb.in*[0] = "L"
Steps *RP1-3*: only the start and the end nodes are left.

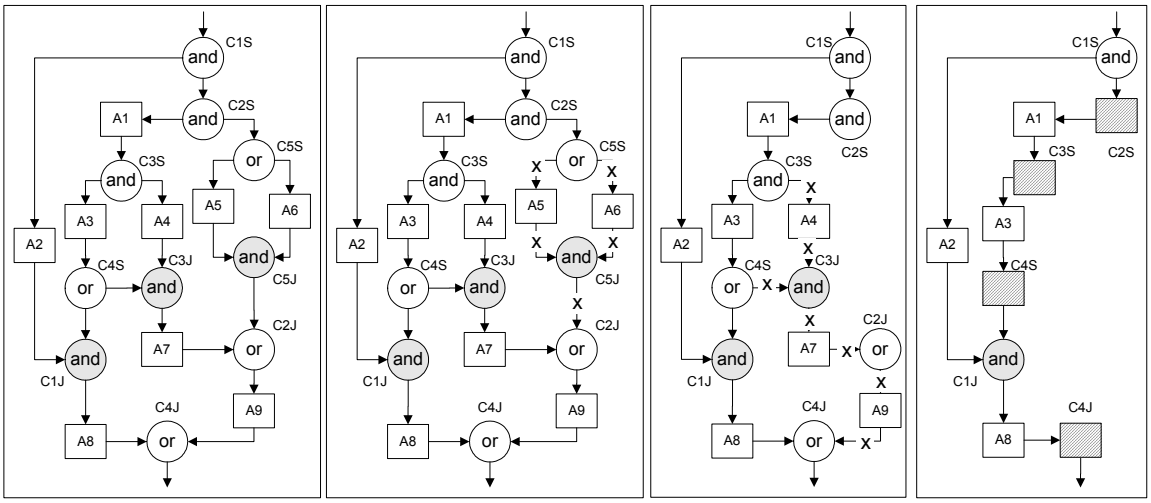
(c) Test 2: *pb.in*[1] = "R"
RP1: remove *A2* and reach *C2S*.
RP2: remove *A1* and reach *C1S*
RP3: remove *C1J*

(d) Find a bypass for Test 2

Figure 4-15: Example 3: Testing Blocked Node *C1J* and *C1J.in* = "LR"

Example 4: Figure 4-16(a) shows a workflow with three blocked nodes, *C1J*, *C3J*, and *C5J*. Now we need to determine whether this workflow is deadlocked. We can test any one of them first, say *C5J*. We remove nodes as shown in Figure 4-16(b). At this

stage, it is still hard to tell whether *C5J* can be bypassed since there are blocked nodes in the remaining workflow. So we determine whether *C3J* can be bypassed (again, we may also choose *C1J* instead of *C3J* here) by repeating Steps *RP1-4*. We remove paths as shown in Figure 4-16(c). After reorganizing the remaining workflow, we get a structured workflow as shown in 4-16(d). Obviously, *C4J*, the downstream node of *C5J* and *C3J* can be reached. Therefore, both *C5J* and *C3J* can be bypassed and this workflow is not deadlocked. Note that *C1J* and *C3J* cannot be blocked at the same time. Similarly, we can also show *C1J* can also be bypassed. In general, these three potentially blocked nodes can be tested in any sequence and we always get the same result.



(a) blocked nodes *C1J*, *C3J*, and *C5J* (b) Test *C5J*, *C5J.in* = "LR" (c) Test *C3J*, *C3J.in* = "R-" (d) Find a bypass

Figure 4-16: A Workflow with Multiple Potentially Blocked Nodes

4.3.3 Multiple Instances

Multiple instances (see Definition 8) are another structural flaw that was illustrated in Figure 4-2. Multiple instances usually arise because of mismatched (AND, OR) pairs, and do not cause deadlocks. In some special cases, an (AND, OR) pair may not lead to multiple instances – for example, if there is a blocked node in one path from the AND-Split node to the OR-Join node. An example of this situation is shown in Figure 4-17. In Figure 4-17, since both *C2J* and *C3J* are blocked nodes, one path from *C1S* to *C1J* is actually blocked. Therefore, there are no multiple instances at *C1J*. However, multiple instances alone do not result in deadlock and these structures are examples of weakly correct workflows (see Definitions 12-13). Our diagnosis algorithm checks for multiple instances and reports them.

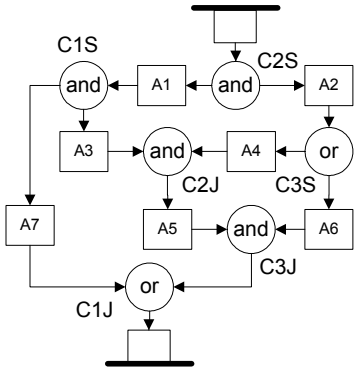


Figure 4-17: A Workflow with (AND, OR) Pair; No Multiple Instances

Figure 4-18 gives a simple algorithm which checks a mismatched (AND, OR) pair, say (s, j), and reports whether this pair leads to multiple instances. It should be note that if a blocked node, say b, is in one branch from s to j and the other branch becomes the bypass of b, then there are no multiple instances.

```
                          Algorithm MULTI_INSTANCES
Input: a mismatched (AND,OR) pair (s, j), blocked nodes B[]
Output: result ∈{0,1} /* 0: multiple instances
                         1: no multiple instances in some workflow instances */
result = 0;
for (each b in B[])
{ if (b is in a path from s to j)
  {  if (j == first downstream node of b in the bypass of b )
        {result = 1;}     /* the bypass of b is the other branch from s to j,
                             therefore, j is only activated once */
  }
}
return(result);
```

Figure 4-18: Algorithm for Checking Multiple Instances

4.4 Finding Equivalent Structured Mappings

So far, we have discussed various structural flaws and the verification of workflow correctness. However, in practice, most workflow tools support structured workflows despite the fact that unstructured workflows could also be correct. Aalst et al. [5] have compared 15 main workflow management systems in terms of a set of selected workflow patterns, and showed none of these systems supports all these patterns, but all of them can support structured workflows. Since most workflow products impose different structural constraints, while structured workflows are widely supported, the mapping may bridge the gap between process modeling and workflow implementation. Moreover, with the mapping technique, during initial process modeling, structural constraints can be released and attention can be focused on precisely capturing business requirements. Later on, workflow verification and implementation issues are considered with the mapping techniques. This is certainly a preferable approach to designing workflow systems. Also, by showing that some correct unstructured workflows have no

equivalent structured mappings, we suggest that workflow tools should give specific considerations to those workflows in order to support a larger variety of process scenarios.

In this section, we will discuss the mappings from unstructured workflows to structured ones. [45] discusses such transformations mainly through examples. In this essay, we will precisely describe the scenarios where structured mappings exist and design an approach to developing equivalent structured mappings. We start with equivalence preserving mappings.

4.4.1 Equivalence Preserving Mapping

The equivalence between two workflows can be checked by bisimulation game [45]. Based on the concept of *bisimulation games*, workflow *A* can simulate workflow *B* if *A* can imitate any movement (e.g., starting or finishing the workflow, or completing an activity,) of *B*. If *A* can simulate *B,* and *B* can also simulate *A*, then we say *A* and *B* are equivalent, or *A* is the *equivalent mapping* of *B*. For example, in Figure 4-19, workflow *wf1* is unstructured because of (*C1S*, *C1J*)$^{\lfloor}_{\rfloor}$(*C2S*, *C2J*). Workflow *wf2* is an equivalent structured mapping of *wf1*.

However, not every workflow has equivalent mapping. In some case, workflow *A* is only quasi-equivalent to workflow *B*.

Definition 14 (Quasi-equivalent or q-equivalent mapping): A mapping from workflow *A* to workflow *B* is quasi-equivalent if *A* can simulate *B*, but *B* cannot simulate *A*. ■

As an example, consider the two workflow patterns in Figure 4-20. Here, based on bisimulation games, Workflow *wf1* can simulate Workflow *wf2*, but *wf2* cannot simulate *wf1*. In *wf1*, the possible completed execution paths (or interleavings) are *A1A2B*, *A2A1B*, *A1BA2*, and *A2BA1*, but only *A1A2B* and *A2A1B* are the possible paths of *wf2*. In other words, *wf1* is more expressive than *wf2*. Thus, this mapping is not *completely equivalence preserving*. However, such a *quasi-equivalent mapping* can help in the verification of complicated workflows involving OR-Join elements as we will see later.

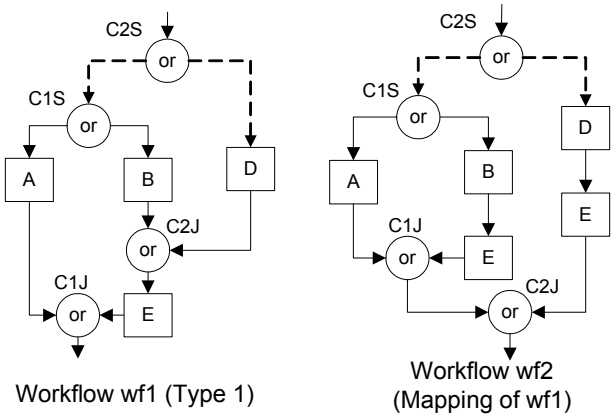


Figure 4-19: A workflow with its Equivalent Structured Mapping

Figure 4-20: An Unstructured Workflow with Q-equivalent Mapping

4.4.2 Possibility of Equivalence Preserving Mapping

Intuitively, if an unstructured workflow has an equivalent structured mapping, it must be strictly correct, because the structured mapping can never simulate any structural flaws, and to preserve equivalence, the original workflow need not produce any structural flaws. Therefore, we state Lemma 7 without proof as follows.

Lemma 7: Only strictly correct (unstructured) workflows can have equivalent structured mappings. ■

Therefore, according to our analytical results, we can enumerate all scenarios of unstructured workflows and discuss their possibilities of structured mappings as follows:

(1) An unstructured workflow with only improper nesting in the form of (AND, AND)$^{\lceil}{}_{\rceil}$(AND, AND) or (OR, OR)$^{\lceil}{}_{\rceil}$(OR, OR) may have structured mappings (see Lemma 4 and Lemma 7). This case is referred to as AND-AND (or OR-OR) improper nesting. We will discuss those workflows in Section 4.4.3.

(2) Unstructured workflows with improper nesting in the form of (AND, AND)$^{\lceil}{}_{\rceil}$(OR, OR) lead to potentially blocked nodes (see Theorem 1 and the example shown in Figure 4-1). This case is referred to as AND-OR Improper nesting. In general, these workflows may not have structured mappings, but there are some exceptions, as we will discuss in Section 4.4.4.

(3) Unstructured workflows with mismatched pairs cannot have equivalent structured mappings, since every structured mapping contains no mismatched pairs and any mapping from one node type to another cannot be completely equivalence preserving. Q-equivalent mappings are an example in this case. We will discuss Q-

equivalent mappings for workflows with mismatched (AND, OR) pairs in Section 4.4.5.

4.4.3 OR-OR/AND-AND Improper Nesting

Unstructured workflows with OR-OR improper nesting (i.e., in the form of (OR, OR) $^{\lceil}_{\rfloor}$ (OR, OR)), such as the one in Figure 4-19, always have equivalent structured mappings. Next, we will develop a simple algorithm as shown in Figure 4-21 for generating those mappings. First, we define adjacent join nodes.

```
                                Algorithm EQUIV_OR
Input: workflow wf, first-order improper nesting (C1S, C1J)⌈⌋(C2S C1J C2J)
Output: wf /* improper nesting is replaced by its equivalent structured
              mapping*/

if (C1J is adjacent to C2J)
{ find pred1∈C1J.Pred[] s.t. a path exists from C2S to pred1 or pred1==C2S;
  find pred2 s.t. pred2∈C1J.Pred[] and pred2 ≠ pred1;
  find pred3∈C2J.Pred[]) s.t. a path exists from C1J to pred3 OR pred3==C1J;
  succ1 = C1J.succ[0];
  succ2 = C2J.succ[0];  /*Note: A Join node has only one successor*/

  disconnect C1J from pred1;
  disconnect C2J from pred3;
  disconnect succ2 from C2J;

  if (pred3 ≠ C1J)            /* Now copy the path from C2J to C1J*/
   {p1 = path from succ1 to pred3;
    p1' = p1;
    n1 = First_node(p1');
    n2 = Last_node(p1');
    connect pred1 to n1;
    connect n2 to C2J;
    disconnect succ1 from C1J;
    disconnect C1J from pred2;
    connect succ1 to pred2; connect both pred3 and C2J to C1J;}
  else {connect pred1 to C2J; connect C2J to C1J; }

  C1J.succ[0] = succ2; }               /* switch C1J with C2J */
```

Figure 4-21: Algorithm for Mapping Workflows with OR-OR Improper Nesting

Definition 15 (Adjacent join nodes): Two join nodes are adjacent, if there are only activities, but no other control nodes between them. ■

For example, In Figure 4-22(a), there exists first-order improper nesting ($C1S^{C2S}\ C1J$) $^{[}_{]}$ ($C2S^{C1J}\ C2J$) and *C1J* is adjacent to *C2J*. An intuitive observation is that, if we push *C1J* down below *C2J*, we can correct such improper nesting. In general, in a workflow with improper nesting, two adjacent join nodes always exist. Next, we use an example to illustrate the algorithm for correcting such improper nesting.

Example: 4-22(a) shows a workflow with 3 split and 3 join control elements. The improper nesting relationships are ($C1S^{C2S}\ C1J$) $^{[}_{]}$ ($C2S^{C1J}\ C2J$), ($C1S^{C3S}\ C1J$) $^{[}_{]}$ ($C3S^{C1J}\ C3J$), and ($C3S^{C2J}\ C3J$) $^{[}_{]}$ ($C2S\ C2J$).

Only ($C1S^{C2S}\ C1J$) $^{[}_{]}$ ($C2S^{C1J}\ C2J$) is a first-order improper nesting. We can remove this improper nesting using the algorithm above, and obtain workflow *wf2* shown in Figure 4-22(b). In *wf2*, only one improper nesting ($C1S^{C3S}\ C1J$) $^{[}_{]}$ ($C3S^{C1J}\ C3J$) remains. We continue this procedure and get the final transformation shown as workflow *wf3* in Figure 4-22(c).

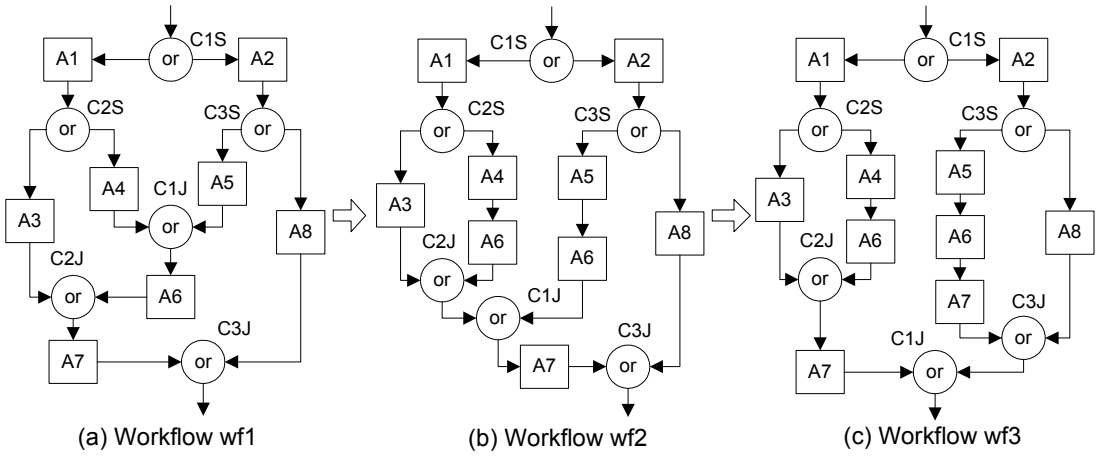


Figure 4-22: Steps in Mapping Unstructured *wf1* into Structured *wf3*

Next, we discuss the structured mappings of AND-AND improper nesting.

Theorem 3: In a workflow that contains (AND, AND) $^{[}_{]}$ (AND, AND) nesting, if there is at least one activity between these two AND-Split nodes, and at least one activity between these two AND-Join nodes, this workflow does not have an equivalent structured mapping.

Proof: The proof is based on construction and uses Figure 4-23. Two scenarios are constructed for creating a structured mapping by: removing activity *D* and pushing *v* down (Figure 4-23(b)), or removing activity *B* and pushing *s* up (Figure 4-23 (c)). Then, we argue that in either case the removed activity cannot be reinserted without disturbing the order of activities in Figure 4-23(a). Therefore, the structured mapping is not possible. ■

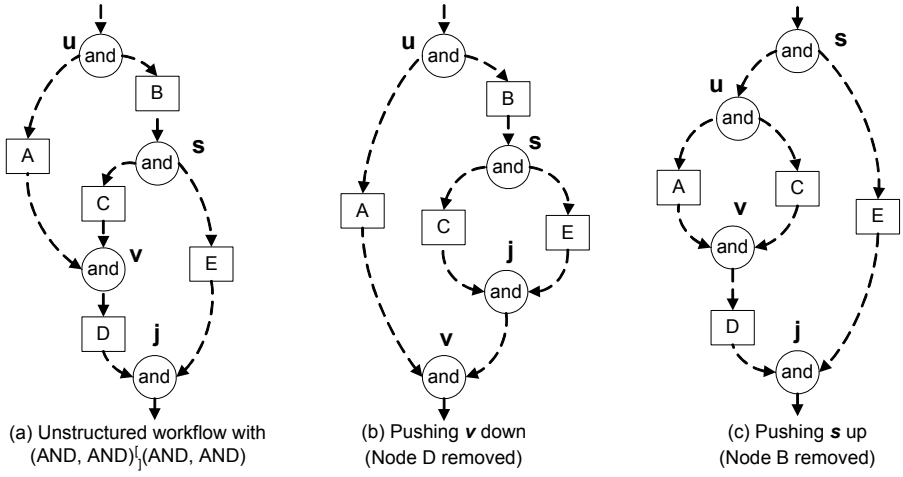


Figure 4-23: AND-AND Improper Nesting; No Structured Mappings

However, Theorem 3 does not cover situations in Figure 4-24(a) where *B* (or *D*) does not exist. In such cases, structured mappings are possible. This is because we can

"merge" two AND-Split nodes, *C1S* and *C2S*, and then split again with re-distributed outgoing arcs, as shown in Figure 4-24 (b).

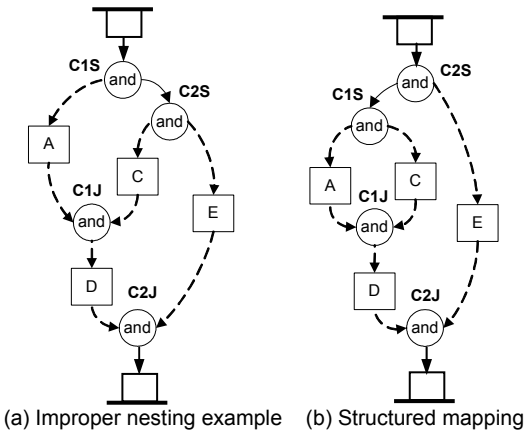


Figure 4-24: Equivalent Structured Mapping of a Special AND-AND Improper Nesting

Figure 4-25 gives an algorithm which develops structured mapping for such workflows.

Algorithm EQUIV_AND

Input: workflow *wf*, improper nesting $(C1S\ ^{C2S}\ C1J)^{[}_{]}(C2S, C2J)$ **or** $(C1S, C1J)^{[}_{]}(C2S\ ^{C2J}\ C2J)$

Output: *wf*

```
/* if a structured mapping is found, the improper nesting is replaced by it */

if((C1S C2S C1J)[](C2S,C2J) and C2S∈C1S.Succ[])     /*C1S is directly connected
                                                       to C2S*/
{find succ1∈C2S.Succ[] s.t. ∃ a path from succ1 to C1J or succ1 == C1J;
 pred1 = C1S.pred[0];  /* assign the only predecessor of C1S to pred1 */
 disconnect C2S from C1S; disconnect succ1 from C2S;
 disconnect C1S from pred1; connect C1S to succ1;
 connect C2S to C1S; connect pred1 to C2S} /*switch C1S with C2S*/

if ((C1S,C1J)[](C2S C1J C2J) and C1J∈C2J.Pred[]) /*C1J is directly connected to
                                                   C2J*/
{find pred1∈C1J.Pred[] s.t. ∃ a path from C2S to pred1 or pred1 == C2S;
 succ1 = C2J.succ[0];  /* assign the only successor of C2J to succ1 */
 disconnect C1J from C2J; disconnect pred1 from C1J;
 disconnect C2J from succ1; connect pred1 to C2J;
 connect C2J to C1J; connect C1J to succ1;} /*switch C2J with C1J*/
```

Figure 4-25: Algorithm for Mapping Workflows with AND-AND Improper Nesting

4.4.4 AND-OR Improper Nesting and Overlapping Structures

In general, unstructured workflows of this type lead to blocked nodes (see Theorem 1) and therefore they do not have any structured mappings. An overlapping structure as shown in Figure 4-26(a) is an exception. An overlapping structure is strictly correct and it has an equivalent structured mapping as shown in Figure 4-26(b).

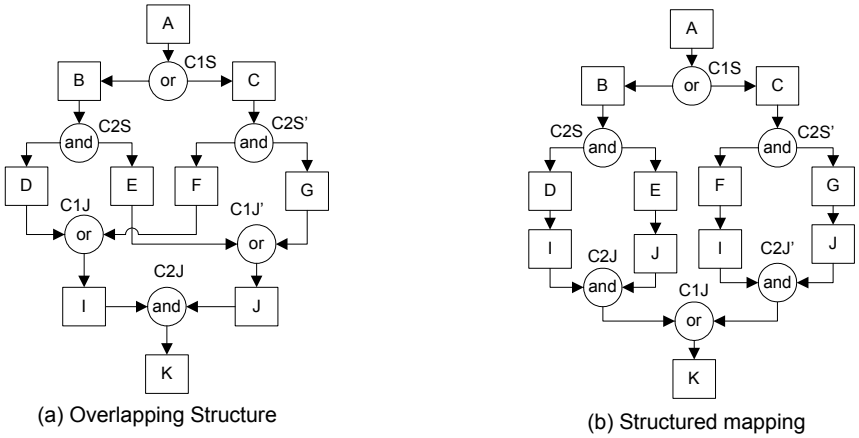


Figure 4-26: An overlapping Structure and Its Mapping

Definition 16 (Overlapping structures): A workflow is called an overlapping structure if it can satisfy the following three conditions:

(1) An OR-Split has two corresponding OR-Join nodes, say (*C1S*, *C1J*) and (*C1S*, *C1J'*), and a AND-Join node corresponds two AND-Split nodes, say (*C2S*, *C2J*) and (*C2S'*, *C2J*); and

(2) Each pair is nested with the other three pairs, i.e., there is only third-order improper nesting among these six nodes: (*C1S*, *C1J*) $\lceil\rceil$ { (*C1S*, *C1J'*), (*C2S*, *C2J*), (*C2S'*, *C2J*) }, (*C1S*, *C1J'*) $\lceil\rceil$ { (*C1S*, *C1J*), (*C2S*, *C2J*), (*C2S'*, *C2J*) }, (*C2S*, *C2J*) $\lceil\rceil$ { (*C1S*, *C1J*), (*C1S*, *C1J'*), (*C2S'*, *C2J*) }, (*C2S'*, *C2J*) $\lceil\rceil$ { (*C1S*, *C1J*), (*C1S*, *C1J'*), (*C2S*, *C2J*) }.

(3) *C1J* and *C1J'* are adjacent to *C2J*. ■

4.4.5 Q-equivalent Mappings for Mismatched (AND, OR) Pairs

Unstructured workflows with mismatched (AND, OR) pairs may have quasi-equivalent mappings, as shown in Figure 4-20. However, in some situations q-equivalent mappings do not exist. Next, we state those situations in Lemma 8.

Lemma 8: An (AND, OR) cannot be (quasi) mapped to an (AND, AND) pair if this pair is a part of improper nesting (AND ^{OR} OR)$^{\lceil}_{\rfloor}$(OR, OR) and in the mapping the AND-Join node becomes a blocked node.

Proof: This statement can be proved using the argument of contradiction. Suppose the (AND, OR) pair can be mapped (AND, AND). Therefore, the corresponding improper nesting becomes (AND ^{OR} AND)$^{\lceil}_{\rfloor}$(OR, OR), which leads to potentially blocked nodes, by Theorem 1. Further, the AND-Join node is proved to be truly blocked in the mapping. However, (AND, OR)$^{\lceil}_{\rfloor}$(OR, OR) does not produce any blocked nodes. Therefore, the original workflow cannot simulate the mapping. Obviously, by Definition 14, the mapping is not quasi-equivalent. ■

4.5 Introducing Loops

So far we only considered acyclic workflows, i.e. in these workflows there were no paths that created cycles. Next, we turn to consider workflows where loops may be present. Loops consist of a cycle having one entrance at an OR-Join element and one exit

from an OR-Split element. These two elements do not have corresponding elements to them in the sense of Definition 4. However, they are said to "correspond" to each other (for loops), and every loop will have such a pair of choice elements. In Figure 4-27, *C1S* and *C1J* are such distinguished nodes. In a structured loop (see Figure 4-4), there are no other exits from or entrances into the loop path. However, in a general loop additional entrances and exits may exist. We call these as situations of improper nesting into the loop. In this section, we are primarily interested in loops with at least one OR-Join that serves as an entrance, and one OR-Split that serves as an exit from the loop. Then, we consider scenarios involving additional entrances and exits.

Definition 17 (**Entrances** and **Exits** of a loop): a join node *j* in a loop is an entrance if one of *j*'s predecessors is not in the loop. A split node *s* in a loop is an exit if one of its successors is not in the loop. A loop must have an OR-Join node as a primary entrance and an OR-Split node as a primary exit. It may have join nodes as additional entrances or split nodes as additional exits. ■

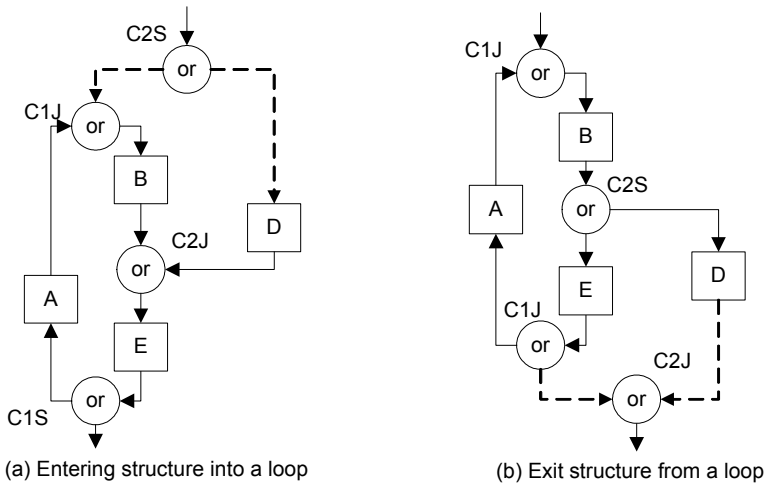


Figure 4-27: Structures Entering and Leaving loops

4.5.1 Scenarios and Taxonomy

Figure 4-27 shows the corresponding scenarios of interest. In both these figures there is a correspondence between *C2S* and *C2J* nodes. In Figure 4-27 (a), *C2J* lies on the loop and *C2S* is outside the loop, while in Figure 4-27 (b), it is the other way around. There are four combinations of values for the *C2S* and *C2J* pairs, and these are considered in Table 4-2 and Table 4-3, which correspond to Figure 4-27 (a) and Figure 4-27(b) respectively. The tables show that in both scenarios, 2 out of 4 cases behave similarly and are acceptable. When the split-join combination is OR-OR, the workflow is strictly correct and also has a corresponding structured representation. The AND-OR combination leads to multiple instances, and in an entering structure, it has a q-equivalent mapping. A third combination AND-AND, causes a deadlock for an entering structure, but works well in the exit structure. The semantics in this case is as follows: if there are multiple passes through the loop, then activity *D* will get invoked repeatedly. However, when the loop is exited, then the AND control element *C2J* will be activated. From a semantic perspective, the results from the most recent execution of *D* should be regarded, while the earlier ones can be ignored. The last structure in the table is OR-AND that leads to deadlock. Clearly the behavior in the case of an entrance versus an exit from the loop is not symmetric. Next, we see how equivalent mappings of structures with loops can be created.

Table 4-2: Behavior of Structures Entering a Loop

Type	(C2S	C2J)	Correctness issues	Structured Transformation
1N	AND	OR	multiple instances	q-equivalent mapping
2N	OR	AND	deadlock	No
3N	OR	OR	strictly correct	Yes
4N	AND	AND	deadlock	No

Table 4-3: Behavior of Structures Exiting a Loop

Type	(C2S	C2J)	Correctness issues	Structured Transformation
1X	AND	OR	multiple instances	No
2X	OR	AND	deadlock	No
3X	OR	OR	strictly correct	Yes
4X	AND	AND	strictly correct	No

4.5.2 Mappings

Next, to appreciate how a q-equivalent structured mapping can be created for a Type 1N workflow from Table 4-2, consider Figure 4-28(a). In the mapping shown here, the whole loop $\{E, A, B, E\}$ is duplicated and *C2J*, an OR-Join node, is mapped to AND-Join. In this mapping, multiple instances can arise of activities E, A and B, which lie inside the loop.

Similarly, a type 3N structure from Table 4-2 has an equivalent structured mapping, as shown in Figure 4-29. A Type 3X workflow, as shown in Figure 4-30, is well behaved, but we can show that it has no structured mapping without using auxiliary variables.

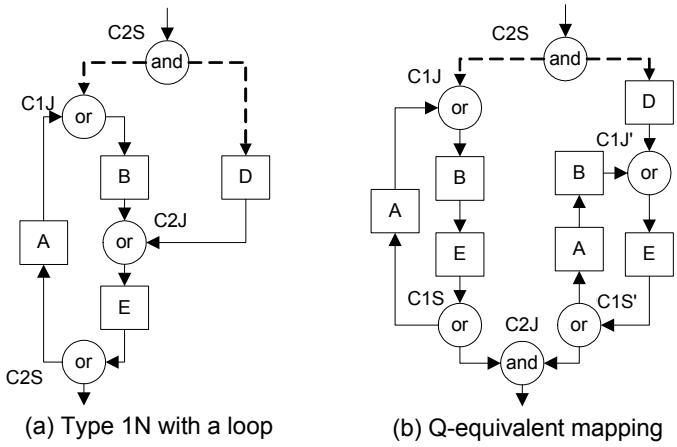


Figure 4-28: Q-Equivalent Mapping of Type 1N with a Loop

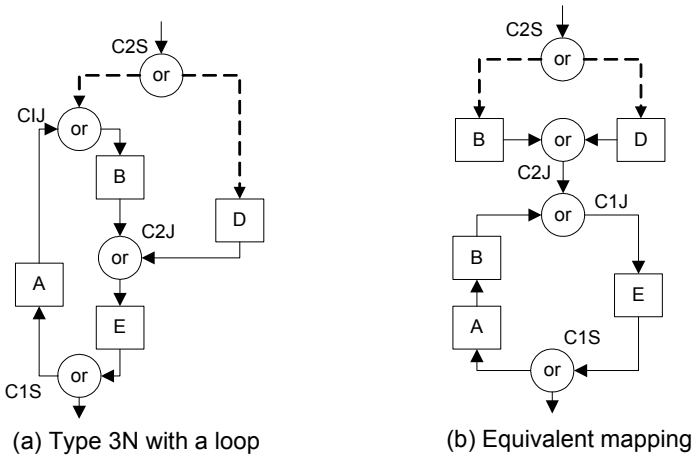


Figure 4-29: Equivalent Structured Mapping of Type 3N with a Loop

4.5.3 Results and Algorithm

Next, we discuss two main results related to loops.

Lemma 9: A workflow pattern of type 3X with a loop cannot be mapped to structured workflows without using auxiliary variables.

Proof sketch (By contradiction): The proof is based on arguing that this loop has two exits nodes, and a different activity follows after each of these exits nodes. A structured mapping of the loop will have only one exit node, and an auxiliary variable would be required to determine which of the two exits was taken in order to make sure that the correct activity follows the exit. ■

Figure 4-30 gives an example of how a mapping for this situation can be produced with auxiliary variables. It is also observed in [45] that certain forms of unstructured workflows cannot be transformed without the use of auxiliary variables.

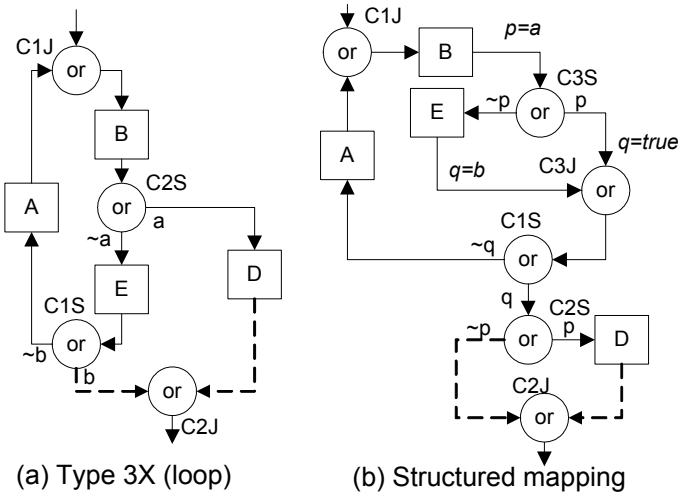


Figure 4-30: Structured Mapping of Type 3X (Loop) (With Auxiliary Variables)

Lemma 10: A Type 1X (or 4X) workflow cannot have a q-equivalent structured (or equivalent) mapping.

Proof sketch (by contradiction): This result is proved by arguing that for a workflow of Type 1X (see Figure 4-31), any structured mapping must contain a structured loop and a parallel structure. The parallel structure would contain *E* and *D* in parallel. If such a structure were inside the loop, then both *E* and *D* would be part of the

loop (but *D* is not); while, if it were outside the loop, then both would be outside (but *E* is in the loop)! ■

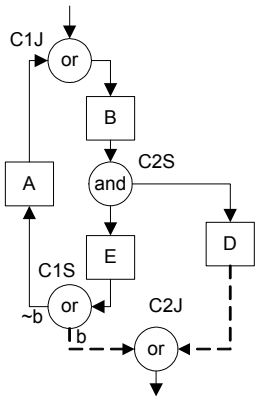


Figure 4-31: Type 1X (Loop)

Finally, we describe an algorithm for analyzing loops, as shown in Figure 4-32. This algorithm first detects loops. A loop is a cycle which goes through an OR-Join node and an OR-Split node, which, however, are not corresponding to each other by Definition 4. Next, we find the corresponding node pairs which enter or exit a loop. If the loop only has one such pair (i.e., (*C2S*, *C2J*) in Table 4-2 and Table 4-3), we look up for the type of the loop using Table 4-2 and Table 4-3. Then, depending upon the type of loop it takes one of three appropriate actions: applying equivalent transformation, reporting multiple instances, or reporting deadlocks. Note that this algorithm is only able to handle cases of loops with only one additional entrance or only one additional exit constructed by a corresponding pair. If a loop enters or exits another loop or it has more than one additional entrance and/or more than one additional exit, our algorithm simply reports it. Simplification techniques, such as cycle reducibility analysis [47], may be needed to analyze such complicated unstructured loops.

```
                                  Algorithm LOOPS

Input: workflow wf
Output: modified wf          /* loops are analyzed and results are displayed*/

/* step 1: Detect loops*/
L0 = Ø; i = 0;
for (each OR-Join node j)
 if (∃ path {j,…,s,…j} s.t. s is an OR-Split node and
                            s is not corresponding to j)
   if (j ∉ any Lk, k ≤ i) {let Li(j,s) be the loop; i++; Li = Ø; }

/*step 2: Analyze loops */
for (each loop L(j,s))
{ entrances = 0; exits = 0 ;     /* number of additional entrances and exits */
  N_pairs[] = Ø; X_pairs[] = Ø; /*array of additional entrance or exit pairs*/
  L.Type = Ø ;                  /*Type of loop L(j,s) will be determined later */

  for (each join node j1 in L, j1≠j)
  { if(∃ n∈j1.pred[] and n∉L)
    { entrances ++;          /*j1 is an entrance if one of its predecessors∉L */
      if (∃ s1 s.t. (s1, j1)) {add (s1, j1) to N_pairs[];} }
  for (each split node s2 in L, s2≠s)
  { if (∃ n∈s2.succ[] and n∉L)
    { exits ++;              /* s2 is an exit if one of its successors∉L */
      if (∃ j2 s.t. (s2, j2)) {add (s2, j2) to X_pairs[]; } }

  if (entrances == 0 and exits == 0) { report L(j,s) is a structured loop; }
  elseif (entrances+exits == 1)
                        /* only 1 additional entrance or 1 additional exit*/
  { determine L.Type using node pairs in N_pairs[] and X_pairs[];
                                    /* From Table 4-2 and Table 4-3 */
    if (L.Type ≠ Ø ) { switch (L.Type)
       { case('1N','3N','3X'): apply equivalent transformation;
         case('1N', '1X'): report multiple instances;
         case('2N','2X','4N'): report deadlocks;} }
    else { report another loop entering or exiting L(j,s) ;} }
  else { report L(j,s) has more than one entrance and more than one exit; }}
```

Figure 4-32: Algorithm for Analyzing Unstructured Loops

4.6 Workflow Diagnosis Algorithm and Results

4.6.1 Algorithm Outline

In this section, we introduce a workflow diagnosis algorithm based on the analysis in the previous section. Figure 4-33 shows the outline of this algorithm.

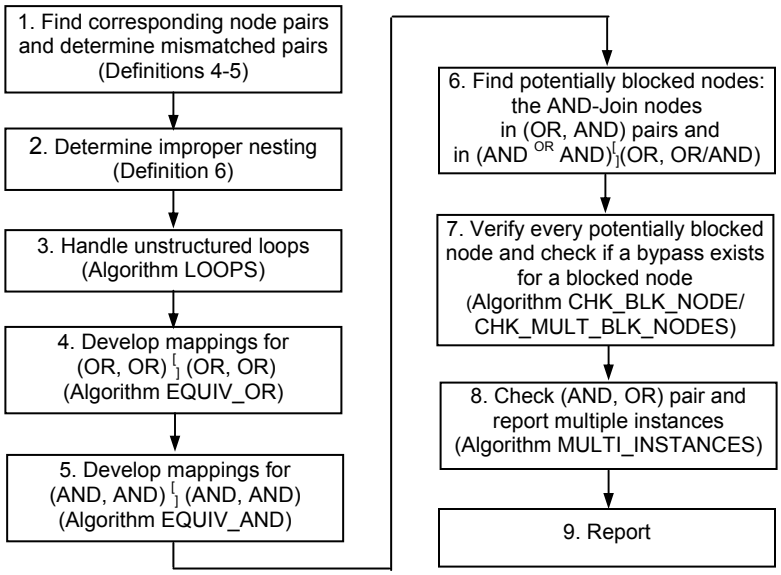


Figure 4-33: The Outline of the Workflow Diagnosis Algorithm

The input of this algorithm is a text file of a workflow graph. A sample input file will be provided later. After the algorithm reads an input file, it executes nine steps in order to provide a detailed analysis report. These nine steps are explained next. The first step is to calculate corresponding node pairs and determine mismatched pairs. The next step is to determine improper nesting based on corresponding pairs. In Step 3, unstructured loops are handled as using Algorithm `LOOPS` (see Figure 4-32). Then, Step 4 deals with improper nestings in the form of (OR, OR)[ ](OR, OR) using Algorithm `EQUIV_OR` (see Figure 4-21). Equivalent structured mappings for such situations are developed. After mapping, improper nesting is reduced. Moreover, this step also finds structured mappings for overlapping structures (by Definition 16) and replaces the original structures using the mappings in order to further reduce improper nesting. Step 5 handles improper nesting of type (AND, AND)[ ](AND, AND) using Algorithm `EQUIV_AND` (see Figure 4-25). Step 6 detects potentially blocked nodes in improper

nestings of the type (AND ^{OR} AND) $^{\lceil}_{\rfloor}$ (OR, OR) or in mismatched (OR, AND) pairs. Step 7 verifies whether a potentially blocked node is indeed blocked, and if it is, whether it has a bypass. The detailed algorithms for this were shown in Figure 4-11 and Figure 4-12. Then, step 8 determines whether an (AND, OR) pair leads to multiple instances using Algorithm `MULTI_INSTANCE` (see Figure 4-18). The last step provides a detailed diagnosis report that records results of each step and draws conclusions on the workflow correctness.

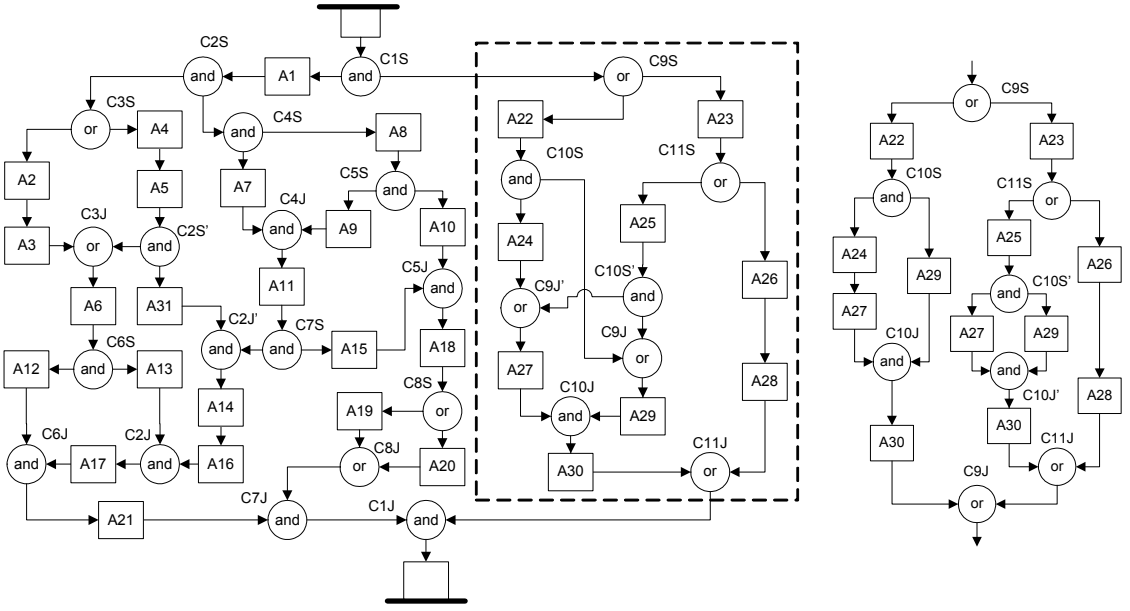
Lemma 11: The Workflow Diagnosis Algorithm is correct.

Proof: This algorithm produces correct results since each step (except step 9, which reports results) is based on Definitions or proven Theorems. Steps 1-2 calculates corresponding node pairs, mismatched pairs and improper nestings per their definitions (see Definitions 4-6). Step 3 handles loops based on our results in Section 4.5, which are theoretically developed or proven (see Lemmas 9-10). Also, Lemma 4 and Lemma 7 ensure the correctness of step 4. Step 5 uses Theorem 3 and Lemma 4. Step 6 directly comes from Theorem 1 and Algorithms `CHK_BLK_NODE` and `CHK_MULT_BLK_NODES` used in step 7 are developed based on Theorem 2 and Lemma 5, as we discussed in Section 4.3.2. Finally, step 8 directly follows the definition of multiple instances (see Definition 8). ■

This algorithm is implemented in C. The detailed algorithm is available on our web site: http://zen.smeal.psu.edu/~emily/bypass_algorithm.htm.

4.6.2 Experimental Results

Figure 4-34(a) shows a workflow that is presented in [82], Sadiq and Orlowska developed an algorithm to verify the correctness of this workflow, but the detailed causes for structural flaws are not provided. Using our algorithm, we can provide a detailed analysis report for it.



(a) A complex workflow

(b) Structured mapping of the dotted part in (a)

Figure 4-34: A Diagnosis Experiment

This is clearly a non-trivial complex workflow with 59 nodes and 14 corresponding pairs. This algorithm can diagnose such a workflow within a few seconds. A part of the input file is shown in Figure 4-35. For example, the second line describes the start node in this workflow. It only has one successor, *C1J* in the column "succ_left", but without any predecessors in the columns "pred_left" and "pred_right". The third line

describes node *C1S*, which has the start node as the only predecessor and two successors, *A1* and *C9S*. A summary of the analysis results as shown in Figure 4-36.

Node	pred_left	pred_right	succ_left	succ_right	Node Type
Start	-	-	C1S	-	Start
C1S	Start	-	A1	C9S	AND-Split
A1	C1S	-	C2S	-	Activity
C9S	C1S	-	A22	A23	OR-Split
			……		
C3J	A3	C2S'	A6	-	OR-Join
			……		
C4J	A7	A9	A11	-	AND-Join
			……		
End	C1J	-	-	-	End

Figure 4-35: A Snippet of a Sample Algorithm Input File

As the report shows, the algorithm first reads the workflow definition from the input file, and determines corresponding pairs and improper nestings. Then, in Step (4), the algorithm detects an overlapping structure (see the dotted part in Figure 4-34(a)) and develops an equivalent structured mapping for it, as shown in Figure 4-34(b). Then, the corresponding pairs and improper nestings are re-calculated and now the number of improperly nested structures is reduced from 16 to 10. Moreover, this step handles (*C9S*, *C9J*)$^{[}_{]}$(*C11S*, *C11J*), an improper nesting of type (OR, OR)$^{[}_{]}$(OR, OR). A mapping is generated and replaces (*C9S*, *C9J*)$^{[}_{]}$(*C11S*, *C11J*). Figure 4-34(b) also shows this structured mapping. In addition, this workflow also contains improper nestings in the form of (AND, AND)$^{[}_{]}$(AND, AND), but there is no structured mapping for any of them, because neither the two of AND-Split nodes nor the two AND-Join nodes are directly connected (see Theorem 3).

However, in Step 6, because of (AND, AND)[]](OR, OR) improper nestings, *C2J* and *C2J′* are potentially blocked nodes. Since *C2J′* is upstream of *C2J*, in Step 8, *C2J′* is tested and it turns out to be a deadlock causing node. Then there is no need for testing *C2J*. In summary, this workflow is not deadlock-free.

```
Analysis Report of Workflow
Step (1)
Find pairs of corresponding nodes: 14 pairs
(C1S,C1J), (C9S,C9J'), (C9S,C9J),(C2S',C2J), (C2S,C2J'), (C3S,C3J)...
The number of workflow nodes: 59
Number of mismatched pairs: 0

Step (2)
Find improper nesting structures:
AND-AND: 8  AND-OR: 5  OR-OR: 3  Total: 16

Step (4)
Find an overlapping structure:(C9S,C9J'), (C9S,C9J), (C10S,C10J),
(C10S',C10J)
A mapping is generated and the workflow is updated
Re-find pairs of corresponding nodes: 14 pairs
(C1S,C1J), (C9S,C9J'), (C9S,C9J),(C2S',C2J),(C2S,C2J'),(C3S,C3J) ...
Number of mismatched pairs: 0
Re-find improper nesting structures:
AND-AND: 7  AND-OR: 2  OR-OR: 1  Total: 10

Find improper nesting of OR-OR: (C9S,C9J)[](C11S,C11J)
A mapping is generated and the workflow is updated
Re-find improper nesting structures:
AND-AND: 7  AND-OR: 2  OR-OR: 0       Total: 9

Step (6)
Potentially blocked nodes: C2J' in (C3S,C3J)[](C2S,C2J'), C2J'.in="R-"
                           C2J in (C3S,C3J)[](C2S',C2J), C2J.in="L-"
Step (7)
C2J' is blocked and cannot be bypassed.
C2J is a downstream node of C2J'. C2J is blocked and has no bypass.

Conclusion
Is the workflow structured? ------- No
Does this workflow have equivalent structured mapping? ----- No
Is the workflow deadlock-free? --- No, C2J' is a deadlock causing node
```

Figure 4-36: Analysis Report of Workflow in Figure 4-34

Therefore, to improve this workflow, first of all, we need to correct improper nesting (*C2S′*, *C2J*)[]](*C3S*, *C3J*). For example, we can suggest placing *C2S′* downstream of *C3J* and therefore (*C2S′*, *C2J*) will be no longer improperly nested with (*C3S*, *C3J*). Alternatively, we could make (*C3S*, *C3J*) an AND pair and then change the improper

nesting in the form of (AND, AND)[ ](OR, OR) to that in the form of (AND, AND)[ ](AND, AND). Certainly, we need to ensure that the workflow can still capture desired semantics after such modifications.

4.7 Discussion and Conclusion

Traditionally, workflows are verified by strict notions of correctness such as structuredness. However, as workflow processes become more complex, structured workflows are not able to offer the desired flexibility and expressive power. In this paper, we have created formal taxonomy of unstructured workflows based on a notion of improper nesting and mismatched pairs. We have shown how this taxonomy can help in analyzing unstructured workflows and determining whether they are correct, and if so whether they can be transformed into equivalent structured mappings. Such an equivalent structured mapping may involve some redundancies, but it gives another way to verify that a workflow is correct. Moreover, as mentioned earlier most tools do not support unstructured workflows; however, if some kinds of unstructured workflows can be mapped into structured ones, this can provide an easy way to increase the expressive power of the workflows that can be supported by these tools.

Moreover, we introduced a relaxed notion of correctness for unstructured workflows and developed an algorithm to diagnose them. Our notion of relaxed correctness is called weak correctness. In a weakly correct workflow there are some blocked nodes; however, they do not prevent the workflow from completing. The

blocked nodes can also represent some exceptions or abnormal situations designed into the workflow as contingency plans, as illustrated by the example shown in Figure 4-1.

However, such workflows must be carefully verified. Our diagnosis algorithm detects structural flaws based on the taxonomy. It provides detailed causes for blocked nodes, deadlocks and multiple instances. This research helps investigate how to achieve workflow correctness and flexibility in process modeling.

The main contributions of this work are to: (1) motivate the need for and formally introduce a new class of weakly correct workflows; (2) Develop ways for checking weak correctness. As shown in Figure 4-37, the class of weakly correct workflows is naturally larger than the class of strictly correct workflows. Intuitively, a weakly correct workflow is deadlock free, though some paths may not finish and multiple instances of certain activities may arise. The diagnosis algorithm can highlight such situations to the users and help them decide if any action needs to be taken. The distinction between workflows with deadlocks and ones with blocked nodes (but no deadlocks) is crucial to our notion of weak correctness. In previous research this distinction has not been made. The complexity of our algorithm is limited by step 7 of Figure 4-33, and it is exponential in the number of potentially blocked nodes in the worst case, but much less in the average case.

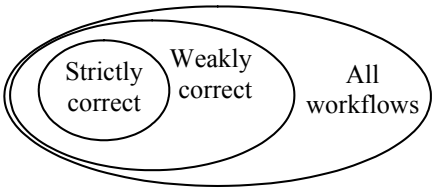


Figure 4-37: Classes of workflows

We expect our future work to focus on applying the taxonomy and the algorithm to dynamic workflows [76, 77] and exception handling. Also, it would be interesting to investigate patterns for supply chain process integration under this framework.

Chapter 5

Discussion and Conclusions

5.1 Summary

In this dissertation, we proposed an integrated framework to study information sharing, event management and process verification. Under this framework, first, inter-organizational processes are modeled formally by UML diagrams and information sharing is captured by a parameterized model based on the ECA rules. By adjusting parameters in the model, we get different supply chain configurations and these configurations are evaluated in terms of supply chain performance. These configurations are stored as organizational memory in supply chains.

Second, we analyzed causes and effects of supply chain events in order to select suitable configurations as a response to significant events. We used time colored Petri nets to model causal and temporal relationships between events. Seven basic event patterns were developed to capture common scenarios of supply chain events. The set of event patterns is extensible. For example, new patterns can be created by combining the basic patterns. By mapping an event rule to a pattern, one can set up a Petri net model for this rule easily. We used dependency graphs to visualize the causal dependencies between events. Moreover, we performed sensitivity analysis and what-if scenarios analysis to identify significant events and suggested resolution strategies for them.

Third, supply chain configurations and event management require correct business processes. We studied process verification based on workflow technologies.

There are two types of workflows, structured and unstructured. Structured workflows are strictly correct but restricted. Unstructured workflows may be incorrect but they can model a variety of process scenarios. Since unstructured workflows provide more flexibility in modeling complicated inter-organizational processes, we proposed an analysis taxonomy for unstructured workflows. This taxonomy categorizes unstructured workflow along two dimensions, mismatched pairs and improper nesting. Using this taxonomy, we analyzed different types of unstructured workflows and showed their correctness of each type. We extended the concept of strict workflow correctness and introduced a notion of weak correctness, which emphasizes proper termination of a workflow. A diagnosis algorithm was developed based on the taxonomy to verify the strict or weak correctness of workflows. This algorithm can detect structural flaws, indicate causes of those flaws, and suggest how to fix them. In addition, the algorithm can show the possibility of transforming unstructured workflows to structured ones and develop equivalent or quasi-equivalent structured mappings.

In summary, information sharing, event management and process management are three key components in supply chain management. These issues were studied thoroughly in this thesis and solutions were proposed to solve issues including supply chain configurability, sense-and-respond capability and process verification. We demonstrated an integrated framework for studying these issues. Under this framework, information sharing is leveraged to achieve supply chain configurations, configurations are means of responding to supply chain events, and correct inter-organizational processes ensure successful execution of information sharing and configurations.

5.2 Contributions

This dissertation makes the following contributions:

(1) We provided a methodology for the design of configurable supply chains based on information sharing. The methodology consists of several steps, many of which can be automated (or partially automated) using well-known technologies like UML, XML, and ECA rules. We explored different supply chain configurations, such as daily information sharing, weekly information sharing, mixed daily and weekly information sharing, and sharing information about the occurrences of events, and showed their effectiveness in terms of supply chain performance. We learned that supply chain changes, such as changes in cost structures, market competitiveness and demand variability, and exceptions can lead to different information sharing requirements and supply chains should adjust their information sharing in a timely manner in order to achieve the best performance.

(2) We developed an innovative approach to modeling events and event dependencies formally in the context of supply chain management. This approach can capture timing and causal relationships between events precisely. We also performed comprehensive simulation experiments to analyze events and compare event resolution strategies in terms of supply chain performance. Therefore, we actually showed a new method to manage supply chain performance through the lens of events. Also, this approach is a new application of Petri nets in the area of supply chain management.

(3) We designed a simple but complete taxonomy for unstructured workflows. This taxonomy allows us to analyze unstructured workflows and draw conclusions on their correctness. We also provided a diagnosis algorithm that can identify structural flaws of a workflow, point out the causes of such flaws, and derive any structured mappings if such mappings exist. In contrast to existing approaches, our diagnosis algorithm not only provides a Yes/No answer on the workflow correctness, but also gives structural evidence of execution problems as well as suggestions for correcting structural flaws.

(4) We developed a new "sense-and-respond" framework based on formal event analysis and dynamic information sharing. This framework shows that information sharing can be used as an effective response to supply chain events.

(5) We proposed a systematic framework to study supply chain events, information sharing and supply chain processes. We demonstrated that in a tightly integrated supply chain environment, any change in one of these three key components can affect the others. This framework provides guidelines for designing proper architectures, such as an electronic hub, in order to achieve successful supply chain integration.

5.3 Implications to Supply Chain Management Practices

5.3.1 A New Supply Chain Infrastructure

The focus of supply chain management has shifted from efficiency and cost-effectiveness to sustainable advantage. Lee's study [52] shows that supply chains with outstanding performance actually possess three critical qualities: *agility*, *adaptability* and *alignment* (i.e., the ability to align the interests of all supply chain partners). To foster these three capabilities, a supply chain needs to sense changes in supply and demand in a timely (often real-time) manner, interpret these changes, and respond to them quickly by modifying strategies in supply, products and technologies. In addition, the alignment of interests can only be achieved by extensive collaboration and sufficient information sharing.

However, traditional ERP systems focus on the integrated transaction processing inside a company but offer limited support on inter-organizational processes. Although supply chain management applications, such as SAP, aim at providing a higher level decision support capability across companies, they are in general not able to allow enough flexibility to support today's business dynamics [98]. These applications are typically built upon a set of process reference models (for example, event-driven process chain (EPC) modeling is used in SAP [2]), which usually allow very limited configurability. Therefore, still most supply chain infrastructures (i.e., supply chain management systems as well as closely related applications and tools) are not capable of providing supply chains agility, adaptability and alignment. It has been observed that sophisticated decision support systems, optimization techniques and artificial intelligence

are merging with supply chain infrastructures in order to make supply chains sensitive to changes and help decision making to respond to these changes quickly [84].

The results of this dissertation can contribute to the design of a supply chain infrastructure which can meet the requirements for agility, adaptability and alignment. Figure 5-1 shows such a supply chain infrastructure. This infrastructure contains an information sharing and event management hub, which analyzes events and facilitates information sharing between partners by choosing suitable supply chain configurations (see Section 2.6). Figure 5-1 also describes the interactions between the hub and ERP or supply chain applications. In this infrastructure, ERP systems or supply chain applications provide planning, transactional and process-related data to the hub. As a feedback, the changes in supply chain configurations will indicate the adjustment in ERP systems or supply chain applications, such as modifications to business processes and system reconfigurations. Kapoor et al. [42] designed a similar infrastructure in support of sense-and-respond capability. Their architecture enables an enterprise to proactively monitor trends in demand and help it act in a timely manner by optimization techniques. On the other hand, our infrastructure focuses on how information sharing can be leveraged to achieve sense-and-respond capability in a supply chain. Certainly, optimization techniques can also be included in our architecture as a new module, for example, to discover demand changes, performance exceptions or event patterns by mining shared data objects. Such a module can make our architecture more powerful and able to support decision making on a wider range of supply chain problems.

Also note that the event management and information sharing hub actually acts as a decision support tool. This tool collects data and events, analyzes them, and helps a

supply chain make decisions in a timely manner. A unique feature is that the decisions are related to information sharing and inter-organizational processes in a supply chain, and, therefore, they are stored as supply chain configurations.

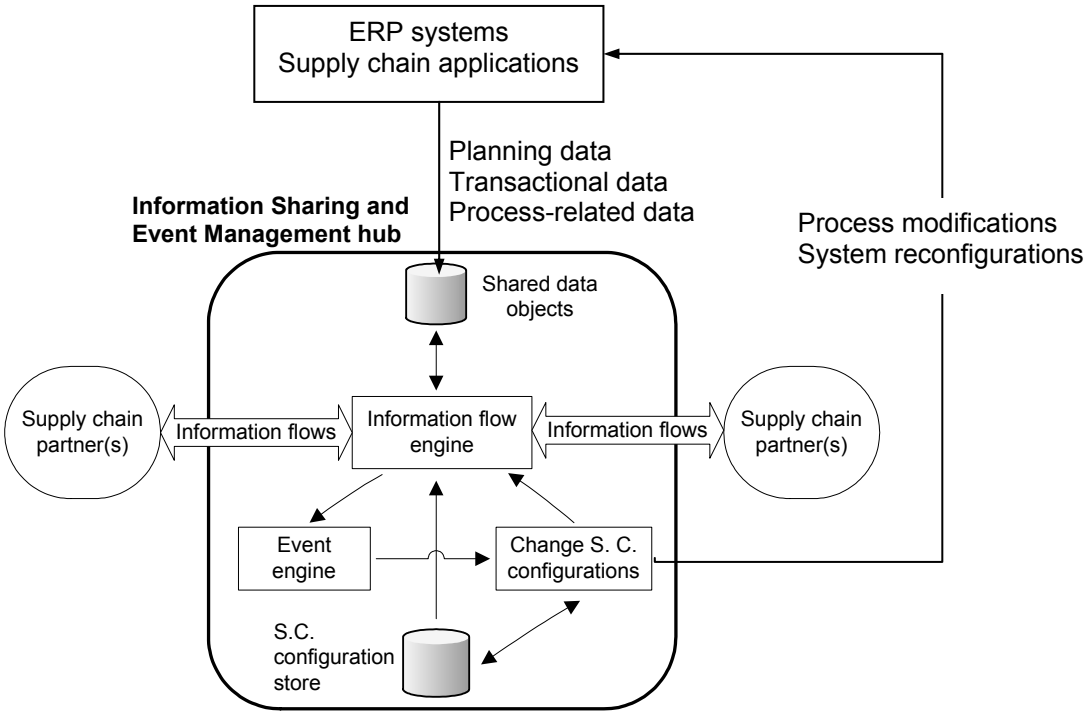


Figure 5-1: A Supply Chain Architecture

5.3.2 Organizational Memory

A supply chain configuration is one type of "organizational memory". Organizational memory refers to "stored information about a decision stimulus and response that, when retrieved, comes to bear on present decisions" [93]. A configuration captures a supply chain's experience in reacting to a particular event and change. When the same event or change recurs or a similar one happens, this configuration can provide

inference capability in responding to it. For example, in Section 2.5.5.3, we showed that when the demand variability is not stable, overall, the configuration with mixed daily and weekly information can achieve a better performance than that with pure daily or weekly information sharing. Therefore, when a supply chain faces a non-stationary demand (for example, because of various promotions), the mixed information sharing could be a feasible strategy to cope with the variation in demand.

Similarly, event rules can be another type of "organizational memory". Typically, an event rule shows the cause-effect relationships between events. The "cause" events are "decision stimuli" and the "effect" events result from the decision taken. Such a decision, which has been verified by previous experience, is clearly defined as a business rule or an event aggregation rule. An illustrating example is that "three out-of-stock events happening during a week" indicate potential supply problems and a supply chain manager should be notified immediately. This event rule clearly tells how to respond to three repetitive occurrences of out-of-stock events.

Moreover, in this research, not only did we clearly identify two types of organizational memory, but we also defined how information from such organizational memory can be stored, retrieved and utilized. Supply chain configurations are stored as ECA rules and can be queried and simulated. For event rules, we designed event patterns to classify this type of "organizational memory". Further, we developed a Petri net based approach to automatically retrieve information from this type of memory for decision making.

5.4 Limitations and Future Work

We recognize some limitations of this research and the opportunity they present for future work. First, we focused on a framework for configuring supply chains only on the operational and tactical levels. Strategic configurations of supply chains have not been explored. A strategic supply chain configuration refers to a set of selected process categories, the detailed information sharing requirements, and supply chain partnerships pertaining to each process category. SCOR model [87] defines 30 process categories, such as make-to-order and engineer-to-order, and a "supply chain configuration" consists of a set of selected process categories. However, SCOR model does not address how to implement such a "configuration" and how to switch from one "configuration" to another. We plan to extend our framework to strategic supply chain configurations. For example, as supply chains are increasingly opting for make-to-order (MTO) or make-to-stock (MTS) operations that match supply to demand more closely, strategic configurations can help the implementation of these types of operations because they describe the specific partnerships (e.g., Vendor Managed Inventory, Collaborative Design etc.), specific information sharing patterns (i.e., by the parameterized information sharing model), and detailed inter-organizational processes. We expect to further investigate information sharing between process categories in the SCOR model and derive configurations for supply chains in a strategic context.

Second, the validation of supply chain configurations needs to be further extended. Each configuration should be validated both syntactically and semantically. Syntactically, the ECA rules in each configuration should have no conflict. We proposed

a relational data model to validate data dependencies between ECA rules (i.e., one information flow causes a change to a shared data object and this change triggers another information flow). Certainly, we also need to validate complicated composite events and XQUERY conditions in ECA rules. Semantically, we need to ensure the validity of each parameter in an ECA rule. For example, if the "condition" parameter in an ECA rule specifies the required order fill rate, this parameter can only be adjusted in a reasonable value range. In general, to ensure a configuration semantically correct, we may need to define specific data range for each parameter. Also, the dependencies between ECA rules should follow business practices. For example, in the VMI arrangement, after the customer shares the demand with the vendor, the vendor should propose a replenishment order for the customer to confirm it, rather than send a ship notice to the customer directly. Therefore, we plan to develop a detailed procedure for verifying supply chain configurations syntactically and semantically.

Third, simulation is the main evaluation method in this research. We used simulation to evaluate supply chain configurations. As we pointed out in Section 2.5.6, for the illustrating examples provided, simulation probably is the most convenient evaluation tool. Other evaluation tool, such as optimization techniques, may be used to evaluate a configuration if theoretical models for configurations can be set up and elegantly solved. In addition, we also used simulation to test our Petri net based approach for event management. The limitation of this approach is that we may not derive all possible event dependency graphs, as we pointed at the end of Section 3.5.2. However, traditional theoretical analysis based on reachability techniques [14, 91] may not be able to discover all dependency graphs for the Petri nets used in this research either, since

these techniques can only perform reachability analysis for relatively small and simple Petri nets while large time color Petri nets are typically required to model complicated supply chain events. Therefore, in future work, it will be useful to design an algorithm or heuristic that can automatically generate dependency graphs containing events of interest.

Finally, although the diagnosis algorithm developed in Chapter 4.6.1 can analyze most types of unstructured workflows, it may not be able to deal with (or completely handle) some special cases. These cases include: (1) unstructured loops with more than one additional entrance and/or more than one additional exit, or unstructured loops entering/exiting other loops; and (2) unstructured workflows with only second-order or higher-order improper nesting in the form of (OR, OR)$^{[}_{]}$(OR, OR). For such unstructured loops in (1), this algorithm simply detects and reports them without further analysis. For (2), this algorithm can conclude that they are strictly correct, but it cannot develop equivalent structured mappings for them. Certainly, these tasks will be tackled as future exercises.

Bibliography

1. Aalst, W.M.P. van der. "The application of Petri nets to workflow management," *The journal of Circuits, Systems and Computes*, 7(1):21-66, 1997.

2. Aalst, W.M.P. van der. "Formalization and Verification of Event-driven Process Chains," *Information and Software Technology*, 41(10):639--650, 1999.

3. Aalst, W.M.P. van der, and B., Hofstede. "Verification of Workflow Task Structures: A Petri-net- based approach," *Information Systems*, 25(1):43-69, 2000.

4. Aalst, W. M. P. van der, Hee, K. M. van. *Workflow Management: Models, Methods, and Systems*, MIT Press 2002.

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

6. Alvarenga, C. A., and Schoenthaler, R.C. "A New Take on Supply Chain Event Management," *Supply Chain Management Review*. March/April. 2003, pp. 29-35.

7. Alonso, G., Casati, F., Kuno, H. and Machiraju, V. *Web Services Concepts, Architectures and Applications*, Springer Verlag, 2004.

8. Amerom, M.V., and Speyer, M. "XML and Supply Chain Management," *Supply Chain Management* 5(1):12-14, 2000.

9. Anderson, D., and Lee, H. "Synchronized Supply Chains: The New Frontier," *Achieving Supply Chain Excellence Through Technology*, Montgomery Research, Volume 1, April 15, 1999 (available online at http://www.ascet.com).

10. Angulo, A., Nachtmann, H and Waller, M. "Supply Chain Information Sharing in a Vendor Managed Inventory Partnership," *Journal of Business Logistics* 25(1):101-116, 2004.

11. Asgekar, V. "Event Management Graduates with Distinction," *Supply Chain Management Review*, September/October 2003, pp. 15-16.

12. Ayers, J. B. *Supply chain project management: a structured collaborative and measurable approach*, St. Lucie Press, Boca Raton, FL, 2004.

13. Ball, M. O., Ma, M., Raschid, and L., Zhao, Z. "Supply Chain Infrastructures: System Integration and Information Sharing," *ACM SIGMOD Record*. 31(1), 2002.

14. Berthomieu, B., and Diaz, M. "Modeling and Verification of Time Dependent Systems Using Time Petri Nets," *IEEE Transactions on Software Engineering*, 17(3):259-275, 1991.

15. Bi, H. and Zhao, L. "Process logic for verifying the correctness of business process models," *Proceedings of International Conference on Information Systems (ICIS 2004),* Washington, D.C. , December 12-15, 2004.

16. Bodendorf, F. and Zimmermann, R. "Proactive Supply-Chain Event Management with Agent Technology," *International Journal of Electronic Commerce*, 9(4): 57-89, Summer 2005.

17. Carlson, D. *Modeling XML Applications with UML: Practical e-Business Applications*, Addison-Wesley, 2001.

18. Casati, F., Du, W. and Shan, M. "Semantic Mapping of Events," HP Labs Technical, HPL-98-74 980421.

19. Choi, Y and Zhao, J. L. "Decomposition-Based Verification of Cyclic Workflows," In *Proceedings of Automated Technology for Verification and Analysis: Third International Symposium* (ATVA 2005), Lecture Notes in Computer Science, (3707):84-98, Springer-Verlag, 2005.

20. Chopra, S., and Meindl, P. *Supply Chain Management*, Prentice Hall, Upper Saddle River, NJ, 2001.

21. Christensen, and S., Hansen, N.D. "Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs", *Application and Theory of Petri Nets 1993*, Marsan, M. A. (ed.), Lecture Notes in Computer Science, 691, pp.186-205. Springer-Verlag, Berlin, 1993.

22. Christiaanse, E. "Performance Benefits through Integration Hubs," *Communication of ACM* 48(4):95-100, 2005.

23. Christopher, M., and Peck, H. "Building the Resilient Supply Chain," *The International Journal of Logistics Management* 5(2):1-13. 2004.

24. Covisint. Covisint Media Kit (available online at http://www.covisint.com), 2004.

25. Duffy, R., and Fearne, A. "The Impact of Supply Chain Partnerships on Supplier Performance," *The International Journal of Logistics Management* 15(1):57-71, November 2004.

26. Finley, F., and Srikanth, S. "7 Imperatives for Successful Collaboration," *Supply Chain Management Review*, January/February, 2005, pp. 30-37.

27. Fox, M.S., Barbuceanu, M. and Teigen, R. "Agent-Oriented Supply Chain Management," *The International Journal of Flexible Manufacturing Systems*, (12): 165-188, 2000.

28. Gardner, R. and Harle, D. "Pattern discovery and specification translation for alarm correlation," In *proceedings of Network Operations and Management Symposium (NOMS'98)*, New Orleans, USA, February 1998, IEEE.

29. Georgakopoulos, D. and Hornick, Mark "An Overview of Workflow Management From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Database*, (3):119-153, 1995.

30. Gosain, S., Malhotra, A. and El Sawy, O.A. "Coordinating for Flexibility in e-Business Supply Chains," *Journal of Management Information Systems*, 21(3): 7-45, 2004.

31. Gruschke, B. "Integrated Event Management: Event Correlation Using Dependency Graphs", In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 98)*, Newark, DE, USA, October 1998.

32. Haeckel, S. H., *Adaptive Enterprise: Creating and Leading Sense-and-Respond Organizations*, Harvard Business School Press, 1999.

33. Hajibashi, M. "E-Marketplaces: The Shape of the New Economy," *Achieving Supply Chain Excellence through Technology*. Montgomery Research. Volume 3, April 15, 2001. (available online at http://www.ascet.com).

34. Hasan, M, Sugla, B., and Viswanathan, R. "A conceptual framework for network management event correlation and filtering systems," In M. Sloman, S. Mazumdar, and E. Lupu, editors, *Integrated Network Management* VI, pages 233–246, Boston, MA, May 1999.

35. Hofstede, A.H.M. ter, Orlowska, M.E, and Rajapakse, J. "Verification Problems in Conceptual Workflow Specifications," *Data and Knowledge Engineering*, 24(3): 239-256, 1998.

36. Holten, R. Dreiling, A., zur Muehlen, and M., Becker, J. "Enabling Technologies for Supply Chain Process Management," *Proceedings of the IRMA 2002 Conference*, Seattle, 2002.

37. Horvath, L. "Collaboration: The key to value creation in supply chain management," *Supply Chain Management: An International Journal* 6(5):205-207, 2001.

38. IBM MQSeries, http://www-306.ibm.com/software/integration/wmq/.

39. IBM Rational XDE, Version 2003.06.12, http://www-306.ibm.com/software/awdtools/developer/rosexde/, 2003.

40. Jensen, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Volume 1, Springer-Verlag, Berlin Heidelberg, 1996.

41. Jensen, K. "An Introduction to the Practical Use of Coloured Petri Nets," *Lectures on Petri Nets II: Applications*, Reisig, W and Rozenberg, G (eds.), Lecture Notes in Computer Science, (1492):237-292, Springer-Verlag 1998.

42. Kapoor, S., Bhattacharya, K., Buckley, S., Chowdhary, P., Ettl, M., Katircioglu, K., Mauch, E., and Phillips, L. "A technical framework for sense-and-respond business management," *IBM Systems Journal*, March, 2005.

43. Keaton, M. "Using the Gamma Distribution to Model Demand when Lead Time is Random, " *Journal of Business Logistics* 16(1): 107-131, 1995.

44. Kelton, D., Sadowski, R. and Sturrock, D. *Simulation with Arena*, Mc Graw Hill, New York, 2004.

45. Kiepuszewski, B., Hofstede, A.H.M, and Bussler, C. "On Structured Workflow Modeling" *In Proceedings CAiSE'2000,* LNCS Vol. 1797, Springer Verlag.

46. Kiepuszewski, B. "Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows", PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.

47. Koehler, J., Hauser, R., Sendall, S. and Wahler, M. "Declarative techniques for model-driven business process integration," *IBM Systems Journal*, March, 2005

48. Lee H., Padmanabhan, V., and Whang, S. "The bullwhip effect in supply chains," *Sloan Management Review* 38(3):93-102, 1997.

49. Lee, H., and Whang, S. "Information sharing in a supply chain," *International Journal of Manufacturing Technology and Management* 1(1):79-93, 2000.

50. Lee, H., and Whang, S. "E-Business and Supply Chain Integration," *Stanford Global Supply Chain Management Forum.* Stanford University Report SGSCMF-W2-2001. Stanford CA. 2001.

51. Lee, H. "Simple Theories for Complex Logistics," *Optimize*, Issue 22, 2004.

52. Lee, H. "The Triple-A Supply Chain," *Harvard Business Review*, October 2004, pp.102-112.

53. Lee, H., Padmanabhan, V. and Whang, S. "The Bullwhip Effect in Supply Chains," *Sloan Management Review* (38):93-102, 1997.

54. Lewis, L. "A case–based reasoning approach to the resolution of faults in communication networks," In *Proceedings of the IFIP TC6/WG6.6 Third International Sysposium on Integrated Network Management*, Hegering, H. G. and Yemini, Y. (eds.) San Francisco, USA, April 1993, pp. 671–682.

55. Li, G., Yan, H., Wang, S. and Xia, Y. "Comparative Analaysis on Value of Information Sharing in Supply Chains," *Supply Chain Management*, 10(1): 34-46, 2005.

56. Linthicum, D. "Understanding Supply Chain Integration," *B2B Application Integration*. Addison-Wesley, Indianapolis, IN, 2001, pp. 313-324.

57. Liu, N.K and Dillon, T. "An approach towards the verification of expert systems using numerical Petri net," *International Journal of Intelligent Systems*, 6(3):255-276. 1991.

58. Liu, R., and Kumar, A. "Leveraging Information Sharing to Increase Supply Chain Configurability," *Proceedings of International Conference on Information Systems* (ICIS 2003), S. T. March, A. Massey, and J. I. DeGross (eds.), Seattle, 2003, pp.523-536.

59. Liu, R., Kumar, A., and Aalst, W.M.P. van der. "A Formal Modeling Approach for Supply Chain Event Management," In *Proceedings of 14th Workshop on Information Technologies and Systems* (WITS 2004), Dutta, A. and Goes, P (eds.). Washington D.C., December 2004, pp. 110-115.

60. Luckham, D. *The Power of Events*, Addison-Wesley, Boston, 2002.

61. Malhotra, A., Gosain, S. and El Sawy, O.A. "Absorptive Capacity Configurations in Supply Chains: Gearing for Partner-enabled Market Knowledge Creation," *MIS Quarterly*, 29(1): 145-187, March 2005.

62. Malone, T. W., and Crowston, K. "The Interdisciplinary Study of Coordination," *ACM Computing Surveys* 26(1):87-119, 1994.

63. Malone, T. W., Crowston, K., Lee, J., Pentland, B.T., Dellarocas, C., Wyner, G. M., Quimby J., Bernstein, A., Herman, G. A., Klein, M., Osborn, C. S., and O'Donnell, E., "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes," *Management Science* 45(3): 425-443, 1999.

64. Marabotti, D. "Information Technology Insights: Supply Chain Event Management Emerges in Enterprise Software," *Chemical Market Reporter*, 262(9):21-22, September 2002.

65. McCarthy, D.R., and Dayal, U. "The Architecture of an Active Database System," *Proceedings of ACM SIGMOD Conference on Management of Data*, J. Clifford, B. G. Lindsay, and D. Maier (eds.). ACM Press, New York, 1989, pp. 215-224.

66. McCrea, B. "EMS Completes the Visibility Picture," *Logistics Management*, 44(6):57-61, June 2005.

67. Meseguer, P. "A New Method to Checking Rule Bases for Inconsistency: A Petri Net Approach," In *Proceedings of the 9th European Conference on Artificial Intelligence* (ECAI-90), Stockholm, August 1990.

68. Mesquite. CSIM 19, http://www.mesquite.com/.

69. Miller H. "The Multiple Dimensions of Information Quality," *Information System Management* 13(2):79-82, Spring 1996.

70. Montgomery, N., and Waheed, R. "Supply Chain Event Management Enables Companies to Take Control of Extended Supply Chains," *AMR Research* 2001. (available online at http://www.amrresearch.com).

71. Murata, T. "Petri Nets: Properties, Analysis and Application," In *Proceedings of the Institute of Electrical and Electronics Engineers*, 77(4): 541-580, April 1989.

72. NØkkentved, C., and Hedaa, L. "Collaborative Processes in esupply Networks," *Proceedings of the 16th International Marketing and Purchasing Group Conference*. Bath, U.K, 2000. (available online at http://www.bath.ac.uk/imp/tracke.htm).

73. OMG. Unified Modeling Language Specification, Version 1.5. Object Management Group, 2003. (Available online at http://www.omg.org/technology/documents/formal/uml.htm).

74. Poirier C. and Quinn, F. "A Survey of Supply Chain Progress," *Supply Chain Management Review*, September/October 2003, pp. 40-48.

75. Ratzer, V. A., Wells, L., Lassen, M. H, Laursen, M., Qvortrup, F. J., Stissing, S. M., Westergaard, M., Christensen, and S., Jensen, K. "CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets," *Applications and Theory of Petri Nets 2003*, W. van der Aalst, E. Best (Eds.), Lecture Notes in Computer Science, (2679): 450 - 462, Springer-Verlag GmbH, 2003.

76. Reichert, M. and Dadam, P "ADEPTflex – Supporting dynamic changes of workflows without losing control," *Journal of Intelligent Information Systems---Special Issue on Workflow Managament*, 10(2):93-129, 1998.

77. Rinderle, S., Reichert, M., and Dadam, P. "Correctness criteria for dynamic changes in workflow systems - a survey," *Data and Knowledge Engineering*, 50(1): 9-34, 2004.

78. RosettaNet. http://www.rosettanet.org, 2004.

79. Ross, David F. *Introduction to e-Supply Chain Management*. St. Lucie Press, Boca Raton, FL, 2003.

80. Russell, N., Aalst, W.M.P.van der, Hofstede, A.H.M. ter, and Edmond, D. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcao e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering* (CAiSE'05), volume 3520 of Lecture Notes in Computer Science, pages 216-232. Springer-Verlag, Berlin, 2005.

81. Sadiq, W. and Orlowska, M. E. "On correctness issues in conceptual modeling of workflows," In *Proceedings of the 5th European Conference on Information Systems (ECIS `97)*, Cork, Ireland, June 19-21, 1997, pp. 943-964.

82. Sadiq, W, and Orlowska, M. E. "Analyzing process models using graph reduction techniques," *Information Systems*, 25(2):117-134, 2000.

83. Shimura, T., Lobo, J. and Murata, T. "An Extended Petri Net Model for Normal Logic Programs," *IEEE Transactions on Knowledge and Data Engineering*, 7(1): 150-162, 1995.

84. Singh, N. "Emerging Technologies to Support Supply Chain Management," *Communications of the ACM*, 46(9):243-247, 2003.

85. Strozniak, P. "Exception Management," *Frontline Solutions*, 3(8): 16-24, August 2002.

86. Sun eInsight Business Process Manager, http://www.seebeyond.com/software/einsight.asp.

87. Supply-Chain Council. "Supply-Chain Operations Reference-Model, Overview of SCOR," Version 6.0. Supply-Chain Council, Inc, Pittsburgh, PA, 2003 (available at http://www.supply-chain.org/scoroverview.asp).

88. Tyworth, J. E., Guo, Y. and Ganeshan, R. "Inventory Control Under Gamma Demand and Random Lead Time," *Journal of Business Logistics*, 17(1): 291-304, 1996.

89. Verbeek, H.M.W., Basten, T. and Aalst, W.M.P. van der. "Diagnosing Workflow Processes using Woflan," *The Computer Journal*, 44(4):246-279. British Computer Society, 2001.

90. Visual Object Modelers. Visual UML Software, version 3.0, 2003.

91. Wang, J. *Timed Petri Nets Theory and Application*, Kluwer Academic Publishers, Boston, 1998, pp. 63-123.

92. Waller, M, Johnson, M. E. and Davis, T. "Vendor Managed Inventory in the Retail Supply Chain," *Journal of Business Logistics*, 20(1):183-203, 1999.

93. Walsh, J. P. and Ungson, G. R. "Organizational Memory," *Academy of Management Review* 16(1):57-91, 1991.

94. Wu, P., Bhatnagar, R., L. Epshtein, Shi, Z. Alarm correlation engine (ace). In *Proceedings of the 1998 IEEE Network Operations and Management Symposium (NOMS'98)*, New Orleans, Louisiana, USA, 1998, pages 733-742.

95. XQuery, "XQuery 1.0: An XML Query Language," W3C, http://www.w3.org/TR/2005/WD-xquery-20050404/, April 2005.

96. Yu, Z., Yan, H., and Cheng, T.C. E. Benefits of Information sharing with supply chain partnerships. *Industrial Management & Data Systems* 101(3):114-119, 2001.

97. Zhang, D and Nguyen, D. "PREPARE: A Tool for Knowledge Base Verification," *IEEE Transactions on Knowledge and Data Engineering*, 6(6):983-989, 1994.

98. Zhao, Z.-Y., Ball, M. and Chen, C.-Y. "A Scalable Supply Chain Infrastructure Research Test-Bed," Smith Papers Online, University of Maryland, 2002 (Available online at http://bmgt3-notes.umd.edu/Faculty/KM/papers.nsf/).

99. Zuberek, W.M. "Timed Petri nets - definitions, properties, and applications," *Microelectronics and Reliability*, 31(4):627-644, 1991.

Appendix A

Simulation of Petri net in Figure 3-18

A.1 Hierarchical CPN Mapping

Here we show how our Petri nets are implemented using CPN Tools. Figure A-1 gives a Colored Petri net (CPN or CP-net) mapping from Figure 3-18. To make this mapping readable, a *hierarchical CPN* is used. In Figure A-1, each transition with a small tag, called HS-tag, is a substitution transaction, and it is mapped onto a so-called subpage. This CPN has three levels. The first level is shown in Figure A-1. The transaction "rule8&9" here can be expanded as a sub-page as shown in Figure A-2 (Level 2). Furthermore, transaction "rule8" in Figure A-2 can be substituted by a sub-page as shown in Figure A-3.

A.2 Implementation of Time Constraints

CPN tools cannot directly support the subtle time semantics used in this paper, but it can support *timed CP-nets*. In a timed CP-net, a global clock is introduced and a token can carry a time stamp. The time stamp describes the earliest model time when the token can be used. In addition, after firing a transition, the output tokens can be delayed for a *fixed time*. The details of timed CP-nets can be found in [41].

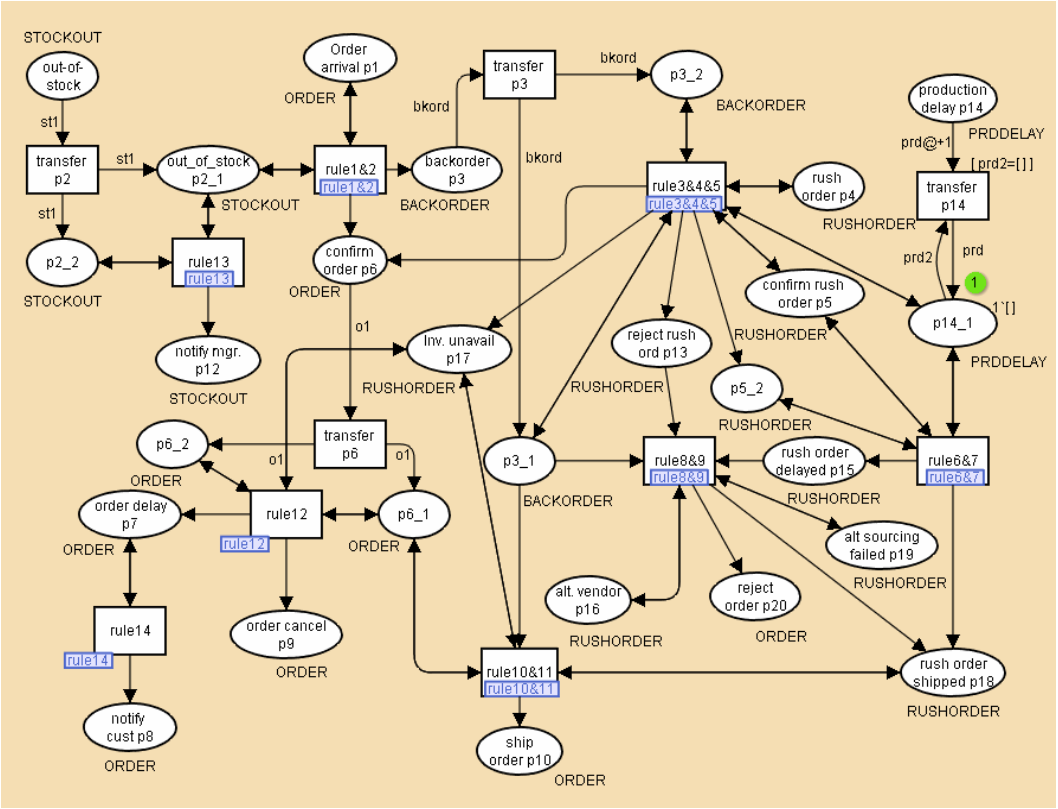


Figure A-1: CP-Net (Level 1)

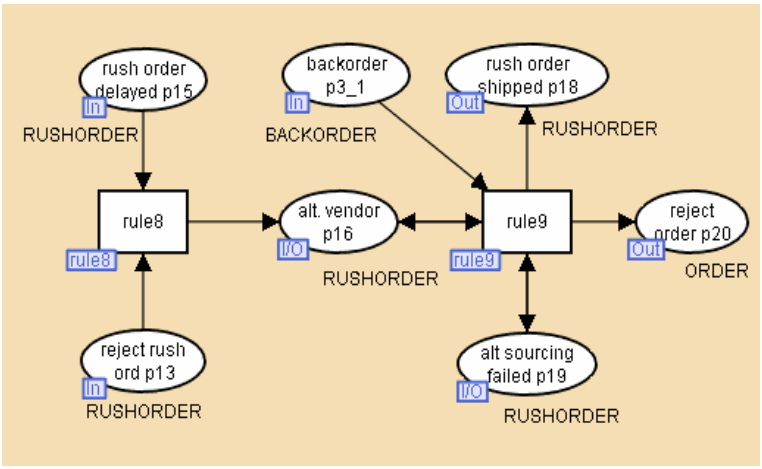


Figure A-2: Subpage for "Rule8&9" (Level 2)

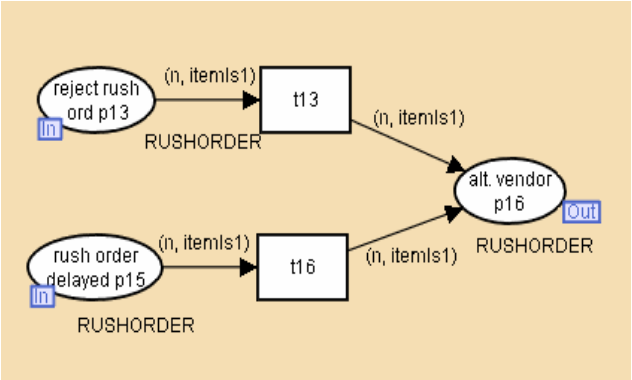


Figure A-3: Subpage for "Rule8" (Level 3)

Since CPN tools are not designed for handling time intervals attached to transitions, we had to improvise and come up with a general approach for doing so. In mapping a time Petri net, a "clock" was introduced with each transition that has an associated time interval. Whenever this transition is enabled, the clock starts and then moves independently until timeout. Figure A-5 is the mapping of the *time* Petri net of Figure A-4 , and shows a *timed* CPN with two "clocks". In Figure A-5, transition "Start 1" fires whenever there is a token in *e1*. According to Figure A-4, Rule 1 should fire in the [2,5] time interval *after* it is enabled. This interval is modeled by the L1.ran() function, which generates a random number between 2 and 5. Therefore, "Timer 1" has an initial token with a color showing the ID of event *e1*, allowed waiting time, and actual waiting time 0. Then transition "Clock 1" fires. For each firing, the actual waiting time is increased by 1 and this token will be delayed for 1 time unit. Transition "Clock 1" fires until the allowed waiting time is reached. This is controlled by a *guard* "c2 < d2" placed on this transition. At that time, if the transition "Rule 1" is still enabled, it fires. The other "clock" controls the expiration of *e1*.

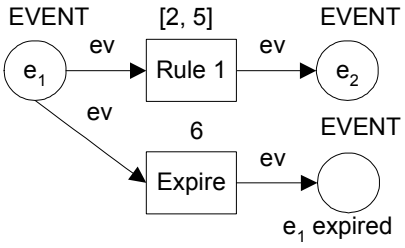


Figure A-4: Time Petri Net

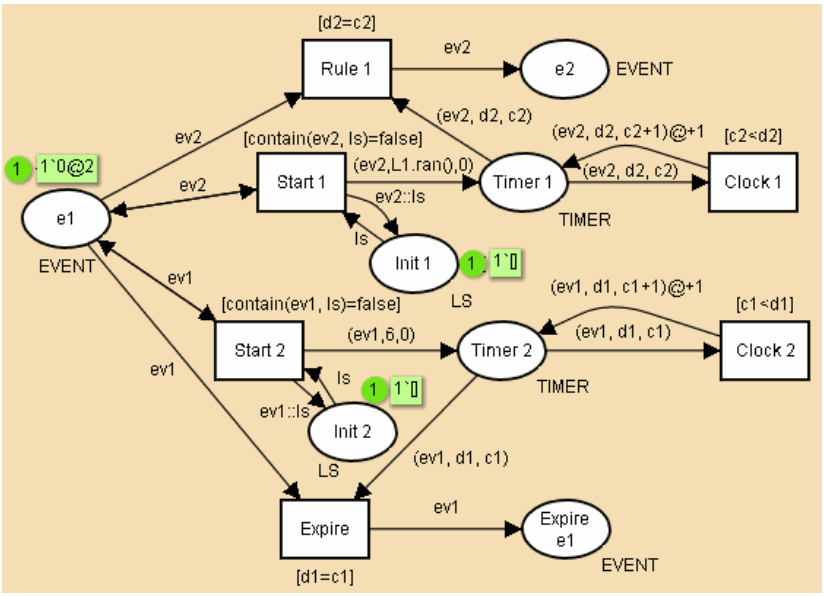


Figure A-5: Implementation with CPN Tools

A.3 Implementation of Inhibitor Arcs

We can substitute inhibitor arcs with an equivalent structure as discussed in [21]. Figure A-7 shows an equivalent Petri net of Rule 1 (see Figure A-6). In Figure A-7, each order arrival into *p1* sends an empty list to the place "stockout list". Then transition "makelist" fires if there is any token in place *p2* (i.e., out-of-stock event) and this list is appended with any item which is currently out-of-stock. In order to ensure this list is

completely generated before it is actually used by Rule 1, Rule 1 always fires 1 time unit later than transition "makelist". This delay is achieved by the place "delay", where a token arrives one time unit after transition "start" fires. Later, this list is used to control the firing of Rule 1. Two functions are used in the guard conditions of the transitions. Function *contain(item1, itemls1)* checks whether *item1* exists in list *itemls1* to prevent duplicates. Function *containls(itemls2, itemls1)* returns true if any item of list *itemls1* is contained in list *itemls2*. Therefore, transition "rule 1" fires only if none of the ordered items is in the stockout list.

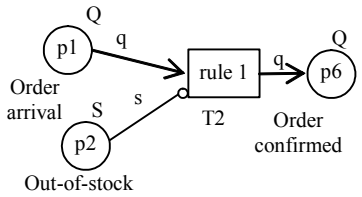


Figure A-6: Rule 1 with an Inhibitor Arc

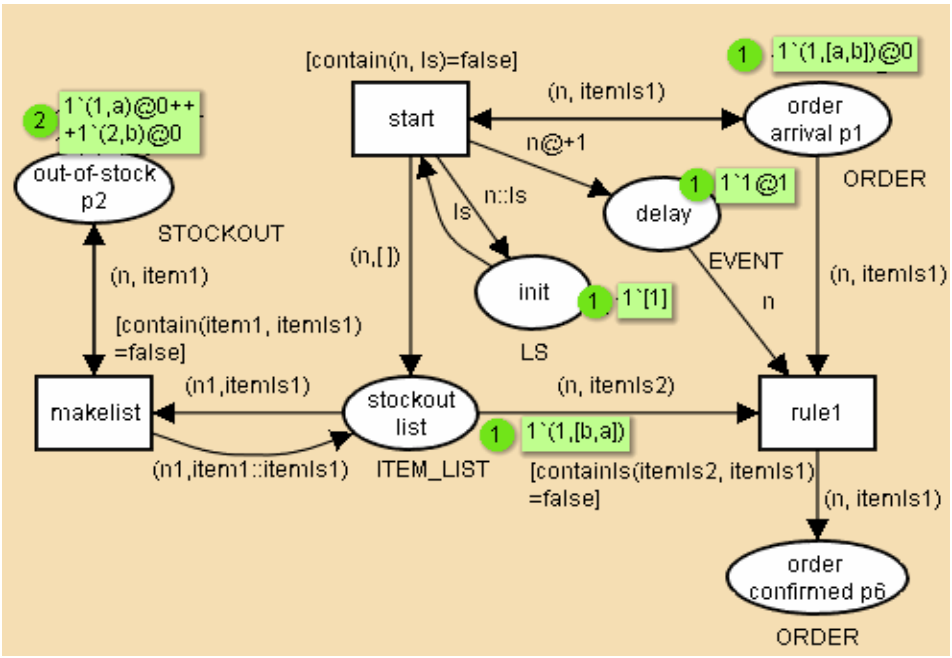


Figure A-7: CPN Implementation of Rule 1

Appendix B

Lemma 12

Lemma 12: If there are two potentially blocked nodes, pb_1 and pb_2, and there is no path between them, the sequence of testing them will not affect the final result of Algorithm `CHK_MULT_BLK_NODES`.

Proof: Suppose we first test pb_1 and then pb_2 (i.e., *sequence* $pb_1\, pb_2$) in workflow *wf*. There are three possible results after testing: (1) no blocked nodes (*result* =1); (2) blocked but not deadlock causing (*result* =2); and (3) deadlock causing (*result* =3). Next, by enumerating all possible cases, we show that the same result can be achieved if pb_2 is tested first followed by pb_1 (in the *sequence* $pb_2\, pb_1$).

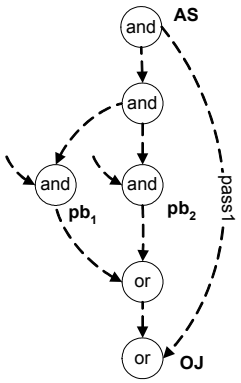
(1) No blocked nodes (*result* =1)

Result = 1 is returned only if neither pb_1 nor pb_2 is a blocked node. In *sequence* $pb_1\, pb_2$, we call algorithm `CHK_MULT_BLK_NODES(wf,1)` to test pb_1 first and *result* = 1. Then we call `CHK_MULT_BLK_NODES(wf,1)` to test pb_2 and still *result* = 1. Obviously, we can reverse the sequence and get the same result for each call of the algorithm.

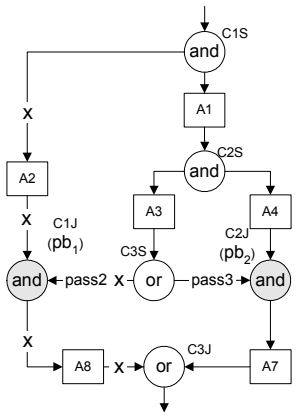
(2) Blocked but not deadlock causing (*result* =2)

In *sequence* $pb_1\, pb_2$, we find a bypass, say *pass1*, from an AND-Split node, say *AS*, to an OR-Join node, say *OJ*. By Theorem 2, *AS* must be an upstream node of both pb_1 and pb_2, and *OJ* must be a downstream node of both pb_1 and pb_2. There are two cases of *pass1*:

Case (a): *pass1 does not contain* pb_2, as shown in Figure B-1(a). In this case, if we reverse the testing sequence, we can still get bypass *pass1* (i.e., *result* = 2).



(a) *pass1* does not contain pb_2



(b) The bypass of pb_1 contains pb_2 (or the bypass the pb_2 contains pb_1)

Figure B-1: Two Types of Bypasses of Blocked Nodes

Case (b): *pass1 contains* pb_2. In other words, pb_2 becomes unblocked in the truncated workflow after testing pb_1. In this case, two exclusive paths, say *pass*2 and *pass3* are the incoming paths of pb_1 and pb_2 respectively. Since either *pass2* or *pass3*, but not both, can be taken, correspondingly, either pb_2 or pb_1, but not both can be blocked. Therefore, when testing one node, one of these two exclusive paths is removed, the other path can be taken for sure, and the other node then becomes unblocked in the truncated workflow. Thus, the testing sequence will only affect which pass (*pass2* or *pass3*) can be removed, but the same result (*result* = 2) is always drawn. An example is shown in Figure B-1(b). During testing pb_1, *pass2* is not taken and therefore, *pass3* becomes a certain input of pb_2 and then pb_2 is not blocked. Similarly, if we test pb_2 first and remove *pass3*, pb_1 becomes unblocked in the truncated workflow. Therefore, this truncated workflow is a bypass of pb_2 and *result* = 2.

(3) Deadlock causing nodes (*result* =3)

Still, there are two possible situations resulting in this result.

Case (i): both nodes are deadlock causing. In this case, we can test either one first and there is no need to test the other.

Case (ii): Only one of them is deadlock causing. Suppose pb_1 is deadlock causing. If we call `CHK_MULT_BLK_NODES(wf,1)` to test pb_1 first, we will get a disconnected workflow where only the start and the end nodes are left. Moreover, in this case, we can show that Proposition 1 described below is correct.

Proposition 1: For any sub-workflow of *wf*, say *wf'*, which also contains pb_1, the start node and the end node, if *result* = 3 is returned after calling `CHK_MULT_BLK_NODES(wf,1)` to test pb_1, we can conclude that testing pb_1 by calling `CHK_MULT_BLK_NODES(wf',1)` returns the same result, and vice versa.

Proposition 1 is correct because if one path from the start node to pb_1 is removed, then the only successor of the start node is removed. As a result, this breaks every path from the start node to the end node and then the workflow is disconnected (i.e. *result* = 3).

Next, we need to show that the same *result* = 3 is drawn in *sequence* $pb_2\, pb_1$.

In *sequence* $pb_2\, pb_1$, after testing pb_2, a truncated workflow, say wf_1, is returned. Obviously, wf_1 is a sub-workflow of *wf*. If we call `CHK_MULT_BLK_NODES(wf1,1)` to test pb_1, according to Proposition 1, we can get *result* = 3.■

Appendix C

Notations

Notations	Descriptions
(s, j)	split node s is a corresponding node of join node j
$(u, v)\ ^{[}_{]}\ (s, j)$	corresponding node pair (u, v) is improperly nested with another pair (s, j). $(u, v)\ ^{[}_{]}\ \{(x_1, y_1), (x_2, y_2),\ldots, (x_n, y_n)\}$ is used to denote n pairs of corresponding nodes nested into (u, v)
$(u\ ^{s}\ v)\ ^{[}_{]}\ (s, j)$	corresponding node pair (u, v) is improperly nested with another pair (s, j) and s is in a path from u to v, but j is not in this path
b.in	*b* is a blocked node and *b.in* is a two-character string that records which incoming paths of *b* is taken
b.in = "L-"	the left incoming path of *b* is taken but the right one cannot
b.in = "R-"	the right incoming path of *b* is taken but the left one cannot
b.in = "LR"	either incoming path of *b* might be taken but not both
pb.pred_left	the left predecessor of potentially blocked node *pb*
pb.pred_right	the right predecessor of potentially blocked node *pb*
wf.PB[]	workflow *wf* has an array of potentially blocked nodes *PB*[]
wf.B[]	workflow *wf* has an array of blocked nodes *B*[]
C1J.Pred[]	array of the predecessors of node *C1J*
C1J.pred[0]	the first predecessor of node *C1J*
C1S.Succ[]	array of the successors of node *C1S*
C1S.succ[0]	the first successor of node *C1S*
First_node(*p1*)	the first node in path *p1*
Last_node(*p1*)	the last node in path *p1*
L(*j, s*)	a loop *L* with join node *j* as its primary entrance and split node *s* as its primary exit
L.Type	the type of Loop *L*, which is one of types in Table 4-2 or Table 4-3

VITA

Rong Liu

Education

Ph.D., Supply Chain and Information Systems, Smeal College of Business, 2006

The Pennsylvania State University

M.S., Management Information Systems, conferred during Ph.D. program, 2005

The Pennsylvania State University

B.E., Industrial Engineering and **B.E.,** Automation and Control, 1996

Shanghai Jiao Tong University, China