Scholar Commons
USF University of South Florida

University of South Florida
Scholar Commons

Graduate Theses and Dissertations | Graduate School

11-7-2003

Workflow Modeling Using Finite Automata

Atul Ravi Khemuka
University of South Florida

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

 Part of the American Studies Commons

Scholar Commons Citation
Khemuka, Atul Ravi, "Workflow Modeling Using Finite Automata" (2003). *Graduate Theses and Dissertations.*
https://scholarcommons.usf.edu/etd/1406

This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.

Workflow Modeling Using Finite Automata

by

Atul Ravi Khemuka

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

Major Professor: Ali Yalcin, Ph.D.

Co Major Professor: William Miller, Ph.D.

Suresh Khator, Ph.D.

Date of Approval:
November 7, 2003

Keywords: supervisory control theory, task, control-flow dependencies, state avoidance, string avoidance

© Copyright 2003 , Atul Ravi Khemuka

Table of Contents

List of Tables

List of Figures

Workflow Modeling Using Finite Automata

Atul Ravi Khemuka

ABSTRACT

A Workflow is an automation of a business process. In general, it consists of processes and activities, which are represented by well-defined tasks. These include 'Office Automation,' 'Health Care' and service-oriented processes such as 'Online Reservations,' 'Online Bookstores' and 'Insurance Claims,' etc. The entities that execute these tasks are humans, application programs or database management systems. These tasks are related and dependent on one another based on business policies and rules.

With rapid increases in application domains that use workflow management systems, there is a need for a framework that can be used to implement these applications. In particular, it is essential to provide a formal technique for defining a problem that can be used by various workflow software product developers.

In this work, a formal framework based on finite state automata that facilitate modeling and analysis of workflows is presented. The workflow and its specifications are modeled separately as finite state automata models. We provide a general framework for specifying control flow dependencies in the context of supervisory control theory. We also identify several properties of supervisory control theory and demonstrate their use for conducting the analysis of the workflows.

Chapter 1
Introduction

The work on business process reengineering and office automation in the1970s led to the evolution of workflow technology. Since then workflow has been a subject of on-going development in the traditional areas of business processes. These include office automation, health care, telecommunication, manufacturing etc. Workflow generally represents processes and activities, which are represented by well-defined tasks. These tasks are related and dependent on one another and are executed either by humans or processes such as application programs or database management systems. As an example of workflow, consider a computing system of hospital management and administration [38]. A workflow for this kind of system may consist of several tasks such as entering the patient data into a database, obtaining information on earlier visits and medical history, ascertaining insurance information, entering the medical attendant's diagnostics, prescribing treatment medicine, assessing cost and billing the patient. Following are some of the formal definition of workflow.

- A workflow is a collection of tasks organized to accomplish some business process [35].
- A workflow is a representation of a given process that is made up of well-defined collection of activities referred as tasks [1].
- The workflow management coalition (WFMC) defines a workflow as a computerized facilitation or automation of a business process, in whole or part [39].

1.1 Task

The most important concept of workflow is a task. A task in a workflow is a logical unit of work that can be processed by the processing entity. Table 1.1 shows common workflow, task and processing entity examples [26].

Table 1.1. Example Workflow

Workflow	*Task*	*Processing Entity*
Mail routing	Email	Mailer
Loan processing	Form processing	Humans, application software
Order processing	Form processing	Humans, application software, DBMS
Service order processing in telecommunication	Transactions, Contracts	Application system, DBMS

A task is modeled as a set of externally observable states as shown in Figure 1.1. Typical states include: *initial*, indicate the start of a task execution; *done*, indicates the successful execution of all operation in a task; *commit*, indicate that all the operation in the task has been completed successfully and their effects are permanently stored in the system; *abort* signifies the failure of the execution of a task and all effects of the task be eliminated as if it had never been executed [28].

Each task begins executing only after begin transition is invoked. At any given point of time, a task can be in any of the executing states. A task moves from one state to another only when the transition between them is enabled. The transaction is enabled either by the workflow *controller* or by the *processing entity*. If the workflow controller enables the transition then the task is *controllable*. Whereas if a processing entity enables the transition then the task is said to be *uncontrollable* [42]. For example, in Figure 1.1 the done transition between the executing state and done state is enabled by the

processing entity, where as the start transition is enabled by the workflow controller. A workflow controller or scheduler is one which is responsible for coordinating the execution of various tasks within a workflow [42]. Where as processing entity is any user or application system that is responsible for completion of task during workflow execution.

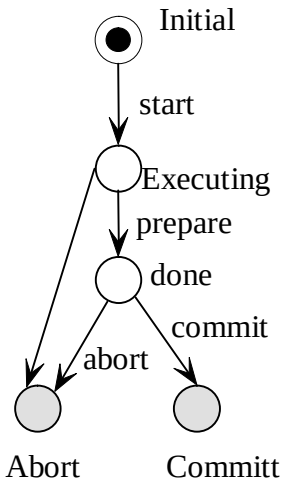


Figure 1.1. Task Structure [26]

1.1.1 Types of Tasks

A task could be transactional or non-transactional in nature. Each of these tasks can be further classified as user task and application task. User tasks are manual tasks that involve human action or human-computer interactions. Whereas application tasks are automated processes that need not involve human interaction, e.g. computer programs. A transactional task is a task that minimally obeys the atomicity property of a transaction and maximally supports the ACID (Atomic Consistent Isolated Durable) property. In this type of task there are four externally visible states: initial, executing, committed and aborted as shown in Figure 1.2. An example of this type of task is a banking transaction. Where as non-transactional task is a task that does not support the atomicity or any of the ACID transactional properties. The externally visible states of a non-transactional task

are initial, executing, failed and done as shown in Figure 1.2. Human activities are usually considered to be the non-transactional tasks [42].

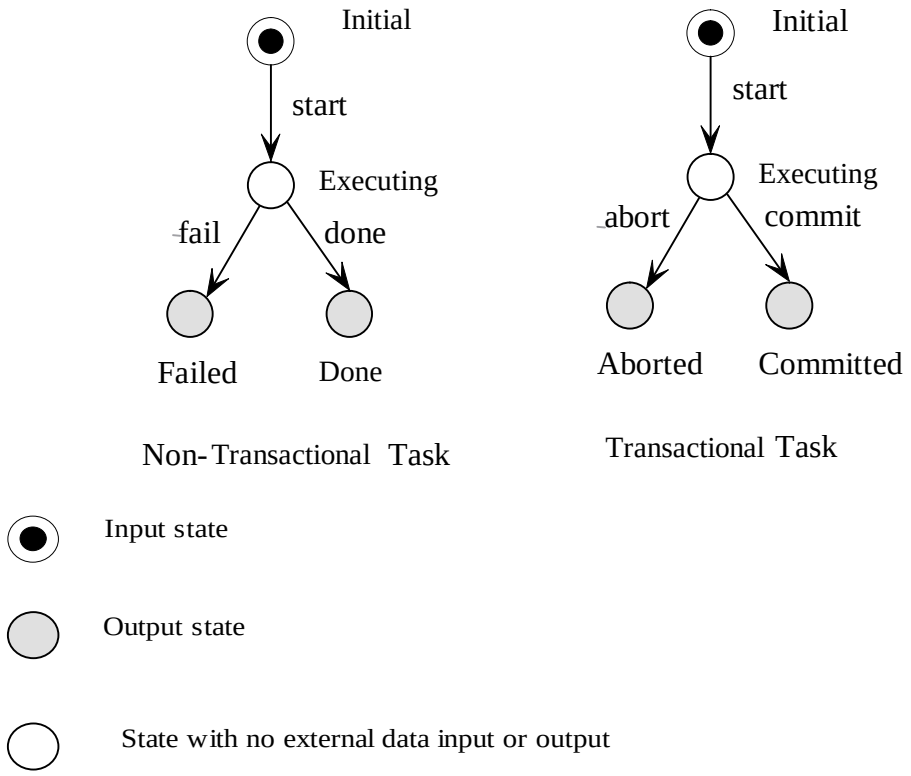


Figure 1.2. Types of Task Structure [26]

1.1.2 Task Structure

Tasks are modeled in the workflow using task structures that represents the execution behavior of each task. The task structure can be defined by providing [26].

- A set of externally visible executing states of a task (e.g. initial, executing, done, commit, abort).
- A set of transitions or primitives between these states (e.g. begin, pre-commit, abort, commit).
- The conditions that enables these transitions (the transition conditions can be used to specify inter-task dependencies).

In general, each task can have a different internal task structure. This depends mainly on the characteristics of the system on which task is executed and some of the properties of processing entity responsible for the execution of a task. A task structure can be transactional or non-transactional. In the workflow environment a user task or script is characterized by the non-transactional task structure, which has failed or done as final state. Whereas in transactional task structure, a task executes a sequence of operations, then requests a commit or abort. If a commit fails then the task is aborted. But this is not always the case as transactional workflow could be two-phase commit, one-phase commit or zero-phase commit.

A two-phase commit first enters prepared to commit state and if the controller decides to commit, the task is guaranteed to commit. Whereas in one-phase commit there is no prepared to commit state, it can commit or abort once the task has been executed. However, in zero-phase commit there is no explicit commit state, i.e. a task either finishes executing or fails to execute [21]. Figure 1.3 shows the two-phase commit and one-phase commit task structure.

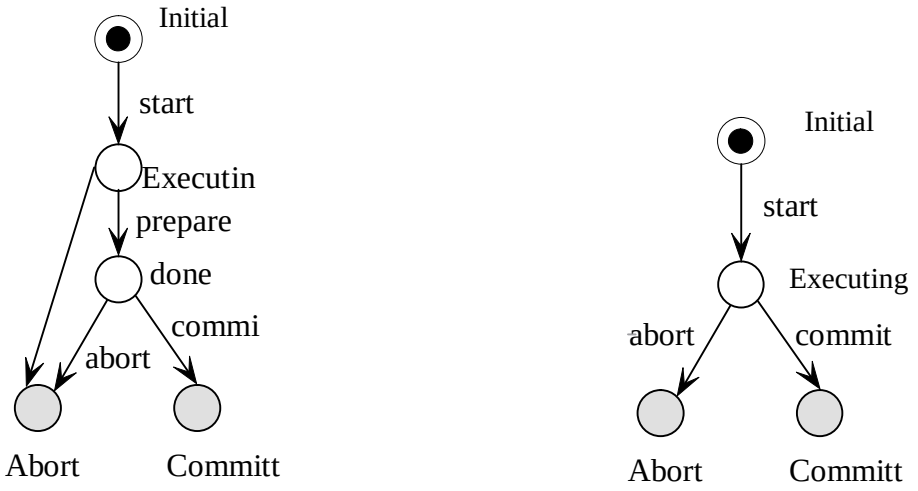


Figure 1.3. Types of Transactional Task [16]

1.2 Task Dependency

Task dependency is a method of describing certain restrictions on the execution of the workflows [28]. Dependencies can be intra-task dependencies or inter-task dependencies. Intra-task dependencies are dependencies within the task and inter-task dependencies are between tasks. Following are the types of inter-task dependencies that exist in the workflow [1].

- Control-flow Dependency: A control-flow dependency between two tasks t_i and t_j specifies the condition under which t_j is allowed to enter state s_j based on the state st_i of t_i. A comprehensive list of task dependencies based on the task primitives (begins, commit and abort) can be found in Appendix A.
- Value Dependency: A value dependency specifies task dependencies based on the out put value generated by certain tasks.
- External Dependency: These dependencies are due to some external factors such as time. These are also termed as temporal dependencies.

1.3 Types of Control-flow Dependencies

Control flow dependencies based on their precedence order and incompatible state can be classified into three types [1].

- Strong-causal
- Weak-causal
- Precedence

1.3.1 Strong-causal Dependency

Strong-causal dependency between two tasks t_i and t_j can be interpreted as t_j can enter state s_j only if t_i enters state s_i. Thus, logically, the combination of $s_i^{'}$ and s_j is not

allowed at any given time, s_i and their corresponding $s_i^{'}$ are shown in Table 1.2. Moreover in order to enforce the dependency, s_i must precede $s_j^{'}$. For Example, a business rule that states that the purchasing department is allowed to order an item only if the inventory falls below a certain level.

- Incompatible state ($s_i^{'}$, s_j)
- Precedence order: s_i <= s_j
- Example: Begin Dependency
- *Begin Dependency* (t_j BD t_i): task t_j cannot begin execution until task t_i has begun

1.3.2 Weak–causal Dependency

Weak-causal dependency between two tasks, t_i and t_j can be interpreted as, t_j must enter state s_j if t_i enters state s_i. Thus, logically, the combination of s_i and $s_j^{'}$ is not allowed at any given time. The weak-causal type specifies the sufficient condition to enforce the dependency. In other word if any of the two task involved in the dependency start execution, the other task should start execution. Thus, the combination of s_i and b_j (begin state of task j) is not an allowed terminating state. In a workflow, the weak-causal type can be depict a situation where a particular workflow state triggers another event. For example, the business rule, which states that the purchasing department is allowed to order items if the inventory falls below a certain level.

- Incompatible state (s_i, $s_j^{'}$)
- Precedence order: None
- Example: Abort Dependency
- *Abort Dependency* (t_j AD t_i): if task t_i aborts then task t_j aborts

1.3.3 Precedence Dependency

Precedence dependency between two task t_i and t_j can be interpreted as, t_i must enter state s_i before t_j enters state s_j if both s_i and s_j occur. For example, the business rule stating that if reordering of item requires approval from both divisional manager and general manager, the approval from divisional must be obtained before that of general manager.

- Precedence order: $s_i <= s_j$
- Incompatible state: None
- Example: Commit Dependency
- *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

Table 1.2. s_i and Corresponding s_i' [1]

s_i	ex_i dn_i cm_i ab_i
s_i'	in_i ex_i ab_i cm_i

1.4 Organization of the Thesis

The Organization of this thesis is as follows: Chapter 1 include basic definitions and terminology on workflows. Chapter 2 include literature review on the current techniques used in workflow modeling, motivation for the research focus, specific problem definition and objectives of the research. Chapter 3 describes the methodology used to solve the problem and highlights some of the analysis techniques useful in the context of the workflow. Chapter 4 describes modeling workflows with uncontrollable events in the context of Supervisory control theory. Chapter 5 includes a case study of online bookstore. Chapter 6 includes contribution, conclusion and suggestions for future research.

Chapter 2
Literature Review

In the 1980s and early 1990s a lot of research was done in relation to database and transaction processing. The database researchers during this period attempted to use the transactional models to model workflows. However, these models where not practical for real world applications. But they can be used as a primary baseline to model workflow applications, and subsequently workflow management systems (WFMS) [40].

This chapter is organized as follows. In Section 2.1, details are presented about transactional models. In Section 2.2, the concept of WFMS and requirements for real world applications are discussed. Section 2.3, describes the process definition tool, used to specify and analyze workflows. In Section 2.4, an issue concerning workflow enactment service, which takes care of control and execution of workflows, is addressed. In Section 2.5, the motivation for the research focus and specific problem definition is discussed.

2.1 Transaction Models

The concept of transaction models [11, 16] allows an application programmer to write applications without the need to deal with consistency and reliability in presence of failure and concurrent users, since transaction provides the well-known ACID properties [10, 16]. Traditionally, transactions are characterized by simple application logic and short duration activities that typically execute within a few minutes or seconds. *Traditional transactions models* are built on the concept of ACID Properties. Although this concept can be useful to model database applications such as airline reservation systems, banking systems and electronic funds transfers. It has been recognized that the

standard model is too restrictive for many advanced database applications [7, 10]. For example, in a cooperative environment, if long-duration activities are executed as atomic transactions, they may significantly delay the execution of shorter activities. Hence an extension to these models is needed to support the development of multi-system applications or workflows that access heterogeneous databases. Consequently, a number of researchers have attempted to extend the traditional transaction model to support more flexible transaction processing. Examples of such models are Nested Transactions [22], Sagas [9], ConTract [25], and ACTA [5].

A crucial limitation of the extended models is that they have been proposed with specific applications in mind, which limits the applicability of these models. Moreover, these models are geared towards processing entities that are DBMSs (database management systems) that provide transactional management features, not the legacy systems or non-DBMS systems [21]. However, the requirements for real-world applications (large scale multi system executing in heterogeneous, autonomous, distributed environment) involve multiple communication paradigms, humans and legacy application systems, far exceeds the capabilities provided by these products [42]. Furthermore, they support only a little for coordinating independent tasks. Therefore, most of the extended models are not practical [21]. Based on these needs, the concept of workflow management systems was born [17, 19].

2.2 Workflow Management System

Workflow management system (WFMS) is a tool to integrate humans, computer systems, information resources and organizational processes to provide a unified solution [17, 19]. Hence, the requirements of WFMS are more challenging than DBMSs. In WFMS, the database might comprise a part of entire solution; involve other users and application tasks that are non-transactional in nature.

In order to standardize the requirement of real word applications, the workflow management coalition (WFMC), founded in 1993, developed a workflow reference model as shown in Figure 2.1. The model outlines the architectural representation of WFMS. According to workflow reference model, an entire WFMS is centered on a workflow engine, which is responsible for enacting task execution, monitoring workflow state, and evaluating conditions related to inter-task dependencies [40].

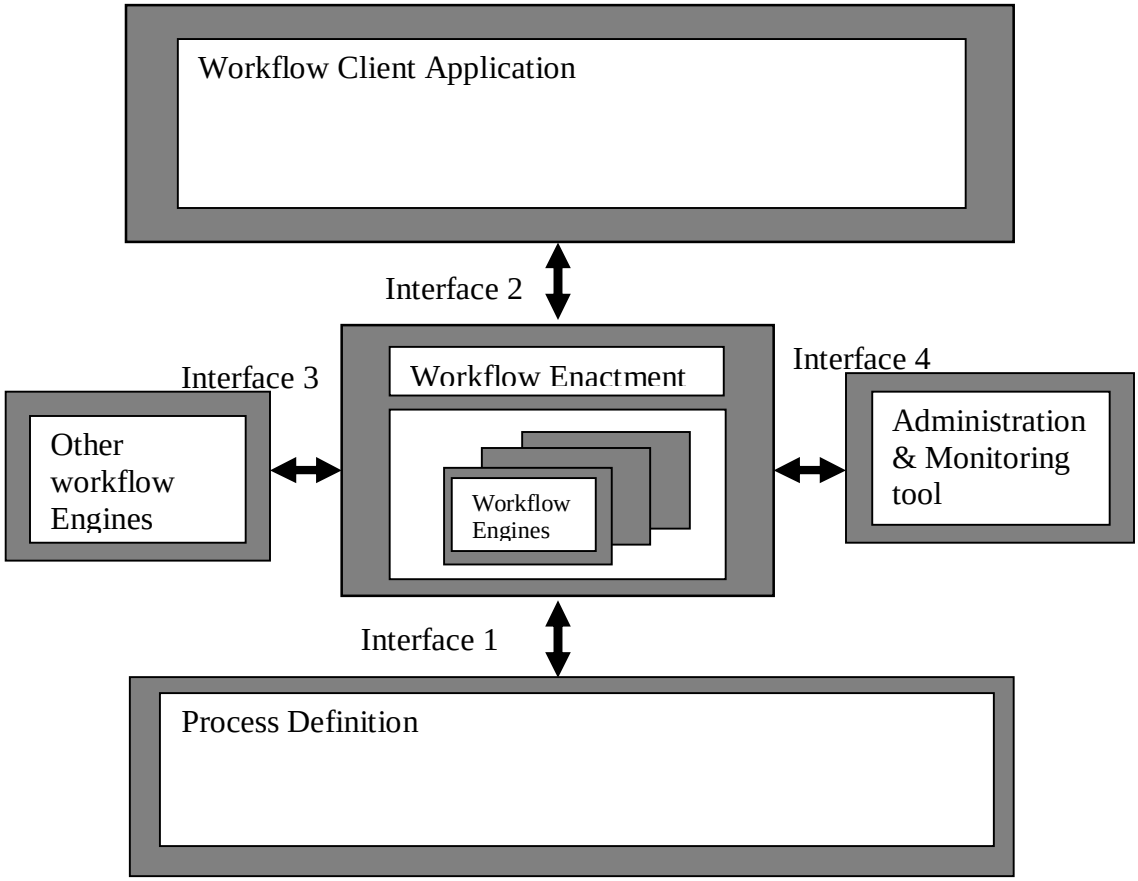


Figure 2.1. Workflow Management System Reference Model [40]

A WFMS consist of several functional components.

- Process definitions tool
- Workflow enactment service

- Administration and monitoring tool
- Interface to interoperate with client application

The process definition tool is used to specify and analyze the workflow process definition. Process definition contains the information regarding the tasks that are to be carried out, the component operation and primitives within the tasks, its starting and completion conditions, rules and dependencies for navigating between tasks [13]. These tools are used at design time. In general, the process definition tool includes the following.

- Formalism for modeling and specification of workflows,
- Specifying the task and information associated with it,
- Specification of business rules (dependencies and constraints),
- Analysis of the workflow model.

The workflow enactment service provides a run-time environment, which takes care of the control and execution of the workflow. In general, execution of workflow includes enforcing all inter-task dependencies and test for workflow safety.

Administration tools provide functions such as managing users, roles and security policy. Monitoring tools are used for tracking and reporting workflow states and data generation during workflow execution. All these components have application interfaces that provide standard means of communication between components and the workflow engine. The scope of this study is limited to the process definition tool and workflow enactment. In Sections 2.3 and 2.4 process definition tool and workflow enactment are described in detail.

2.3 Process Definition Tool

A process is specified using the process definition tool. In the following Sections, we describe each component of process definition tool in detail.

2.3.1 Formal Modeling and Specification of Workflows

A formal specification provides a formal framework for modeling and analysis of workflows, which develops a higher confidence in the correctness of workflows. A number of formal modeling techniques have been proposed [1, 3, 32, 41] of which Petri Nets is considered to be the state-of-the-art. Van der Aalst [31] identifies three reasons for using Petri Nets in workflow modeling. Firstly, Petri Nets possess formal semantics despites their graphical nature. Secondly, instead of being purely event-based, Petri Nets can explicitly model states, and lastly it is a theoretical proven analysis technique.

Other than Petri Nets, technique such as state chart has also been proposed for modeling WFMS [41]. Although state chart can model the behavior of workflow, they have to be supplemented with logical specification for supporting analysis. Singh et al [27] uses event algebra to model the inter-task dependencies and temporal logic. Attia et al [3] have used computational tree logic (CLT) to model tasks by providing their states together with significant event corresponding to the state transitions (start, commit, rollback etc) that may be forcible, rejectable, or delayable.

2.3.2 Task Specification

Some researchers [13, 29, 33, 35, 36] have treated task as a single unit which precludes its ability to specify certain types of dependencies such as weak-causal type, i.e. an activity cannot start before the completion of another activity. However, in [1, 3, 27, 41] each task is decomposed into a number of primitives (begin, done, abort, commit)

and represents states in between these primitives as well. For example, Atluri and Huang [1] modeled task as an ordinary Petri-net as shown in Figure 2.2.

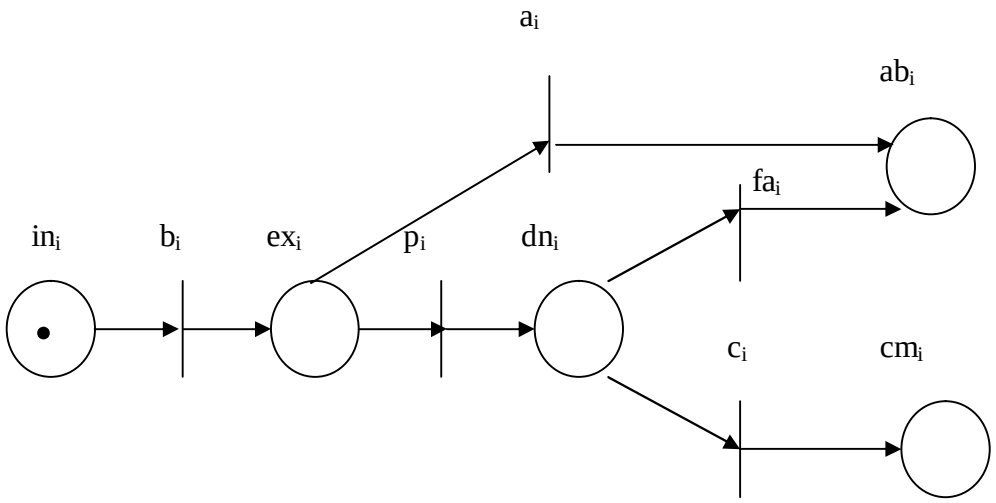


Figure 2.2. A Petri Net Representation of 2PC Task Structure [1]

The events in the task can be controllable or uncontrollable. Atluri, Wodtke, and Singh [1, 27, 41] use the task structure in which all the events are *controllable*. However, Krishnakumar et al and Attia et al [3, 20] consider that the events, which are enabled by user or *processing enti*ties are *uncontrollable* and the events that are enabled by the scheduler are controllable. An *uncontrollable event* is one which cannot be prevented from occurring.

2.3.3 Dependencies

Klein [18] proposed two types of control flow dependencies: order dependencies $e_1 < e_2$, and existence dependencies $e_1 \longrightarrow e_2$. Several researchers have used this to model the workflow system [3, 28].

- Order dependency: $e_1 < e_2$; if e_1 and e_2 both occur, then e_1 must precede e_2. That is, if e_2 occur, then $e_1$cannot occur subsequently.
  Example: Commit Dependency.

- Existence dependencies: $e_1 \longrightarrow e_2$; If event e_1 occurs, then event e_2 must also occur. There is no implied ordering on the occurrences of e_1 and e_2.
  Example: Abort dependency.

These dependencies are also termed as *casual* and *precedence dependencies* [1, 6]. The former specifies a logical implication; the later specifies a precedence constraint. Atluri and Huang [1] further classify casual type dependency into *weak causal* and *strong casual* based on the implied logical relationship as discussed in Section 1.2.

2.3.4 Analysis of Workflow

Analysis in workflows is to check the process definition for errors [34]. As the process definition is so important, it is useful to analyze it thoroughly prior to its enactment. Such analysis can encompass checking the semantic correctness of a process definition as well as performing a simulation in order to gain insight into the process [43].

There are two types of analysis for workflow models.

- Validation, i.e., to check that the model behaves the same as the real system.
- Verification, i.e., to check the logical correctness of a workflow, which is the absence of dead locks and livelocks.

2.3.4.1 Validation

Validation is done by comparing the values form the model with the real system. The values that are compared for validation of the model are performance indicator such as average completion time, level of service, and utilization. Most of workflow management systems use simulation as a tool for validation [13, 29, 34, 36].

2.3.4.2 Verification

Atluri [1] developed an algorithm based on reachability property of Petri Nets, to check for the logical correctness of the process. Whereas others [29, 34, 36] have checked the soundness property for the workflow model, a workflow model is sound if it fulfils the following three requirements.

- For each token put in a place start, one (and only one) token eventually appears in the place end.
- When the token appears in the place end, all the other places are empty; and
- For each transition (task), it is possible to move from the initial state to a state in which the transition is enabled.

The first requirement means that every task is completed successfully over a period of time. The second requirement means that once the task is completed, no reference to it remains in the system. The last requirement excludes "dead tasks".

2.4 Workflow Enactment Tool

A *workflow enactment tool* is the heart of the workflow system. It consists of several functional components, which will be discussed in Section 2.4.1 and 2.4.2.

2.4.1 Enforcing Dependency

In traditional applications, the dependencies between different tasks are encoded in the program, which are enforced automatically during the execution. However, in advanced applications, different tasks should not be grouped into a single program [12]. Thus, all dependencies cannot be encoded directly into an application program; as a result, they have to be specified as an additional constraint on the execution of task, which in turn need a separate enforcement mechanism. The enforcement mechanism, decides whether a task is allowed to enter certain state based on the dependencies. The

task is allowed to enter a particular state only if by doing so will not violate any dependencies [3].

Attie [3] uses event attributes to determine whether a dependency is enforceable or not. Each dependency is modeled as a finite state machine, which is responsible for enforcing the dependency. This can be done manually, or an extension of the computational logic tree (CLT) [8]. Attie [3] only dependencies are specified as finite state machines and not the tasks. As a result, they cannot handle uncontrollable events. Using the concept introduce by Attie [3] more similar research has been reported [14, 27, 28]. Wallace et al [38] also, specify both tasks and dependencies as finite state automata, moreover they have adapted the technique of supervisory control.

2.4.2 Workflow Safety

Safety is an important property in workflow analysis. A workflow is said to be safe if it always terminates in one of the specified acceptable states [1]. In other words, it never terminates in an unacceptable state.

Atluri [1] developed an algorithm that makes use of a reachability property of Petri nets to check for termination in unacceptable states. Van der Aalst [36, 39] has defined task as a single unit. Hence the workflow has only one acceptable state i.e. final task. For a workflow to be safe it should terminate in the final task, which is checked during the verification of workflow. Whereas [3, 28, 38] have used a scheduler in which a task is allowed only if it does not violate any of the dependencies and it's on the legal path. A legal path is one, which leads to one of the acceptable states.

In Sections 2.1 to 2.4 we have reviewed the existing methods for modeling the workflow process. Amongst these methods, Petri Nets is the most used technique for modeling and analysis of workflows. Even though Petri Nets are powerful design tools, they have some limitations, which make them unsuitable for modeling workflow problems. These limitations in context of modeling workflows are discussed in Chapter 3.

Chapter 3
Motivation and Problem Statement

With rapid increases in application domains that use workflow management systems, there is a need for a framework that can be used to implement these applications. In particular it is essential to provide a formal technique for defining a problem that can be used by various workflow product developers. Developing a simple and homogeneous language, based on formal techniques is one way to facilitate the above requirement. In this work, we describe an architecture based on Finite Automata for modeling and analysis of business processes.

Despite efforts by the WFMC [39, 40], there is lack of standardization for workflow management particularly regarding the problem definition aspect. Workflow system developers use their own languages and techniques, for modeling and analysis of workflow processes. There may be several reasons for this; techniques available are not simple enough, limitations of current techniques, inability to address some of the real world requirements.

Due to their graphical nature and their ability to perform analytical computations, Petri Nets are widely used for modeling and analysis of workflows. The analysis methods based on Petri Nets require every reachable state to be examined after the model is developed. Therefore, as the system gets larger and more complex the analysis becomes computationally difficult or even impossible.

In Petri Net based modeling techniques for workflows, both tasks and dependencies are modeled as a single system, which results in a vague specification of the business rules. With tasks and dependencies as a single system it's difficult to model

the system correctly. Furthermore, if there is a small change in the business rules the whole system needs to be remodeled.

Apart from standardization in problem definition and limitations of Petri Net as a tool for modeling workflows, the current workflow modeling techniques do not illustrate the concept of uncontrollable events, which are required for most of the real world applications.

In order to address the fore-mentioned shortcomings of current WFMS modeling formalisms, we propose a modeling approach based on finite automata formalism. Automata are basic class of DES models, which have strong theoretical and practical applications. Automata represent every state explicitly, tasks and dependencies can be modeled separately, they do not require examining of all the states for analysis and it can model uncontrollable events.

3.1 Objectives

The overall goal of this thesis is

- To provide a comprehensive framework for modeling workflow process definitions.

- Within this goal the objectives are,
    - Distinguish between the events enabled by processing entity (uncontrollable events) and workflow controller (controllable events).
    - Represent business policies described by natural languages as formal languages.
    - Provide logically correct and maximally permissible control structure.

Chapter 4
Finite Automata Theory

In this chapter Section 4.1 provides an overview of finite automata theory; Section 4.2 illustrates the concepts of state avoidance and string avoidance; and in Section 4.3 workflow process is framed and modeled as an avoidance problem, with the help of an example; Section 4.4 describes the analysis techniques for workflows; Section 4.5 is a chapter summary.

4.1 Finite Automata

Finite automata (FA) is a formalism used to model discrete event systems. In FA models, all the states and transitions are explicitly represented and the model always resides in one of its finite number of states. The finite automata model is represented by directed graph, in which a node represents a state and an arc represents an event [4].

An FA model can be formally defined by 5- tuple $G = (\sum, Q, \delta, q_0, Q_m)$ Where;

- Σ is a finite alphabet of event labels,
- Q is the set of states q,
- $\delta : \Sigma \times Q \longrightarrow Q$ is the transition function,
- q_o is the initial state,
- Q_m is the set of marked (or final) states, $Q_m \subseteq Q$

4.1.1 Example

Figure 4.1 shows a finite automata consisting of three states (claim, under consideration, ready) and three events (record, pay, send letter). This network models the process for dealing with an insurance claim. As the *claim* is received, it is first recorded, after which either a payment is made or a letter is sent explaining the reason for rejection. The ready state is marked, as it is the final state of the process [43].

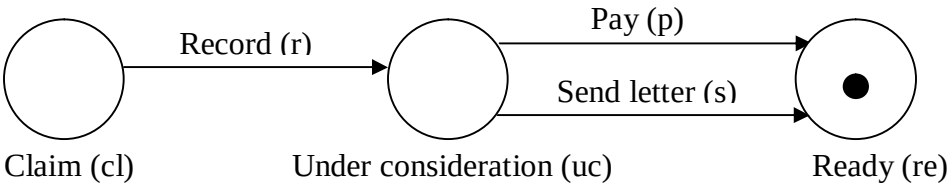


Figure 4.1. Insurance Claim Process

- Σ: {r, p, s}
- Q: {cl, uc, re}
- δ : {(r x cl $\longrightarrow$ uc), (p x uc $\longrightarrow$ re), (s x uc $\longrightarrow$ re)}
- q_o : {cl}
- $Q_{m:}$ {re}

4.2 Avoidance Problem

Avoidance problems are those problems where certain states or events of the system are undesirable and hence needs to be avoided. The business rules (dependencies) in a workflow impose certain restrictions on the behavior of the system. Restrictions imposed include specifying certain states or sequence of events of the system which need to be avoided as they violate some conditions required for the desired behavior of the system. Hence the business rules (dependencies) and avoidance problems are similar as both address the undesirable states and events of the system.

The avoidance problem can be further classified into *state avoidance* and s*tring* (path) *avoidance*. Each will be discussed separately in Sections 4.2.1 and 4.2.2.

4.2.1 State Avoidance

In the state avoidance problem some states of the system are not acceptable as they violate conditions that we wish to impose on the system [2]. These states are termed as *illegal state*. The general idea of state avoidance problems is explained with the help of Figure 4.2.

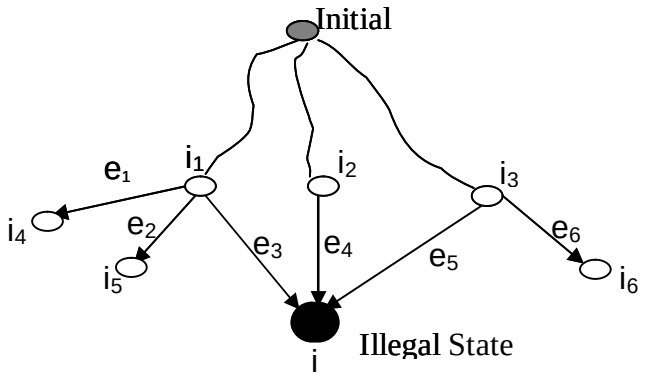


Figure 4.2. State Avoidance [2]

Assume that the state *i* in Figure 4.2 is an illegal state. The illegal state *i* can be reached from *n* number of states by firing one of the events e_3, e_4, and e_5 when the system is in state i_1, i_2 and i_3 respectively. Hence, illegal state *i* should be removed and events e_3, $e_{4,}$ & e_5 that take the system to illegal state *i* should be disabled at the states i_1 , i_2 and i_3.

4.2.2 String Avoidance

In string avoidance problems some strings of the system are not acceptable as they violate conditions that we wish to impose on the system [2]. These strings are termed as *illegal string*. The general idea of string avoidance problem is presented in Figure 4.3.

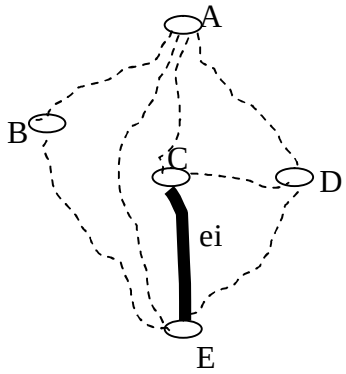


Figure 4.3. String Avoidance

In Figure 4.3, state *E* can be reached from state *A* through a number of strings (ACE, ABE, ADE, ACDE and AE) i.e. following a sequence of events. The objective of string avoidance is to avoid a string that contains an illegal event (e_i). The illegal event e_i is shown with a bold line in Figure 4.3. All the dotted strings are acceptable. Hence the event e_i is disabled.

4.3 Modeling Workflow Specifications

A workflow is a set of tasks and inter-task dependencies (business rules). Each task in a workflow specifies some dependencies between its events, and business rules add to this a set of dependencies between events of different tasks. Each task is modeled as finite automata; these automata are then shuffled to obtain all the reachable states of the workflow model G [2].

Shuffling operation is a cross product of all the states of all the automata (tasks) to describe the overall behavior. That is, for a system with k automata each having n_i states, $i = 1, \ldots k$ the number of states of the combined system after shuffling is $\prod_{i=1}^{k} n_i$.

$$G_1 || G_2 \ldots || G_n = G$$

Based on the inter-task dependencies we identify *illegal states* and *illegal events*, which are then, removed from the workflow model to get the controlled system which is called as *Generator* model.

In general modeling workflow as state avoidance and string avoidance problem, there are three major steps.

- Construct a workflow model,
- Identify the *illegal states* and *illegal events* (based on the business rules),
- Remove all the *illegal states* and disable all the *illegal events.*

In the next Section a practical example is presented to illustrate the technique of state avoidance and string avoidance.

4.3.1 Example: **Airline Example**

Consider a travel agency that processes requests for airline and hotel reservations as shown in Figure 4.4. Once the flight reservation is made it cannot be canceled, but cancellation of a hotel is allowed. There are three tasks involved in this workflow.

- Task 1 - Purchasing an airline ticket (t_a),
- Task 2 - Booking a hotel (t_h), and
- Task 3 - Cancel a hotel reservation (t_h).

Based on the booking regulation, traveler's preferences, or economic reasons, certain constraints are defined between tasks in terms of following dependencies.

- Booking of hotel cannot start until purchasing an airline ticket starts (t_a BD t_h).
- If hotel booking aborts then purchasing airline ticket must aborts too (t_h AD t_a).

- Certain restriction or combinations of airline and hotels are preferred. For example, flight U offers a discount rate with hotel H, and flight N offers an upgrade package with hotel M. which in turn imposes the following constraint: purchasing a airline ticket must commit before that of hotel booking if both commit (t_a CD t_h).
- If purchasing of airline ticket aborts but hotel booking commits, then hotel booking has to be canceled. (t_a BAD (t_h BAC t_c)).

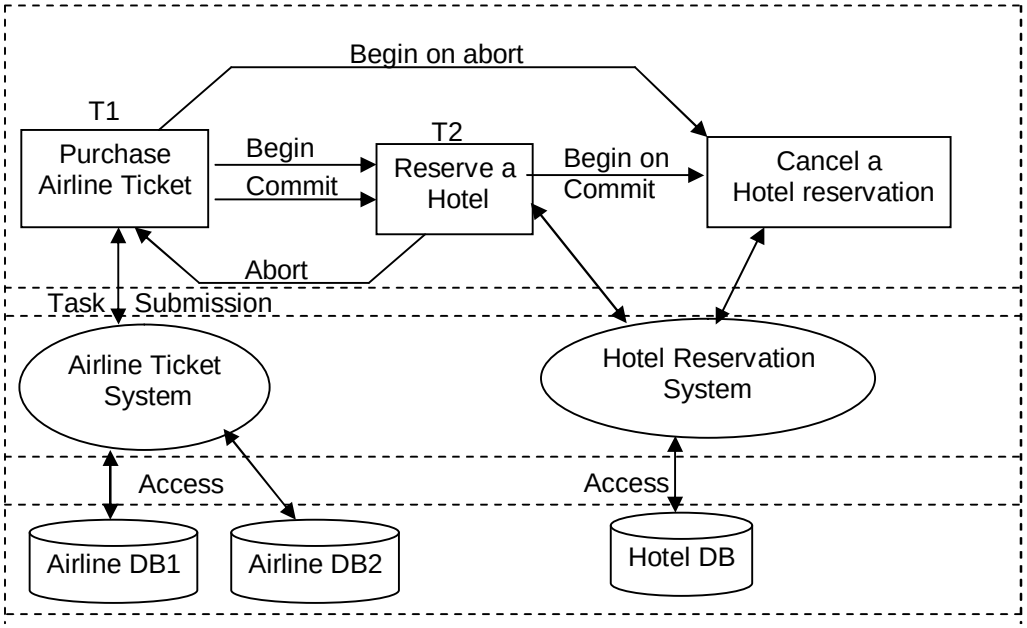


Figure 4.4. Travel Agency Systems [27]

4.3.2 Workflow Model

A task begins with a start event *st* and terminates with commit event (*c)*, an abort event (*a)* or it doesn't start. There is also a pre-commit event (*pc)* that precedes commit and abort events. Since the states following the terminating events and initial state are final states, they are marked. Both tasks t_a and t_h are modeled as separate automata G_a and G_h as shown in Figure 4.5 and these automata are then shuffled to get a workflow model $G_{ah} = G_a \,||\, G_h$ in Figure 4.6.

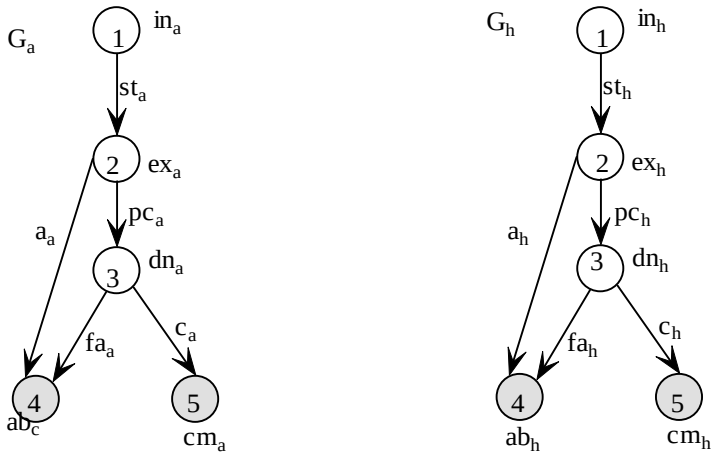


Figure 4.5. Individual Task Automata

Airline task

- Σ: {st_a, pc_a, a_a, fa_a, c_a}
- Q: {in_a, ex_a, dn_a, ab_a, cm_a}
- d : {(st_a x $in_a \longrightarrow ex_a$), ($pc_a$ x $ex_a \longrightarrow dn_a$), ($a_a$ x $ex_a \longrightarrow ab_a$), ($c_a$ x $dn_a \longrightarrow cm_a$), ($fa_a$ x $dn_a \longrightarrow ab_a$)}
- q_o : {in_a}
- $Q_{m:}$ {in_a, ab_a, cm_a}

Hotel task

- Σ: {st_h, pc_h, a_h, fa_h, c_h}
- Q: {in_h, ex_h, dn_h, ab_h, cm_h}
- d : {(st_h x $in_h \longrightarrow ex_h$), ($pc_h$ x $ex_h \longrightarrow dn_h$), ($a_h$ x $ex_h \longrightarrow ab_h$), ($c_h$ x $dn_h \longrightarrow cm_h$), ($fa_h$ x $dn_h \longrightarrow ab_h$)}
- $q_{o:}$ {in_h}
- $Q_{m:}$ {in_h, ab_h, cm_h}

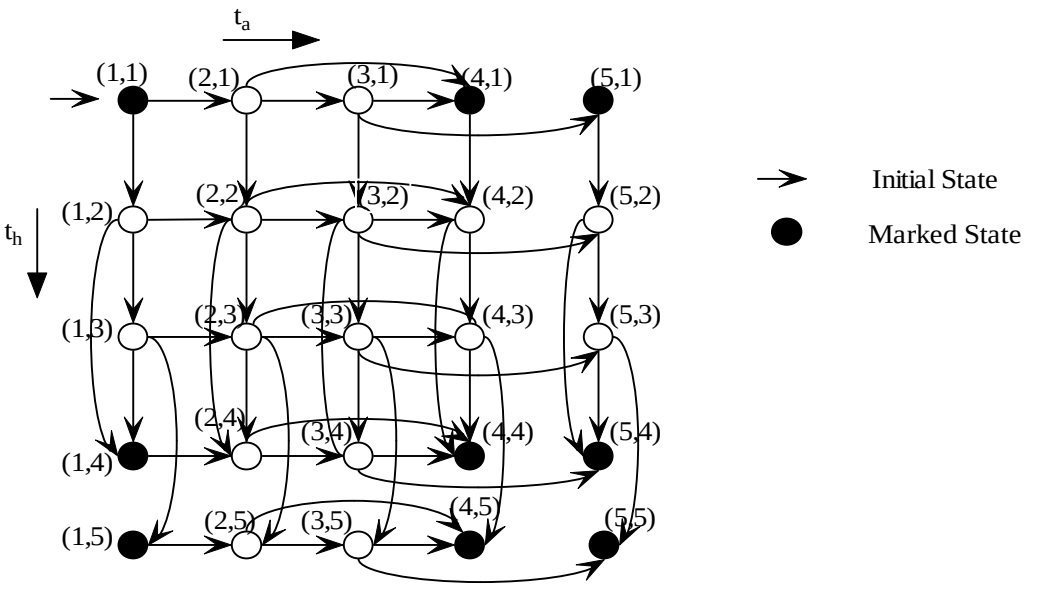


Figure 4.6. Workflow Model G_{ah}

4.3.3 Identifying Illegal States and Illegal Events

Strong-causal and weak-causal dependencies specify incompatible states as described in Section 1.2, where as precedence dependency specifies a precedence order. Based on this specification, we classify and model strong-causal and weak-causal dependencies as state avoidance problems and precedence dependency as string avoidance problem.

For strong-causal dependency the incompatible (illegal) states is (s_i, s_j') and for weak-causal dependency the incompatible state is (s_i', s_j) and state $(s_i \quad b_j)$ is not an acceptable terminating state. Whereas to identify the illegal events for precedence dependency we define a pair (s_j, e_i) where s_j is the state of task j and e_i is the illegal event when precedence order is $s_i' <= s_j$. The complete list of e_i and its corresponding s_j is shown in Table 4.1.

Table 4.1. e_i and Corresponding s_j

e_i	st_i pc_i c_i a_i, fa_i
s_j	ex_j dn_j cm_j ab_j

- Dependency 1: *Begin Dependency (BD):* task t_h cannot begin execution until task t_a has begun.
- *Incompatible or illegal state set*: (s_a', s_h) task t_a is in initial state and task t_h is in execution state (1, 2).
- Dependency 2: *Abort Dependency (AD)*: if task t_h aborts then task t_a aborts.
- *Incompatible or illegal state set*: (s_a, s_h') task t_h is in abort state and task t_a is in commit state (5, 4) and state $(s_h \ b_a)$ is not an acceptable terminating state.
- Dependency 3: *Commit Dependency (CD)*: if both task t_a and t_h commit then the commitment of t_a precedes the commitment of t_h
- *Illegal event*: (s_h, e_a) task hotel is in done state and has an illegal event commit. That is, (event c_a from state (3, 5)).

4.3.4 Removing Illegal States and Disabling Illegal Events

In this step we remove illegal states, illegal events, all events leading to and from illegal state and unmark the states that are not acceptable terminating state, from the workflow model G_{ah} *to* get the generator model G_{ahg} as shown in Figure 4.7.

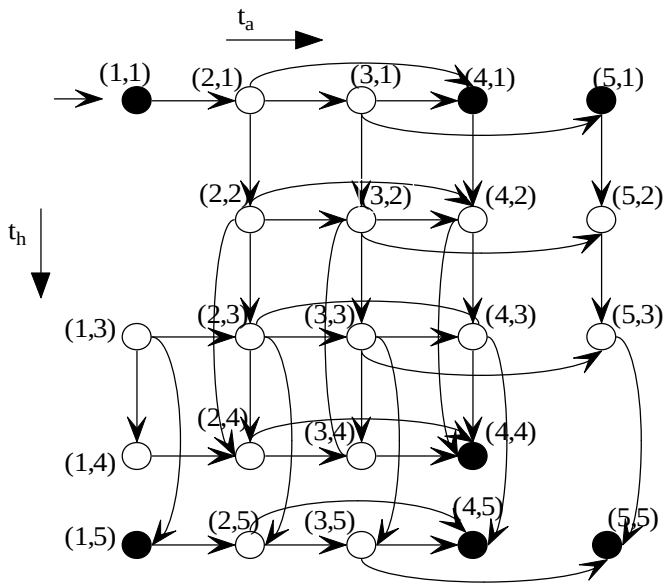


Figure 4.7. Generator Model

In Sections 4.1 to 4.3 we have seen how to model the process definition using state avoidance and string avoidance techniques based on finite automata. A process definition of a workflow is a blueprint of a business process, so it is vitally important that it does not contain errors. For example in the above model the workflow cannot reaches state {(1, 3), (1, 4), (1, 5)} from initial state, which is undesirable. Hence it is useful to analyze the process definition prior to its enactment. The next Sections highlight some of the analysis techniques, which are useful in the context of analysis of workflows.

4.4 Analysis of Workflow Model

For workflow models, the following types of analysis are performed: (1) check for the logical correctness of the model, i.e. deadlocks and livelocks; (2) identifying inconsistency in the dependency specifications; (3) test for workflow safety i.e. to check whether the workflow terminates in one of the acceptable final states [1].

In Section 4.4.1 we introduce a simple technique to check for the logical correctness of the model i.e. absence of deadlocks; In Section 4.4.2 we turn our attention

to check for the errors that can be made while the defining business rules i.e. checking for inconsistent dependencies; Sections 4.4.3 shows how finite automata facilitates some properties for the business process i.e. workflow safety.

4.4.1 Logical Correctness of the Model

When a workflow model reaches an unmarked state such that no further events can be executed, we say that the system is deadlocked because it enters an absorbing state without terminating the current task. Another issue is when there is a set of states in G that forms a strongly connected component (i.e. these states can reach from one another) and no transitions going out of the set. If the system enters this set of states then we get what is called a livelock. Hence to check for the presence of deadlock and livelock, we check the *trim* property for the generator model. If the generator model is *trim* then the model is free from deadlocks and livelocks [4]. If the generator model is not trim then we calculate the trim generator to get the deadlock free model. The following definitions are necessary in determining the trim property of the generator model.

- *Definition 1: Accessible states set*: $Q_a = \left\{ q \in Q \middle| \left(\exists \omega \in \Sigma^*\right) \delta(\omega, q_0) = q \right\}$, i.e. the set of all the states that can be reached from the initial state is called the *accessible states subset* [2].
- *Definition 2: Co-accessible state set*: $Q_{ca} = \left\{ q \in Q \middle| \left(\exists \omega \in \Sigma^*\right) \delta(\omega, q) \in Q_m \right\}$, i.e. the set of all the states q from which some marked state can be reached is called the *co-accessible states subset* [2].
- *Definition 3: Trim*: The generator G is *trim* if it is *accessible* (i.e. $Q=Q_a$) and *co-accessible* (i.e. $Q=Q_{ca}$).

Example: Figure 4.8 shows a DES model, where state q_0 represents initial state and state q_1 represents the marked state [13].

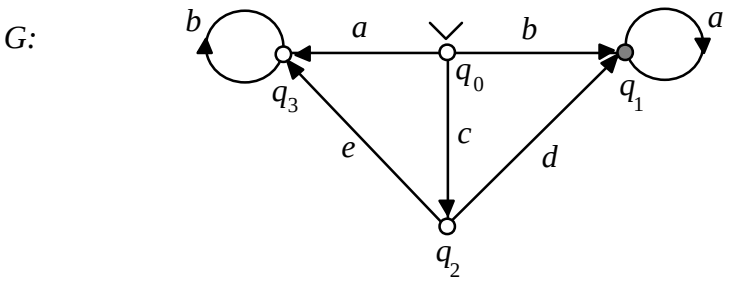


Figure 4.8. DES Model [2]

For the above example: $Q = \{q_0, q_1, q_2, q_3\}$, $Q_a = \{q_0, q_1, q_2, q_3\}$, $Q_{ca} = \{q_0, q_1, q_2\}$. Since Q is equal to Q_a but not Q_{ca}, G is not trim.

To check the trim property for the airline example refer to Figure 4.9 below. All the states that are red in color are both *accessible* and co-accessible, where as states that are blue in color are *co-accessible*.

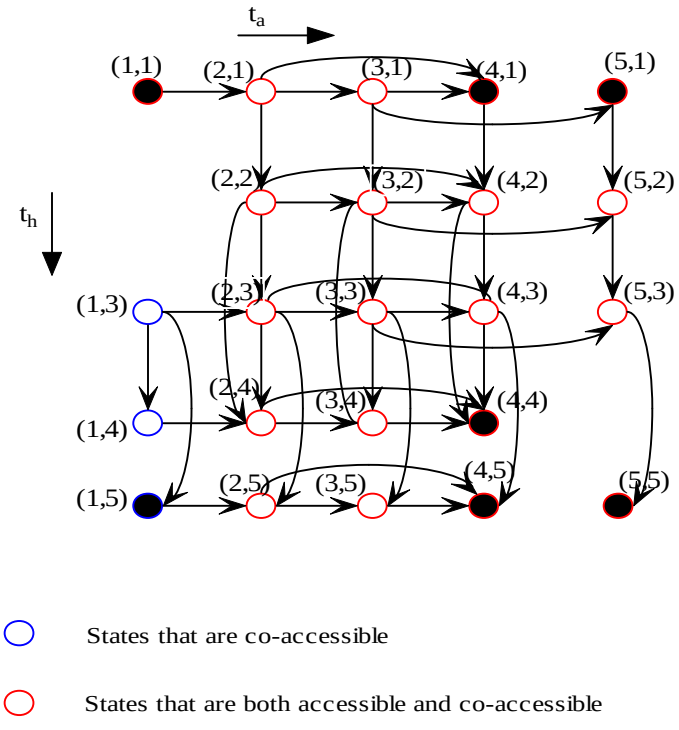


Figure 4.9. Generator Model G_{ahg}

Q = {(1, 1), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}

Q_a= {(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}

Q_{ca}= {(1, 1), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)}

Since Q ¹ Q_a ¹ Q_{ca}, $G_{a\text{hg}}$ is not trim; hence there is a deadlock or livelock in the system. To get the deadlock free model we find a trim generator G_t. A trim generator can be obtained by replacing Q with $Q_t = Q_a \cap Q_{ca.}$

Q_t = {(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}∩ {(1, 1), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)}

Q_t ={(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}

A trim generator consists of states that are both accessible and co-accessible (states with red color in Figure 4.9). A trim generator for the airline example is shown in Figure 4.10.

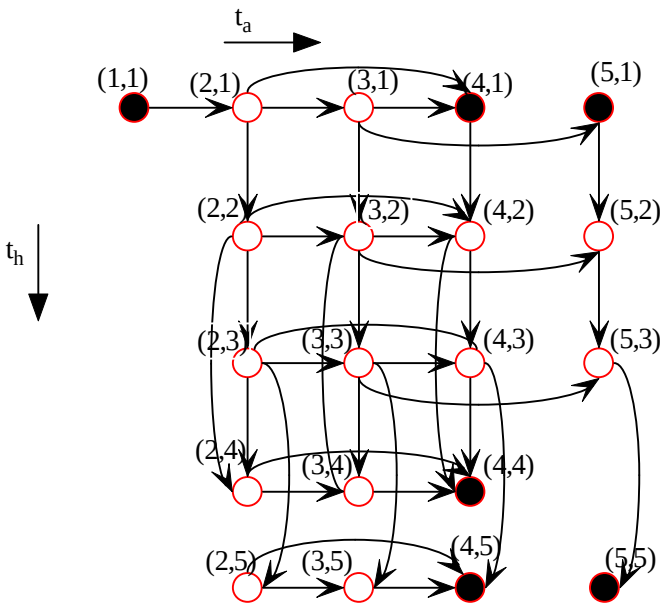


Figure 4.10. Trim Generator G_t

4.4.2 Inconsistent Dependency Specification

Inconsistency is a conflicting state of the system, which results because of contradicting dependencies. The inconsistency may be due to the following two reasons. (1) Customization of workflows based on user preferences. (2) The workflow specifications may change, e.g. designer may add new dependencies or delete some existing ones [1]. Below is an example to depict the inconsistencies:

In the airline example we have abort dependency, which states if hotel booking (t_a) aborts, airline ticket must abort (t_h). If add two more dependencies.

- if airline tickets aborts then train reservation begins (t_t),
- if train reservation commits (t_t) then hotel reservation must commit (t_h).

According to first dependency, if (t_h) aborts (t_a) aborts and third dependency specifies that (t_h) must commit even if (t_a) aborts, which is inconsistent. Therefore, in

order to avoid inconsistency in the dependency specification it is important to check, sets of dependencies before the enactment of business process.

In Section 4.4.2.1 we provide a methodology for checking inconsistent dependency specification; In Section 4.4.2.2 we check and the identify type of dependency that is inconsistent.

4.4.2.1 Formalism for Checking Inconsistency

We define the following terms before we provide the methodology for how to check for inconsistent specifications.

- *Definition 4:* For strong-causal dependency between task t_i and t_j an incompatible state (*iss*) is defined as ($s_i^{'}, s_j$) and resultant state (*rss*) is defined as (s_i, s_j).
- *Definition 5:* For weak-causal dependency between task t_i and t_j an incompatible state (*iss*) is defined as ($s_i, s_j^{'}$) and resultant state (*rss*) is defined as (s_i, s_j).
- *Definition 6:* For precedence dependency between task t_i and t_j an resultant state (*rss*) is defined as (s_i, s_j) and illegal event (*ies*) is defined as (e_i).
- *Definition 7: Incompatible state set (ISS (W))*: In a workflow, *ISS (W)* is the set of all incompatible states (*iss*).
- *Definition 8: Resultant State Set (RSS (W)*): In a workflow, *RSS (W)* is the set of all resultant states (*rss*).

In general identifying inconsistent dependencies is a three-step procedure.

- S*tep1*: Determine the *trim* model of the workflow,
- *Step2*: Check the model for inconsistent dependency specification,
- S*tep3*: If the dependencies specified are inconsistent, identify the types of dependencies that are inconsistent.

Step 1. The procedure to check for trim and how to get the trim model is explained in Section 4.6

Step 2. To check the model for inconsistent dependency specification, we check the following condition, if it's satisfied then all the dependencies specified are consistent.

- $\forall\ rss : rss \in Q_t$

Step 3. To identify the type of dependency that is inconsistent, we check the following conditions.

- There exists at least one incompatible-state, such that $iss \in RSS\ (W)$, If this condition is satisfied, then there is inconsistent specification due to weak-causal dependency.
- There exists at least one resultant state, such that $rss \in Q$ but $rss \notin Q_t$, If this condition is satisfied, then there is inconsistent specification due to strong-causal or precedence dependency.

Note: In the airline example explained in Section 4.3 all the dependencies specified are consistent i.e. there is no inconsistency in the dependency specification.

4.4.2.2 Checking for Inconsistent Workflow Specification

Consider an airline example with additional dependencies specified in Section 4.4.2, which states that if airline ticket aborts (t_a) hotel booking commits (t_c). Due to the additional dependency there is one more incompatible state (4, 4), so we remove state (4, 4) from the generator model of airline example as shown in Figure 4.7. The generator model for airline example with additional dependency i.e. after removing state (4, 4) is shown in Figure 4.11.

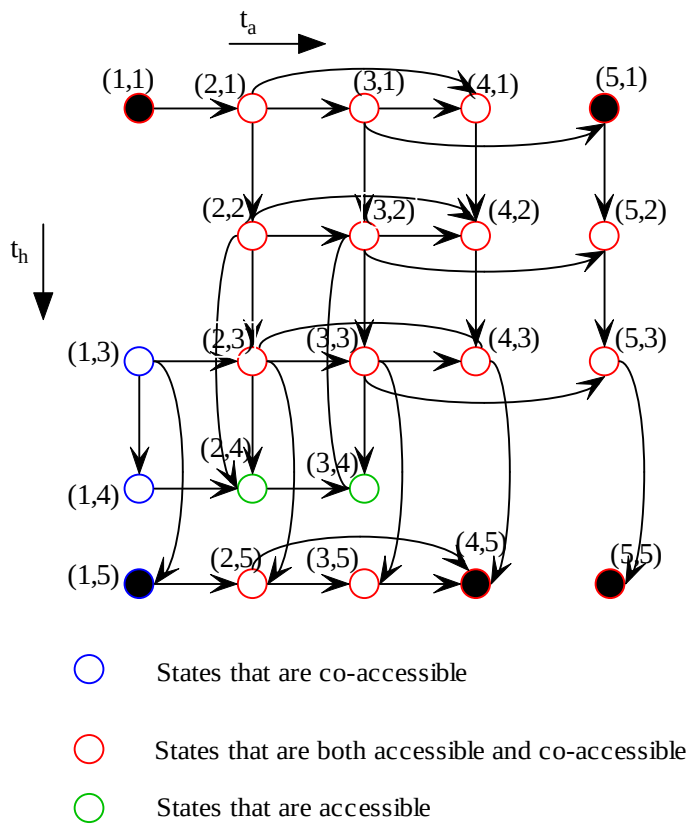


Figure 4.11. Generator Model G_{ahg} with Additional Dependencies

Step 1: In this step we first check the trim property for generator model G_{ahg}, if its *trim* that means model is deadlock free. If the generator model G_{ahg} is not *trim* then we find the *trim* generator to calculate the deadlock free model.

Q = {(1, 1), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}

Q_a= {(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}

Q_{ca}= {(1, 1), (1, 4), (2, 1), (2, 2), (2, 3), (2, 5), (3, 1), (3, 2), (3, 3), (3, 5), (4, 1), (4, 2), (4, 3), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)}

Since $Q \neq Q_a \neq Q_{ca}$, G_{ahg} is not trim; hence there is a deadlock in the system. Now to calculate the deadlock free model we find a trim generator G_{tac}.

Q_t = {(1, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}∩ {(1, 1), (1, 4), (2, 1), (2, 2), (2, 3), (2, 5), (3, 1), (3, 2), (3, 3), (3, 5), (4, 1), (4, 2), (4, 3), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5)}.

Q_t = {(1, 1), (2, 1), (2, 2), (2, 3), (2, 5), (3, 1), (3, 2), (3, 3), (3, 5), (4, 1), (4, 2), (4, 3), (4, 5), (5, 1), (5, 2), (5, 3), (5, 5)}.

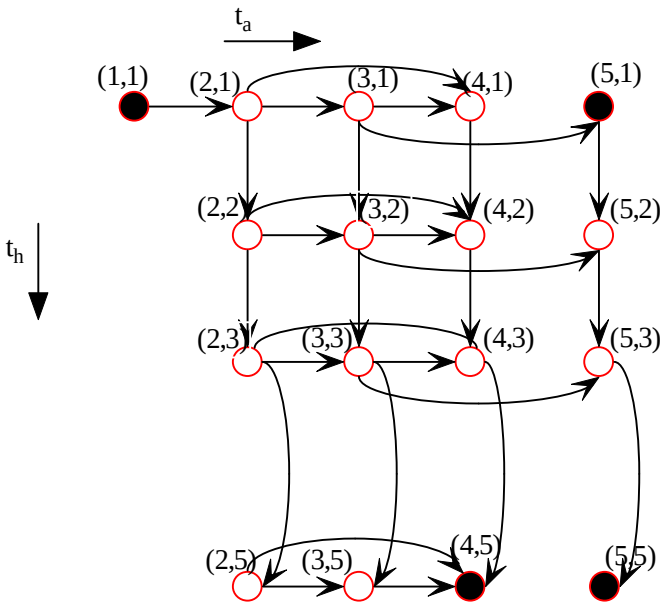


Figure 4.12. Trim Generator with Additional Dependencies

Step 2: In this step we first find *RSS (W)* and then check the condition for consistency.
Additional Dependency: if airline ticket aborts (t_a) hotel booking commits (t_c)

Resultant State (rss): (4, 5).
RSS (W): {(2, 2), (4, 4), (5, 5), (4, 5)}

Note: States (4, 4), of *RSS (W)* is not a member of set Q_t, hence there is an inconsistent dependency specification.

Step 3: In this step we first determine ISS (W) and IES (W) and then check the conditions to identify the type of inconsistent dependency.

Incompatible state set (ISS (W)): {(5, 4), (1, 2), (4, 4)}
Illegal Event set IES (W): {(event c_a from state (3, 5))}
Resultant State Set (RSS (W)): {(2, 2), (4, 4), (5, 5), (4, 5)}

Note: Incompatible states (4, 4) ∈ *RSS (W),* hence according to the first condition Section 4.4.2 we can say that the inconsistency is due to weak causal dependency.

4.4.3 Testing for Safety

A workflow is said to be safe if it always terminates in one of the specified acceptable states [1]. In other words, it never terminates in an unacceptable state. An unacceptable state is either an illegal state or state which is not marked. That is, the workflow terminates in an incompatible state or it has terminated without completing all the tasks.

Hence to check for proper termination, we check the following condition and if both the conditions are satisfied then we say workflow is safe.

- Condition 1: $iss \notin Q_g$. $\forall iss \in ISS(W)$.
- Condition 2: Generator model is deadlock free (*trim*).

Condition 1: In state and string avoidance technique, all the incompatible states are removed from the generator model G_g as explained in Section 4.3.3 hence the incompatible state will not be a member of Q_g.

Condition 2: The procedure for checking deadlock and to calculate the deadlock free generator model is explained in Section 4.4.1. Hence by using the technique of state avoidance and string avoidance, we can say that if the workflow is trim it is also safe.

4.5 Chapter Summary

In state avoidance and string avoidance techniques, each task is modeled as finite automata; these automata are then shuffled to obtain the workflow model. Based on the dependencies (business rules) undesirable states and strings are identified and by removing the undesirable states and strings for the workflow model, acceptable behavior of the workflow is determined. But in this technique there is no mechanism that enforces the correctness conditions i.e. enables or disables the events in the workflow model, based on dependencies to determine the acceptable behavior.

In state avoidance and string avoidance techniques we have considered that all the events are controllable. But this is not true for all the workflow applications, if the event requires a processing entity then that event is uncontrollable as describe in Section 1.1.

To illustrate the effect of uncontrollable events, we will use the airline example with only one dependency (Abort dependency). According to the task structure describe in Section 1.1, events pc_a and a_a are executed by the processing entity, hence uncontrollable. Figure 4.13 shows two task, namely, airline reservation and hotel booking with uncontrollable events and Figure 4.14 shows the workflow model.

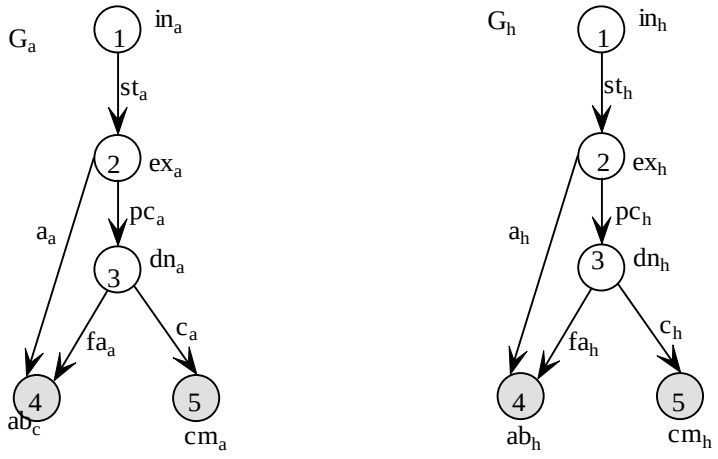


Figure 4.13. Task with Uncontrollable Events

Airline task

- Σ_c: {st_a, fa_a, c_a} $\Sigma_{u:}$ {pca, a_a}

Hotel task

- Σ_c: {st_h, fa_h, c_h} $\Sigma_{u:}$ {pc_h, a_h}

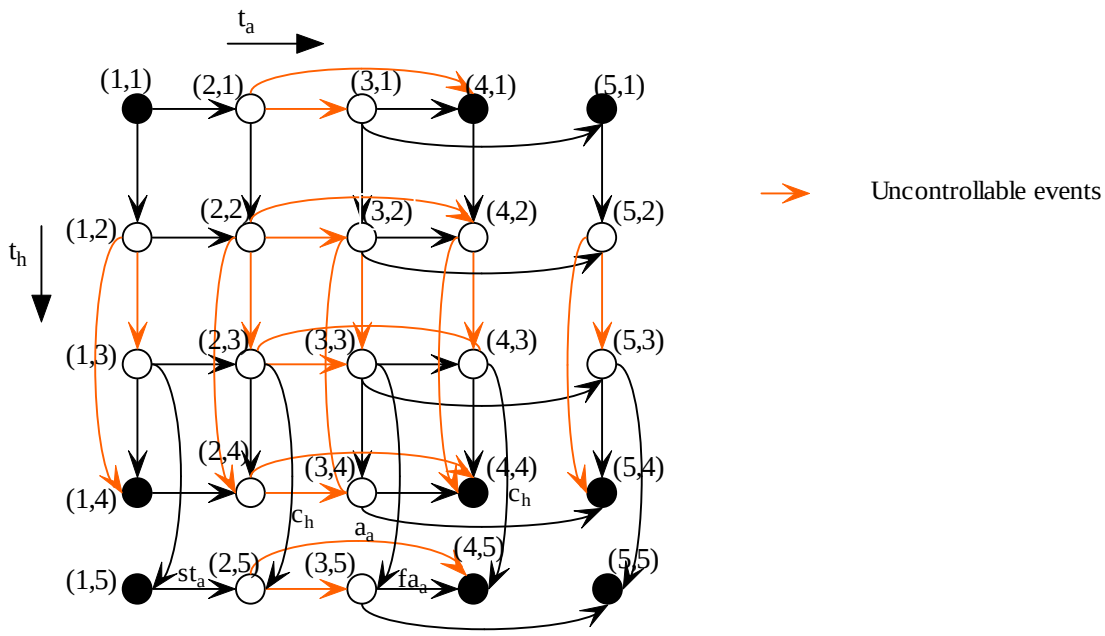


Figure 4.14. Workflow Model G_{ah} with Uncontrollable Events

- *Abort Dependency (AD)*: if task t_a aborts then task t_h aborts,
- *Incompatible or illegal state set*: (s_a, s'_h) task t_a is in abort state and task t_h is in commit state (4, 5).

To determine the controlled behavior (generator model) we remove illegal state (4, 5) and disable events a_a, fa_a, c_h that leads to state (4, 5). But event a_a is uncontrollable so it cannot be disabled, hence to calculate the controlled behavior the workflow model should be prevented from reaching the state (2, 5) from which event a_a is generated. So to prevent the workflow from reaching state (2, 5) event c_h, st_a needs to be disabled.

The point what we want to illustrate is that, even for a simple example with single dependency, uncontrollable events can very rapidly complicate finding the desired model. Hence formal methods are required to solve complex problems with uncontrollable events.

In avoidance techniques only tasks are modeled as automata and not the dependencies. The dependencies are specified in everyday language and based on this illegal state and illegal strings are identified, which preclude the ability to calculate the desired models automatically.

In order to address the fore-mentioned issues, we use *supervisory control theory* for modeling and analysis of workflows. In *supervisory control theory* both controllable and uncontrollable events are distinguish and control-flow dependencies can be modeled as finite automata, which can be directly coupled with uncontrolled process model to calculate the desired model.

Chapter 5
Supervisory Control Theory

Supervisory control theory has two distinct entities: the uncontrolled process model and the supervisor. The uncontrolled process model is any system that needs to be controlled, where as a *supervisor* is an agent that enforces correctness conditions, by enabling or disabling the controllable events in the uncontrolled process model [23, 24]. The basic idea of Supervisory control theory can be understood with the help of Figure 5.1 [2].

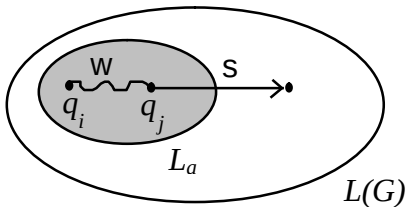


Figure 5.1. Supervisory Control Theory

The uncontrolled process model generates a set of sequence of events. This set is referred as the language generated by the uncontrolled process model L(G) as shown in Figure 5.1. Once a set of control specifications C are imposed on the uncontrolled process model, some of the sequences in L(G) will no longer be acceptable i.e. the system is now required to operate in the shaded region in Figure 5.1. Language L_a is the admissible or desired language for the system. For example, the sequence $\omega\sigma$ is in language L(G) but it is not acceptable with respect to language L_a . Hence to keep the generated sequence of events within the shaded region of Figure 5.1, event σ needs to be disabled when the system is at state q_j. It is the function of the supervisor to disable event σ to keep the system behavior within the boundaries of the specification language L_a [2].

In this chapter Section 5.1 provides formal definitions and an overview of supervisory control theory; Section 5.2 illustrates the technique of supervisory control theory to model workflow processes.

5.1 Formal Definition

A symbol σ represents an event in a system. A string ω is a finite sequence of events that take place in a system. An alphabet Σ is a finite set of symbols. A language L is a set of strings from some alphabet. A finite automaton is formally defined by a five-tuple:

$$G= (\Sigma, Q,\delta, q_o, Q_m),$$

Where Q is the set of states q, Σ is a finite alphabet of events, $\delta: \Sigma \times Q \rightarrow Q$ is the transition function, $q_o \in Q$ is the initial state and $Q_m \subseteq Q$ is the set of marked (final) states.

- $L(G) = \{\omega | \ \omega \in \Sigma^*, \delta(\omega,q_0) \in Q$ is defined$\}$, where ω is a string and Σ^* is the set of all strings over the alphabet Σ. In other words L (G) is the set of all possible sequences of events (strings) which take the initial state to some reachable state in Q.

- $L_m(G) = \{\omega | \ \omega \in \Sigma^*, \delta(\omega,q_0) \in Q_m \}$. $L_m(G)$ is the marked language generated by G which represents the sequence of events that take the initial state to some marked (final) state, Q_m.

In supervisory control theory [23], the uncontrolled process behavior is described by a finite automaton that is modified to accept controls. This is done by defining a set of events, $\Sigma_c \subseteq \Sigma$, which accept control. Those events that are uncontrollable are represented by Σ_u, $\Sigma = \Sigma_u \cup \Sigma_c$ and $\Sigma_u \cap \Sigma_c = \emptyset$. A supervisor is an agent that enables or disables controllable events in the uncontrolled process model such that the language generated satisfies some specifications. Formally the supervisor consists of a finite automaton S and

an output function Ψ(control pattern). *S*=(S, Ψ), where: S= (Σ, X, ξ, xo, X_m) and ψ:Σ x X→(0:disable, 1: enable) such that ψ(σ,x)= 0 or 1 if $\sigma \in \Sigma_c$ and 1 if $\sigma \in \Sigma_u$.

The supervisor and the uncontrolled process model are coupled to form a closed loop system. Assume that at a given time the uncontrolled process model is in state q_i and the supervisor is in state x_j. An event $\sigma \in \Sigma$ can occur in the uncontrolled process in state q_i. According to the state x_j only a subset of the $\sigma \in \Sigma_c$ may be permitted by the supervisor based on the control patternψ. This concept of a closed loop system is illustrated in Figure 5.2 [23].

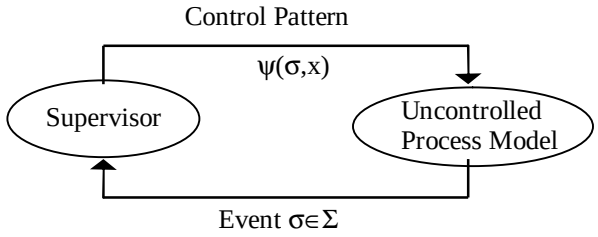


Figure 5.2. Closed Loop Coupled System

The closed loop system is denoted by S/G. Two languages generated by S/G are of interest.

- L(*S* /G)= L(*S*) ∩ L(G) is the sequences of events of the closed loop system,
- L_m(*S* /G)= L_m(*S*) ∩ L_m(G) is the marked sequences of events of the closed loop system.

The basic problem in supervisory control is to restrict the behavior of the uncontrolled process model G in order to stay inside the admissible behavior as shown in Figure 5.1. The basic supervisory control problem and the solution to the basic supervisory control problem is formally discussed in Section 5.1.1.

5.1.1 Basic Supervisory Control Problem (BSCP)

Consider DES G with event set E, uncontrollable events set $E_{uc} \subseteq E$, and admissible marked language $L_a = \overline{L}_a \subseteq L(G)$, find a supervisor *S* such that [4].

- Condition 1: $L(S/G) \subseteq L_a$
- Condition 2: $L_m(S/G)$ is maximally permissible

Where, $\overline{L}_a$ is the prefix closure of L_a. The prefix closure of any language *L* is the language denoted by $\overline{L}$ and consisting of all the prefixes of all the strings in *L* [4].

5.1.1.1 Solution of BSCP

The solution for basic supervisory control problem according to the *Controllability Theorem* (CT) is to choose S such that [4]:

$$L(S/G) = L_a^{\uparrow C}$$

as long as $L_a^{\uparrow C} \neq \varnothing$. *S* can be realized by building a recognizer of $L_a^{\uparrow C}$ that is an automaton whose marked language is $L_a^{\uparrow C}$. The recognizer for $L_a^{\uparrow C}$ can be build from L_a by using the algorithm for *Computation* of $\uparrow C$, which is provided in Appendix (B) [4].

If we obtain $L_a^{\uparrow C} = \varnothing$, then this is not an allowed control behavior. This means that there exist a string of uncontrollable events from the initial state of G that does not belong to L_a.

5.1.1.2 Controllability Theorem (CT)

Theorem 1 [4]**:** Consider DES $G = (E, Q, \delta, x_0, x_m)$ where $E_{uc} \subseteq E$ is the set of uncontrollable events. Let the language $L_a^{\uparrow C} \subseteq$ L(G) where $L_a \neq \emptyset$. There exists a supervisor S such that for G such that:

$$L\ (S/G) = L_a^{\uparrow C}$$

If and only if the following condition holds:

- $L_a^{\uparrow C}\ \Sigma_u \cap L\ (G) \subseteq L_a^{\uparrow C}$

Proof **:** For the proof refer to [4].

In the basic supervisory control problem (BSCP) *blocking* is not considered. A system is said to be blocked if it reaches a state from where no further events can be executed i.e. If a uncontrolled process model G reaches a state x where $\delta\ (x) = \emptyset$ and $x \notin X_m$. We can also say that the system is blocked because it enters a deadlock state without having terminated the task at hand. Another issue is when uncontrolled process model G enters a set of states that forms a strongly connected component and no transitions going out of the set. Since there is no way out of this set, the system is blocked due to livelock. Hence for a system that requires nonblocking behavior, blocking must be considered in the supervisory control problem.

In basic supervisory control problems when blocking is of concern, the admissible behavior is obtain from $L_m(G)$ and it is required that a supervisor is nonblocking. The admissible behavior for basic supervisory control problem where blocking is of concern is denoted by L_{am} as it is a sublanguage of marked languages [4].

5.1.2 Basic Supervisory Control Problem – Nonblocking Case (BSCP-NB)

Consider DES G with event set E, uncontrollable events set $E_{uc} \subseteq E$, and admissible marked language $L_{am} \subseteq L_m(G)$, with L_{am} assumed to be $L_m(G)$ closed, find a nonblocking supervisor *S* such that [4].

- Condition 1: $L_m(S/G) \subseteq L_{am}$
- Condition 2: $L_m(S/G)$ is maximally permissible

5.1.2.1 Solution of BSCP-NB

According to *Nonblocking Controllability Theorem (NCT)*, the solution to the supervisory control problem when the behavior required is nonblocking is to choose S such that [4]:

$$L(S/G) = \overline{L_{am}^{\uparrow C}} \quad \text{and} \quad L_m(S/G) = L_{am}^{\uparrow C}$$

as long as $L_{am}^{\uparrow C} \neq \varnothing$. It is important to note that since L_{am} is assumed to be $L_m(G)$ closed then $L_{am}^{\uparrow C}$ is also $L_m(G)$ closed which guarantees that:

$$L_m(S/G) = L_{am}^{\uparrow C}$$

Whenever

$$L(S/G) = \overline{L_{am}^{\uparrow C}}$$

S can be realized by building a recognizer for $L_{am}^{\uparrow C}$. The recognizer for $L_{am}^{\uparrow C}$ can be build from L_{am} by using the algorithm for *Computation of* $^{\uparrow C}$, which is provided in Appendix (B) [4].

5.1.2.2 Nonblocking Controllability Theorem (NTC)

Theorem 2 [4]: Consider DES $G = (E, Q, \delta, x_0, x_m)$ where $E_{uc} \subseteq E$ is the set of uncontrollable events. Let the language $L_{am}^{\uparrow C} \subseteq L_m(G)$ where $L_{am}^{\uparrow C} \neq \emptyset$. There exists a nonblocking supervisor *S* for G such that:

$$L_m(S/G) = L_{am}^{\uparrow C} \quad \text{and} \quad L(S/G) = \overline{L_{am}^{\uparrow C}}$$

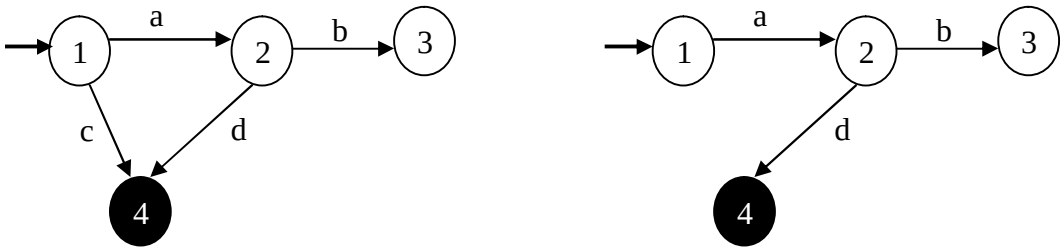
If and only if the following conditions hold:

- Condition 1: Controllability: $\overline{L_{am}^{\uparrow C}} \Sigma_u \cap L(G) \subseteq \overline{L_{am}^{\uparrow C}}$
- Condition 2: $L_m(G)$-closure: $L_{am}^{\uparrow C} = \overline{L_{am}^{\uparrow C}} \cap L_m(G)$

Proof: For the proof refer to [4].

- *Definition 9:* Controllability: A language L_a is said to be controllable if $\overline{L_a} \Sigma_u \cap L(G) \subseteq \overline{L_a}$. This means, given a string ω, which is a prefix of L_a, if we add an uncontrollable event $\sigma \in \sum_u$ such that $\omega\sigma \in L(G)$. Then if adding event σ does not causes the string to exit from the prefix closure $\overline{L_a}$, then *L* is said to be controllable [2]
- *Definition 10: $L_m(G)$-closed* : A language *L* is said to be $L_m(G)$-closed if $L = \overline{L} \cap L_m(G)$. This mean, the intersection of prefix closure $\overline{L}$ and $L_m(G)$ is same as *L*

Example $L_m(G)$- closed: Consider a system with states (1, 2, 3, and 4) and events (a, b, c, and d) as shown in Figure 5.3(a). In this system state 4 is a marked state and state 1 is initial state. Once the control specifications are imposed, event *c* is undesirable and hence removed as shown in Figure 5.3(b).



(a) (b)

Figure 5.3. System for Closure

- L (G) = {a, ab, c, ad}
- L_m (G) = {c, ad}
- L_a = {a, ab, ad}
- $\overline{L}_a$ = {a, ab, ad}
- L_{am} = {ad}
- $\overline{L}_{am}$ = {a, ad}

A language L_a is not L_m (G) closed as $L_a \neq \overline{L}_a \cap L_m(G)$. Whereas Language L_{am} is L_m (G) closed as $L_{am} = \overline{L}_{am} \cap L_m(G)$.

5.2 Applying Supervisory Control to Workflow Processes

Considering the airline example explained in Section 3.3 which consists of following tasks and inter-task dependencies.

- Task 1 - Purchasing an airline ticket (t_a),
- Task 2 - Booking a hotel (t_h), and
- Dependency 1: Booking of hotel cannot start until purchasing an airline ticket starts (t_a BD t_h).

- Dependency 2: If hotel booking aborts then purchasing airline ticket must aborts too (t_h AD t_a).

5.2.1 Uncontrolled Process Model

Both tasks t_a and t_h are modeled as separate automata G_a and G_h as shown in Figure 5.4 and these automata are then shuffled to get the uncontrolled process model $G = G_a \,||\, G_h$ as shown in Figure 5.5.

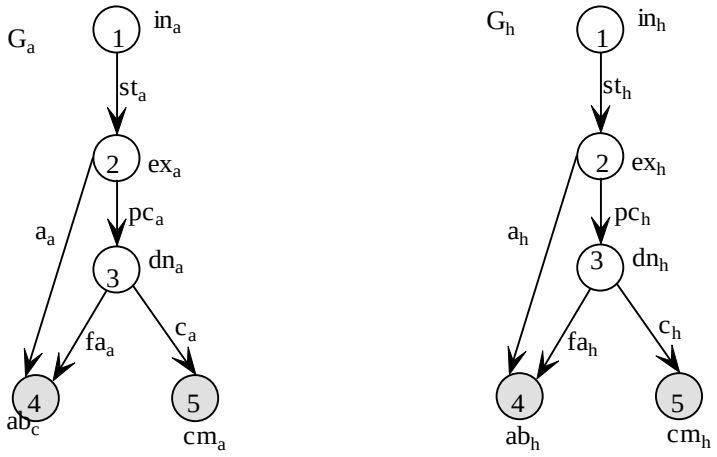


Figure 5.4. Individual Task Automata

Airline task

- Σ_c: {st_a, fa_a, c_a} $\Sigma_{u:}$ {pca, a_a}
- Q: {in_a, ex_a, dn_a, ab_a, cm_a}
- d : {(st_a x $in_a \rightarrow ex_a$), (pc_a x $ex_a \rightarrow dn_a$), (a_a x $ex_a \rightarrow ab_a$), (c_a x $dn_a \rightarrow cm_a$),
- (fa_a x $dn_a \rightarrow$ aba)}
- q_o : {in_a}
- $Q_{m:}$ { in_a, ab_a, cm_a}

Hotel task

- Σ_c: {st_h, fa_h, c_h} $\Sigma_{u:}$ {pc_h, a_h}
- Q: {in_h, ex_h, dn_h, ab_h, cm_h}
- d : {(st_h x in_h ⟶ ex_h), (pc_h x ex_h ⟶ dn_h), (a_h x ex_h ⟶ ab_h), (c_h x dn_h ⟶ cm_h), (fa_h x dn_h ⟶ ab_h)}
- $q_{o:}$ {in_h}
- $Q_{m:}$ { in_h , ab_h, cm_h}

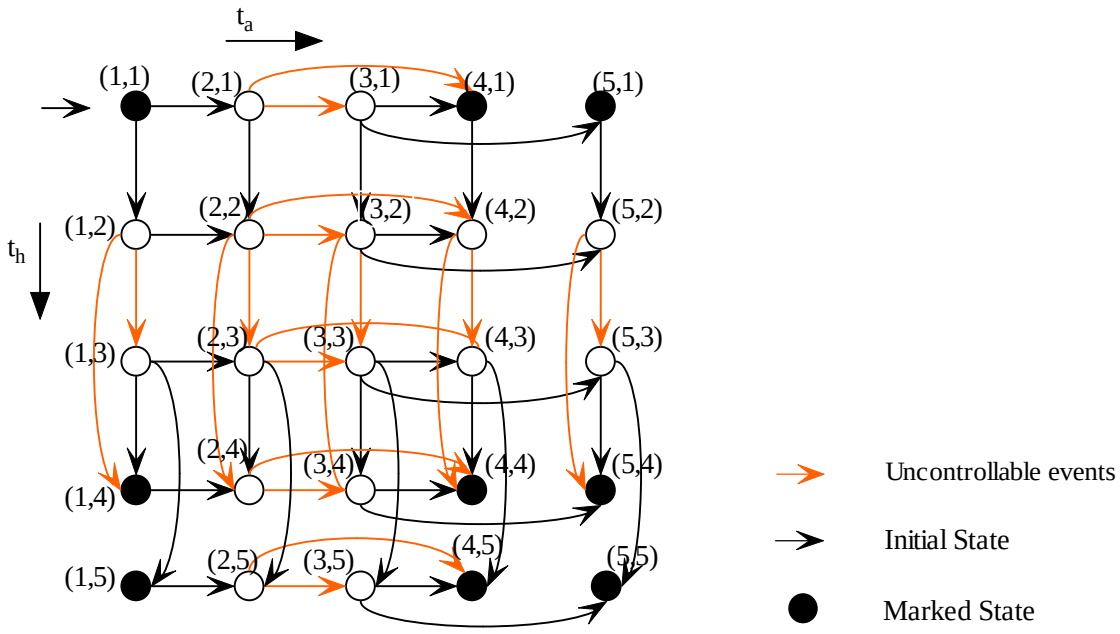


Figure 5.5.Uncontrolled Process Model (G)

5.2.2 Admissible Language

In workflow processes a workflow model should have certain properties, such as it should be safe and deadlock free. A workflow is safe if it terminates in a compatible or marked state. Whereas, a workflow model is deadlock free if from any state by allowing a sequence of events it is possible to reach some final state (marked state). To satisfy the property of safety and deadlock free, the workflow model should be nonblocking i.e. a basic supervisory control problem where blocking is of concern (BSCP-NB).

The solution for this problem according BSCP-NC as explained in Section 5.1.2 is to choose a nonblocking supervisor S such that:

$$L(S/G) = \overline{L_{am}^{\uparrow C}} \quad \text{and} \quad L_m(S/G) = L_{am}^{\uparrow C}$$

The existence of such a nonblocking supervisor is guaranteed If and only if the following conditions hold.

- Condition 1: Controllability: $\overline{L_{am}^{\uparrow C}}\ \Sigma_u \cap L\ (G) \subseteq \overline{L_{am}^{\uparrow C}}$
- Condition 2: $L_m(G)$-closure: $L_{am}^{\uparrow C} = \overline{L_{am}^{\uparrow C}} \cap L_m(G)$

To check for the existence of nonblocking supervisor, we first compute the recognizer for $L_{am}^{\uparrow C}$, which is the supremal controllable sublanguage of L_{am}.

5.2.3 Computation of L_{am}

The admissible behavior L_{am} is the intersection of marked language generated by the uncontrolled process model $L_m(G)$ and the marked language generated by control specification $L_m(C)$ i.e. $L_{am} = L_m(G) \cap L_m(C)$. Control specification C is the shuffle product of individual control specification (dependency) automata C_i for dependency *i*. However it's not easy to describe the control specification as a formal language at all times. A wrong specification model will lead to the construction of an incorrect supervisor [23, 24]. In order to automate this we develop control specifications for weak-causal, strong-causal and precedence type dependencies which can be coupled directly with the uncontrolled process model to determine the admissible language and hence the supervisor.

5.2.3.1 Control Specification Models

Control-flow dependencies involve two tasks and specify certain restrictions over the execution of these tasks based on precedence of events and combination of states that are not allowed at a given time. The set of incompatible states (combination of states not allowed at any given time) and precedence order arising from control-flow dependencies based on their type are listed in Table 5.1.

Table 5.1. Dependency Specification

Dependency	*Incompatible state*	*Precedence order*
Strong-causal	$(s_i^{'}, s_j)$	$s_i <= s_j$
Weak-causal	$(s_i, s_j^{'})$	None
Precedence	None	$s_i <= s_j$

A task T_i has a set of states Q_i and at any given point of time a task can be in only one state $s_i \in Q_i$. Thus we define, all the states other than state s_i as complementary states $\hat{s_i}$ i.e. $\hat{s_i} = Q - s_i$. Table 5.2 shows all $s_i \in Q_i$ and its complementary state $\hat{s_i}$.

Table 5.2. Complementary States

s_i	ex_i	dn_i	cm_i	ab_i
$\hat{s_i}$	in_i, dn_i, cm_i, ab_i	in_i, ex_i, cm_i, ab_i	in_i, ex_i, dn_i, ab_i	in_i, ex_i, dn_i, cm_i

Based on complementary states, a pair of tasks can be in one of the state sets $(\hat{s_i}, \hat{s_j})$, $(\hat{s_i}, s_j)$, $(s_i, \hat{s_j})$, and (s_i, s_j). The precedence relationships result in a situation where state set (s_i, s_j) can be reached from state set $(\hat{s_i}, s_j)$ or state set $(s_i, \hat{s_j})$ by following a sequence of events (path), from which only one of the sequences is a legal

path. In order to differentiate state set (s_i, s_j) that is reached from state set $(\hat{s}_i, s_j)$ and state set $(s_i, \hat{s}_j)$, state set (s_i, s_j) is modeled as two different nodes $(\bar{s}_i, s_j)$ and $(s_i, \bar{s}_j)$ respectively and an event e_i is defined where $\delta(\hat{s}_i, e_i) = s_i$. Figure 5.6 illustrates the transition graph of the control specification model for any control-flow dependency. For this transition graph by looking in Table 5.3 events e_i and e_j can be identified based on the type of dependency as illustrated in Section 5.2.3.1.

Formally a control specification is described by a finite automaton $C = (\Sigma, Y, \zeta, y_o, Y_m)$ where Σ is a set of events, Y is a set of states, ζ is a transition function, y_o is an initial state and Y_m is a set of marked states. From Figure 5.6,

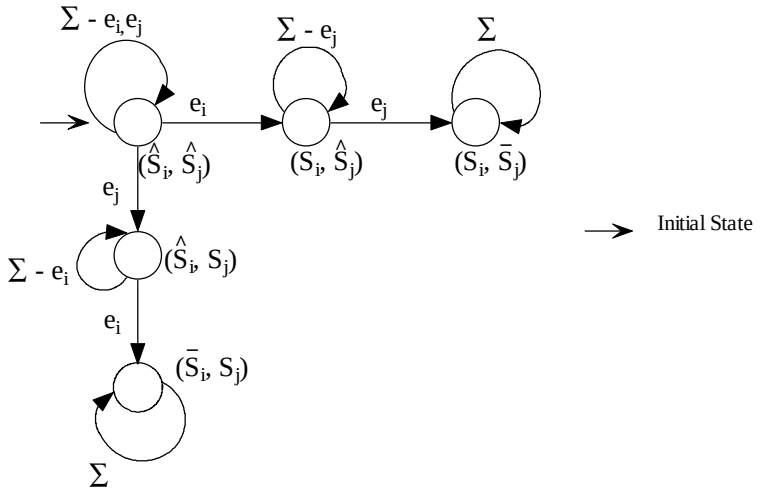


Figure 5.6. Structure for Control Specification

Y: $\{(\hat{s}_i, \hat{s}_j), (\hat{s}_i, s_j), (s_i, \hat{s}_j), (\bar{s}_i, s_j), (s_i, \bar{s}_j)\}$

Σ: $\Sigma_i \cup \Sigma_j = \{b_i, pc_i, c_i, a_i, fa_i, b_j, pc_j, c_j, a_j, fa_j\}$

ζ: $\{(e_i \times (\hat{s}_i, \hat{s}_j) \rightarrow (s_i, \hat{s}_j)), (e_j \times (\hat{s}_i, \hat{s}_j) \rightarrow (\hat{s}_i, s_j)), (e_i \times (\hat{s}_i, s_j) \rightarrow (\bar{s}_i, s_j)),$

$(e_j \times (s_i, s_j') \rightarrow (s_i, \bar{s}_j))\}$ Where $e_i \in \Sigma_i$ and $e_j \in \Sigma_j$

Considering that all tasks start from their respective initial states and this state is included in state set ($\hat{s}_i$, $\hat{s}_j$), then the combined initial state of the specification is y_o= {($\hat{s}_i$, $\hat{s}_j$)}.

The set of final (marked) states, Y_m, are determined based on the types of dependencies. Generally, if none of the states included in a specification model state set violate the conditions that the dependency specifies, then the state is marked. Conversely, if any member of the state set is illegal (incompatible) or is reached through an illegal string then the state set is not marked.

In order to identify the illegal states and illegal strings based on the type of dependencies, we have provided Table 5.3 which can be used to construct the specification model for a given dependency. Table 5.3 lists a complete set of incompatible states, illegal strings event e_i and event e_j for all the control-flow dependencies.

In Table 5.3 e_i is an event which takes the task t_i from state $s_i^{'}$ to state s_i and is defined as $\delta(s_i^{'}, e_i) = s_i$. Similarly e_j is an event which takes the task t_j from state $s_j^{'}$ to state s_j and is defined as $\delta(s_j^{'}, e_i) = s_j$, whereas illegal state and illegal string are explained in Section 3.3.3. In the next Sections 5.2.3.1.1 we describes the methodology to determine the specification models of Strong-causal, weak-causal and precedence type dependencies.

Table 5.3. Incompatible States and Illegal Strings

Type of dependency	*Incompatible state or illegal state*	*Illegal string*	e_i	e_j
Begin	$(s_i^{'}, s_j)$	Event e_i from $(s_i^{'}, s_j)$	b_i	b_j
Begin on commit	$(s_i^{'}, s_j)$	Event e_i from $(s_i^{'}, s_j)$	c_i	b_j
Begin on abort	$(s_i^{'}, s_j)$	Event e_i from $(s_i^{'}, s_j)$	a_i, fa_i	b_j
Serial	$(s_i^{'}, s_j)$	Event e_i from $(s_i^{'}, s_j)$	c_i, a_i, fa_i	b_j
Terminating	$(s_i^{'}, s_j)$	Event e_i from $(s_i^{'}, s_j)$	c_i, a_i, fa_i	c_j, a_j, fa_j
Strong commit	$(s_i, s_j^{'})$	None	c_i	c_j
Abort	$(s_i, s_j^{'})$	None	a_i, fa_i	a_j, fa_j
Forced commit on abort	$(s_i, s_j^{'})$	None	a_i, fa_i	c_j
Exclusion	$(s_i, s_j^{'})$	None	c_i	a_j, fa_j
Weak begin on commit	None	Event e_i from $(s_i^{'}, s_j)$	c_i	b_j
Weak abort	None	Event e_i from $(s_i^{'}, s_j)$	a_i	c_j
Commit	None	Event e_i from $(s_i^{'}, s_j)$	c_i	c_j

5.2.3.1.1 Strong–causal Dependency

Consider the begin dependency between tasks t_i and t_j (strong-causal dependency) which indicates that, task t_j cannot begin execution until task t_i has begun. For the begin dependency, Table 5.3 indicates that state $(s_i^{'}, s_j)$ is an incompatible state which is a member of $(\hat{s_i}, s_j)$ and hence state $(\hat{s_i}, s_j)$ will not be marked. In order to determine if there is a state set which is reached by executing an illegal sequence of events (illegal string). We look into Table 5.3, which indicates that event e_i from state $(s_i^{'}, s_j)$ is an illegal string which is a member of $(\hat{s_i}, s_j)$. By executing event e_i from $(\hat{s_i}, s_j)$ the only state set that can be reached in Figure 5.6 is $(\overline{s_i}, s_j)$. Event e_i corresponds to the b_i event for this dependency. Since this is an illegal string, (task t_j is in s_j indicating task t_j has

already begun before task t_i, which violates the precedence constraint) state set $(\bar{s}_i, s_j)$ is also unmarked. The resulting generator for the begin dependency is illustrated in Figure 5.7.

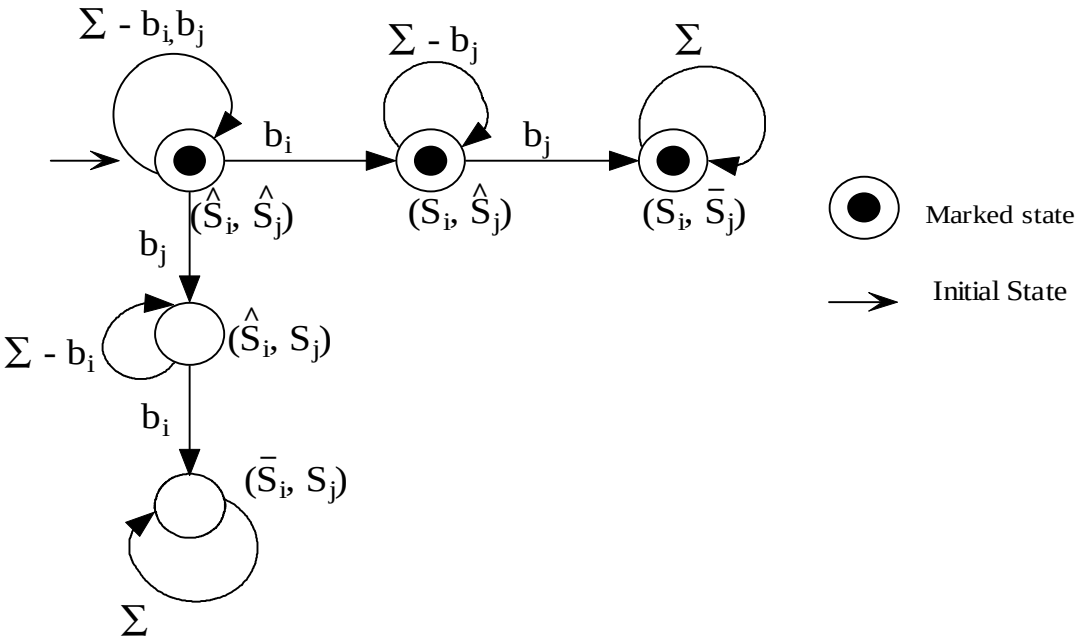


Figure 5.7. Control Specification Model (Begin Dependency)

5.2.3.1.2 Weak-Causal Dependency

Consider an abort dependency between tasks t_i and t_j (weak-causal dependency) which indicates that, if task t_i aborts task t_j has to abort too. Table 5.3 indicates that state (s_i, s_j') is an incompatible state which is a member of $(s_i, \hat{s}_j)$ and hence state set $(s_i, \hat{s}_j)$ will be unmarked. The resulting generator for the abort dependency is shown in Figure 5.8.

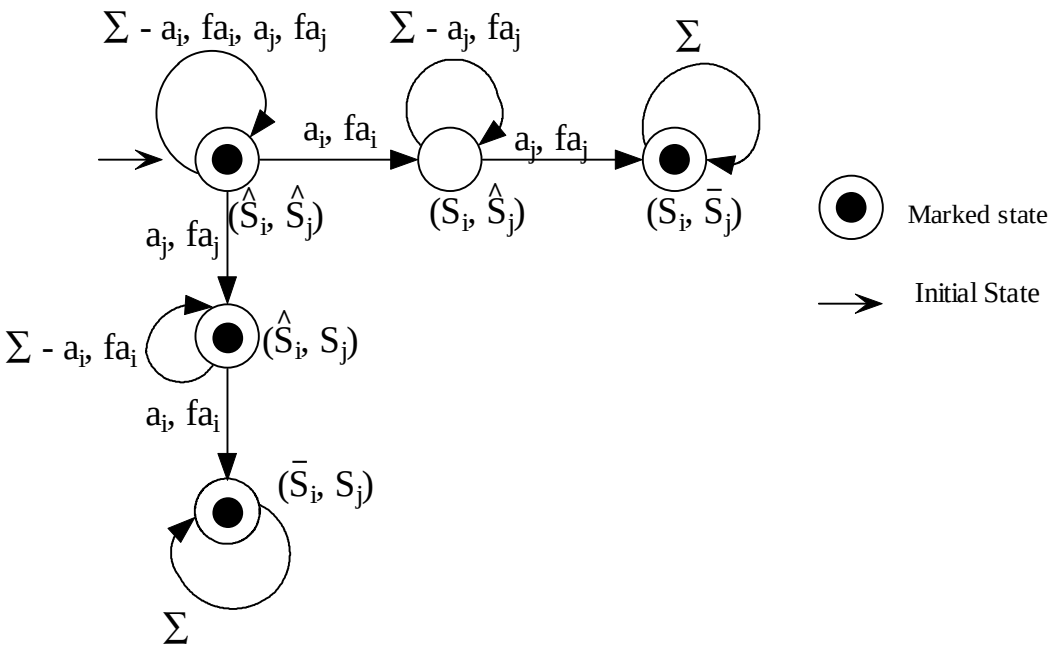


Figure 5.8. Control Specification Model (Abort Dependency)

5.2.3.1.3 Precedence Dependency

Consider a commit dependency between tasks t_i and t_j (precedence dependency), which indicates that, if both task t_i and t_j commits, then task t_i commits first. Table 5.3 indicates that there is no incompatible state. However, there is a precedence order which signifies that there is an illegal string. Table 5.3 indicates that event e_i from state (s_i, s_j') is illegal string, state (s_i, s_j') is a member of $(\hat{s}_i, s_j)$. By executing event e_i from $(\hat{s}_i, s_j)$ the only state set that can be reached in Figure 5.6 is $(\bar{s}_i, s_j)$. Event e_i corresponds to the c_i event for this dependency. Since this is an illegal string, (task t_j is in c_j indicating task t_j has already committed before task t_i, which violates the precedence constraint) hence state set $(\bar{s}_i, s_j)$ is also unmarked. The resulting generator for the commit dependency is illustrated in Figure 5.9.

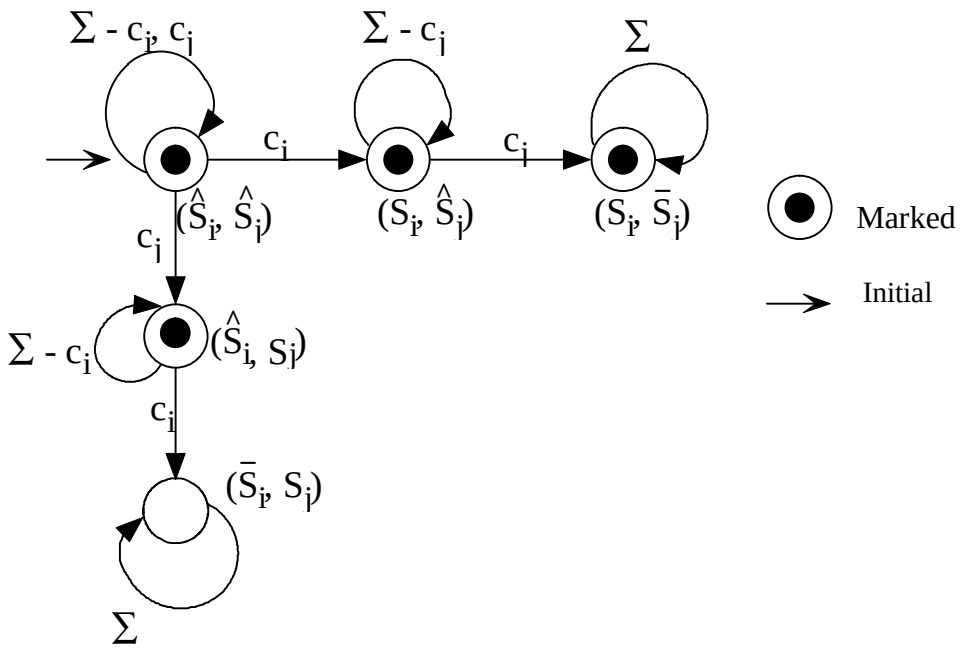
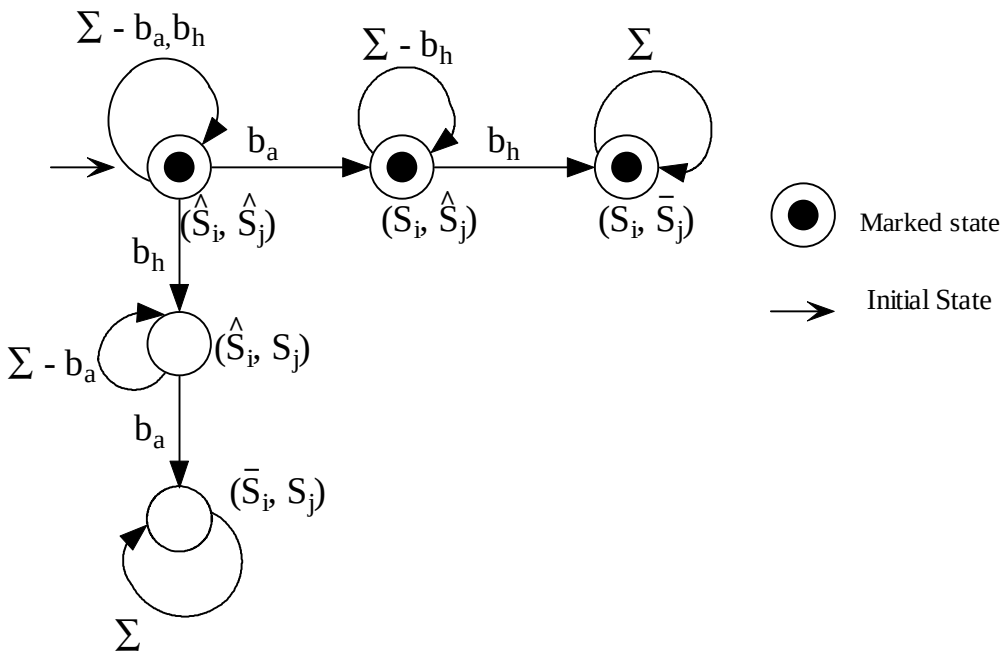


Figure 5.9. Control Specification Model (Commit Dependency)

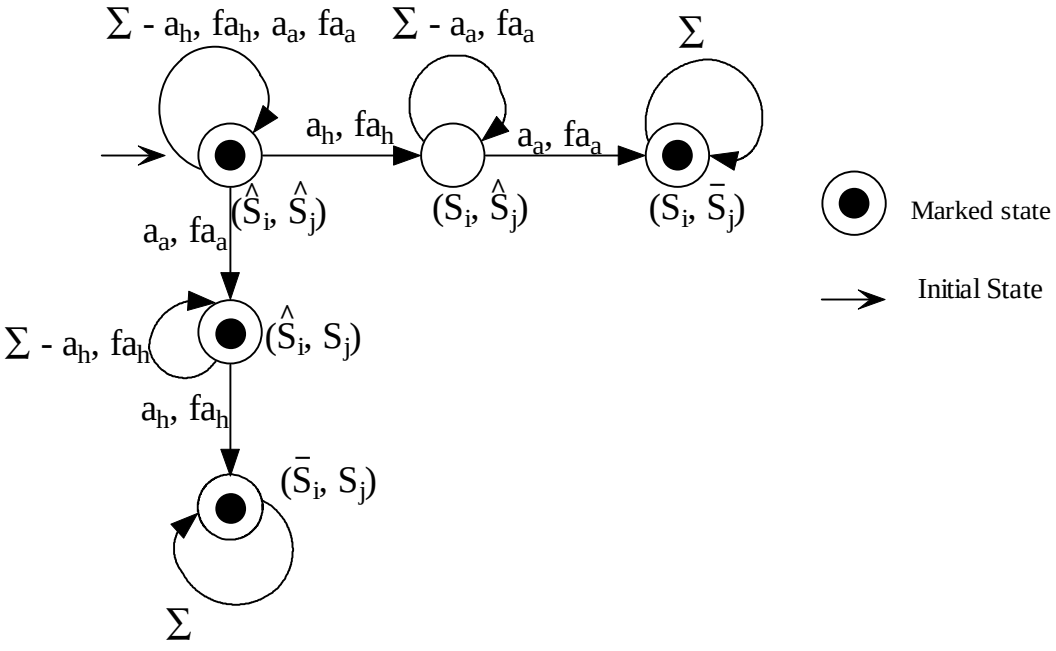
In Section 5.2.3.1 we have introduced the methodology for building the specification models. In the next Section we use this methodology to develop the specification model for the airline example explained in Section 5.2.

5.2.3.2 Specification Model for Airline Example

The specification model consists of the dependencies between pairs of tasks. Each dependency *i* is modeled as finite automata C_i. The automata representing the individual specification for begin and abort dependencies (C_a and C_b) between tasks t_a and t_h are shown in Figure 5.10(a) and 5.10(b) respectively. These automata are then shuffled to obtain the specification model C as depicted in Figure 5.11.



(a) Begin Dependency



(b) Abort Dependency

Figure 5.10. Specification Model

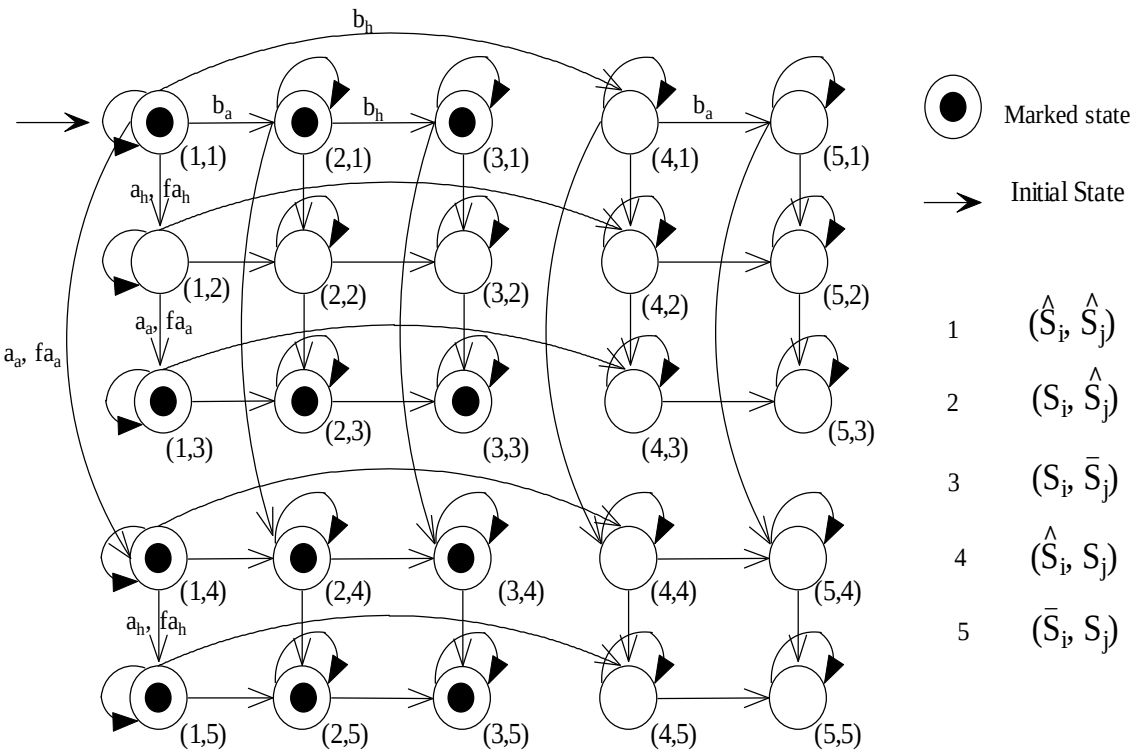


Figure 5.11. Total Specification Model ($C = C_a \,||\, C_b$)

Once the uncontrolled process model and specification model are determined, the next step is to couple uncontrolled process model and specification model to obtain the admissible language $L_{am} = L_{am}(C/G) = L_m(C) \cap L_m(G)$. The recognizer for admissible language L_{am} is shown in Figure 5.12.

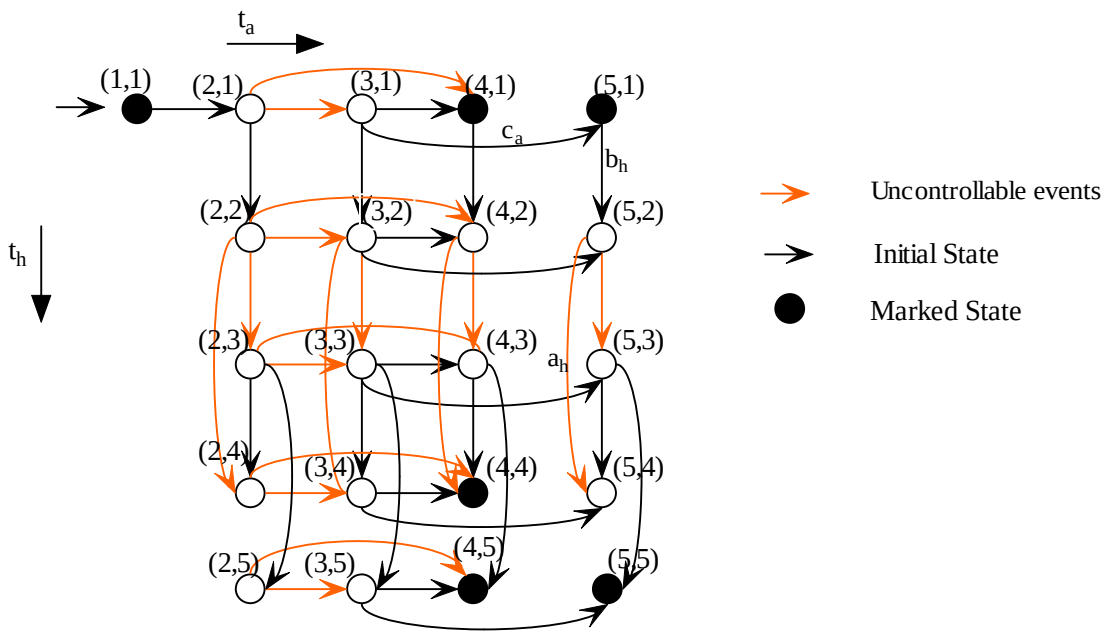


Figure 5.12. Recognizer for L_{am}

5.2.3.3 Recognizer for $L_{am}^{\uparrow C}$

The recognizer for $L_{am}^{\uparrow C}$ is supremal controllable sublanguage of L_{am}. The supremal controllable sublanguage refers to maximum permissible language with respect to the admissible language. The supremal language $L_{am}^{\uparrow C}$ can be computed using the algorithm [4]. This procedure is well understood with the help of Figure 5.13 and the algorithm is included in Appendix B.

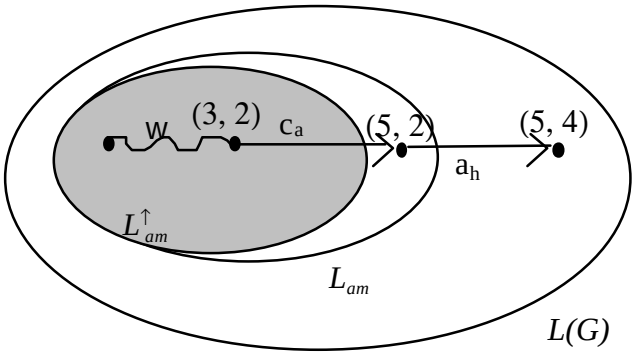


Figure 5.13. Supremal Sublanguage

Initially, the admissible language for the system is $L_{am} = L_m(C/G)$. However, at state (5, 2) the supervisor cannot prevent the system from crossing the boundaries of the admissible language L_{am} since $a_h \in \Sigma_u$. Hence, the idea is to prevent the system from entering in to state (5, 2). In Figure 5.13 state (5, 2) is accessible from states {(5, 1), (3, 3)} by an events c_a, $b_h \in \Sigma_c$. Since events c_a, b_h are controllable events, these can be disabled by the supervisor at states (5, 1), (3, 3) so that state (5, 2) is not reached. Once the system is prevented from reaching state (5, 2) there can be some states in the system which are blocked. For example if we remove state (5, 2) it is not possible to reach any of the required marked states of the coupled model from state (5, 4). Hence a trim operation is done which removes all the states that are not reached from initial state or states from where some final state cannot be reached. The trim operation is described in Section 4.4.1. According to this iterative procedure, admissible language L_{am} is reduced to a supremal controllable sublanguage $L_{am}^{\uparrow C}$ shown by the shaded region in Figure 5.13. The recognizer for $L_{am}^{\uparrow C}$ is shown in Figure 5.14, that excludes states (5, 2) and (5, 4).

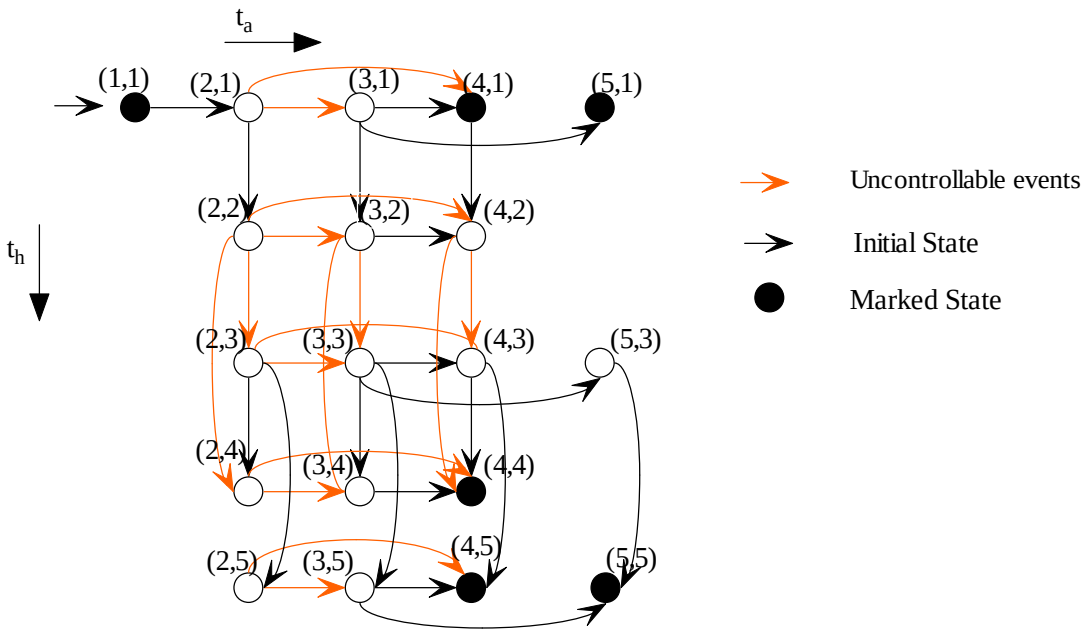


Figure 5.14. Recognizer for $L_{am}^{\uparrow C}$ (C/G)

5.2.3.4 Existence of Supervisor

To test for the existence of a nonblocking supervisor we check supremal controllable sublanguage $L_{am}^{\uparrow C}(C/G)$ for controllability and L_m(G)-closure conditions described in Section 5.2.3. If both these conditions are satisfied, then there exists a nonblocking supervisor S such that:

$$L(S/G) = L_{am}^{\uparrow C}(C/G) \text{ and } L_m(S/G) = \overline{L_{am}^{\uparrow C}}(C/G)$$

The controllability condition depends on the system under consideration i.e. the admissible behavior $L_{am}^{\uparrow C}(C/G)$. If we obtain $L_{am}^{\uparrow C}(C/G)=\varnothing$, then there exist a string of uncontrollable events from the initial state of *G* that does not belong to $L_{am}(C/G)$ i.e. there exist a dependency or combination of dependencies among tasks that are inconsistent.

Whereas if $L_{am}^{\uparrow C}(C/G) \neq \varnothing$, the condition of controllability is satisfied i.e. given a string ω, which is a prefix of $\overline{L_{am}^{\uparrow C}}(C/G)$, if we add an uncontrollable event $\sigma \in \sum_u$ such that $\omega\sigma \in L(G)$, then adding event σ does not causes the string to exit from the prefix closure $\overline{L_{am}^{\uparrow C}}(C/G)$.

$L_m(G)$-closure condition depends on the construction of the admissible behavior. When the admissible behavior is " admissible marked behavior" then it satisfy L_m(G)-closure condition. The following points support this argument [4].

- Marking is the property of the uncontrolled process model G, modeled by proper construction of Q_m.

- Specifications are stated in term of marked languages $L_m(C)$, as described in Section 5.2.3.2.
- The admissible marked language is obtained by forming $L_{am} = L_m(C) \cap L_m(G)$.
- Such a L_{am} is guaranteed to be $L_m(G)$-closed, since $\forall w \in L_{am}$ and $w \in L_m(G)$.

Since $L_{am}^{\uparrow C}(C/G) \neq \varnothing$ as shown in Figure 5.14 and the admissible behavior is marked i.e. $L_{am} = L_m(C) \cap L_m(G)$ as describes in Section 5.2.3.2 the condition of controllability and $L_m(G)$-closure are satisfied. There exist a supervisor *S* such that.

$$L(S/G) = \overline{L}_{am}^{\uparrow C}(C/G) \text{ and } L_m(S/G) = \overline{L}_{am}^{\uparrow C}(C/G)$$

5.2.4 Supervisor

The supervisor *S* consists of a finite automata S and an output function Ψ(control pattern) as describe in Section 5.1. The finite automaton S is the automaton generated by $\overline{L}_{am}^{\uparrow C}(C/G)$ i.e. the recognizer for $\overline{L}_{am}^{\uparrow C}(C/G)$. Whereas the output function Ψ (control pattern) for the supervisor is calculated for the states of S and events $\sigma \in \Sigma_c$ (set of controllable events). Based on the control pattern Ψ, the supervisor disables the controllable events such that the uncontrolled process model satisfies the language $\overline{L}_{am}^{\uparrow C}(C/G)$. The control pattern for the supervisor is shown in Table 5.4 which is in the form of 0 and 1 (0: disable, 1: enable).

Table 5.4. Control Pattern

	b_a	af_a	c_a	b_h	af_h	c_h
(1, 1)	1	-	-	0	-	-
(2, 1)	-	-	-	1	-	-
(3, 1)	-	1	0	1	-	-
(4, 1)	-	-	-	1	-	-
(2, 2)	-	-	-	-	-	-
(3, 2)	-	1	0	-	-	-
(4, 2)	-	-	-	-	-	-
(2, 3)	-	-	-	-	1	-
(3, 3)	-	1	1	-	1	1
(4, 3)	-	-	-	-	1	1
(5, 3)	-	-	-	-	0	1
(2, 4)	-	-	-	-	-	-
(3, 4)	-	1	0	-	-	-
(4, 4)	-	-	-	-	-	-
(2, 5)	-	-	-	-	-	-
(3, 5)	-	1	1	-	-	-
(4, 5)	-	-	-	-	-	-
(5, 5)	-	-	-	-	-	-

* The events for which the control patter is marked '-'means that the event has no restriction in that state.

Consider state (1, 1) in Table 5.4 at which event $b_h = 0$, which means when the uncontrolled process model reaches state (1, 1) the supervisor disables the event b_h to keep the uncontrolled process model under the specified behavior $L_m^{\uparrow C}(C/G)$ i.e. the supervisor prevents hotel booking task t_h to start before airline reservation task t_a starts.

Similarly consider state (3, 4) which represents a state when airline task is in its done state and hotel task is in abort state. At state (3, 4) event $c_a = 0$ as Shown in table 5.4, which is the commit event of the airline task and is disabled. As this event c_a takes the system to a state where airline task will be in commit state and hotel task is in its abort state, which is an incompatible state according to abort dependency between hotel task and airline task.

Chapter 6
Case Study

In this chapter online bookstores workflow architecture is illustrated, which is modeled using the supervisory control theory using the formalism described in chapter 5.

6.1 Online Bookstore

The workflow architecture for online bookstore is shown in Figure 6.1 which is centered around a workflow engine. The workflow components have application interfaces that provide standard means of communication between these components and the workflow engine. All workflow components have specific functions to perform in a workflow.

- The *Process Definition Tool* is used at design stage to specify the process. Process definition contains information regarding the tasks, its starting and completion conditions, and rules and dependencies for navigating between tasks.
- The *Administration and Monitoring Tool* are used for managing users, roles, security policy and for tracking and reporting workflow states and data generation.

The workflow engine reads the information from process definitions. This information is used by the engine to determine the step(s) to be performed and present them to the user through a user interface. The user then takes the appropriate action and notifies the workflow engine. Based on the user's action the engine determines the future steps to be taken. When all the steps are completed, the workflow terminates.

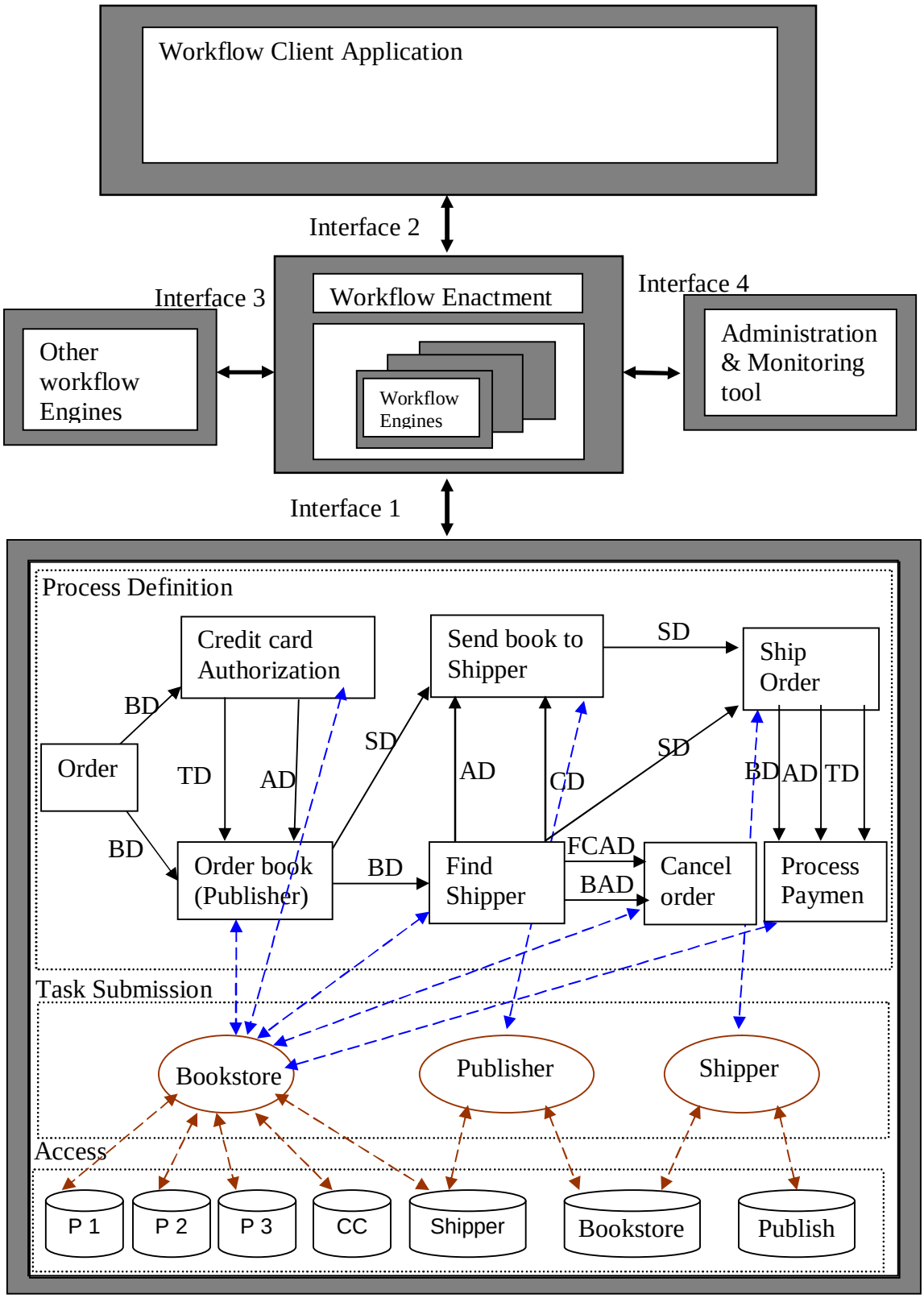


Figure 6.1. Online Bookstore

6.1.1 Process Definition

The online bookstore is a virtual company that has no books in stock. It has a pool of publishers who supply books to the online bookstore when ordered. The bookstore has access to these publisher's databases. The customer places an order (*Order*) with the bookstore. The bookstore checks the availability of the book with a publisher by accessing the publisher's database. If the book is available, the bookstore transfers the order to the publisher (*Order Book*). If the book is not available, the bookstore decides to search for an alternative publisher or rejects the order. At the same time the bookstore checks the credit card information provided by the user (C*redit Card Authorization*). If the book is available and the credit card information provided by the user is correct, the customer is informed and the bookstore continues to process the order. After ordering the books with the publisher, the bookstore searches for a shipper and sends a request to the shipper (*Find Shipper*). 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 (*Cancel Order*). If the shipper accepts the request, the publisher is informed. Then the publisher prepares the book for shipment and sends it to the shipper (*Send Book to Shipper*). The shipper prepares and ships the order (*Ship Order*). The shipper notifies online bookstore and the online bookstore or its billing company then processes the payment (*Process Payment*).

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).
- Order 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 aborts 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 (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* as shown in Figure 6.2. These tasks are.

- Credit Card Authorization (T_2)
- Order Books (T_3)
- Find Shipper (T_4)
- Cancel Order (T_6)

The dependencies between these tasks are listed above and shown in Figure 6.2 [35]. The dependencies between these tasks are.

- T_2 AD T_3 (C1)
- T_3 TD T_2 (C2)
- T_4 BD T_3 (C3)
- T_4 BAD T_6 (C4)
- T_6 FCAD T_4 (C5)

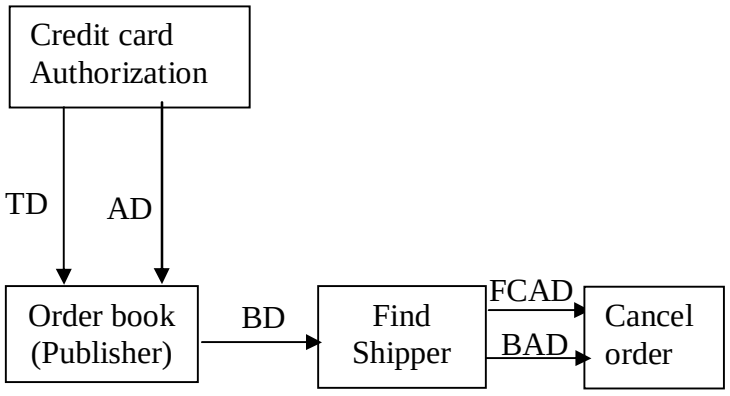


Figure 6.2. Online Bookstore Workflow

6.1.3 Uncontrolled Process Model

The above-mentioned tasks are represented as finite state automata models. The uncontrolled process model (G), is the combined representation of all the tasks together, and is determined by taking the shuffle product of individual tasks as describe in Section 6.2.1.

$G = T_2 \,||\, T_3 \,||\, T_4 \,||\, T_6$

6.1.4 Specification Model

The inter task dependencies are modeled as finite state automata also and are referred as specification models. Similar to the tasks, individual dependency specifications are combined to form the total specification (C) as describe in Section 5.2.3.2.

$C = C1 \,||C2 \,||\, C3 \,||\, C4 \,||\, C5$

The admissible language is obtained by the couple product of specification C and the uncontrolled process model G i.e. $L_{am} = L_m(C/G)$.

The operations on the finite state automata such as Shuffle, Couple, finding the Supremal Sublanguage etc. were performed with the help of the software XPTCT designed by W.M. Wonham.

6.1.5 Supervisor and Inconsistency

We construct a nonblocking supervisor *S* such that:

$$L(S/G) = \overline{L}_{am}^{\uparrow C}(C/G) \text{ and } L_m(S/G) = \overline{L}_{am}^{\uparrow C}(C/G)$$

6.1.5.1 Inconsistent Supervisor

The supremal controllable sublanguage $\overline{L}_{am}^{\uparrow C}(C/G)$ is shown in Figure 6.3. This figure shows that only credit card authorization (T_2) and order books with publisher task (T_3) has executed. This indicates the presence of a one or more dependency specification that is inconsistent with the given task structure between find shipper and cancel order task. In order to identify which dependency specification is inconsistent we have checked all the dependencies and the task structures involved in them individually and found that a combination of forced commit on abort dependency (FCAD) and begin on abort (BAD) between tasks *Find Shipper* and *Cancel Order* is inconsistent. These dependency acts in the following way.

- Begin on abort (BAD): Online bookstore looks for a shipper to ship the order to the customer. If a shipper is not available i.e. the Find Shipper task aborts, then cancel order task begin. This is depicted in Figure 6.4.
- Forced commit on abort (FCAD): Online bookstore looks for a shipper to ship the order to the customer. If a shipper is not available i.e. the Find Shipper task aborts, then cancel order task should commit. However with the given task structure the cancel order task cannot be forced to commit. It can abort from its execution state due to an uncontrollable *Abort* event that leads the task to the aborted state. With the current task structure, after the order has been placed, the publisher can deny cancellation of an order. This is depicted in Figure 6.5.

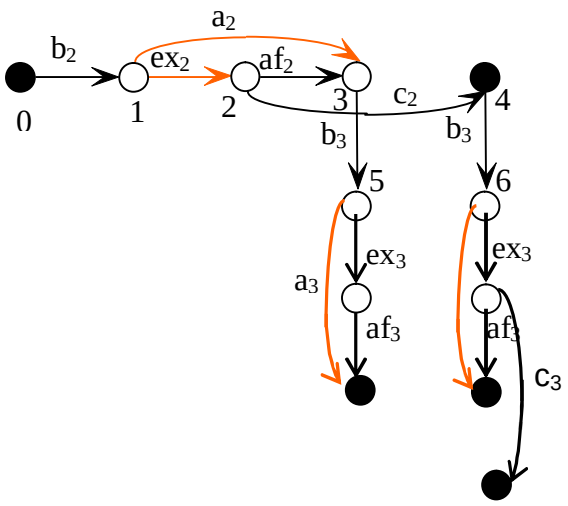


Figure 6.3. Recognizer for $\overline{L}_{am}^{\uparrow C}$ (C / G)

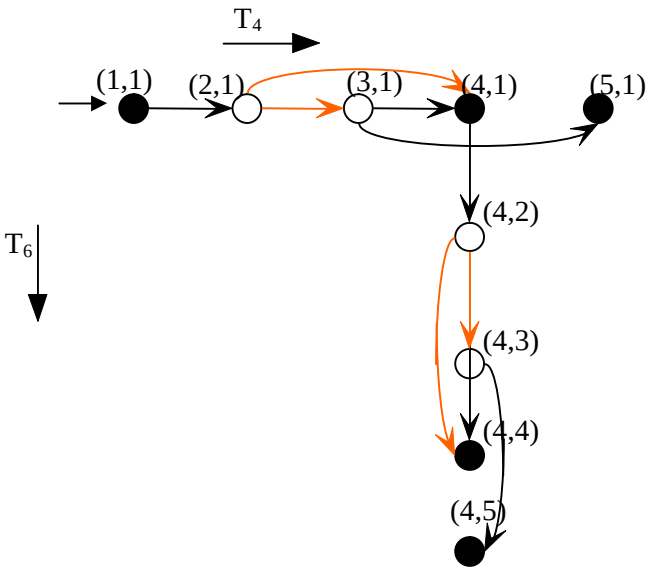


Figure 6.4. Begin on Abort

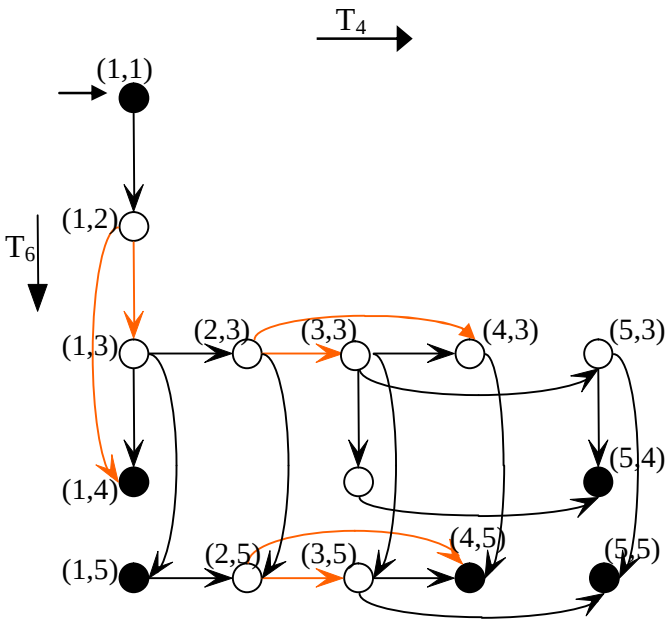


Figure 6.5. Forced Commit on Abort

In both forced commit on abort and begin on commit dependencies are combined i.e. if both FCAD and BAD are specified between find shipper task and cancel order task. The only feasible state is when both find shipper and cancel order tasks are in there initial states (1, 1) i.e. both find shipper and cancel order task can not execute.

6.1.5.2 Modified Supervisor

At this juncture there are two approaches that a company can adopt. The first approach is if the denial of the publisher to cancel the order is unacceptable, the terms of business agreement between the two concerned parties should be modified. In this case it means that the online bookstore can cancel the order with publisher at any point in the transaction, i.e. bookstore can force the Cancel Order task to commit. This involves adopting a different task structure. This task structure will not have an uncontrollable abort event *a* which can abort the cancel order task during its execution. The current and the suggested task structure for *Cancel Order* are shown in Figure 6.6.

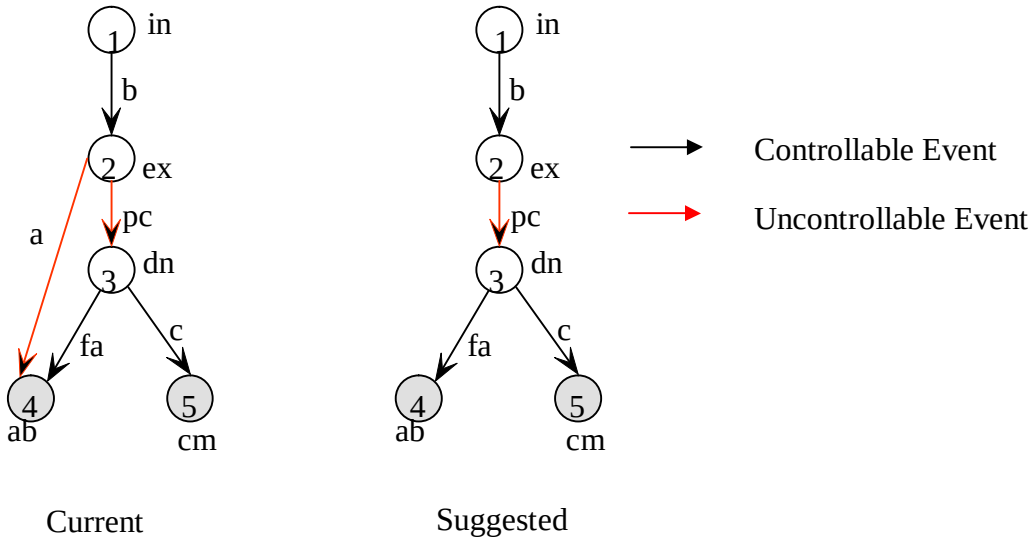


Figure 6.6. Cancel Order Task Structures

The second approach is to accept the current policy that cancellation of order cannot be forced to commit. This might result in losses for the company. With this approach the FCAD dependency between tasks *Find Shipper* and *Cancel Order* cannot be imposed. The control pattern and the recognizer for the supervisor with this approach is shown in Table 6.1.

Table 6.1. Control Pattern

	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	-	-

Table 6.1 (Continued)

	b_2	fa_2	c_2	b_3	fa_3	c_3	b_4	fa_4	c_4	b_6	fa_6	c_6
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	-	-	-	-	-	-	-	-	-	-	-	-

Table 6.1 (Continued)

	b_2	fa_2	c_2	b_3	fa_3	c_3	b_4	fa_4	c_4	b_6	fa_6	c_6
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	-	-	-	-	-	-	-	-	-	-	-	-

Consider begin event for order book task b_3 in table 6.1. This event is disabled (b_3 =0) until the system reaches state 3 or state 4 i.e. until credit card authorization task either commits or aborts, order book task cannot begin. This is due to the terminating dependency between credit card authorization task and order book task which specifies that order book task cannot commit or abort, unless credit card authorization task commits or aborts.

Similarly consider an abort dependency between credit card authorization task and order book task (T2 AD T_3), which specifies that if credit card authorization task abort then order book task must abort too. Hence at state 7, which represents that credit card authorization task is in abort state and order book task is in done state, event c_3 is disabled, as this event takes the system to a state where credit card authorization task is aborted and order book task is committed which violates abort dependency between these task.

Chapter 7
Conclusion and Future Research

In this chapter: Section 7.1 lists the contributions of the research; Section 7.2 conclusion of the research; Section 7.3 discusses future research directions.

7.1 Contribution

- We have modeled and analyzed the workflow process definition using the state avoidance and string avoidance techniques as described in Section 4.2 - 4.4 which provides insight to the modeling and control of workflow systems as DES.

- The Methodology for modeling control-flow specifications (business policies) presented in Section 5.2.3.1 is a valuable contribution as this facilitates automatic design of workflow controllers.

- We have provided a mathematical framework based on Finite Automata formalism for modeling, control and analysis of workflow process definitions. Within this framework uncontrolled events are considered and tasks, and dependencies are modeled separately leading to robust system design (Chapter 5).

    - The framework presented is independent of task structure and can be used with two-phase commit, one-phase commit and zero-phase commit task structures.

    - The framework facilitates identification of inconsistency in business policies as illustrated in Section 6.1.5.1.

- The existence of nonblocking supervisor for workflows show in Section 5.2.3.4 is a valuable contribution as it facilitates computing the workflow model which guarantees certain desirable properties (deadlock free, livelock free and safe termination).

7.2 Conclusion

In this research, we have modeled workflows using Finite Automata. A finite automaton provides the basis for formal proofs that develops a higher confidence in the correctness of the workflow; it also facilitates analysis of the workflow, which reduces human error and cost of testing.

More specifically, the workflow process definition is modeled using supervisory control theory, where both the uncontrolled process and the specification (business policies) are modeled separately and then combined to obtain the controlled process. If business policies are updated or new business policies are added, it is not necessary to remodel the system. Only the specification model needs to be modified.

Several properties of supervisory control theory such as nonblocking, controllability, closure, accessibility and co-accessibility have been discussed. These properties are effectively used for identifying inconsistencies in business policies, testing for safe termination of the workflow (process) and checking for deadlocks and livelocks in the workflow.

In short; there is a need for standardization in the problem definition tool. The rapid increases in application domains that use workflow management systems; requires a formal framework that can be used by various workflow product developers to implement these applications. To achieve this goal we have provided a formal comprehensive framework for modeling a process definition, that can be used at design time for modeling and analysis of workflow applications

7.3 Future Research

We have modeled the workflow processes using a centralized supervisory control architecture, where each dependency (specification) is expressed as finite automata and the specification model is generated by taking a shuffle product of individual dependency (specification) automata. However the result suffers from an exponential state space increase. For example, the shuffle product of n specification automata with each automata having m states, results in m^n states.

Similarly the uncontrolled process model is the shuffle product of individual task automata. Hence as the number of tasks increase, the uncontrolled process model suffers from a state space explosion. The point we want to illustrate is that centralized control architecture suffers from scalability. Moreover some of the systems are distributed, heterogeneous and autonomous in nature, and therefore do not lend themselves to centralized control.

In this work we have dependencies (business policies) to described restriction on the execution of the workflow which are of control-flow type; however there can be other types of business policies that are based on the output value generated by certain task or some external factors such as time.

The fore-mentioned computational complexity and state space explosion can be resolved by modeling the workflow as a modular and decentralized supervisory control. Whereas a workflow with business policies based on output value or time can be model using high-level finite automata formalism.

References

1. N. R. Adam, V. Atluri and W. Huang, "Modeling and Analysis of Workflows Using Petri Nets", Journal of Intelligent Information Systems, Special Issue on Workflow and Process Management, pp. 131-158, March 1998.

2. G. Alpan, "Design and Analysis of Supervisory Controllers for DEDS", Ph.D. Discertation, Rutgurs University, 1997.

3. P. C. Attie, M. P. Singh, A. Sheth and M. Rusibkiewicz, "Specifying Interdatabase Dependencies," Proceedings 19th International Conference on Very Large Database, pp.134-145, 1993.

4. C. G, Cassandras, "Introduction to Discrete Event Systems," Kluwer Academic Publishers, 1999.

5. P. Chrysanthis, "A Framework for Modeling and Reasoning about Extended Transactions" PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, 1991.

6. A. K. Elemagarmid, Y. Lue, W. Litwin and M. Rusinkiewicz, "A Multidatabase Transaction Model for InterBase," Proceeding of 16th International Conference on Very Large Database, pp 507-518, 1992.

7. A. K. Elmagarmid, "Database Transaction for Advance Applications," Morgan Kaufmann, 1992.

8. E. A. Emerson, "Temporal and Model Logic," In Handbook of Theoretical Computer Science, Volume B, 1990.

9. H Garcia-Molina and K. Salem, "SAGAS," In Proceedings of the ACM SIGMOD, International Conference on Management of Data, pp 249-259, 1987.

10. J. N. Gray, "The Transaction Concept: Virtues and limitations," Proceeding of the 7th VLDB, September 1981.

11. J. Gray and A. Reuter, "Transaction Processing: Concept and Techniques," Morgan Kaufmann, 1993.

12. D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin, “Chronological Scheduling of Transactions with Temporal Dependencies,” The VLDB Journal, pp. 1-28, 1994.

13. Georgakopoulos, M. Hornick and A. Sheth, “An Overview of Workflow Management: From Process Modeling of Workflow Automation Infrastructure”, Distributed and Parallel Database, pp 119-153, August 1995.

14. R. Gunthor, “Extended Transaction Processing Based on Dependencies Rule,” Proceedings of RIDE-IMS, pp. 207-214, 1993.

15. D. Hauschildt, H.M.W. Verbeek and W.M.P. van der Aalst, “WOFLAN: A Petri Net Based Workflow Analyzer,” Computing Science Reports 97/12, Eindhoven University of Technology, 1997.

16. T. Haerder and A. Reuter, “Principles of Transaction-oriented Database Recovery,” ACM Computing Survey, 15(4), December 1983.

17. K. Hayes and K. Lavery, “Workflow Management Software: the Business Opportunity,” Technical report, 1991.

18. J. Klein, “Advanced Rule Driven Transactional Management,” Proceeding of the IEEE COMPCON, 1991.

19. T.M. Koulopoulos, “TheWorkflow Imperative,” Van Nostrand Reinhold, 1995.

20. 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.

21. C. Liu, and D Keo, “Modeling and Scheduling of Transactional Workflows”, January 1996.

22. J. Moss, “Nested transaction: An Introduction,” In Bhargava, 1987.

23. P. J. Ramadge and W. M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," SIAM J. Control and Optimization, vol. 25, no. 1, 1987.

24. P. J. Ramadge and W. M. Wonham, "On the Supremal Controllable Sublanguage of a Given Language", SIAM J. Control and Optimization, vol. 25, no. 3, 1987.

25. A. Reuter, “ConTract: A Means for Extending Control Beyond Transaction Boundaries,” In proceedings of 3rd International Workshop on High Performance Transaction System, 1989.

26. M. Rusinkiewiez and A. Sheth, “Specification and Execution of Transactional Workflows”, Addison-wesley, chapter 29, 1995.

27. M.P. Singh, G. Meredith, C. Tomlinson, and P.C. Attie, “An Event Algebra for Specifying and Scheduling Workflows,” Proceedings 4th International Conference on Database System for Advance Application, pp. 53-60, 1995.

28. J. Tang, and J. veijalainen, “Enforcing Inter-task Dependencies in Transactional Workflow”, Technical Report J-2/95, VTT Information Technology, January 1995.

29. W.M.P Van der Aalst, “A Class of Petri nets For Modeling and Analysis Bussiness Process,” Computer Science Reports, Eindhoven University of Technology, 1995.

30. W.M.P Van der Aalst, “Petri Nets Based Workflow Management Software,” In NSF Workshop on Workflow and Process Automation in Information Systems: State-of-art- and Future Directions, 1996.

31. W.M.P. van der Aalst, “Three Good Reasons for Using a Petri Net-Based Workflow Management System”, Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC’96), pp. 179–201, Nov 1996.

32. W.M.P. van der Aalst, “Verification of Workflow Nets”, Proceedings of Application and Theory of Petri Nets, Volume 1248 of Lecture Notes in Computer Science, pp. 407-426, 1997.

33. W.M.P. van der Aalst, D. Hauschildt and H.M.W Verbeek, “A Petri Nets Based Tool to Analyze Workflows”, Proceedings of Petri Nets in System Engineering (PNSE’97), pp. 78–90, Sept 1997.

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

35. W.M.P. van der Aalst and A.H.M. ter Hofstede, “Verification of Workflow Task Structures”, A Petri Nets based Approach Information Systems, pp. 43-69, 2000.

36. W.M.P. van der Aalst, “Workflow Verification: Finding Control- Flow Errors Using Petri-Net- Based Techniques”, Business Process Management, LNCS 1806, pp. 161-183, 2000.

37. W.M.P. van der Aalst, “Workflow Management,” MIT Press Cambridge, 2002.

38. C. Wallace, P. Jensen and N.Sopparkar, “Supervisory Control of Workflow Scheduling,” Proceedings of ATMA, pp. 36-46, 1996.

39. WFMC, “Glossary, A Workflow Management Coalitions Specification,” Technical Report, Workflow Management Coalition, 1994.

40. WFMC, “Workflow Management Coalition Terminology and Glossary,” Technical report, Workflow Management Coalition, 1996.

41. D. Wodtke and G. Weikum, “A Formal Foundation for Distributed Workflow Execution Based State Charts,” Proceedings 18th International Conference on Database theory, 1997.

42. D. Worah, A. Sheth, “Transactions in Transactional Workflows”, Advanced Transaction Models and Architectures, Kluwer Academic Publishers.1997

Appendices

Appendix A: Types of Dependencies [5]

- *Begin Dependency* (t_j BD t_i): task tj cannot begin execution until task ti has begun.

- *Abort Dependency* (t_j AD t_i): if task t_i aborts then task t_j aborts.

- *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 .

- *Strong Dependency* (t_j SD t_i): if task t_i commits then task t_j commits.

- *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 words if task t_j commit and task t_i aborts then the commitment of t_j precedes the abortion of t_i.

- Terminating Dependency (t_j TD t_i): t_j cannot commit or abort until t_i either commits or aborts.

- *Exclusion Dependency* (t_j ED t_i): if task t_i commits and task t_j has begun executing then task t_j aborts.

- *Forced Commit on Abort Dependency* (t_j FCAD t_i): if task t_i aborts then task t_j commits.

- *Serial Dependency* (t_j SD t_i): t_j cannot begin executing until t_i either commits or aborts.

- *Begin on Commit Dependency* (t_j BCD t_i): t_j cannot begin executing until t_i commits.

Appendix A (Continued)

- *Begin on Abort Dependency* (t_j BAD t_i): t_j cannot begin executing until t_i aborts.

- *Weak Begin on Commit Dependency* (t_j WBAD t_i): if t_i commits, t_j can begin executing after t_i commits.

Table A1. Dependencies Classification [1]

Precedence Dependencies	*Weak Causal Dependencies*	*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 B: Standard Algorithm for • C [4]

This algorithm is used to compute the supremal controllable sublanguage as explained in Section 4.2.2.3. The general description of the algorithm is as follows:

The language M is generated by uncontrolled process model G i.e. $L(G)$ = M. The language $L_{am} \subseteq$ M which is marked by automata H. The goal is to calculate $L_{am}^{\uparrow C}$ with respect to M and $\sum_{uc}$ where, $\sum_{uc}$ is uncontrolled event set.

In step 1 H_0 is first calculated, which is a recognizer for admissible behavior L_{am}. Step 2 is an iterative procedure, in this step for iteration number 1 the states of automata H_0 is checked if any of the states violates the active event set constraint imposed by the controllability conditions describe in Section 4.1.2.2 and are deleted from automata H_0. Then in Step 2.2 trim operation is performed on automata H_0 which produces H_1. The same procedure is repeated until $H_{i+1} = \varnothing$ or $H_{i+1} = H_i$.

If $H_{i+1} = \varnothing$, the algorithm terminates at step 2.2 and $L_{am}^{\uparrow C} = \varnothing$, whereas if $H_{i+1} = H_i$ the algorithm terminates at step 3 and $L_{am}^{\uparrow C} = L_m(H_{i+1})$.

B.1. Step 0

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

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

B.2. Step1

Let $H_0 = (Y_0, E, g_0, W_{H0}, (y_0, x_0), Y_{0,m}) = H \times G$

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 $L(H_0) = \overline{L}_{am}$.

Appendix B (Continued)

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

Set i = 0.

B.3. Step 2: Calculate

B.3.1 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"

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

B.3.2 Step 2.2

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

If H_{i+1} is the empty automaton, i.e. (y_0, x_0) is deleted in the above calculation, then $L_{am}^{\uparrow C} = \varnothing$ and stop

Otherwise, set

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

B.4. Step 3

If $H_{i+1} = H_i$, then

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

and STOP. Otherwise, set i⟵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.