11-13-2003

Distributed Supervisory Control of Workflows

Pranav Deshpande
University of South Florida

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

Table of Contents

List of Tables

List of Figures

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*paydamage.*

Figure 1.1 Accident Insurance Claim Workflow

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.

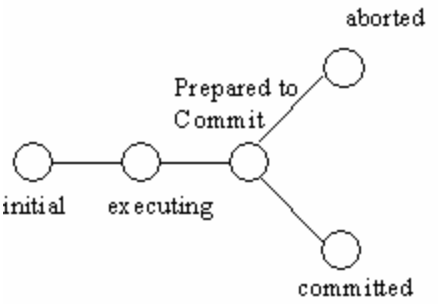

Figure 1.2 States in a Task

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

'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.

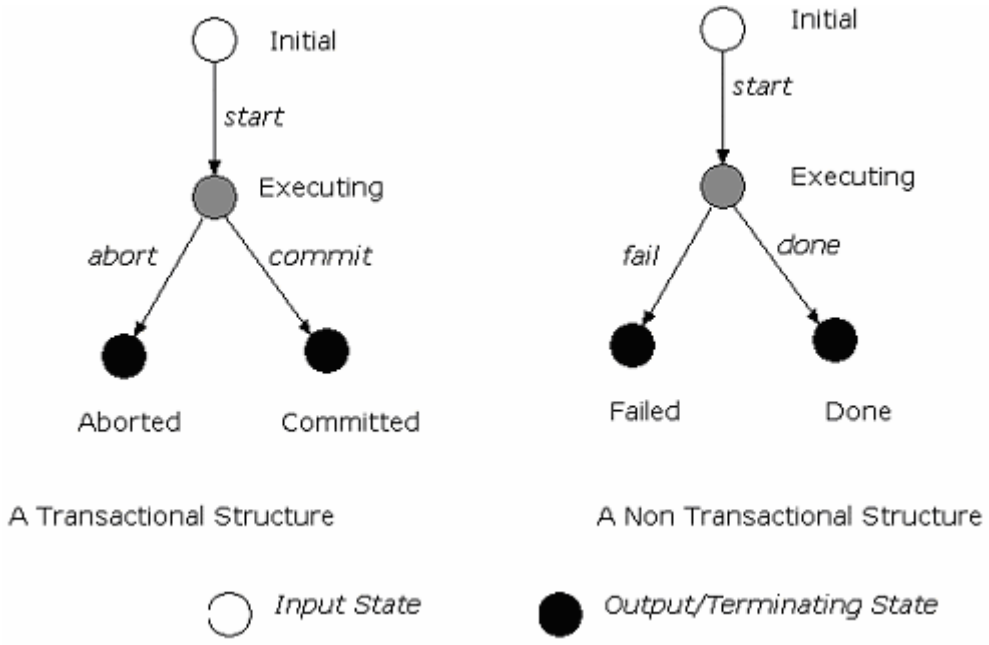

Figure 1.3 Types of Task [17]

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.

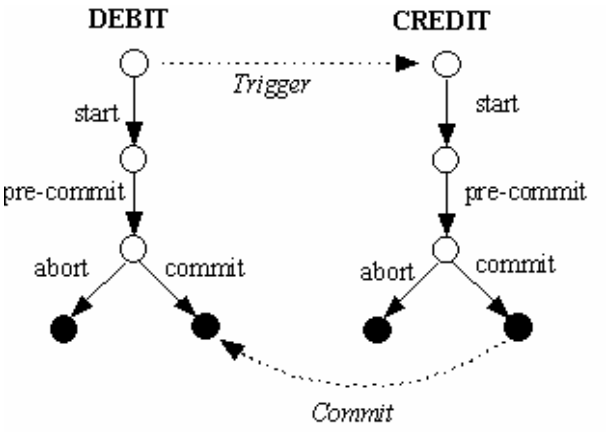

Figure 1.4 Fund Transfer Workflow [33]

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.

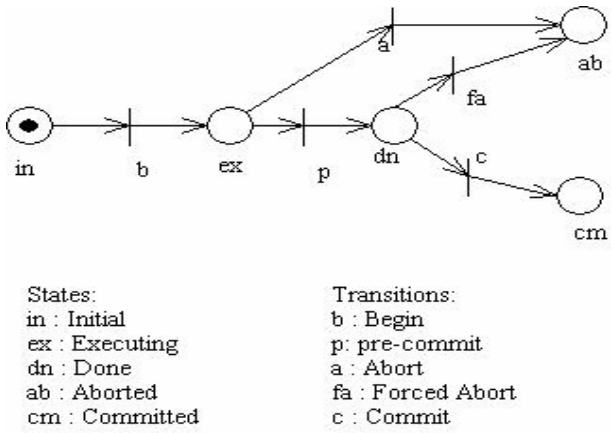

Figure 2.1 Petri Net Representation of a Task [1]

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.

$e_1 \rightarrow e_2$: If e_1 occurs then e_2 must occur. There is no implied ordering on the occurrences of the two events.

$e_1 < e_2$: If e_1 and e_2 both occur then e_1 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_j 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

Σ 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.

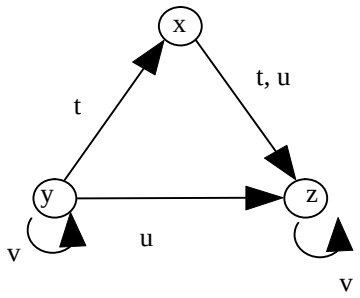

Figure 3.1 Finite State Automata Example

G can be represented formally as

$G = (Q, \Sigma, f, q_0, Q_m)$ where

$Q = \{y, z, x\}$; 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)* = $\{w : w \in \Sigma^* \text{ and } f(w, q_0) \text{ defined}\}$. 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. $L_m(G) = \{w : w \in L(G) \text{ and } f(w, q_0) \in Q_m\}$. 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.

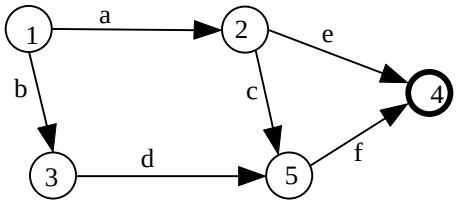

Figure 3.2 Example

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 $L_aL_b := \{s \in E^* : (s = s_as_b)$ and $(s_a \in L_a)$ 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:

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

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,

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

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

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} and

$L(G)$ = {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 $f(q,\sigma) = \emptyset\ \forall s \in \Sigma$, and $q \notin Q_m$. 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 ω 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.

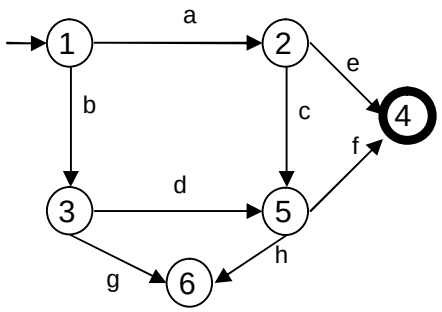

Figure 3.3 Blocking Automaton

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 *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. $\overline{L_1 \cap L_2} = \overline{L_1} \cap \overline{L_2}$ [21].

3.2.9 Controllability

The event set Σ is divided into two sets, *controllable events set* (Σ_C) and *uncontrollable events set* (Σ_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 $\overline{K}\,\Sigma_U \cap L(G) \subseteq \overline{K}$ [8]. This means that given a string ω, 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 σ causes the string to exit from the prefix closure $\overline{K}$, then *K* is not controllable since σ is an uncontrollable event. This is illustrated in Figure 3.4.

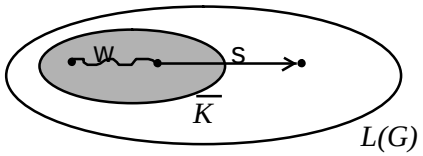

Figure 3.4 Controllability

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.

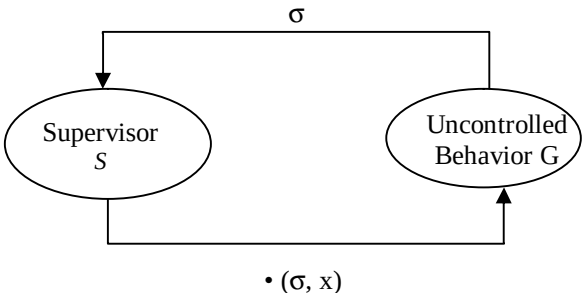

Figure 3.5 Supervisor-Uncontrolled Model Feedback System

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* Ψ 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 •

S = (S, Ψ),

S = (X, Σ, d , x_0, X_m) and

Ψ: Σ ×X → (0: disable, 1: do not disable)

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

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

When an event • executes, both *S* and G are updated. The closed loop behavior of S/G is described by the language L(S/G) = L(G) ∩ 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) ∩ L_m (S), where L_m (*S*/G) consists of the sequences of events that are marked by both G and S.

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.

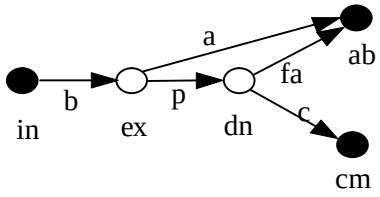

Figure 3.6 Task represented by automaton T

Task T_i is defined as

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

Where

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

Σ_i consists of two sets:

Σ_{Ui} = uncontrollable events = {p, a}

Σ_{Ci} = controllable events = {b, fa, c}

f_i (b, in) = ex; f_i (p, ex) = dn; f_i (a, ex) = ab; f_i (fa, dn) = ab; f_i (c, dn) = cm;

q_0 = in,

Q_{im} = {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.

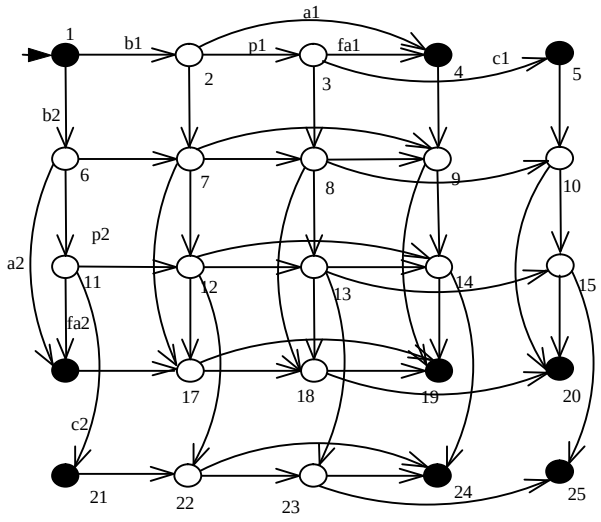

Figure 3.7 Uncontrolled Model

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.

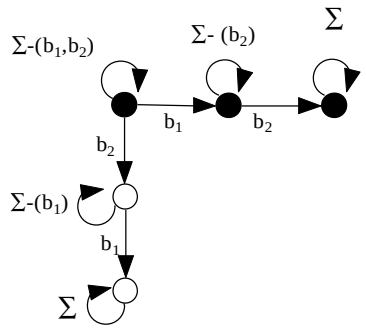

Figure 3.8 C_1-Specification Model for Begin Dependency [15]

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

b_2 executes followed by b_1, the system cannot reach a marked state. This ensures that transition b_2 is executed only after b_1 has already executed.

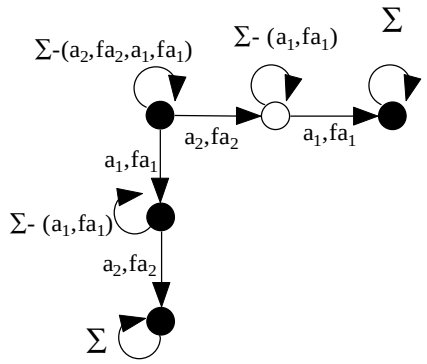

Figure 3.9 C_2-Specification Model for Abort Dependency [15]

Figure 3.9 shows the specification model for the abort dependency. The specification model remains in the marked state if any transition except a_1 and a_2 are executed. However once task T_2 aborts then the specification reaches a marked state only when task T_1 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.

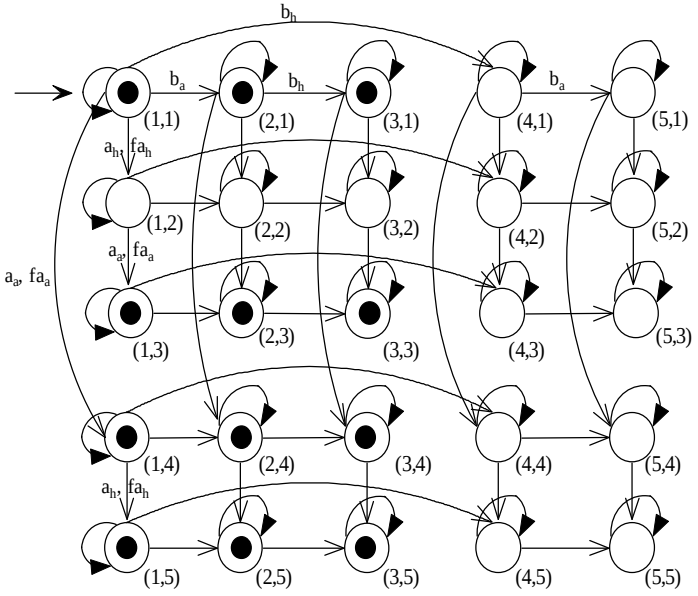

Figure 3.10. C-Specification Automaton [15]

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 $L_m(C/G) = L_m(G) \cap L_m(C)$.

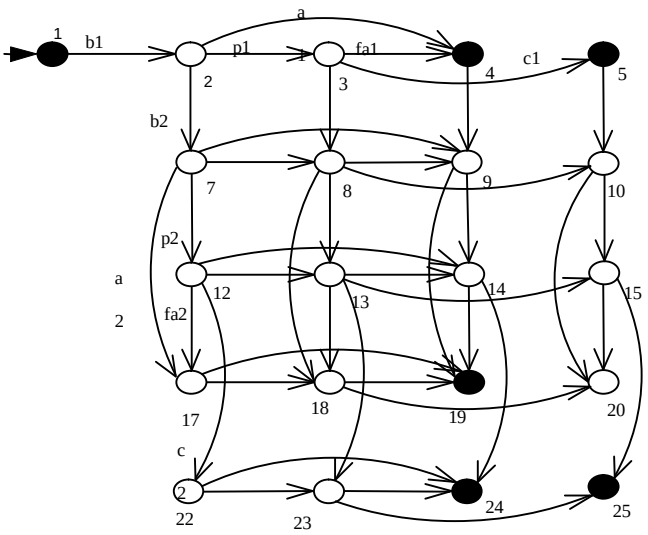

Figure 3.11 C/G-Coupled Model for Airline Workflow

In Figure 3.11, task 2 can execute it's 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

- T_1 initial, T_2 initial (1)
- T_1 aborted, T_2 initial (4)
- T_1 committed, T_2 initial (19)
- T_1 abort, T_2 abort (19)
- T_1 abort, T_2 commit (24)
- T_1 commit, T_2 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 $w \in L$ there is at least one string s such that $ws \in L_m$. 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 $L_m^{\uparrow C}\ (C/G)$, which is the supremal controllable sublanguage of $L_m(C/G)$. The supremal controllable sublanguage is shown in Figure 3.12.

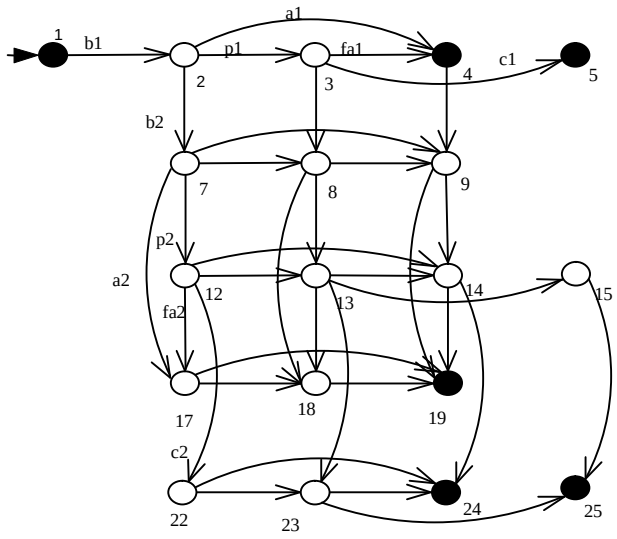

Figure 3.12 Supremal Controllable Sublanguage

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 3.1 Control Pattern • for Airline Workflow

Events	b_1	fa_1	c_1	b_2	fa_2	c_2
States						
1	1	-	-	0	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	0	-	-
8	-	1	0	-	-	-
12	-	-	-	-	1	1
13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	0	1
18	-	1	0	-	-	-
23	-	1	1	-	-	-

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*5 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 3.2 States Space Increase in Specification model

Number of Inter-task dependencies	States in specification model
3	$5^3 = 125$
4	$5^4 = 625$
5	$5^5 = 3125$
10	$5^{10} = 9765625$

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 3.3 Task-State Comparisons

Number of tasks	Number of states in the uncontrolled model
2	$5^2 = 25$
5	$5^5 = 3125$
10	$5^{10} = 9765625$
25	$5^{25} = 298023223876953125$

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 $S_{mod} = S_1(S, \Psi_1) \bullet S_2(S, \Psi_2)$ [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.

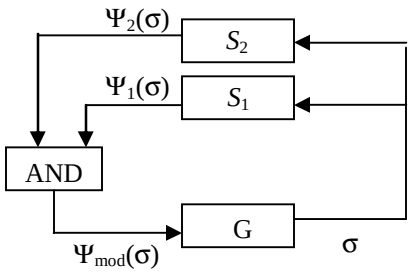

Figure 4.1 Modular Supervisory Control Architecture

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

$L_a = L_{a1} \cap L_{a2.....}L_{an}$ Where

$L_{ai} = \overline{L}_{ai} \subseteq L(G)$ for i=1,2….,n, find a modular supervisor S_{mod} (according to the architecture in Figure 4.1) such that

$$L(S_{mod}/G) = L_a^{\uparrow c}$$

4.1.2 Modular Supervisory Control Solution

Build realizations R_i of S_i such that

$L(S_i/G) = L_{ai}^{\uparrow c}$ for i =1,……,n

S_{mod} is the modular supervisor, where

$S_{mod}(s) := S_{mod12}(s) = S_1(s) \cap S_2(s)\ldots\ldots\ldots \cap S_n(s)$

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

$L(S_{mod}/G) = L_{a1}^{\uparrow c} \cap L_{a2}^{\uparrow c} \ldots\ldots. L_{an}^{\uparrow c}$

$= (L_{a1} \cap L_{a2}\ldots\ldots L_{an})^{-C}$

$= L_a^{\uparrow c}$

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…,n be individual nonblocking supervisors for G. Then $S_{mod1....n}$ is nonblocking if and only if $L_m(S_1/G)$, $L_m(S_2/G)$…..and $L_m(S_n/G)$ are non-conflicting languages, i.e., if and only if

$$\overline{L_m(S_1/G) \cap L_m(S_2/G).....\cap L_m(S_n/G)} = \overline{L_m(S_1/G)} \cap \overline{L_m(S_2/G)} \ldots \cap \overline{L_m(S_n/G)}$$

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

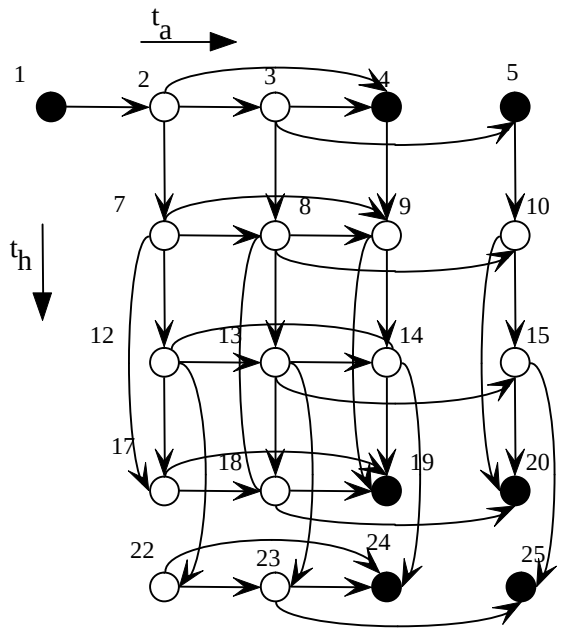

Figure 4.2 Coupled Model for Begin Dependency

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_1/G. Hence we say that $L_{am1} = L_m(C_1/G)$.

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 $L_m^{\uparrow c}(C_1/G)$, 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 Ψ_1 for the begin dependency is shown in Table 4.1. The control pattern for the begin dependency realizes $L_m^{\uparrow c}(C_1/G)$.

Table 4.1 Control Pattern Υ_1 for the Begin Dependency

Events	b_1	fa_1	c_1	b_2	fa_2	c_2
States						
1	1	-	-	0	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	1	-	-
8	-	1	1	-	-	-
12	-	-	-	-	1	1
13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	1	1
18	-	1	1	-	-	-
23	-	1	1	-	-	-

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

(Ψ_1 (b_2, 1) = 0).

However in state 2, since the ticket reservation has begun, the hotel booking can begin too, hence (Ψ_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.

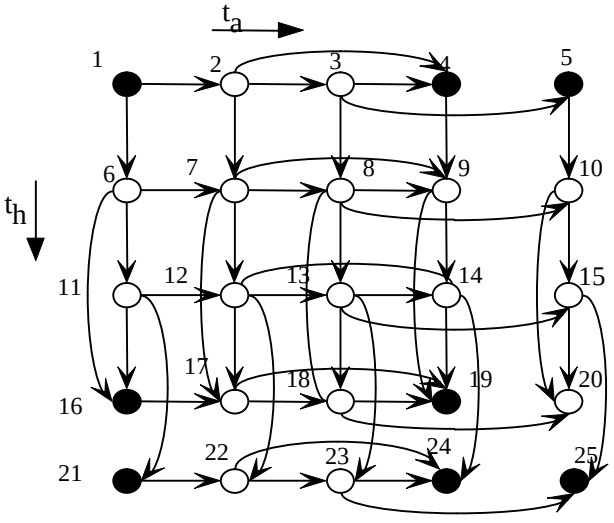

Figure 4.3 Coupled Model for Abort Dependency

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 $L_m(S_2/G) = L_{am2}^{\uparrow c} = L_m^{\uparrow c}(C_2/G)$, based on the theorem in sections 4.1.1 and 4.1.2. The supremal controllable sublanguage $L_m^{\uparrow c}(C_2/G)$ 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 the 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 $L_m^{\uparrow c}(C_2/G)$.

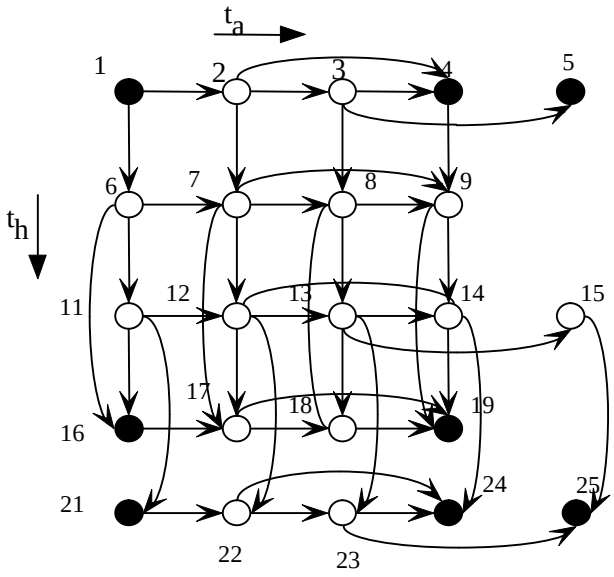

Figure 4.4 Supremal Controllable Sub-language for Abort Dependency

Table 4.2 Control Pattern Y_2 for Abort Dependency

Events	b1	fa_1	c_1	b_2	fa_2	c_2
States						
1	1	-	-	1	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	0	-	-
6	1	-	-	-	-	-
8	-	1	0	-	-	-
11	1	-	-	-	1	1
12	-	-	-	-	1	1
13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	0	1
16	1	-	-	-	-	-
18	-	1	0	-	-	-
21	1	-	-	-	-	-
23	-	1	1	-	-	-

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. (Ψ_2 (c_1, 18) = 0).

4.3.7 Control Pattern Υ for the Workflow

The resultant control in the form of control pattern ψ, 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 (Ψ_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 (Ψ_1 (b_2, 1) = 0). Hence the resultant control pattern disables the event (Ψ (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. (Ψ_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 Ψ does not allow it (Ψ (c_3, 18) = 0).

Table 4.3 Control Pattern Υ for the Airline Workflow

Events•	B1	fa_1	c_1	b_2	fa_2	c_2
States•						
1	1	-	-	0	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	0	-	-
6	0	-	-	-	-	-
8	-	1	0	-	-	-
11	0	-	-	-	1	1
12	-	-	-	-	1	1

13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	0	1
16	0	-	-	-	-	-
18	-	1	0	-	-	-
21	0	-	-	-	-	-
23	-	1	1	-	-	-

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 $K_1 = L_m^{\uparrow c}(C_1/G)$ and $K_2 = L_m^{\uparrow c}(C_2/G)$ as well as $L_m^{\uparrow c}(C_2/G) \cap L_m^{\uparrow c}(C_2/G)$. So we have to show that

$$\overline{L_m^{\uparrow C}(C_1/G) \cap L_m^{\uparrow C}(C_2/G)} \supseteq \overline{L_m^{\uparrow c}(C_1/G)} \cap \overline{L_m^{\uparrow c}(C_2/G)} \text{ i.e.}$$

$$\overline{K_1 \cap K_2} \supseteq \overline{K_1} \cap \overline{K_2}$$

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.

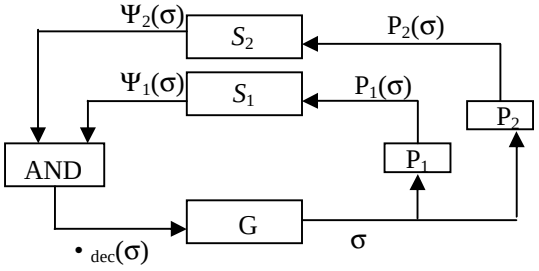

Figure 5.1 Decentralized Supervisory Control [8]

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 Σ_{io} of the event set Σ of the system. Also the

supervisor S_i only has control over Σ_{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

$P_i(\sigma) = \sigma$, if $\sigma \in \Sigma_o$

$P_i(\sigma) = \theta$, if $\sigma \in \Sigma - \Sigma_o$

where Σ_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,....,n, jointly controlling the given system G with event set $\bullet$. 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 $L(S_{dec}/G)$ and $L_m(S_{dec}/G)$. 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, Σ, 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,....n, consider the set of controllable events $S_{i,c}$, and the set of observable events $S_{i,o}$; overall, $\cup_{i=1}^{n} \Sigma_{i,c} = \Sigma_c$ and $\cup_{i=1}^{n} \Sigma_{i,o} = \Sigma_o$. Let P_i be the natural projection from Σ to $\Sigma_{i,o}$, i =1,....n. Consider also the language $L_m(C/G) \subseteq L_m(G)$, where $L_m(C/G) \neq 0$ and C is the specification. There exists a nonblocking decentralized supervisor S_{dec} for G such that

$L_m(S_{dec}/G) = L_m^{\uparrow C}(C/G)$ and $L(S_{dec}/G) = \overline{L}_m^{\uparrow C}(C/G)$ if and only if these three conditions hold [11]:

- $L_m^{\uparrow C}(C/G)$ is controllable with respect to $L(G)$ and Σ_{uc}
- $L_m^{\uparrow C}(C/G)$ is $L_m(G)$ closed
- $L_m^{\uparrow C}(C/G)$ is co-observable with respect to $L(G)$, P_i, and $\Sigma_{i,c}$, i = 1,...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 Σ_{ko} 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 *Local Uncontrolled Model*, where k represents the dependency.

Thus

$P_k(\sigma) = \sigma$, if $\sigma \in \Sigma_k$

$P_k(\sigma) = e$, if $\sigma \in (\Sigma - \Sigma_k)$, where $\Sigma_k = \Sigma_i \cup \Sigma_j$, where i and j 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

$\Sigma_{12} = \{b_1, p_1, a_1, fa_1, c_1, b_2, p_2, a_2, fa_2, c_2\}$.

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)

$G_{ij} = G_i || G_j$.

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

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

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 $L_m^{\uparrow C}(C/G)$ satisfies the first two conditions of controllability and $L_m(G)$ closure described by the nonblocking controllability theorem in [8,15].

The fact that $L_m^{\uparrow C}(C/G)$ 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 σ needs to be disabled, then at least one of the supervisors must unambiguously know that it must disable σ. 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.

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

If the event • 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,

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

there is no ambiguity associated with disabling • and we can say that $L_m^{\uparrow C}(C/G)$ is co-observable with respect to $L(G)$, P_i, and $\Sigma_{i,c}$ for i= 1,...,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 $L_m^{\uparrow C}(C_1 / G_{12})$ 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.

Table 5.1 Y_1 Control Pattern for Begin Dependency

Events	b_1	fa_1	c_1	b_2	fa_2	c_2
States						
1	1	-	-	0	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	1	-	-
8	-	1	1	-	-	-
12	-	-	-	-	1	1
13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	1	1
18	-	1	1	-	-	-
23	-	1	1	-	-	-

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 $L_m^{\uparrow C}(C_2 / G_{12})$. We find the supremal controllable sublanguage $L_m^{\uparrow C}(C_2 / G_{12})$ using the algorithm explained in Appendix 2. The control pattern that realizes the supremal controllable sublanguage $L_m^{\uparrow C}(C_2 / G_{12})$ is shown in Table 5.2.

Table 5.2 Y_2 Control Pattern for the Abort Dependency

	b1	fa_1	c_1	b_2	fa_2	c_2
1	1	-	-	1	-	-
2	-	-	-	1	-	-
3	-	1	1	1	-	-
4	-	-	-	1	-	-
5	-	-	-	0	-	-

6	1	-	-	-	-	-
8	-	1	0	-	-	-
11	1	-	-	-	1	1
12	-	-	-	-	1	1
13	-	1	1	-	1	1
14	-	-	-	-	1	1
15	-	-	-	-	0	1
16	1	-	-	-	-	-
18	-	1	0	-	-	-
21	1	-	-	-	-	-
23	-	1	1	-	-	-

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 $L_m^{\uparrow C}(C_3 / G_{13})$. The supremal controllable language $L_m^{\uparrow C}(C_3 / G_{13})$ is shown in Figure 5.2.

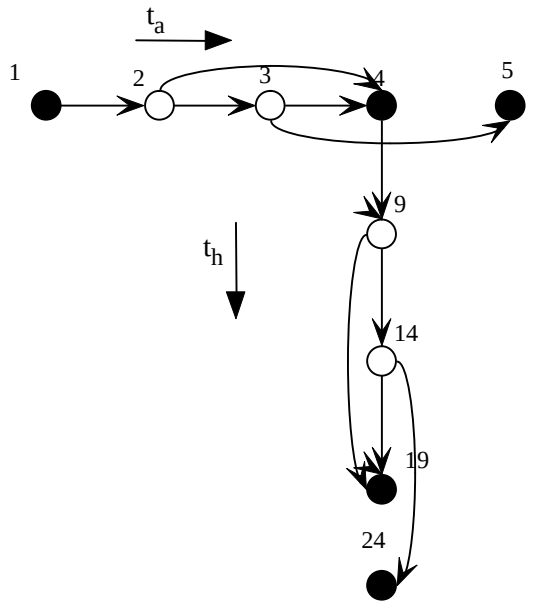

Figure 5.2 Supremal Language for Begin on Abort Dependency

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 $L_m^{\uparrow C}(C_3 / G_{13})$. The control pattern is shown in Table 5.3.

Table 5.3 Y_3 Control Pattern for Begin on Abort Dependency

	b_1	p_1	a_1	fa_1	c_1	b_3	p_3	a_3	fa_3	c_3
1	1	-	-	-	-	0	-	-	-	-
2	-	1	1	-	-	0	-	-	-	-
3	-	-	-	1	1	0	-	-	-	-
4	-	-	-	-	-	1	-	-	-	-
5	-	-	-	-	-	0	-	-	-	-
9	-	-	-	-	-	-	1	1	-	-
14	-	-	-	-	-	-	-	-	1	1
19	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-

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

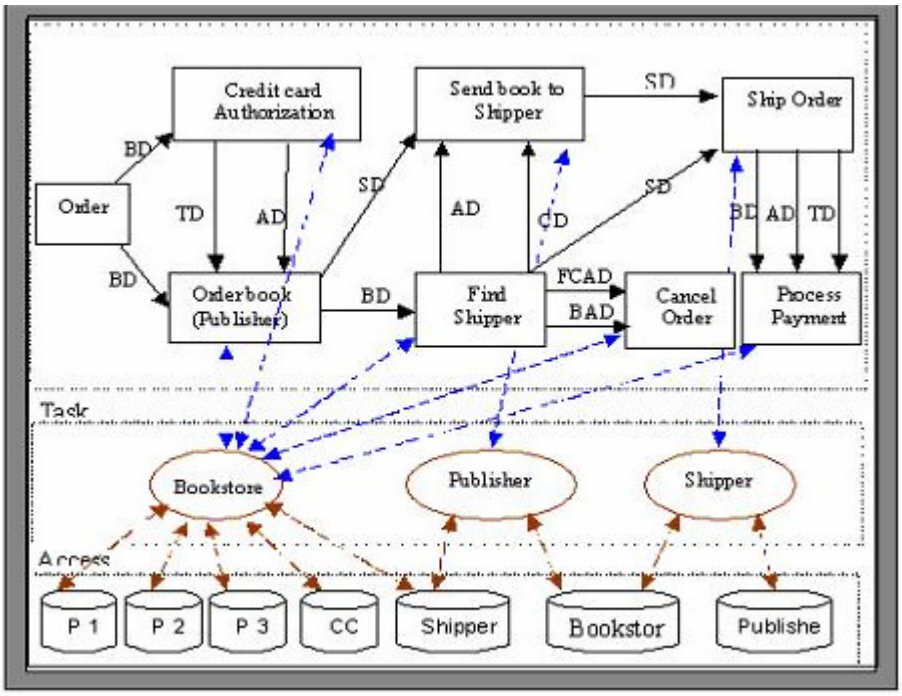

Figure 6.1 Online Bookstore Architecture

.

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

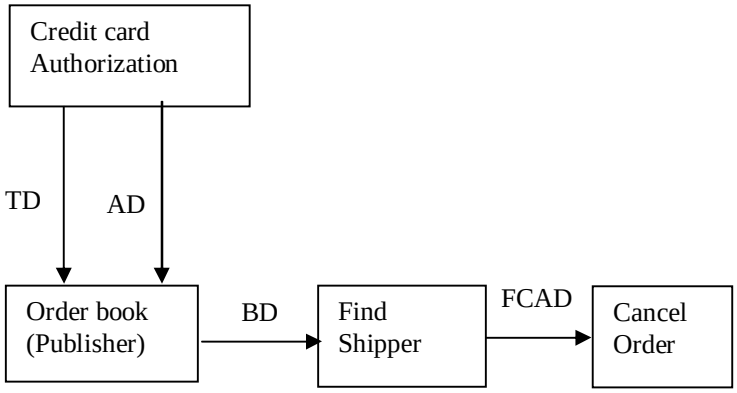

Figure 6.2 Bookstore Substructure

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 6.1 Supervisory Control Elements

Dependency	*Local Uncontrolled Model*	*Specification Model*	*Admissible Language*	*Supervisor*
Abort	$P_{23}(G) = G_2 \parallel G_3$	C_3 (Fig. 3.9)	$L_m^{\uparrow c}(C_3 / G_{23})$ (Figure 4.4)	S_3
Terminating Dependency	$P_{23}(G) = G_2 \parallel G_3$	C_4	$L_m^{\uparrow c}(C_4 / G_{23})$	S_4
Begin Dependency	$P_{34}(G) = G_3 \parallel G_4$	C_6 (Fig. 3.8)	$L_m^{\uparrow c}(C_6 / G_{34})$ (Figure 4.2)	S_6
Begin on Abort Dependency	$P_{46}(G) = G_4 \parallel G_6$	C_9	$L_m^{\uparrow c}(C_9 / G_{46})$ (Figure 5.2)	S_9
Forced Commit on Abort	$P_{46}(G) = G_4 \parallel G_6$	C_{10}	$L_m^{\uparrow c}(C_{10} / G_{46})$	S_{10}

6.2.1 Supervisor for Abort Dependency (S_3)

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

Table 6.2 Control Pattern for the Abort Dependency (Y_3)

	b_2	p_2	a_2	fa_2	c_2	b_3	p_3	a_3	fa_3	c_3
1	1	-	-	-	-	1	-	-	-	-
2	-	1	1	-	-	1	-	-	-	-
3	-	-	-	1	1	1	-	-	-	-
4	-	-	-	-	-	1	-	-	-	-
5	-	-	-	-	-	1	-	-	-	-
6	1	-	-	-	-	-	1	1	-	-
7	-	1	1	-	-	-	1	1	-	-
8	-	-	-	1	1	-	1	1	-	-
9	-	-	-	-	-	-	1	1	-	-
10	-	-	-	-	-	-	1	1	-	-
11	1	-	-	-	-	-	-	-	1	1
12	-	1	1	-	-	-	-	-	1	0
13	-	-	-	1	1	-	-	-	1	1
14	-	-	-	-	-	-	-	-	1	0
15	-	-	-	-	-	-	-	-	1	1
16	1	-	-	-	-	-	-	-	-	-
17	-	1	1	-	-	-	-	-	-	-
18	-	-	-	1	1	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-
21	0	-	-	-	-	-	-	-	-	-
23	-	-	-	0	1	-	-	-	-	-
25	-	-	-	-	-	-	--	-	1	-

6.2.2 Supervisor for Terminating Dependency (S_4)

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

Table 6.3 Control Pattern for Terminating Dependency (Y_4)

	b_2	p_2	a_2	fa_2	c_2	b_3	p_3	a_3	fa_3	c_3
1	1	-	-	-	-	0	-	-	-	-
2	-	1	1	-	-	0	-	-	-	-
3	-	-	-	1	1	0	-	-	-	-
4	-	-	-	-	-	1	-	-	-	-
5	-	-	-	-	-	1	-	-	-	-
9	-	-	-	-	-	-	1	1	-	-
10	-	-	-	-	-	-	1	1	-	-
14	-	-	-	-	-	-	-	-	1	1
15	-	-	-	-	-	-	-	-	1	1
19	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-

6.2.3 Supervisor for Begin Dependency (S_6)

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

Table 6.4 Control Pattern for the Begin Dependency (Y_6)

	b_3	p_3	a_3	fa_3	c_3	b_4	p_4	a_4	fa_4	c_4
1	1	-	-	-	-	0	-	-	-	-
2	-	1	1	-	-	1	-	-	-	-
3	-	-	-	1	1	1	-	-	-	-
4	-	-	-	-	-	1	-	-	-	-
5	-	-	-	-	-	0	-	-	-	-
7	-	1	1	-	-	-	1	1	-	-
8	-	-	-	1	1	-	1	1	-	-
9	-	-	-	-	-	-	1	1	-	-
10	-	-	-	-	-	-	1	1	-	-
12	-	1	1	-	-	-	-	-	1	1
13	-	-	-	1	1	-	-	-	1	1
14	-	-	-	-	-	-	-	-	1	1
15	-	-	-	-	-	-	-	-	1	1
17	-	1	1	-	-	-	-	-	-	-
18	-	-	-	1	1	-	-	-	-	-
19	-	-	-	-	-	-	-	-	-	-
20	-	-	-	-	-	-	-	-	-	-

22	-	1	1	-	-	-	-	-	-	-
23	-	-	-	1	1	-	-	-	-	-
24	-	-	-	-	-	-	-	-	-	-
25	-	-	-	-	-	-	-	-	-	-

6.2.4 Supervisor for Begin on Abort Dependency (S_9)

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

Table 6.5 Control Pattern for Begin on Abort Dependency (Y_9)

	b_4	p_4	a_4	fa_4	c_4	b_6	p_6	a_6	fa_6	c_6
1	1	-	-	-	-	0	-	-	-	-
2	-	1	1	-	-	0	-	-	-	-
3	-	-	-	1	1	0	-	-	-	-
4	-	-	-	-	-	1	-	-	-	-
5	-	-	-	-	-	0	-	-	-	-
9	-	-	-	-	-	-	1	1	-	-
14	-	-	-	-	-	-	-	-	1	1
19	-	-	-	-	-	-	-	-	-	-

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

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

Table 6.6 Control pattern for Forced Commit on Abort Dependency (Y_9)

	b_4	p_4	a_4	fa_4	c_4	b_6	p_6	a_6	fa_6	c_6
1	0	-	-	-	-	1	-	-	-	-
6	0	-	-	-	-	-	1	1	-	-
11	1	-	-	-	-	-	-	-	1	1
12	-	1	1	-	-	-	-	-	0	1
13	-	-	-	1	1	-	-	-	1	1
14	-	-	-	-	-	-	-	-	0	1
15	-	-	-	-	-	-	-	-	1	1

16	0	-	-	-	-	-	-	-	-	-
18	-	-	-	0	1	-	-	-	-	-
21	1	-	-	-	-	-	-	-	-	-
22	-	1	1	-	-	-	-	-	-	-
23	-	-	-	1	1	-	-	-	-	-

Table 6.7 Centralized Control Pattern [15]

	b_2	fa_2	c_2	b_3	fa_3	c_3	b_4	fa_4	c_4	b_6	fa_6	c_6
0	1	-	-	0	-	-	0	-	-	0	-	-
1	-	-	-	0	-	-	0	-	-	0	-	-
2	-	1	1	0	-	-	0	-	-	0	-	-
3	-	-	-	1	-	-	0	-	-	0	-	-
4	-	-	-	1	-	-	0	-	-	0	-	-
5	-	-	-	-	-	-	1	-	-	0	-	-
6	-	-	-	-	-	-	1	-	-	0	-	-
7	-	-	-	-	1	0	1	-	-	0	-	-
8	-	-	-	-	-	-	1	-	-	0	-	-
9	-	-	-	-	-	-	-	-	-	0	-	-
10	-	-	-	-	1	1	1	-	-	0	-	-
11	-	-	-	-	-	-	1	-	-	0	-	-
12	-	-	-	-	-	-	-	-	-	0	-	-
13	-	-	-	-	1	0	-	-	-	0	-	-
14	-	-	-	-	-	-	-	-	-	0	-	-
15	-	-	-	-	-	-	-	1	1	0	-	-
16	-	-	-	-	-	-	-	-	-	1	-	-
17	-	-	-	-	-	-	1	-	-	0	-	-
18	-	-	-	-	1	1	-	-	-	0	-	-
19	-	-	-	-	-	-	-	-	-	0	-	-
20	-	-	-	-	-	-	-	1	1	0	-	-
21	-	-	-	-		-	-	-	-	1	-	-
22	-	-	-	-	1	0	-	1	1	0	-	-
23	-	-	-	-	1	0	-	-	-	1	-	-
24	-	-	-	-	-	-	-	1	1	0	-	-
25	-	-	-	-	-	-	-	-	-	1	-	-
26	-	-	-	-	-	-	-	-	-	0	-	-
27	-	-	-	-	-	-	-	-	-	-	-	-
28	-	-	-	-	-	-	-	-	-	0	-	-
29	-	-	-	-	1	1	-	1	1	0	-	-
30	-	-	-	-	1	1	-	-	-	1	-	-
31	-	-	-	-	-	-	-	1	1	0	-	-
32	-	-	-	-	-	-	-	-	-	1	-	-

33	-	-	-	-	-	-	-	-	-	0	-	-
34	-	-	-	-	-	-	-	-	-	-	-	-
35	-	-	-	-	1	0	-	-	-	0	-	-
36	-	-	-	-	1	0	-	-	-	-	-	-
37	-	-	-	-	-	-	-	-	-	0	-	-
38	-	-	-	-	-	-	-	-	-	-	-	-
39	-	-	-	-	-	-	-	-	-	-	1	1
40	-	-	-	-	-	-	-	-	-	-	-	-
41	-	-	-	-	-	-	-	1	1	0	-	-
42	-	-	-	-	-	-	-	-	-	1	-	-
43	-	-	-	-	1	1	-	-	-	0	-	-
44	-	-	-	-	1	1	-	-	-	-	-	-
45	-	-	-	-	-	-	-	-	-	0	-	-
46	-	-	-	-	-	-	-	-	-	-	-	-
47	-	-	-	-	-	-	-	-	-	-	1	1
48	-	-	-	-	-	-	-	-	-	-	-	-
49	-	-	-	-	1	0	-	-	-	-	1	1
50	-	-	-	-	1	0	-	-	-	-	-	-
51	-	-	-	-	-	-	-	-	-	-	1	1
52	-	-	-	-	-	-	-	-	-	-	-	-
53	-	-	-	-	-	-	-	-	-	-	-	-
54	-	-	-	-	-	-	-	-	-	0	-	-
55	-	-	-	-	-	-	-	-	-		-	-
56	-	-	-	-	1	1	-	-	-	-	1	1
57	-	-	-	-	1	1	-	-	-	-		
58	-	-	-	-	-	-	-	-	-	-	1	1
59	-	-	-	-	-	-	-	-	-	-	-	-
60	-	-	-	-	-	-	-	-	-	-	-	-
61	-	-	-	-	1	0	-	-	-	-	-	-
62	-	-	-	-	-	-	-	-	-	-	-	-
63	-	-	-	-	1	-	-	-	-	-	1	1
64	-	-	-	-	-	-	-	-	-	-	-	-
65	-	-	-	-	1	1	-	-	-	-	-	-
66	-	-	-	-	-	-	-	-	-	-	-	-
67	-	-	-	-	-	-	-	-	-	-	-	-
68	-	-	-	-	-	-	-	-	-	-	-	-

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 6.8 Comparison of Results

	Supervisor S_3 (Table 6.2)	Supervisor S_4 (Table 6.3)	Supervisor S_6 (Table 6.4)	Supervisor S_9 (Table 6.5)	Centralized Supervisor
Step 1	b_2: 1; b_3: 1	b_2: 1, b_3: 0	b_3: 1, b_4: 0	b_4: 1, b_6: 0	b_2: 1, b_3: 0, b_4: 0, b_6: 0
Step 2	p_2: 1, a_2: 1, b_3: 1	p_2: 1, a_2: 1, b_3: 0	b_3: 1, b_4: 0	b_4: 1, b_6: 0	b_3: 0, b_4: 0, b_6: 0
Step 3	fa_2: 1, c_2: 1, b_3: 1	fa_2: 1, c_2 :1, b_3: 0	b_3: 1, b_4: 0	b_4: 1, b_6:0	fa_2: 1, c2: 1, b_3: 0, b4: 0,

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 ($\Psi_9(b_6,1)=1$. 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 S4 move to their state 3. The centralized controller moves to state 2 and the supervisors S6 and S9 remain in their state1. Step 3 in Table 6.8 denotes this state. In step 3 the resultant control pattern allows fa_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

1. N. Adam, V. Atluri and W. Huang, *Modeling and Analysis of Workflows Using Petri Nets*, In Journal of Intelligent Information Systems, 10, 131-158 (1998).

2. G. Alonso, D. Agarwal, A. El Abbadi, M. Kamath et al., *Advanced Transaction Models in Workflow Context*, Research Report, IBM Research Division, San Jose, USA. 22.

3. G. Alpan, *Design and analysis of Supervisory Controllers for DEDS*, Ph.D. Dissertation, Rutgers University 1997.

4. P. Attie, M. Singh, A. Sheth and M. Rusinkiewicz, *Specifying and Enforcing Inter-task Dependencies*, In Proceedings of the 19th International Conference on Very Large Databases, Dublin, 1993, pp. 134-145.

5. A. P. Barros, A. H. M. ter Hofstede, *Towards the Construction of Workflow Suitable Conceptual Modeling Techniques*, Information Systems Journal 8 (4) (1998) 313-337.

6. R. Boel, and G. Stremersch, editors, *Discrete Events Systems, Analysis and Control*, Kluwer Academic Publishers, 2000.

7. F. Casati, S. Ceri, B. Pernici, G. Pozzi, *Conceptual Modeling of Workflows*, in Proceedings of 14th Object Oriented and Entity-Relationship Approach, Gold Coast, Australia, Lecture Notes in Computer Science, vol. 1021, 12-15 Dec.1995, 341-354.

8. C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 1999.

9. P. Chrysanthis, *ACTA: A Framework for Modeling and Reasoning about Extended Transactions*, PhD Thesis, Department of Computer and Information Science, University Of Massachusetts, Amherst (1991).

10. B. Curtis, M. I. Kellner, and J. Over, *Process Modeling*, Comm. Of the ACM, 35(9): 75-90, 1992.

11. J. Cury and M. Queiroz, Website Tutorial, www.laas.fr/~francois/SVF/seminaires.

12. C. A. Ellis, Information Control Nets: *A Mathematical Model of Office Information Flow*, Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems, ACM Press, 1979.

13. A. K. Elmagarmid, Y. Lue, W. Litwin, and M. Rusinkiewicz, *A Multidatabase Transaction Model for InterBase.* In Proceedings 16th International Conference on Very Large Database, pages 507-518, 1992.

14. G. K. Janssens, J. Verelst, and B. Weyn, *Techniques for Modeling Workflows and Their Support of Reuse.*

15. A. R. Khemuka, *Workflow Modeling and Analysis*, Master's Thesis, IMSE, University of South Florida, Tampa, 2003.

16. J. Klein. *Advanced Rule Driven Transactional Management.* Proceeding of the IEEE COMPCON, 1991.

17. N. Krishnakumar and A. Sheth, *Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations*, Distributed and Parallel Databases, 3(2), pp.155-186, 1995.

18. N. Krishnakumar and A. Sheth, *Specification of Workflows with Heterogeneous Tasks in METEOR.*

19. R. Kumar and V. Garg, *Modeling and Control of Logical Discrete Event Systems*, Kluwer Academic Publishers, 1995.

20. F. Lin and W. Wonham, *Decentralized Supervisory Control of Discrete-Event Systems*, Information Sciences, Vol. 44, 199-224, 1988.

21. P. J. Ramadge and W. M. Wonham, *Supervisory Control Of A Class of Discrete Event Processes*, SIAM Journal of Control and Optimization, 1987, Pages 206-229.

22. M. Rusinkiewicz and A. Sheth, *Specification and Execution of Transactional Workflows*, In Modern Database Systems: The Object Model, Interoperability and Beyond, W. Kim, Ed., Addison-Wesley/ACM Press, 1994.

23. K. Salimifard and M. Wright, *Petri-Net Based Modeling of Workflow Systems: An Overview*, In European Journal of Operations Research 134 (2001) pg 664-676.

24. M. Singh, G. Meredith, C. Tomlinson, and P. Attie, *Event Algebra for specifying and scheduling workflows.* Proce. Fourth International Conference on Database system for advance application pages 53-60, 1995.

25. J. Tang, and J. Veijalainen, *Enforcing Inter-task Dependencies in Transactional Workflows*, Research Report No. J-2/95. VTT Information Technology, Finland, January 1995.

26. W. van der Aalst and A.H.M. ter Hofstede. *Verification of Workflow Task Structures: A Petri-net-based Approach.* Information Systems, 25(1): 43{69, 2000).

27. W. van der Aalst, *A class of Petri nets For Modeling and Analysis Business Process.* Computer Science reports, Eindhoven University of Technology, Eindhoven 1995.

28. W. van der Aalst, A. Hirnschall and H. M. W. Verbeek, *An Alternative Way to Analyze Workflow Graphs.*

29. W. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski. *Advanced Workflow Patterns*. In O. Etzion and P. Cheuermann, editors, Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000), volume 1901 of Lecture Notes in Computer Science, pages 18-29, Eilat, Israel, September 2000. Springer-Verlag.

30. W. van der Aalst, *Petri Nets based Workflow Management Software*. In NSF Workshop on workflow and process Automation in Information Systems: State-of-art- and Further Directions, 1996.

31. W. van der Aalst, *The Application of Petri Nets to Workflow Management,* The Journal of Circuits, Systems and Computers (8:1), 1998, pp. 21-66.

32. W. van der Aalst., *Three Good Reasons for Using a Petri-Net-based Workflow Management System,* In S. Navathe and T. Wakayama, Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), pages 179-201, Cambridge, Massachusetts, Nov 1996.

33. C. Wallace, P. Jensen and N. Sopparkar, *Supervisory Control of Workflow Scheduling,* Proceedings of ATMA. 36-46, 1996.

34. D. Wodtke, and G. Weikum, *A formal Foundation for Distributed Workflow Execution Based State Charts*. In Proc. International Conference on Data base theory, 1997, Delpin, Greece.

35. D. Worah, and A. Sheth, *Transactions in Transactional Workflows,* In: Jajodia, S. and Kerschberg, L. (Eds.), Advanced Transaction Models and Architectures: Kluwer Academic Publishers.1997.

Appendices

Appendix 1. Types of Dependencies

1. *Begin Dependency* (t_j BD t_i): task tj cannot begin execution until task ti has begun.
2. *Abort Dependency* (t_j AD t_i): if task t_i aborts then task t_j aborts.
3. *Commit Dependency* (t_j CD t_i): if both task t_i and t_j commit then the commitment of t_i precedes the commitment of t_j .
4. *Strong Dependency* (t_j SD t_i): if task t_i commits then task t_j commits.
5. *Weak Abort Dependency* (t_j WAD t_i): if task t_i aborts and task t_j has not yet committed then task t_j aborts. In other works if task t_j commit and task t_i then the commitment of t_j precedes the abortion of t_i.
6. *Terminating Dependency* (t_j TD t_i): t_j cannot commit or abort until t_i either commits or aborts.
7. *Exclusion Dependency* (t_j ED t_i): if task t_i commits and task t_j has begun executing then task t_j aborts.
8. *Forced Commit on Abort Dependency* (t_j FCAD t_i): if task t_i aborts then task t_j commits.
9. *Serial Dependency* (t_j SD t_i): t_j cannot begin executing until t_i either commits or aborts.
10. *Begin on Commit Dependency* (t_j BCD t_i): t_j cannot begin executing until t_i commits.
11. *Begin on Abort Dependency* (t_j BAD t_i): t_j cannot begin executing until t_i aborts.
12. *Weak Begin on Commit Dependency* (t_j WBAD t_i): if t_i commits, t_j can begin executing after t_i commits.

Table A.1 Dependency Classification [1]

Precedence Type	Weak Causal Type	Strong Causal Dependencies
Commit	Strong Commit	Begin on Commit
Weak Begin on Commit	Forced Commit on Abort	Begin on Abort
Weak Abort	Exclusion	Begin
	Abort	Serial
		Terminating

Appendix 2. Standard algorithm for $\uparrow$ C

Step 0:

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

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

$$Y_i^{'} = \{(y,x) \in Y_i : \Gamma(x) \cap \Sigma_{uc} \subseteq \Gamma_{Hi}((y,x))\}$$

$g_i^{'} = g_i \mid Y_i^{'}$ where the notation | stands for " restricted to"

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

Step 2.2

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

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

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

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 $L_{am}^{\uparrow C}$ 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.