Distributed supervisory control of workflows

Pranav Deshpande

University of South Florida

Follow this and additional works at: http://scholarcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation
Distributed Supervisory Control Of Workflows

by

Pranav Deshpande

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Industrial Engineering
Department of Industrial and Management Systems Engineering
College of Engineering
University of South Florida

Major Professor: Ali Yalcin, Ph.D.
Suresh Khator, Ph.D.
Jose L. Zayas-Castro, Ph.D.

Date of Approval:
November 14, 2003

Keywords: discrete event system, modular control, shuffle product, wfmc, ramadge, wonham

© Copyright 2003, Pranav Deshpande
Table of Contents

List of Tables......................................................................................................................... iv

List of Figures ......................................................................................................................... v

ABSTRACT............................................................................................................................. vii

Chapter 1. Introduction .......................................................................................................... 1
  1.1 Task ................................................................................................................................. 2
  1.2 Type of Tasks.................................................................................................................. 3
  1.3 Task Structure ............................................................................................................... 3
  1.4 Task Dependency .......................................................................................................... 5
  1.5 Fund Transfer Workflow Example.................................................................................. 6

Chapter 2. Literature Review ................................................................................................. 8
  2.1 Modeling Formalism ...................................................................................................... 8
  2.2 Task ................................................................................................................................. 9
  2.3 Task Structure ............................................................................................................... 10
  2.4 Inter Task Dependencies............................................................................................... 12

Chapter 3. Problem Description ............................................................................................ 14
  3.1 Finite State Automata .................................................................................................. 14
  3.2 Languages ..................................................................................................................... 16
    3.2.1 Concatenation ......................................................................................................... 17
    3.2.2 Prefix-Closure ....................................................................................................... 17
    3.2.3 Accessible States .................................................................................................... 17
    3.2.4 Co-Accessible States ............................................................................................. 17
    3.2.5 Shuffle Product ....................................................................................................... 17
    3.2.6 Blocking ................................................................................................................ 18
    3.2.7 Trim Generator ....................................................................................................... 19
    3.2.8 Non-Conflicting Languages ................................................................................... 19
    3.2.9 Controllability ......................................................................................................... 19
    3.2.10 Supremal Controllable Sublanguage ..................................................................... 20
  3.3 Supervisory Control Theory............................................................................................. 20
  3.4 Problem Description ...................................................................................................... 22
    3.4.1 The Uncontrolled Model ......................................................................................... 22
    3.4.2 The Specification Model ........................................................................................ 24
    3.4.3 The Coupled model............................................................................................... 26
    3.4.4 Identify Admissible language ............................................................................... 27
    3.4.5 Construct Supervisor $S$ ....................................................................................... 27
  3.5 Motivation....................................................................................................................... 29
  3.6 Objectives ...................................................................................................................... 31
Appendix 1. Types of Dependencies ................................................................. 72

Appendix 2. Standard algorithm for \( \uparrow \) C ................................................................. 74
List of Tables

Table 3.1 Control Pattern • for Airline Workflow ................................................................. 28
Table 3.2 States Space Increase in Specification model ............................................................. 29
Table 3.3 Task-State Comparisons .......................................................................................... 30
Table 4.1 Control Pattern $\Psi_1$ for the Begin Dependency .................................................... 36
Table 4.2 Control Pattern $\Psi_2$ for Abort Dependency ............................................................ 39
Table 4.3 Control Pattern $\Psi$ for the Airline Workflow .......................................................... 40
Table 5.1 $\Psi_1$ Control Pattern for Begin Dependency .............................................................. 49
Table 5.2 $\Psi_2$ Control Pattern for the Abort Dependency .......................................................... 49
Table 5.3 $\Psi_3$ Control Pattern for Begin on Abort Dependency ............................................... 51
Table 6.1 Supervisory Control Elements .................................................................................. 57
Table 6.2 Control Pattern for the Abort Dependency ($\Psi_3$) ...................................................... 58
Table 6.3 Control Pattern for Terminating Dependency ($\Psi_4$) .................................................. 59
Table 6.4 Control Pattern for the Begin Dependency ($\Psi_6$) ...................................................... 59
Table 6.5 Control Pattern for Begin on Abort Dependency ($\Psi_9$) ............................................. 60
Table 6.6 Control pattern for Forced Commit on Abort Dependency ($\Psi_9$) .............................. 60
Table 6.7 Centralized Control Pattern [15] .............................................................................. 61
Table 6.8 Comparison of Results ............................................................................................. 63
Table A.1 Dependency Classification [1] .................................................................................. 73
List of Figures

Figure 1.1 Accident Insurance Claim Workflow................................................................. 2
Figure 1.2 States in a Task.................................................................................................. 3
Figure 1.3 Types of Task [17]............................................................................................ 4
Figure 2.1 Petri Net Representation of a Task [1]............................................................... 11
Figure 3.1 Finite State Automata Example........................................................................ 15
Figure 3.2 Example............................................................................................................ 16
Figure 3.3 Blocking Automaton.......................................................................................... 19
Figure 3.4 Controllability................................................................................................... 20
Figure 3.5 Supervisor-Uncontrolled Model Feedback System.......................................... 21
Figure 3.6 Task represented by automaton T.................................................................... 23
Figure 3.7 Uncontrolled Model........................................................................................ 24
Figure 3.8 C₁-Specification Model for Begin Dependency [15].......................................... 24
Figure 3.9 C₂-Specification Model for Abort Dependency [15].......................................... 25
Figure 3.10. C-Specification Automaton [15]................................................................... 25
Figure 3.11 C/G-Coupled Model for Airline Workflow..................................................... 26
Figure 3.12 Supremal Controllable Sublanguage.............................................................. 27
Figure 4.1 Modular Supervisory Control Architecture....................................................... 32
Figure 4.2 Coupled Model for Begin Dependency............................................................ 35
Figure 4.3 Coupled Model for Abort Dependency.............................................................. 37
Figure 4.4 Supremal Controllable Sub-language for Abort Dependency............................ 39
Figure 5.1 Decentralized Supervisory Control [8].............................................................. 43
Figure 5.2 Supremal Language for Begin on Abort Dependency ........................................ 50
Figure 6.1 Online Bookstore Architecture ........................................................................ 54
Figure 6.2 Bookstore Substructure .................................................................................. 56
Distributed Supervisory Control of Workflows

Pranav Deshpande

ABSTRACT

The need for redesigning existing business processes to improve their efficiency makes it essential to adequately represent, study, and automate them. The WFMC defines “workflow” as computerized facilitation or automation of a business process in whole or part. It is actually a representation of the given process, which is made up of well-defined collection of activities called tasks.

Modeling and specification of a workflow involves the following steps: 1) Provide formalism for modeling and specification of workflow 2) specify the tasks together with the associated information and 3) enter the applicable business rules in form of inter-task dependencies.

Earlier attempts at modeling of workflows are based on a centralized control approach, has limited applicability for modeling and control of real life workflow due to computational complexity. In this thesis, a distributed supervisory control approach is described and shown to be computationally tractable. The application of such an approach is demonstrated with a case study.
Chapter 1. Introduction

The need for redesigning existing business processes in order to improve their efficiency makes it essential to adequately represent, study, and automate them. Business processes are market-centered descriptions of an organization’s activities comprising both material processes and information processes [12]. Workflow Management supports both business process specification and automated extension of business procedure. The workflow management system is the specific software, which controls the automated aspects of a workflow. The workflow management software defines, manages, and executes workflow through the execution of software. The Workflow Management Coalition (WFMC) was formed to promote workflow and establish standards for Workflow Management Systems (WFMS). The WFMC released a glossary, which provides a common set of terms for workflow users, vendors and researchers.

The WFMC defines “workflow” as computerized facilitation or automation of a business process in whole or part. It is actually a representation of the given process, which is made up of well-defined collection of activities called tasks.

As an example let us consider a workflow for “Accident Insurance Claim” process in Figure 1.1. The way the process works is that a person submits a claim for insurance. The claim is received and a check is then made on the person’s insurance. The place where the person wants to repair his or her vehicle is then contacted to make further investigations. When these things work out correctly a letter is sent to the concerned person and the damage is paid. There are five tasks that can be identified in this workflow submit claim, check insurance, contact garage, send letter and pay damage.
Although most workflow processes are used within a department, intra organizational workflow can exist. Salimifard and Wright [23] describe three types of workflow based on the level of persistence in process definition and routing of tasks. *Production Workflow* is characterized by a fixed definition of tasks and order of execution. *Administrative workflow* cases follow a well-defined procedure but alternative routing of cases is possible. *Ad Hoc workflow* handles cases derived from a predefined process and modifies the same template to meet the specific needs of different workflow. This research mainly concentrates on production workflow. The workflow referred to throughout this research is a production workflow.

1.1 Task

A *task* is the key concept of a workflow. The tasks, while serving the given function in the process, have a certain information input requirement and may produce information as output. Each task has a definite structure and consists of several events that are to be executed by carrying out the activities in the task. All tasks have a starting state and one or many terminating states. The events are represented as transitions between the states. A task is shown in Figure 1.2. If the task is started, then the task moves from *Initial* state (In) to *Executing* state (Ex). Generally the terminating states are Failed/Abort and Done/Commit depending upon the task structure. Once the task reaches a termination state, no further transitions are allowed.
1.2 Type of Tasks

Tasks can be classified as *Transactional* and *Non Transactional*. A transactional task is one that minimally supports atomicity and maximally supports all ACID (Atomic, Consistent, Isolated, and Durable) properties. The externally visible states of a transactional task are: *initial*, *executing*, *aborted* and *committed*.

A non-transactional task is used when a generic application that does not enforce atomicity or isolation is to be modeled in a workflow. Such a task cannot be micromanaged. A non-transactional task can be initiated or forced to fail but that's all that can be done [35].

1.3 Task Structure

According to Rusinkiewicz and Sheth [22] a task structure can be defined by providing: 1) a set of externally visible execution states of a task 2) a set of legal transitions between these states and 3) the conditions that enable these transitions. The type of task structure to be used depends on the type of workflow that is being modeled. Krishna Kumar and Sheth [17] classify the tasks based on their processing entities. The tasks that involve humans either directly or in interaction with some computer programs are referred to as ‘User’ tasks. The processing entities for the user tasks are humans who use software like business automation software such as spreadsheets and document processing systems. The tasks that do not involve humans are called

![Figure 1.2 States in a Task](image-url)
‘application’ tasks. The processing entities for application tasks are usually modern application systems, servers supported by client server processing systems etc.

Krishna Kumar and Sheth [17] have also provided task structures, which differ from each other depending upon whether or not the transitions are controllable. The controllability or non-controllability of task depends on the interface or the processing entity. A user task is characterized by a non-transactional task structure, which reaches a failed state if a system error is encountered during its execution. If the task is executed successfully it reaches the done state. The application tasks are characterized by transactional task structures, which have aborted and committed state as its final states. The transactional and non-transactional task structures are shown in Figure 1.3. The authors also state that a workflow can be a combination of different task structures, the structure of each task depending upon the task properties.

![Figure 1.3 Types of Task](image)

**Figure 1.3 Types of Task** [17]
1.4 Task Dependency

Tasks that constitute a workflow are related to each other and are dependent on each other. These task dependencies are called intra-workflow dependencies. Task dependencies may also exist across workflows and are referred to as inter workflow dependencies [1]. Dependencies can be broadly classified into ‘Static’ and ‘Dynamic’ depending upon the time when they are enforced.

Static dependencies between the workflow transactions are defined before the actual execution takes place. Generally a precondition is defined for the execution of each task, so that information about all the possible tasks and their dependencies is known in advance. But only those tasks whose preconditions are satisfied are executed [22]. Dynamic dependencies develop as the workflow progresses through its execution and the enforcement is done usually by a well-defined set of rules [22]. Only the static dependencies are considered in this work.

Task dependencies can also be classified based on their precondition. The classification is done as follows:

- Control Flow Dependency: A control flow dependency between a pair of tasks \( t_1 \) and \( t_2 \) specifies the condition under which a particular task (say \( t_1 \)) is allowed to enter a particular state based on the state of the other task (say \( t_2 \)). An example of such a dependency is the Begin on Commit (BC) dependency, which states that task \( t_2 \) cannot ‘begin’ (enter executing state) unless and until task \( t_2 \) ‘commits’ (enter the committed state). The control flow dependencies are considered for this thesis.

- Value Dependency: The value dependencies encompass relations between tasks based on the values generated by the related tasks.

- External Dependency: If external agents or parameters cause the dependencies, the dependencies are called external dependencies. Usually such dependencies are caused with the external parameter being time.
1.5 Fund Transfer Workflow Example

The control flow dependencies will be demonstrated with the help of an example from [33]. Consider a simple workflow that demonstrates the transfer of funds between bank accounts by debiting one account and crediting the other. The workflow contains two tasks, Credit and Debit. Both tasks involve a start event, a termination event (either commit or abort), and an intermediate pre-commit event. The workflow consisting of the two tasks is illustrated in Figure 1.4. Solid circles represent the states of termination.

Each task is atomic, i.e. it must either execute to completion or not execute at all. The failure of the credit task is allowed but the failure of debit task is not. No work should be committed if debit aborts (fails). Hence the task credit can complete successfully only if the debit task commits. The dotted lines represented the inter task dependencies.

![Fund Transfer Workflow](image)

**Figure 1.4 Fund Transfer Workflow [33]**

The following two dependencies are identified:

- Trigger Dependency: If debit is to start, credit must also start, with debit preceding credit.
- Commit Dependency: Debit must commit before credit is allowed to commit.

The previous sections explain the basics of workflow. The next chapter presents a literature review describing work relevant to our research. Chapter 3 describes of the problem
under study with the help of an example. Chapter 4 contains modular supervisory control architecture in general and as applied to the example workflow from chapter 3. Chapter 5 introduces a decentralized solution to the problems faced by centralized supervisory control. Chapter 6 contains a case study of an Online Bookstore workflow. The online bookstore workflow is modeled using decentralized supervisory control. Chapter 7 contains the conclusion, contributions and future research directions.
Chapter 2. Literature Review

This chapter is a review of the literature we have studied for this work. We take a look at some of the earlier work done in this context by other researchers. A workflow management system consists of the following three components [17]:

- Modeling and specification of the workflow
- Analysis and prototyping of the workflow
- Co-ordination

This thesis concentrates only on the first component of the workflow management system. Modeling and specification of a workflow involves the following steps: 1) Provide formalism for modeling and specification of workflow 2) specify the tasks together with the associated information and 3) enter the applicable business rules in form of inter-task dependencies. The work done by various researchers in the above three areas, is reviewed in the following order:

- Section 2.1- Modeling Formalism
- Section 2.2 – Task
- Section 2.3- Task Structure
- Section 2.4- Inter-task Dependencies

2.1 Modeling Formalism

Researchers have taken different approaches [17,18,31,33] to model workflow. The most significant among them is using Discrete Event Systems (DES) to model and schedule
workflows. A large number of researchers have used Petri nets as a modeling formalism [13,18,30]. The main purpose of workflow management system according to Van Der Aalst [32] is the support of the definition, execution, registration and control of processes. As the processes are a dominant factor in workflow management, it is important to have a properly established framework to model and analyze workflow processes.

The fact that the Petri Nets have formal semantics has led to the following advantages [32]:

- A workflow procedure specified in terms of a Petri net is unambiguous.
- A Petri net description of a workflow can serve as a contract between sub-departments.
- The interpretation of a Petri-net-based workflow procedure does not change when a new version of the WFMS is released.
- The workflow primitives identified by the Work Flow Management Coalition (WFMC) can be easily mapped on to Petri nets.

Formal modeling using state charts [34] has also been proposed. Some of the other formal modeling techniques include the ‘Object Oriented approach [5], modified version of the Entity-Relationship (ER) model [7], and the transaction model [2]. Many researchers have used slight variations of the classical Petri net such as Workflow Nets [32] and Information Control Nets [12]. Recently, Khemuka [15] has used Finite State Automata as modeling formalism.

2.2 Task

Tasks are the building blocks of a workflow and various applications have different interpretations of tasks based on the environment in which the workflow is based. Some of the commonly used definitions of task are presented below:

According to van der Aalst [28] a task stands for work required to reach an objective. A task represents the basic unit of computation within an instance of the workflow enactment...
process [35]. A task can either be transactional or non-transactional in nature. Each of these categories can further be divided based on whether the task is an application or a user-oriented task [17, 35]. Application tasks are typically computer programs or scripts that would be complex in nature. A user task requires a human performing certain actions that might entail interaction with a GUI-capable terminal.

Traditional transactions had a single flat task structure. Many of the Petri net-based architectures [29, 30, 31, 32] represent the task as a single unit, which is considered atomic. However all the researchers have now recognized that a rich internal structure is necessary for a task, if it has to adequately represent real world workflow applications.

2.3 Task Structure

Van der Aalst [26] makes a note about having a task structure to model the task instead of a transition. The idea of having a task structure to model a task instead of a single transition helps in description of the internal behavior of a task, which may be necessary while modeling complex workflow problems. The modeling of a task as a task structure also captures the behavior of the task between the start of a task and the completion, during which certain undesirable situation might arise such as the failure of the system or lack of required information. The idea of task structure was further developed and used by many researchers like Adam et al [1], Rusinkiewicz and Sheth [22], Worah and Sheth [35].

In [17] Krishna Kumar and Sheth discuss specification of workflows that involve heterogeneous tasks. The execution behavior of each task is represented using task structures. Different kinds of task structures are proposed for different kinds of tasks. The authors make use of three task structures namely, transactional task structure (characterizes application tasks i.e. tasks that do not require humans), non-transactional task structure (characterizes user tasks i.e. tasks that involve humans), and open 2-Phase-Commit (2-PC) transaction structure (characterizes
transactions supported by database management systems). The three structures differ in their termination states as well their inherent properties.

Rusinkiewicz and Sheth [22] state that the fundamental problem with many transaction models is that they have a predefined set of properties that may or may not be required by the semantics of a particular activity. The structure of the task should be chosen based on the behavior or capabilities of its processing entity, which underlines the fact that the tasks can have different internal task structure depending upon the type of activities and the type of processing entities.

The differentiation of task structures can also be based on whether the events in the structure are controllable or uncontrollable. Some researchers have taken the approach that all events in a task are controllable [1,24,34].

Adam et al [1] present a PN based framework for modeling and analyzing workflow. They decompose the task into a number of primitives such as Begin, Commit and Abort and also the states between these primitives. This approach has allowed them to model the workflow as an ordinary Petri net thereby facilitating the use of already existing and available analysis tools and techniques for analyzing the workflow. The task structure used by Adam et al [1] is a 2PC where all the events are controllable.

![Figure 2.1 Petri Net Representation of a Task [1]](image-url)
In Figure 2.1 it is seen that there are two transitions leading to the abort state. The ‘abort’ transition that originates from the ‘executing’ state and reaches abort state represents the event that a task can abort during its operation due to failure of its processing entity. But once the task reaches done state, it can either commit or be forced to abort due to failures of mechanisms other than the processing entity.

In [15] the 2-PC task structure is used to represent a task. Every task is modeled as Finite State Automaton and the workflow is represented as a combination of the individual task automata.

### 2.4 Inter Task Dependencies

According to Van der Aalst [32] the minimum capacity any workflow specification language should have is the ability to capture moments of choice, sequential composition, parallel execution and synchronization. These task dependencies are captured by task structures.

Klein [16] and Attie et al. [4] specify inter task dependencies as constraints on the occurrence and temporal order of certain significant events.

\[ e_1 \rightarrow e_2: \text{If } e_1 \text{ occurs then } e_2 \text{ must occur. There is no implied ordering on the occurrences of the two events.} \]

\[ e_1 < e_2: \text{If } e_1 \text{ and } e_2 \text{ both occur then } e_1 \text{ must precede } e_2. \]

Tang et al. [25] have used the same theory for their work. Rusinkiewicz and Sheth [22] have defined inter-task dependencies to represent preconditions required for the execution of tasks. The sources of preconditions of a task can be 1) execution state of other task, 2) Output values of other tasks, and 3) External variables.

Rusinkiewicz and Sheth [22] have categorized these inter-task dependencies into three broad categories: **Control Flow Dependencies**, **Value Dependencies** and **External Dependencies**. The control flow dependencies can then be further categorized into causal type (dependency due
to a cause) and precedence type (dependency due to a previously decided order). Adam et al [24] also use the same dependency classification that is described below:

- **Strong Causal Type**: This type of a dependency generally implies a logical relationship \( st_i \Leftarrow st_j \). It can be said that the strong causal dependency specifies the necessary condition of a relation. If such a dependency exists between two tasks \( tw_i \) and \( tw_j \) then it implies that \( tw_j \) can enter state \( st_j \) only if \( tw_i \) enters state \( st_i \). This dependency generates an incompatible set \{ \( st'_i \), \( st_j \) \}

- **Weak Causal Type**: This dependency implies a logical relationship \( st_i \Rightarrow st_j \). This dependency specifies the sufficient condition for the relationship. The dependency can be interpreted as \( tw_i \) must enter state \( st_j \) if \( tw_i \) enters state \( st_i \). The incompatible set generated by a weak causal dependency is \{ \( st_i \), \( st'_j \) \}

- **Precedence Type**: This dependency enforces a condition like: \( tw_i \) must enter state \( st_i \) before \( tw_j \) enters state \( st_j \), if both \( st_i \) and \( st_j \) are to occur. The precedence type dependency does not imply any logical relationship and hence does not generate an incompatible set

The WFMC has provided a comprehensive list of the control flow dependencies. These dependencies are listed in Appendix 1. Based on the work done by various researchers like Adam et al [1], Van der Aalst [27][26], Attie et al [4], workflow can be modeled as discrete event systems (DES). This thesis work also considers workflow as DES and extends the work done by [15], wherein the dependencies are modeled as Finite State Automata specifications and the enforcement of dependencies is done using *Ramadge and Wonham Supervisory Control Theory*. 
Chapter 3. Problem Description

We aim to design a supervisory control architecture that ensures the safe and satisfactory termination of the workflows. Khemuka [15] described a centralized approach to control of workflows where the tasks and the dependencies between these tasks are represented as Finite State Automata. The enforcement of these dependencies is accomplished based on supervisory control theory proposed by Ramadge and Wonham [21]. Before the description of the problem, it is necessary to present some of the basics of Finite State Automata (FSA).

3.1 Finite State Automata

Automata can be described as the most basic class of Discrete Event System (DES) models. An automaton is a device that is capable of representing a language according to well-defined rules. Automata are used as a modeling formalism since they are easy to use, intuitive, amenable to all the unary and composition operations, and easy to analyze.

A DES is formally modeled by a 5-tuple $G= (Q, \Sigma, f, q_0, Q_m)$ where:

- $Q$ is the set of states $q$
- $\Sigma$ is the set of events
- $f : \Sigma \times Q \rightarrow Q$ is the transition function
- $q_0$ is the initial state
- $Q_m$ is the set of marked (or final) states, $Q_m \subseteq Q$

$G$ is interpreted as a device that starts in $q_0$ and executes state transitions by following its transition function [21]. The words $G$ and Generator are used to describe the automaton that
generates the languages of interest. Consider the finite state automata example represented by its state transition diagram in Figure 3.1.

\[ G = (Q, \Sigma, f, q_0, Q_m) \]

\( Q = \{y, z, x\}; \) the states of the system.

\( \Sigma = \{t, u, v\}; \) the events in the system.

The transition function at each state is

\[ f(u, y) = z \]

\[ f(v, y) = y \]

\[ f(u, z) = f(t, z) = x \]

\[ f(v, z) = z \]

\[ f(t, x) = y. \]

initial state \( q_0 = y; \)

marked (final) state(s) \( Q_m = \{z\}. \)
3.2 Languages

When we talk of an automata G, two languages are of interest; the language generated by G denoted by \( L(G) \), and the language marked by G denoted by \( L_m(G) \). The language generated by G is \( L(G) = \{ \omega : \omega \in \Sigma^* \text{ and } f(\omega, q_0) \text{ defined } \} \). The language generated by automaton G can be interpreted as the set of all the sequences of events that take the system from initial state to some reachable state in G. This language represents the directed paths from the initial state through which some state can be reached. \( L(G) \) is prefix-closed by definition, since a path is possible only if all its prefixes are also possible.

The language marked by automaton G can be interpreted as the set of all the strings that take the system from its initial state to some marked state i.e. final state or a state of satisfactory completion. \( L_m(G) = \{ \omega : \omega \in L(G) \text{ and } f(\omega, q_0) \in Q_m \} \). The marked language is a subset of the generated language consisting of only those strings from \( L(G) \) that trace a path from the initial state to the final state. \( L_m(G) \) need not be prefix-closed since all the states of Q (state set) need not be marked.

![Figure 3.2 Example](image)

In the automaton shown in Figure 3.2 ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, and ‘f’ are the events; 1 to 5 are the states with 1 being the initial state and 4 the marked or final state. Following the definitions for the two languages given earlier we have,

\( L(G) = \{ a, ac, acf, ae, b, bd, bdf \} \) and

\( L_m(G) = \{ ae, acf, bdf \} \).
3.2.1 Concatenation

A string is in \( L_aL_b \), if it can be written as the concatenation of a string in \( L_a \) with a string in \( L_b \) [33]. Let \( L_a, L_b \subseteq E^* \), then \( L_aL_b = \{ s \in E^* : (s = s_as_b) \text{ and } (s_a \in L_a) \text{ and } (s_b \in L_b) \} \).

3.2.2 Prefix-Closure

The prefix closure of \( L \) is the language denoted by \( \overline{L} \), consisting of all the prefixes of all the strings in \( L \). In general \( L \subseteq \overline{L} \). \( L \) is said to be prefix closed if \( L = \overline{L} \). Thus language \( L \) is prefix-closed if any prefix of any string in \( L \) is also an element of \( L \).

3.2.3 Accessible States

The set of all the states that can be reached from the initial state is called the accessible states subset [14]. \( Q_a \) denotes the accessible states subset, and is described as:

\[
Q_a = \left\{ q \in Q \left| \exists \omega \in \Sigma^* \delta(\omega, q_0) = q \right. \right\}.
\]

In Example 2 in Figure 3.2, \( Q_a = \{1, 2, 3, 4, 5\} \).

3.2.4 Co-Accessible States

The set of all the states \( q \) from which some marked state can be reached is called the co-accessible states subset [14]. The co-accessible states subset is denoted by \( Q_{ca} \), where,

\[
Q_{ca} = \left\{ q \in Q \left| \exists \omega \in \Sigma^* \delta(\omega, q) \in Q_m \right. \right\}.
\]

In Example 2 in Figure 3.2, \( Q_{ca} = \{1, 2, 3, 4, 5\} \).

3.2.5 Shuffle Product

The shuffle product of two automata is the concurrent behavior of the two automata [8]. It can also be viewed as the Cartesian product of the two automata. Formally:

Automaton 1: \( G_1 = (Q_1, \Sigma_1, f, q_{01}, Q_{m1}) \)
Automaton 2: \( G_2 = (Q_2, \Sigma_2, f, q_{02}, Q_{m2}) \)

Shuffle Product: \( G_1 \parallel G_2 = G = Ac (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, f, (q_{01}, q_{02}), Q_{m1} \times Q_{m2}) \)

3.2.6 Blocking

An automaton \( G \) is said to be blocking if \( \overline{L_m}(G) \subset L(G) \) and non-blocking when \( \overline{L_m}(G) = L(G) \) \([8]\). This implies that for every string \( \omega \in L(G) \) there is at least one string \( s \) such that \( \omega s \in L_m(G) \). In other words, an automaton is non-blocking if every string starting from the initial state can be completed to some string that leads to a marked state. Based on the definitions of prefixes and marked languages in Example 2 in Figure 3.2, we have

\[
\overline{L_m}(G) = \{a, ac, ae, acf, b, bd, bdf\} \quad \text{and} \quad L(G) = \{a, ac, ae, acf, b, bd, bdf\}
\]

which are equal. Hence we say that the automaton in Figure 3.2 is non-blocking. In this particular example, the magnitude of the problem being small, it is easily seen that there exists a path from all the states to a marked state.

An automaton \( G \) could reach a state \( q \) where \( f(q, \sigma) = \emptyset \ \forall \sigma \in \Sigma, \) and \( q \not\in Q_m \). This situation is referred to as a deadlock since the system has not reached a final state and no further event can be executed. If deadlock happens then the aforementioned condition for blocking is satisfied, because there is some string \( \omega \) in \( L(G) \) that cannot be completed to a string which is a part of \( L_m(G) \). Figure 3.3 shows an example of deadlock and blocking.
Also, an automaton can reach a set of unmarked states that are strongly connected. This means that the states are reachable from one another but there is no transition going out of the set. In such a case there is always at least one transition that can be executed but it can never reach any of the marked states. This situation is called a \textit{livelock}. Hence it can be said that if automaton G is blocking, livelocks and deadlocks can occur.

3.2.7 Trim Generator

A generator (automaton) G is said to be trim, when it is accessible as well as co-accessible, i.e. $Q = Q_a$ and $Q = Q_{ca}$ [21]. Hence for a trim generator $Q = Q_a = Q_{ca}$, which implies that every state that is reachable from initial state by some path, also has a path from itself to a marked state. A trim generator is non-blocking by definition [3]. The automaton in Figure 3.2 is trim.

3.2.8 Non-Conflicting Languages

Languages $L_1$ and $L_2$ are non-conflicting if, whenever they share a prefix, they also share a word containing the prefix, i.e. $\overline{L_1} \cap \overline{L_2} = \overline{L_1} \cap \overline{L_2}$ [21].

3.2.9 Controllability

The event set $\Sigma$ is divided into two sets, \textit{controllable events set} ($\Sigma_c$) and \textit{uncontrollable events set} ($\Sigma_u$). The examples of uncontrollable events are resource failures or the completion of a process, and a controllable event can be the initiation of a process.

A language $K$ is controllable with respect to $G$, if $K \cap L(G) \subseteq K$ [8]. This means that given a string $\omega$, which is a prefix of $K$, if we add an uncontrollable event $\sigma \in \Sigma_u$, the word $\omega \sigma$ is a
sequence of events in $G$. If adding the event $\sigma$ causes the string to exit from the prefix closure $\overline{K}$, then $K$ is not controllable since $\sigma$ is an uncontrollable event. This is illustrated in Figure 3.4.

![Figure 3.4 Controllability](image)

3.2.10 Supremal Controllable Sublanguage

When a given language $L$ is uncontrollable, it is useful to find the supremal controllable sublanguage $L^\uparrow c$ of $L$. The supremal controllable sublanguage $L^\uparrow c$ is the unique largest controllable sublanguage of $L$ [21]. An iterative procedure for determination of the supremal controllable sublanguage from a given automata is explained later in this document.

3.3 Supervisory Control Theory

The behavior represented by a DES modeled by a finite state automaton $G$ may not be satisfactory under all conditions. This unsatisfactory behavior is modified by an external controller by restricting the behavior of $G$ represented by $L(G)$, to a subset of $L(G)$ [8]. This subset of $L(G)$ is the satisfactory behavior and is also called the admissible language $L_a$. The modifications are based on specifications, which are rules that define the system.

A *supervisor* is an agent that disables controllable events such that the behavior of $G$ conforms to the specifications. The supervisor and the process are coupled to form a closed loop system as shown in Figure 3.5.
This feedback mechanism between $S$ and $G$ functions in following way: Let’s assume that the uncontrolled behavior $G$ is in state $q_i$ and the supervisor is in state $x_j$ at a given time. A subset of events $\sigma \in \Sigma$ can occur in the uncontrolled behavior $G$ in state $q_i$. According to state $x_j$ only a subset of these events are permitted. The supervisor issues a control pattern $\Psi$ such that some controllable events are disabled if they take the system to an undesirable state.

Formally the supervisor $S$ consists of a finite automaton $S$ and output function $\bullet$

\[ S = (S, \Psi), \]
\[ S = (X, \Sigma, \delta, x_0, X_m) \]
\[ \Psi: \Sigma \times X \rightarrow \{0: \text{disable}, 1: \text{do not disable}\} \]

s.t. $\Psi(\sigma, x) = 0$ or $1$ if $\sigma \in \Sigma_c$ i.e. $\sigma$ is a controllable event,

$\Psi(\sigma, x) = 1$ if $\sigma \in \Sigma_u$ i.e. $\sigma$ is an uncontrollable event.

When an event $\bullet$ executes, both $S$ and $G$ are updated. The closed loop behavior of $S/G$ is described by the language $L(S/G) = L(G) \cap L(S)$ [3], where $L(S/G)$ consists of the sequence of events of uncontrolled process language that survives under supervision. The marked behavior of $S/G$ is described by the language $L_m(S/G) = L_m(G) \cap L_m(S)$, where $L_m(S/G)$ consists of the sequences of events that are marked by both $G$ and $S$. 

---

Figure 3.5 Supervisor-Uncontrolled Model Feedback System
3.4 Problem Description

This section summarizes the work by Khemuka [15] which motivates the objectives of this thesis and provides the background for the problems addressed in this thesis.

Consider the example discussed in [15, 18], a travel agency that processes requests for airline and hotel reservations. Once the flight reservation is made, it cannot be canceled, whereas cancellation of hotel reservation is allowed. The following two tasks are identified:

- Purchase an airline ticket (Task $T_1$)
- Book a hotel (Task $T_2$)

Based on booking regulations, traveler’s preferences, or economic reasons, the following dependencies are defined:

- Booking of hotel cannot start until purchasing an airline ticket starts (Begin).
- If hotel booking aborts, then purchasing airline ticket must abort too (Abort).

3.4.1 The Uncontrolled Model

Each task $T_i$ in the set of tasks $T$ is a finite state automaton and all the notations used regarding the tasks are consistent with the definitions of DES provided earlier in this discussion. The task is shown in Figure 3.6. Each task starts with a begin event ‘b’ in the initial state (in) and terminates with either a commit event ‘c’ leading to committed state (cm), or an abort event ‘a’ leading into aborted state (ab). Hence cm and ab are the final or marked states. The initial state is marked since it denotes that the task did not begin executing. The marked state will be represented with a solid circle throughout this document. There is a pre-commit event that precedes termination and the event is uncontrollable. Once the task reaches executing state (ex), the task can either get completed and reach done (dn) state or abort state. Thus all the events emanating from ex state are uncontrollable. The supervisor has no control over the uncontrollable events.
Task $T_i$ is defined as

$$T_i = \{ Q_i, \Sigma_i, f_i, q_0, Q_{im} \}$$

Where

$Q_i = \{ \text{in, ex, dn, ab, cm} \}$

$\Sigma_i$ consists of two sets:

$$\Sigma_{U_i} = \text{uncontrollable events} = \{ p, a \}$$

$$\Sigma_{C_i} = \text{controllable events} = \{ b, fa, c \}$$

$f_i (b, \text{in}) = \text{ex}; f_i (p, \text{ex}) = \text{dn}; f_i (a, \text{ex}) = \text{ab}; f_i (fa, \text{dn}) = \text{ab}; f_i (c, \text{dn}) = \text{cm};$

$q_0 = \text{in},$

$Q_{im} = \{ \text{in, ab, cm} \}$

The uncontrolled model shown in Figure 3.7 is the concurrent behavior of the two tasks $T_1$ and $T_2$. The uncontrolled model is obtained by $Shuffle Product$ of the finite state automata models of the individual tasks.
3.4.2 The Specification Model

The second step in the supervisory control of workflows is determining the finite automata model of the specifications, which represent the inter task dependencies. The begin dependency which is of the strong causal type, states that task $t_2$ cannot begin unless task $t_1$ has begun.

\[
\Sigma - (b_1, b_2) \rightarrow \Sigma - (b_2) \rightarrow \Sigma
\]

Figure 3.8 $C_1$-Specification Model for Begin Dependency [15]

In Figure 3.8, the specification model remains in the initial (marked) state when events other then $b_1$ and $b_2$ are executed. Only when $b_2$ follows $b_1$ the system reaches a marked state. If
b₂ executes followed by b₁, the system cannot reach a marked state. This ensures that transition b₂ is executed only after b₁ has already executed.

\[ \sum (a_2, fa_2, a_1, fa_1) \sum (a_1, fa_1) \]

Figure 3.9 C₂-Specification Model for Abort Dependency [15]

Figure 3.9 shows the specification model for the abort dependency. The specification model remains in the marked state if any transition except a₁ and a₂ are executed. However once task T₂ aborts then the specification reaches a marked state only when task T₁ aborts too. The combined specification model C, shown in Figure 3.10, is determined by the shuffle of the two specification automata shown in Figures 3.8 and 3.9.

Figure 3.10. C-Specification Automaton [15]
3.4.3 The Coupled model

The coupled model $C/G$ is obtained by taking the couple product of the uncontrolled model and the specification model. The coupled model generates the language $L(C/G)$ and marks the language $L_m(C/G)$ where $L(C/G) = L(G) \cap L(C)$ and $L_m(C/G) = L_m(G) \cap L_m(C)$.

![Diagram](image)

**Figure 3.11 C/G-Coupled Model for Airline Workflow**

In Figure 3.11, task 2 can execute its begin event only after task 1 has begun. Both the tasks can either commit or abort, but if the hotel booking aborts ($a_2$) then the ticket reservation has to abort (i.e. it cannot commit). Hence the following states are allowed and are marked:

- T₁ initial, T₂ initial (1)
- T₁ aborted, T₂ initial (4)
- T₁ committed, T₂ initial (19)
- T₁ abort, T₂ abort (19)
- T₁ abort, T₂ commit (24)
- T₁ commit, T₂ commit (25)
3.4.4 Identify Admissible language

It is important that a workflow model should be safe and deadlock free. A workflow is safe if it terminates in a compatible state or marked state. A workflow should be executed without running into deadlocks, which requires that $L_a$ is non-blocking. A language $L$ is non-blocking if for any string $\omega \in L$ there is at least one string $s$ such that $\omega s \in L_m$. Therefore, the admissible language $L_a \subseteq L_{am}$ must hold and be non-blocking. Hence the admissible language for workflow should be the marked admissible language $L_{am}$. The admissible language $L_{am}$ is the language that marks the automata $C/G$ i.e. $L_{am} = L_m(C/G)$.

3.4.5 Construct Supervisor $S$

As nonblocking is of concern the supervisor is determined using the Basic Supervisory Control Problem-Nonblocking (BSCP-NB). The solution theorem and proof for the BSCP-NB can be found in [15,8]. According to the BSCP-NB the supervisor should realize the language $L_{am}^{tc}(C/G)$, which is the supremal controllable sublanguage of $L_m(C/G)$. The supremal controllable sublanguage is shown in Figure 3.12.

![Figure 3.12 Supremal Controllable Sublanguage](image-url)
The existence of such a supervisor is guaranteed by the *Non-blocking Controllability* Theorem (NCT) if the two conditions of Controllability and $L_m(G)$-closure are satisfied. The detailed explanation of NCT can be found in [15, 8]. $L_m^{\uparrow C}(C/G)$ is controllable by definition of the supremal controllable sublanguage satisfying condition of controllability. $L_m^{\uparrow C}(C/G)$ is found to be $L_m(G)$ closed [3].

The next step in the supervisory control problem is to determine the control pattern that realizes the supremal controllable sub language $L_m^{\uparrow C}(C/G)$. The control pattern for the workflow, shown in Table 3.1, consists of all the states in the admissible language $L_m^{\uparrow C}(C/G)$ and at every state the events that are allowed in that state are marked 1 and all the events that are disallowed are given a value 0. For the sake of brevity, the states in which no controllable events are defined have not been included in the control pattern in Table 3.1.

<table>
<thead>
<tr>
<th>Events</th>
<th>States</th>
<th>b₁</th>
<th>f₁</th>
<th>c₁</th>
<th>b₂</th>
<th>f₂</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
For example, the Begin dependency states that Hotel Booking (Task 2) cannot begin until the Ticket Reservation (Task 1) begins. In state 1, the ticket reservation has not begun and hence \( b_2 \) which is hotel booking begin is disabled \((\Psi(b_2, 1) = 0)\). In state 2, ticket reservation has already begun and hence hotel booking is allowed to begin \((\Psi(b_2, 2) = 1)\). Also according to Abort dependency if the hotel booking aborts then the ticket reservation has to abort. Hence the ticket reservation is not allowed to commit once the uncontrolled model reaches state 18, where airline ticket reservation is in done state and the hotel booking has aborted, i.e. \((\Psi(c_1, 18) = 0)\). The events that have a ‘-‘ in its cell indicate that the events are not defined in that state. State 1 where both Airline ticket reservation and hotel booking are in the initial state, is marked as it does it not violate the dependencies. Similarly the states 4 and 5 do not violate the dependencies and hence are marked.

### 3.5 Motivation

The workflows in real world may involve more than a few tasks and also a large number of inter task dependencies. The combined specification model of the dependencies is the shuffle of all the individual specifications. In the airline example the shuffle of the two dependencies gives us a structure with \(5^2\) i.e. 25 states and 87 events. Table 3.2 shows the estimated number of states in the specification models for workflows with the given number of tasks.

<table>
<thead>
<tr>
<th>Number of Inter-task dependencies</th>
<th>States in specification model</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(5^3 = 125)</td>
</tr>
<tr>
<td>4</td>
<td>(5^4 = 625)</td>
</tr>
<tr>
<td>5</td>
<td>(5^5 = 3125)</td>
</tr>
<tr>
<td>10</td>
<td>(5^{10} = 9765625)</td>
</tr>
</tbody>
</table>
Thus it can be seen that the number of states in the specification model increases exponentially with increase in the number of inter-task dependencies. This problem of state space explosion will be addressed in the next chapter where modular supervisory control architecture is explained and applied to the airline example.

The uncontrolled model of the workflow is the shuffle product of various tasks constituting the workflow. In Figure 3.10, we can see that for a two-task workflow the uncontrolled model has 25 states. The number of states in uncontrolled models depends on the number of tasks in the workflow. Table 3.3 shows a comparison of the number of states in the uncontrolled model based on the number of tasks in workflow. From Table 3.3 we can see that with increase in the number of tasks in the workflow, the number of states in the uncontrolled model grows exponentially as well.

<table>
<thead>
<tr>
<th>Number of tasks</th>
<th>Number of states in the uncontrolled model</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$2^5 = 32$</td>
</tr>
<tr>
<td>5</td>
<td>$5^5 = 3125$</td>
</tr>
<tr>
<td>10</td>
<td>$5^{10} = 9765625$</td>
</tr>
<tr>
<td>25</td>
<td>$5^{25} = 298023223876953125$</td>
</tr>
</tbody>
</table>

Workflows containing more than a few tasks and dependencies become impossible to represent and control. From the above example it is clear that a centralized supervisory control approach can suffer from exponentially increasing computational complexity when applied to real life workflows.
3.6 Objectives

This research attempts to realize the following objectives:

- Design modular supervisors for workflows to reduce the computational complexity arising due to the increase in state space of the specification models in centralized supervisory control approach

- Show that the proposed modular controllers designed are controllable and nonblocking - i.e. workflow under such a modular control would always terminate in a safe and satisfactory state

- Develop a decentralized supervisory control architecture that divides the workflow into easily manageable modules to reduce the computational complexity arising due to increase in state space of the uncontrolled model of the workflow in centralized supervisory control approach

- Show that the proposed decentralized controllers are controllable and nonblocking

- Combine the modular and decentralized supervisory control approaches to design a distributed supervisory control architecture
Chapter 4. Modular Supervisory Control

Modular supervisory control is introduced as a solution to the problem of state space increase faced by the centralized supervisory control [21]. In modular control, the control action of the supervisor is given by combination of the control action of two or more supervisors. Consider the case of two supervisors $S_1$ and $S_2$ each defined for $G$, the modular supervisor $S_{mod}$ is determined as $S_{mod} = S_1(S, \Psi_1) \cdot S_2(S, \Psi_2)$ [8].

It is sufficient that an event be disabled by one of the supervisors for that event to be effectively disabled by the modular control supervisor. The modular control architecture [8] is depicted in Figure 4.1.

![Figure 4.1 Modular Supervisory Control Architecture](image)

**Figure 4.1 Modular Supervisory Control Architecture**

### 4.1 Modular Supervisory Control Problem (MSCP)

This section defines the general problem of modular supervisory control and presents the solution to it.
4.1.1 Modular Supervisory Control Problem

Given a DES G with event set E, uncontrollable event set \( E_{uc} \subseteq E \), and admissible language

\[
L_a = L_{a1} \cap L_{a2} \ldots \cap L_{an}
\]

Where

\[
L_{ai} = \overline{L_{ai}} \subseteq L(G) \text{ for } i=1,2,\ldots,n.
\]

Find a modular supervisor \( S_{mod} \) (according to the architecture in Figure 4.1) such that

\[
L(S_{mod}/G) = L^+_a
\]

4.1.2 Modular Supervisory Control Solution

Build realizations \( R_i \) of \( S_i \) such that

\[
L(S_i/G) = L^+_a \text{ for } i = 1, \ldots , n
\]

\( S_{mod} \) is the modular supervisor, where

\[
S_{mod}(s) := S_{mod1}(s) \cap S_{mod2}(s) \cap \ldots \cap S_{modn}(s)
\]

With this choice of modular supervisor \([8]\) \( S_{mod} \)

\[
L(S_{mod}/G) = L^+_a \cap L^+_a \cap \ldots \cap L^+_a
\]

\[
= (L_{a1} \cap L_{a2} \cap \ldots \cap L_{an})^+_c
\]

\[
= L^+_a
\]

4.1.3 MSCP (Non-blocking Case)

When non-blocking is of concern along with the MSCP, then it is also necessary that the supremal controllable sublanguages for the admissible languages should be non-conflicting. This condition of non-conflicting ensures that the resultant supervisor is nonblocking \([21]\).
Theorem: Let $S_i$, $i = 1, 2, \ldots, n$ be individual nonblocking supervisors for $G$. Then $S_{\text{mod}1 \ldots n}$ is nonblocking if and only if $L_m(S_1/G)$, $L_m(S_2/G)$, \ldots and $L_m(S_n/G)$ are non-conflicting languages, i.e., if and only if

$$L_m(S_1/G) \cap L_m(S_2/G) \cap \cdots \cap L_m(S_n/G)$$

For the proof to this theorem refer [8].

4.2 Modular Supervisory control of workflows

Workflows in real applications are large and involve the enforcement of several dependencies together on the system. In other words, a constraint is often the intersection of two or more languages and can often be described, as the system should satisfy a property of one kind as well as a property of another kind. This process results in an exponential state space increase as explained in section 3.8.

Modular supervisory control reduces the computational complexity in the construction of the supervisor. If the specification language $L$ for the basic supervisory control problem is given as the intersection of two prefix-closed languages $L_1$ and $L_2$, using modular control theory we synthesize $S_1$ and $S_2$ and use these two supervisors in conjunction. Using this modular approach the total (worst case) computational complexity for supervisor synthesis is reduced from $O(n_1n_2m)$ to $O(\max(n_1, n_2) m)$ where $n_1$ is the number of states in specification automata 1 ($C_1$), $n_2$ is the number of states in specification automata 2 ($C_2$), and $m$ is the number of states in uncontrolled model $G$ [8].

4.3 Applying modular control to Airline example

Applying modular supervisory control to the airline workflow example involves three steps:

- Construct specification automata for the dependencies
• Obtain the coupled model and supervisors for the dependencies individually
• Obtain conjunction of the supervisors to get the final supervisory controller

The shuffle product of individual tasks, as explained in section 3.5.1, determines the uncontrolled model. The uncontrolled model for a two-task workflow is the same as in Figure 3.7. The specifications of the two dependencies, ‘Begin’ (Strong Causal) and Abort (Weak Causal) are shown in Figures 3.8 and 3.9. The coupled models for these specification models are obtained by taking the couple product of each of the specification model and the uncontrolled model.

4.3.1 Coupled Model for Begin (Strong Causal) Dependency

In Figure 4.2 it can be seen that event b_2 from state 1 is disabled as the dependency states that task 2 cannot begin until task 1 begins. Hence b_2 is enabled only from state 2 where b_1 has taken place. In state 1 both the airline ticket reservation and the hotel booking are in the initial state and hence the state is marked. Once the begin condition is satisfied the coupled model can end in any of the six states, namely, 4, 5, 19, 20, 24 and 25. Hence all six states are marked.
4.3.2 Admissible Behavior

For the workflow to be safe and deadlock free we require that admissible behavior should be non-blocking. Hence we choose marked language $L_{am1}$ as the admissible language. In this case $L_{am1}$ is the language that marks the automaton $C_i/G$. Hence we say that $L_{am1} = L_m(C_i/G)$.

4.3.3 Construct Supervisor $S_1$

Since non-blocking is of concern we identify the problem as MSCP-NB and choose a supervisor $S_1$ for the begin dependency, based on the theorems detailed in sections 4.1.1 and 4.1.2. The supremal controllable sublanguage $L_m^c(C_i/G)$, for the begin dependency is the same as the coupled model for the begin dependency shown in Figure 4.2. The supremal controllable sublanguage is determined using the algorithm explained in Appendix 2.

The next step in the supervisory control pattern is to construct the control pattern for the supervisor $S_1$. The control pattern $\Psi_1$ for the begin dependency is shown in Table 4.1. The control pattern for the begin dependency realizes $L_m^c(C_i/G)$.

<table>
<thead>
<tr>
<th>Events</th>
<th>$b_1$</th>
<th>$fa_1$</th>
<th>$c_1$</th>
<th>$b_2$</th>
<th>$fa_2$</th>
<th>$c_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The uncontrollable events cannot be disabled and are not included in the Table 4.1. The events with the ‘-’ mark mean that those events are not defined in that state. The begin dependency states that the hotel booking cannot begin until the ticket reservation begins. In state 1, the ticket reservation has not begun and hence \( b_2 \) i.e. hotel booking begin is disabled

\[ (\Psi_1 (b_2, 1) = 0). \]

However in state 2, since the ticket reservation has begun, the hotel booking can begin too, hence \( (\Psi_1 (b_2, 2) = 1). \)

### 4.3.4 Coupled Model for Abort (Weak Causal) Dependency

In Figure 4.3 state 19 can be reached when both ticket purchase and hotel booking abort. This state is exactly what the dependency states and is acceptable and hence is marked. In state 20, hotel booking aborts but purchase airline ticket commits. According to the dependency this state is not acceptable and hence is unmarked. State 25 is obtained when both ticket and hotel commit, and state ‘24’ where hotel booking commits but ticket purchase aborts. Both these states are acceptable and hence marked. Similarly state 4, where airline ticket reservation has aborted and hotel booking is yet to begin, and state 5 where ticket reservation has committed and hotel booking is yet to begin, are acceptable and are marked.

![Coupled Model for Abort Dependency](image)

**Figure 4.3 Coupled Model for Abort Dependency**
4.3.5 Admissible Behavior

For the workflow to be safe and deadlock free, we require that admissible behavior should be non-blocking. Hence we choose marked language \( L_{am2} \) as the admissible language. In this case \( L_{am2} \) is the language that marks the automaton \( C_2/G \). Hence we say that \( L_{am2} = L_m(C_2/G) \).

4.3.6 Construct Supervisor \( S_2 \)

Similar to the begin dependency we model the problem as MSCP-NB and choose a supervisor \( S_2 \) for the abort dependency, such that \( L_m(S_2/G) = L_{am2}^c = L_m^c(C_2/G) \), based on the theorem in sections 4.1.1 and 4.1.2. The supremal controllable sublanguage \( L_m^c(C_2/G) \) shown in Figure 4.4 is determined using the algorithm explained in Appendix 2. If we examine the Figure 4.3 we can see that due to the presence of uncontrollable events \( p_1, p_2, a_1, \) and \( a_2 \) the strings in \( L_m(C_2/G) \) may lead to an unmarked state. For example the \( b_2 \) event executing from state 5 is in \( L_m(C_2/G) \), and it leads to state 10. But from state 10 there is an uncontrollable \( a_2 \) that can lead to an unmarked state 20. The supervisor cannot disable the \( a_2 \) event once the workflow reaches state 10. Hence to prevent the workflow from reaching an unmarked state, the \( b_2 \) event, which is controllable, should be disabled. The same procedure is repeated for all such strings that lead to an unmarked state, i.e. strings that lead the automata out of \( L_m(C_2/G) \). The resultant structure is shown in Figure 4.4. The control pattern for the abort dependency is shown in Table 4.2. The control pattern realizes the supremal controllable sublanguage \( L_m^c(C_2/G) \).
Figure 4.4 Supremal Controllable Sub-language for Abort Dependency

Table 4.2 Control Pattern $\Psi_2$ for Abort Dependency

<table>
<thead>
<tr>
<th>Events</th>
<th>b1</th>
<th>fa₁</th>
<th>c₁</th>
<th>b₂</th>
<th>fa₂</th>
<th>c₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
The abort dependency states that if the hotel booking aborts then the ticket reservation must abort too. In state 18 the hotel booking is in aborted state and hence the ticket reservation is not allowed to commit i.e. ($\Psi_2 (c_1, 18) = 0$).

4.3.7 Control Pattern $\Psi$ for the Workflow

The resultant control in the form of control pattern $\Psi$, shown in Table 4.3, is obtained as a conjunction of the two sub-controllers. The resultant controller disables any event that is disabled by any one of the controllers. For example, the booking of a hotel is allowed to begin in state 1 according to the control pattern for the Abort dependency ($\Psi_2 (b_2, 1) = 1$). However since the ticket reservation has not begun, the hotel booking is not allowed to begin in state 1 of control pattern for the Begin dependency ($\Psi_1 (b_2, 1) = 0$). Hence the resultant control pattern disables the event ($\Psi (b_2, 1) = 0$).

In state 18, in the control pattern for the abort dependency, the hotel booking has aborted, so to satisfy the abort dependency the ticket reservation is not allowed to commit i.e. ($\Psi_2 (c_3, 18) = 0$). Hence although the airline ticket reservation is allowed to commit from the corresponding state in the control pattern for the Begin dependency, the resultant control pattern $\Psi$ does not allow it ($\Psi (c_3, 18) = 0$).

Table 4.3 Control Pattern $\Psi$ for the Airline Workflow

<table>
<thead>
<tr>
<th>Events*</th>
<th>B1</th>
<th>f_{a_1}</th>
<th>c_{1}</th>
<th>b_{2}</th>
<th>f_{a_2}</th>
<th>c_{2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>States*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
4.4 Nonblocking Modular Control in Consistent Workflows

For the resultant supervisor to be nonblocking, the individual supervisors should be nonblocking and the admissible languages should be non-conflicting. According to the definition of non-conflicting languages, if the admissible languages share a prefix (a sequence of events that have not completed the tasks in the workflow), then there must exist a word (a sequence of events that complete the tasks) that contains that prefix. Otherwise, the workflow is said to have an inconsistency. Two cases are shown to prove that the admissible languages are non-conflicting in case of workflows.

Case 1: $\Sigma_1 \cap \Sigma_2 = 0$, In this case the dependencies modeled do not have a common task. Since there is no common prefix, no conflict can arise.

Case 2: $\Sigma_1 \cap \Sigma_2 \neq 0$, this means that there are strings shared by both specifications or the dependencies have common tasks. Assuming that there are no inconsistencies, there must exist a word common to both $K_1 = L^T(C_1 / G)$ and $K_2 = L^T(C_2 / G)$ as well as $L^T(C_1 / G)$ and $L^T(C_2 / G)$. So we have to show that

$$L^T_m(C_1 / G) \cap L^T_m(C_2 / G) \supseteq L^T_m(C_1 / G) \cap L^T_m(C_2 / G)$$

i.e.

$$K_1 \cap K_2 \supseteq K_1 \cap K_2$$

Let $s \in K_1 \cap K_2$, then $s$ is the sequence of events that denotes the completion of a set of tasks within the specification given. Then it must also be true that $s \in K_1$ and $s \in K_2$. Let $w$ be
any prefix of the word $s$, then $w \in \overline{K_1 \cap K_2}$, $w \in \overline{K_1}$ and $w \in \overline{K_2}$, else $s$ would not be in $K_1 \cap K_2$ or $K_1$ or $K_2$.

For both Case 1 and Case 2, the admissible languages are nonblocking. Hence, we can say that in case of workflows the modular supervisor is nonblocking.
Chapter 5. Decentralized Supervisory Control

In this chapter the general decentralized supervisory control problem is defined, and the conditions that guarantee the existence of a decentralized solution are described. Decentralized control represents the situation where there are several control stations that are jointly controlling a given system that is inherently distributed [21]. Although most current workflow schedulers are centralized the workflow environments are mostly distributed. In such cases it is appropriate to have modular supervisors installed at such distributed locations and the resultant supervisory controller is the conjunction of all such individual supervisors. In such an architecture, the remote supervisors are not able to exert control over events occurring at other remote locations. This process is illustrated in Figure 5.1 that represents the common architecture for a decentralized supervisory control system.

![Figure 5.1 Decentralized Supervisory Control [8]](image)

In the Figure 5.1 associated with each supervisor $S_i$, there is a projection $P_i$. Projections restrict events observed by the supervisor to a subset $\Sigma_{io}$ of the event set $\Sigma$ of the system. Also the
supervisor \( S_i \) only has control over \( \Sigma_{ic} \). Such supervisors are called *partial observation supervisors.*

### 5.1 Projections

Associated with every partial observation supervisor is an event set that the supervisor cannot observe. A projection associated with a supervisor erases the events that belong to the unobservable event set from the strings that are input to the supervisor. Thus projection of a string can be interpreted as the supervisor’s view of the complete string.

Given an automaton \( G = (Q, \Sigma, f, q_0, Q_m) \) and a projection \( P_i \) associated with supervisor \( S_i \), then

\[
P_i(\sigma) = \sigma, \text{ if } \sigma \in \Sigma_o
\]

\[
P_i(\sigma) = \varepsilon, \text{ if } \sigma \in \Sigma - \Sigma_o
\]

where \( \Sigma_o \) is the set of observable events

\( \Sigma - \Sigma_o \) is the set of unobservable events.

### 5.2 General Decentralized Supervisory Control Problem

The general decentralized supervisory control problem can be defined in the following way. We have a set of \( n \) partial observation supervisors, each associated with a different projection \( P_i, i = 1, \ldots, n \), jointly controlling the given system \( G \) with event set \( \cdot \). The task is to define \( S_i(\sigma) \) for \( \sigma \in L(G) \) such that

\[
S_i(\sigma) = S_{pi}[P_i(\sigma)].
\]

The net control action on \( G \) will be the intersection of the sets of events enabled by each supervisor i.e.

\[
S_{dec}(\sigma) = \bigcap_{i=1}^{n} S_i(\sigma)
\]
The resulting controlled behavior is described by the languages $L(S_{\text{dec}}/G)$ and $L_m(S_{\text{dec}}/G)$. The existence of such a decentralized supervisor is guaranteed by Controllability and Co-Observability Theorem described in the next sections.

5.3 Controllability and Co-Observability Theorem (CCOT)

The controllability and co-observability theorem defines the conditions for the existence of a decentralized solution for the given supervisory control problem. Consider DES $G = (Q, \Sigma, f, q_0, Q_m)$, where $\Sigma_{uc} \subseteq \Sigma$ is the set of uncontrollable events, $\Sigma_c = \Sigma \setminus \Sigma_{uc}$ is the set of uncontrollable events, and $\Sigma_o \subseteq \Sigma$ is the set of observable events. For each site $i$, $i = 1, \ldots, n$, consider the set of controllable events $\Sigma_{i,c}$, and the set of observable events $\Sigma_{i,o}$; overall, $\bigcup_{i=1}^n \Sigma_{i,c} = \Sigma_c$ and $\bigcup_{i=1}^n \Sigma_{i,o} = \Sigma_o$. Let $P_i$ be the natural projection from $\Sigma$ to $\Sigma_{i,o}$, $i = 1, \ldots, n$. Consider also the language $L_m(C/G) \subseteq L_m(G)$, where $L_m(C/G) \neq \emptyset$ and $C$ is the specification. There exists a nonblocking decentralized supervisor $S_{\text{dec}}$ for $G$ such that

$$L_m (S_{\text{dec}}/G) = L_m^C (C / G) \text{ and } L (S_{\text{dec}}/G) = \overline{L}_m^C (C / G)$$

if and only if these three conditions hold [11]:

- $L_m^C (C / G)$ is controllable with respect to $L(G)$ and $\Sigma_{uc}$
- $L_m^C (C / G)$ is $L_m(G)$ closed
- $L_m^C (C / G)$ is co-observable with respect to $L(G)$, $P_i$, and $\Sigma_{i,c}$, $i = 1, \ldots, n$

Proof: the proof of this theorem can be found in [8].

5.4 Decentralized Supervisory Control of Workflows

In this section we describe the local uncontrolled models, specifications and existence of a decentralized solution for workflow.
5.4.1 The Uncontrolled Model

A local supervisor is defined for every inter task dependency. We also define a local event set \( \Sigma_{k_0} \) for each supervisor \( S_k \), which denotes the events observed by the supervisor. We define projections based on inter task dependencies. The effect of a projection \( P_k \) on a string is to erase the events that do not belong to the tasks associated with that dependency. Given a uncontrolled behavior \( G \) represented by \( L(G) \), \( P_k (L(G)) \) is interpreted as the supervisors view of the uncontrolled model \textit{Local Uncontrolled Model}, where \( k \) represents the dependency.

Thus

\[ P_k (\sigma) = \sigma, \text{ if } \sigma \in \Sigma_k \]

\[ P_k (\sigma) = \varepsilon , \text{ if } \sigma \in (\Sigma - \Sigma_k), \text{ where } \Sigma_k = \Sigma_i \cup \Sigma_j, \text{ where } i \text{ and } j \text{ represent the tasks involved in dependency } k. \]

For Example the local event set for a supervisor \( S_1 \) that controls a dependency between Task 1 and Task2 is

\[ \Sigma_{12} = \{b_1, p_1, a_1, f_{a_1}, c_1, b_2, p_2, a_2, f_{a_2}, c_2\}. \]

Hence the local uncontrolled model for a supervisor \( S_k \), acting on task \( T_i \) and task \( T_j \) is effectively the Shuffle Product of the tasks \( T_i \) and \( T_j \)

\[ G_{ij} = G_i \parallel G_j. \]

Note that all the controllable events in a local controllable set are observable with respect to the local supervisor i.e.

\[ \Sigma_{io} \subseteq \Sigma_{io} \forall \ i = 1,.....n, \]

5.4.2 The Specification Models

Each specification model represents one inter task dependency expressed over the local event set. The specification model only contains events that belong to the local event set of that supervisor. There may be more than one specification for a pair of tasks where more than one dependency are associated with these tasks, as described in Chapter 4.
5.4.3 Supervisors

We construct individual supervisors for the dependencies. We use the basic supervisory control problem—nonblocking case (BSCP-NB) and the nonblocking controllability theorem to construct these individual supervisors over their local uncontrolled models.

5.5 Existence of a Decentralized Solution

We now show that a decentralized solution exists for a workflow. The language $L_{m}^{\uparrow \mathcal{C}} (C / G)$ satisfies the first two conditions of controllability and $L_{m}(G)$ closure described by the nonblocking controllability theorem in [8,15].

The fact that $L_{m}^{\uparrow \mathcal{C}} (C / G)$ satisfies the condition of co-observability can be shown using the definition of co-observability. The definition of co-observability states that if an event $\sigma$ needs to be disabled, then at least one of the supervisors must unambiguously know that it must disable $\sigma$. Our definition requires that the local supervisors observe all the events in the local event set (Section 5.4.1).

Consider a case where two supervisors have a set of common controllable events.

$\Sigma_{1c} \cdot \Sigma_{2c} = \{\sigma\}$

If the event $\bullet$ needs to be disabled, it must be disabled by any one of these two local supervisors. Since all controllable events are also observable with respect to the local supervisors,

$\Sigma_{ic} \subseteq \Sigma_{io} \forall i = 1,\ldots,n,$

there is no ambiguity associated with disabling $\bullet$ and we can say that $L_{m}^{\uparrow \mathcal{C}} (C / G)$ is co-observable with respect to $L(G)$, $P_{o}$ and $\Sigma_{ic}$ for $i=1,\ldots,n$.
5.6 Airline Example

Consider the airline example discussed in Chapter 3. Suppose we have the condition that there is another task ‘Booking of a Train Ticket’. There is a dependency involved between airline booking and train booking. We have that the train booking cannot start until the airline booking aborts.

The following three tasks are identified:

- Purchase an airline ticket (Task $T_1$)
- Book a hotel (Task $T_2$)
- Book a train ticket (Task $T_3$)

Based on booking regulations, traveler’s preferences, or economic reasons, the following dependencies are defined:

- Booking of hotel cannot start until purchasing an airline ticket starts (Begin)
- If hotel booking aborts, then purchasing airline ticket must abort too (Abort)
- Booking of a train cannot start until airline ticket aborts (Begin on Abort)

5.6.1 Begin Dependency

The begin dependency is defined between Task1 and Task2. The uncontrolled model is the projection of $G$ over $P_{12}$ i.e. the shuffle product of the automata for tasks 1 and 2. The uncontrolled model $G_{12}$ is similar to the two task uncontrolled model shown in Figure 3.7. The specification model $C_1$ for the begin dependency is similar to Figure 3.8. The admissible language $L_{mC}^C (C_1 / G_{12})$ is obtained from the coupled model $C_1/G_{12}$ by the algorithm explained in Appendix 2. The control pattern for the begin dependency is presented below.
5.6.2 Abort Dependency

The uncontrolled model for the abort dependency is the same as for the begin dependency, since the abort dependency is also specified between Task1 and Task2. The specification model for the abort dependency is similar to the one shown in Figure 3.9. As nonblocking is of concern the admissible language will be \( L^{1c}_m(C_2 / G_{12}) \). We find the supremal controllable sublanguage \( L^{1c}_m(C_2 / G_{12}) \) using the algorithm explained in Appendix 2. The control pattern that realizes the supremal controllable sublanguage \( L^{1c}_m(C_2 / G_{12}) \) is shown in Table 5.2.

### Table 5.2 \( \Psi_2 \) Control Pattern for the Abort Dependency

<table>
<thead>
<tr>
<th>Events</th>
<th>( b_1 )</th>
<th>( f_{a1} )</th>
<th>( c_1 )</th>
<th>( b_2 )</th>
<th>( f_{a2} )</th>
<th>( c_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
5.6.3 Begin on Abort Dependency

The ‘begin on abort’ dependency is defined between Task 1 and Task 3. So the uncontrolled model for the ‘begin on abort’ dependency is the shuffle product of automata of tasks 1 and 3. As nonblocking is of concern the admissible language is the supremal controllable sublanguage $L^{sc}_m (C_3 / G_{13})$. The supremal controllable language $L^{sc}_m (C_3 / G_{13})$ is shown in Figure 5.2.

![Figure 5.2 Supremal Language for Begin on Abort Dependency](image-url)
The Begin on abort dependency states that train reservation cannot start until Airline
ticket reservation aborts. Hence in Figure 5.2, the train reservation can only begin from state 4
where airline ticket reservation has aborted. In state 5, since the airline ticket reservation
commits, the train reservation does not begin at all, which is allowed and hence state 5 is marked.

The control pattern for begin on abort dependency realizes $L^c_{tb}(C_3 / G_{13})$. The control
pattern is shown in Table 5.3.

<table>
<thead>
<tr>
<th></th>
<th>b₁</th>
<th>p₁</th>
<th>a₁</th>
<th>fa₁</th>
<th>c₁</th>
<th>b₃</th>
<th>p₃</th>
<th>a₃</th>
<th>fa₃</th>
<th>c₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The resultant control pattern is the conjunction of the individual control patterns. It’s
sufficient for any one of the supervisors to disable an event for that event to be disabled. For
example, according to control pattern for the begin dependency, in state 1 only ticket reservation
can begin ($\Psi_1(b_1, 1) = 1$). But according to the control pattern for the abort dependency both
ticket reservation and hotel booking and begin ($\Psi_2(b_1, 1) = \Psi_2(b_2, 1) =1$). In the control pattern
for the ‘begin on abort dependency’ $b_1$ is allowed, whereas $b_2$ is not disabled. Since one
supervisor does not allow hotel booking to begin, the resultant controller disables it ($\Psi(b_2, 1) = 0$).
Suppose ticket reservation task begins ($b_1$ event executes). All the supervisors observe $b_1$ and
move to the next state. In this state hotel booking can begin ($b_2$) but train reservation cannot due
to the ‘begin on abort’ dependency between airline ticket reservation and train reservation. Hence

\[ \Psi(b_3, 2) = 0. \]

The next chapter consists of a case study involving an online bookstore workflow. The bookstore workflow is modeled using distributed supervisory control.
Chapter 6. Case Study

This chapter presents a case study of the online bookstore example in [15]. The online bookstore workflow is controlled using distributed, both modular and decentralized supervisory control. The result is compared to that obtained using a centralized supervisory control approach in [15].

6.1 Online Bookstore Architecture

Figure 6.1 shows workflow architecture for an online bookstore. The online bookstore is a virtual company that has no books in stock. It has a pool of publishers who supply books to them when ordered. The bookstore has access to these publisher’s databases. The customer places an order with the bookstore.

The bookstore checks the availability of the book with a publisher by accessing the publisher’s database. At the same time bookstore checks the credit card information provided by the user. If the book is available and the credit card information provided by the user is correct, the customer is informed and the bookstore transfers the order to the publisher. If the book is not available, the bookstore decides to search for an alternative publisher and repeats the previous step or rejects the order.
After ordering the books with the publisher, the bookstore searches for a shipper and sends him a request. The shipper evaluates the request and either accepts or denies it. If the bookstore does not find a shipper or if the shipper cannot fulfill the request, the bookstore cancels the order with the publisher and notifies the customer. If the shipper accepts the request, the publisher is informed. Then the publisher prepares the book for shipment and sends it to the shipper. The shipper prepares and ships the order. The shipper notifies the online bookstore and the online bookstore or its billing company then processes the payment.

To model the Online Bookstore workflow we need to identify tasks that constitute the workflow and the relations (i.e. inter-task dependencies) between these tasks. We have identified the following eight tasks in this workflow:

- Task 1: Order
- Task 2: Credit Card Authorization
- Task 3: Order Book (publisher)
- Task 4: Find Shipper
- Task 5: Send Book to Shipper
- Task 6: Cancel Order (Publisher)
- Task 7: Ship Order
- Task 8: Process Payment

To comply with business policies and customer preferences, we identify certain constraints within the workflow. These constraints, represented in the form of inter-task dependencies are as follows.

- Credit Card Authorization cannot start until Order Placement starts (T_2 \text{BD} T_4)
- Ordering Books with publisher cannot start until Order Placement starts (T_3 \text{BD} T_4)
- If Credit Card Authorization aborts then Ordering Books with publisher must abort too (T_2 \text{AD} T_3)
- Ordering Books with publisher cannot commit or abort until Credit Card Authorization either commits or aborts (T_3 \text{TD} T_2).
- Send Book To Shipper cannot begin executing until Order Books with publisher either commits or aborts (T_5 \text{SD} T_3)
- \textit{Find Shipper} cannot start until \textit{Order Placement} starts (T_4 \text{BD} T_1)
- If \textit{Find Shipper} task aborts then \textit{Send Book To Shipper} task must abort too (T_4 \text{AD} T_5)
- If both \textit{Find Shipper} task and \textit{Send Book To Shipper} task commits, then find shipper task commits first (T_4 \text{CD} T_5)
- Cancel Order of books with publisher cannot begin executing until Find Shipper aborts (T_4 \text{BAD} T_6)
- If \textit{Find Shipper} task aborts then task \textit{Cancel Order} of books with publisher commits (T_6 \text{FCAD} T_4)
• Ship Order Task cannot begin executing until Send Book To Shipper task either commits or aborts (T_7 SD T_5)
• Ship Order Task cannot begin executing until Find Shipper task either commits or aborts (T_7 SD T_4)
• Process Payment task cannot start until Ship Order Task starts (T_8 BD T_7)
• If Process Payment task aborts then Ship Order Task must aborts too (T_8 AD T_7)
• Ship Order Task cannot commit or abort until Process Payment either commits or aborts (T_7 TD T_8)

6.1.2 Online Bookstore Workflow

In this online bookstore workflow, three parties namely Online Bookstore, Publisher and Shipper are involved. Consider the tasks executed by the Online Bookstore shown in the Figure 6.2.

These tasks are:
• Credit Card Authorization (T_2)
• Order Books (T_3)
• Find Shipper (T_4)
- Cancel Order ($T_6$)

The following inter task dependencies exist between these tasks:
- $T_2$ AD $T_3$ ($C_3$)
- $T_3$ TD $T_2$ ($C_4$)
- $T_4$ BD $T_3$ ($C_6$)
- $T_4$ BAD $T_6$ ($C_9$)
- $T_6$ FCAD $T_4$ ($C_{10}$)

### 6.2 Distributed Supervisory Control of Online Bookstore Workflow

For the online bookstore example, the uncontrolled process model is subdivided into 4 local uncontrolled models based on the dependencies. Individual supervisors are then constructed for these dependencies so that the supervisors realize the local admissible languages. The admissible languages are determined using the Modular Supervisory Control Problem described in Section 4.1. Table 6.1 shows the local uncontrolled model, specification model, admissible language and the corresponding supervisor for each dependency.

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Local Uncontrolled Model</th>
<th>Specification Model</th>
<th>Admissible Language</th>
<th>Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abort</td>
<td>$P_{23}(G) = G_2 \parallel G_3$</td>
<td>$C_3$ (Fig. 3.9)</td>
<td>$L^c_m (C_3 / G_{23})$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>Terminating Dependency</td>
<td>$P_{23}(G) = G_2 \parallel G_3$</td>
<td>$C_4$</td>
<td>$L^c_m (C_4 / G_{23})$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>Begin Dependency</td>
<td>$P_{34}(G) = G_3 \parallel G_4$</td>
<td>$C_6$ (Fig. 3.8)</td>
<td>$L^c_m (C_6 / G_{34})$</td>
<td>$S_6$</td>
</tr>
<tr>
<td>Begin on Abort</td>
<td>$P_{46}(G) = G_4 \parallel G_5$</td>
<td>$C_9$</td>
<td>$L^c_m (C_9 / G_{46})$</td>
<td>$S_9$</td>
</tr>
<tr>
<td>Forced Commit on Abort</td>
<td>$P_{46}(G) = G_4 \parallel G_5$</td>
<td>$C_{10}$</td>
<td>$L^c_m (C_{10} / G_{46})$</td>
<td>$S_{10}$</td>
</tr>
</tbody>
</table>
6.2.1 Supervisor for Abort Dependency ($S_3$)

We choose a supervisor $S_3$ such that $L_m(S_3/G_{23}) = L_m^c (C_3 / G_{23})$. Table 6.2 shows the control pattern for the abort dependency.

<table>
<thead>
<tr>
<th></th>
<th>$b_2$</th>
<th>$p_2$</th>
<th>$a_2$</th>
<th>$fa_2$</th>
<th>$c_2$</th>
<th>$b_3$</th>
<th>$p_3$</th>
<th>$a_3$</th>
<th>$fa_3$</th>
<th>$c_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

6.2.2 Supervisor for Terminating Dependency ($S_4$)

We choose a supervisor $S_4$ such that $L_m(S_4/G_{23}) = L_m^c (C_4 / G_{23})$. Table 6.3 shows the control pattern for the Terminating dependency.
### Table 6.3 Control Pattern for Terminating Dependency ($\Psi_4$)

<table>
<thead>
<tr>
<th></th>
<th>$b_2$</th>
<th>$p_2$</th>
<th>$a_2$</th>
<th>$fa_2$</th>
<th>$c_2$</th>
<th>$b_3$</th>
<th>$p_3$</th>
<th>$a_3$</th>
<th>$fa_3$</th>
<th>$c_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 6.2.3 Supervisor for Begin Dependency ($S_b$)

We choose a supervisor $S_{BD}$ such that $L_m(S_6/G_{34}) = L_m(C_6/G_{34})$. Table 6.4 shows the control pattern for the Begin dependency.

### Table 6.4 Control Pattern for the Begin Dependency ($\Psi_b$)

<table>
<thead>
<tr>
<th></th>
<th>$b_3$</th>
<th>$p_3$</th>
<th>$a_3$</th>
<th>$fa_3$</th>
<th>$c_3$</th>
<th>$b_4$</th>
<th>$p_4$</th>
<th>$a_4$</th>
<th>$fa_4$</th>
<th>$c_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.2.4 Supervisor for Begin on Abort Dependency ($S_9$)

We choose a supervisor $S_9$ such that $L_m(S_9/G_{46}) = L_m^c(C_9/G_{46})$. Table 6.5 shows the control pattern for the abort dependency.

**Table 6.5 Control Pattern for Begin on Abort Dependency ($Ψ_9$)**

<table>
<thead>
<tr>
<th>$b_4$</th>
<th>$p_4$</th>
<th>$a_4$</th>
<th>$fa_4$</th>
<th>$c_4$</th>
<th>$b_6$</th>
<th>$p_6$</th>
<th>$a_6$</th>
<th>$fa_6$</th>
<th>$c_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

6.2.5 Supervisor for Forced Commit on Abort Dependency ($S_{10}$)

We choose a supervisor $S_{BAD}$ such that $L_m(S_{10}/G_{46}) = L_m^c(C_{10}/G_{46})$. Table 6.6 shows the control pattern for the abort dependency.

**Table 6.6 Control pattern for Forced Commit on Abort Dependency ($Ψ_9$)**

<table>
<thead>
<tr>
<th>$b_4$</th>
<th>$p_4$</th>
<th>$a_4$</th>
<th>$fa_4$</th>
<th>$c_4$</th>
<th>$b_6$</th>
<th>$p_6$</th>
<th>$a_6$</th>
<th>$fa_6$</th>
<th>$c_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

60
<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>18</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.7 Centralized Control Pattern [15]

<table>
<thead>
<tr>
<th></th>
<th>b_2</th>
<th>fa_2</th>
<th>c_2</th>
<th>b_3</th>
<th>fa_3</th>
<th>c_3</th>
<th>b_4</th>
<th>fa_4</th>
<th>c_4</th>
<th>b_6</th>
<th>fa_6</th>
<th>c_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>29</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>30</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
6.3 Comparison of results

The resultant control pattern is the conjunction of the individual supervisors. On conjunction the resultant supervisor only executes events of tasks 2 and 3. The control pattern for
the centralized supervisory approach [15] confirms this. The reason for the events of tasks 4 and 6 not executing can be attributed to the dependencies between the tasks. The control pattern for the Begin on Abort dependency (Table 6.5) contains states 2, 3, 4, 5 that are not allowed in the control pattern for Forced Commit on Abort dependency. Hence when these dependencies act together, both tasks Find Shipper and Cancel Order are not allowed to begin at all. This is certainly an inconsistency in the business specifications. Hence we do not impose the Forced Commit on abort dependency, i.e. $C_{10}$. The resultant control is the conjunction of control patterns for dependencies 3, 4, 6, and 9.

The comparison of the results between the control pattern obtained by the centralized supervisory control approach and the distributed supervisory control is presented in Table 6.8.

<table>
<thead>
<tr>
<th>Supervisors</th>
<th>Supervisor $S_3$ (Table 6.2)</th>
<th>Supervisor $S_4$ (Table 6.3)</th>
<th>Supervisor $S_6$ (Table 6.4)</th>
<th>Supervisor $S_9$ (Table 6.5)</th>
<th>Centralized Supervisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>$b_2$: 1, $b_3$: 1</td>
<td>$b_2$: 1, $b_3$: 0</td>
<td>$b_3$: 1, $b_4$: 0</td>
<td>$b_4$: 1, $b_6$: 0</td>
<td>$b_2$: 1, $b_3$: 0, $b_4$: 0, $b_6$: 0</td>
</tr>
<tr>
<td>Step 2</td>
<td>$p_2$: 1, $a_2$: 1, $b_3$: 1</td>
<td>$p_2$: 1, $a_2$: 1, $b_3$: 0</td>
<td>$b_3$: 1, $b_4$: 0</td>
<td>$b_4$: 1, $b_6$: 0</td>
<td>$b_3$: 0, $b_4$: 0, $b_6$: 0</td>
</tr>
<tr>
<td>Step 3</td>
<td>$fa_2$: 1, $c_2$: 1, $b_3$: 1</td>
<td>$fa_2$: 1, $c_2$: 1, $b_3$: 0</td>
<td>$b_3$: 1, $b_4$: 0</td>
<td>$b_4$: 1, $b_6$: 0</td>
<td>$fa_2$: 1, $c_2$: 1, $b_3$: 0, $b_4$: 0</td>
</tr>
</tbody>
</table>

In Step1 all the individual supervisors are in state 1 and the centralized supervisor $S$ is in state 0. In state 1, when all the tasks are in their initial state in the centralized control pattern we can see that only Order Task is allowed to begin ($\Psi(b_2,1)=1$). Although supervisor $S_3$ allows both Credit Card Authorization and Order Books task to begin ($\Psi_3(b_2,1)=1, \Psi_3(b_3,1)=1$), supervisor $S_4$ does not allow $b_3$ ($\Psi_4(b_3,1)=0$), supervisor $S_6$ does not allow the Find Shipper task to begin.
($\Psi_6(b_4,1)=0$) and supervisor $S_9$ does not allow Cancel Order task to begin ($\Psi_6(b_6,1)=1$). So finally only $b_2$ is enabled in state 1, which is exactly the same control as generated by the centralized control pattern shown in Table 6.7.

When Credit Card Authorization begins ($b_2$ is executed) Supervisors $S_3$ and $S_4$ are the only supervisors that observe $b_2$ and hence move to their state 2. Supervisors $S_6$ and $S_9$ remain in their state 1. The centralized supervisor moves to its state 1. According to Step 2 in Table 6.8, the only events allowed in this stage are the pre-commit and abort event of the Credit card Authorization task. The events $b_3$, $b_4$, and $b_6$ are disallowed by at least one of the supervisors and hence none of these events is allowed by the resultant control.

At this stage if $p_2$ executes, then the supervisor $S_3$ and $S_4$ move to their state 3. The centralized controller moves to state 2 and the supervisors $S_6$ and $S_9$ remain in their state 1. Step 3 in Table 6.8 denotes this state. In step 3 the resultant control pattern allows $f_{a_2}$ and $c_2$. This is the same control exhibited by the centralized controller. Thus it is seen that the control obtained by distributed control approach and the centralized approach are the same.

### 6.4 Inconsistent Dependency Specification

With the centralized supervisory control the identification of the exact dependency that causes any inconsistency involves checking every dependency. This process gets difficult in real workflows with more than a few dependencies.

In the distributed supervisory control approach, every dependency is modeled separately and hence it is easy to identify the exact dependency causing the inconsistency.
Chapter 7. Conclusions, Contributions and Future Work

In this chapter we present the contributions made by this research and the conclusions derived from this research in Section 7.1 and 7.2 respectively. We also present some future research directions in Section 7.3.

7.1 Contributions

- We have designed Modular Supervisors for workflows to reduce the computational complexity arising from the state explosion in centralized supervisory control from $O(5^m, 5^n)$ to $O(5^m, 5)$, where $m$ is the number of tasks in the uncontrolled model and $n$ is the number of inter task dependencies
- The modular supervisors are shown to be controllable and nonblocking
- We have designed Decentralized supervisors for workflows that reduce the computational complexity arising due to the state space explosion in centralized supervisory control from $O(5^m, 5^n)$ to $O(5^2, 5)$
- The decentralized supervisors are shown to be controllable and nonblocking
- The Modular and Decentralized Supervisory controllers are combined to obtain a scalable Distributed Supervisory Control architecture for workflows

7.2 Conclusions

- This thesis extends the work in [3] where standard templates are developed to represent the control flow dependencies that exist between the tasks of the workflow. These
templates are based on Finite State Automata theory, which gives the models a strong mathematical foundation

- Specifically we address the problem of state space explosion and computational complexity arising from the centralized approach to supervisory control. Distributed supervisory control approach is proposed as a solution
- The properties of the FSA such as controllability, nonblocking, supremal controllable sublanguages etc. are used effectively to ensure safety and successful termination of the workflow
- Unique properties of workflows such as consistency are shown to be useful for showing that the modular and decentralized supervisors are nonblocking, which is a difficult problem
- Uncontrollable events are considered in the development of the distributed supervisory control approach
- The distributed approach in which the specifications are modeled individually allows us to identify inconsistencies in the workflow
- The 2-PC task structure is used to represent a task in this work. However the formalism can be used with other task structures such as 1-PC and 0-PC

7.3 Future Work

The supervisors that we have designed and implemented handle the control flow dependencies. In real workflows there may be dependencies and constraints due to time (temporal dependencies) and value (value dependencies). The supervisory control design can be extended to handle these dependencies.
We have assumed that all the events within the scope of a local supervisor are observable. But this might not be the case in large workflows and hence unobservability can be incorporated into workflows.
References


Appendix 1. Types of Dependencies

1. *Begin Dependency* \((t_j \text{ BD } t_i)\): task \(t_j\) cannot begin execution until \(t_i\) has begun.

2. *Abort Dependency* \((t_j \text{ AD } t_i)\): if \(t_i\) aborts then \(t_j\) aborts.

3. *Commit Dependency* \((t_j \text{ CD } t_i)\): if both \(t_i\) and \(t_j\) commit then the commitment of \(t_i\) precedes the commitment of \(t_j\).

4. *Strong Dependency* \((t_j \text{ SD } t_i)\): if \(t_i\) commits then \(t_j\) commits.

5. *Weak Abort Dependency* \((t_j \text{ WAD } t_i)\): if \(t_i\) aborts and \(t_j\) has not yet committed then \(t_j\) aborts. In other words, if \(t_j\) commits and \(t_i\) aborts then the commitment of \(t_j\) precedes the abortion of \(t_i\).

6. *Terminating Dependency* \((t_j \text{ TD } t_i)\): \(t_j\) cannot commit or abort until \(t_i\) either commits or aborts.

7. *Exclusion Dependency* \((t_j \text{ ED } t_i)\): if \(t_i\) commits and \(t_j\) has begun executing then \(t_j\) aborts.

8. *Forced Commit on Abort Dependency* \((t_j \text{ FCAD } t_i)\): if \(t_i\) aborts then \(t_j\) commits.

9. *Serial Dependency* \((t_j \text{ SD } t_i)\): \(t_j\) cannot begin executing until \(t_i\) either commits or aborts.

10. *Begin on Commit Dependency* \((t_j \text{ BCD } t_i)\): \(t_j\) cannot begin executing until \(t_i\) commits.

11. *Begin on Abort Dependency* \((t_j \text{ BAD } t_i)\): \(t_j\) cannot begin executing until \(t_i\) aborts.

12. *Weak Begin on Commit Dependency* \((t_j \text{ WBAD } t_i)\): if \(t_i\) commits, \(t_j\) can begin executing after \(t_i\) commits.
Table A.1 Dependency Classification [1]

<table>
<thead>
<tr>
<th>Precedence Type</th>
<th>Weak Causal Type</th>
<th>Strong Causal Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit</td>
<td>Strong Commit</td>
<td>Begin on Commit</td>
</tr>
<tr>
<td>Weak Begin on Commit</td>
<td>Forced Commit on Abort</td>
<td>Begin on Abort</td>
</tr>
<tr>
<td>Weak Abort</td>
<td>Exclusion</td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td>Abort</td>
<td>Serial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Terminating</td>
</tr>
</tbody>
</table>
Appendix 2. Standard algorithm for $\uparrow C$

Step 0:

Let $G = (X, \Sigma, f, X_m, x_0)$ be an automaton that generates $M$, i.e., $L(G) = M$.

Let $H = C/G = (Y, E, g, y_0, Y_m)$ be such that $L_m(H) = L_{am}$ and $L(H) = \overline{L_{am}}$, where it is assumed that $L_{am} \subseteq L(G)$.

Step 1:

Let $H_0 = (Y_0, E, g_0, (y_0, x_0), Y_{0,m}) = H \times G$

where $Y_0 \subseteq Y \times X$.

Treat all states of $G$ as marked for the purpose of determining $Y_{0,m}$.

By assumption $L_m(H_0) = L_{am}$ and $L(H_0) = \overline{L_{am}}$.

States of $H_0$ will be denoted by pairs $(y, x)$.

Set $i = 0$.

Step 2: Calculate

Step 2.1

$Y_i' = \{(y, x) \in Y_i : \Gamma(x) \cap \Sigma_{ac} \subseteq \Gamma_{H}(y, x)\}$

$g_i' = g_i \mid Y_i'$ where the notation $\mid$ stands for “restricted to”

$Y_{i,m}' = Y_{i,m} \cap Y_i'$

Step 2.2

$H_{i+1} = \text{Trim}(Y_i', E, g_i', (y_0, x_0), Y_{i,m}')$.

If $H_{i+1}$ is the empty automaton, i.e. $(y_0, x_0)$ is deleted in the above calculation,
then \( K^\uparrow_c = 0 \) and stop.

Otherwise, set

\[ H_{i+1} =: (Y_{i+1}, E, g_{i+1}, (y_0, x_0), Y_{i+1,m}). \]

Step 3

If \( H_{i+1} = H_i \), then

\[ L_m(H_{i+1}) = L_{am}^\uparrow C \quad \text{and} \quad L(H_{i+1}) = \overline{L}_{am}^\uparrow C \]

and STOP.

Otherwise, set \( i \leftarrow i+1 \) and go to Step 2.

We make the following comment about step 1 above. By definition, a state of \( H_0 \) is marked if and only if the corresponding state of \( H \) is marked. This is because we want \( H_0 \) to be equivalent to \( H \) and therefore state marking in \( G \) should not affect \( H_0 \). If the given \( L_{am}^\uparrow \) happens to be a subset of \( L_m(G) \), then the second component of all the marked states of \( H_0 \) will be marked in \( G \).