A State Machine Approach to Reliable and Dynamically Reconfigurable Distributed Systems

Alvin Sek See Lim

Technical Report #1132

January 1993

A STATE MACHINE APPROACH TO RELIABLE AND DYNAMICALLY RECONFIGURABLE DISTRIBUTED SYSTEMS

by

ALVIN SEK SEE LIM

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1993

Abstract

Maintaining consistency among distributed processes in the presence of failures and reconfigurations is a difficult problem, especially when the processes may synchronize with one another in complex ways and execute for a long period of time. Atomic transactions are commonly used to simplify the management of concurrency and failure by preserving serializability and failure atomicity. The major drawback of preserving these properties is that they restrict the types of synchronization that can be specified. This thesis focuses on a general set of correctness conditions for preserving consistency based on a general synchronization model of process interaction.

Instead of extending established but inappropriate concepts used in atomic transactions, this thesis explores three more fundamental questions: (1) How can consistency be maintained without enforcing serializability and failure atomicity? (2) How can applications be specified that will assist the system maintain consistency automatically? (3) How can we implement mechanisms for managing synchronization, recovery and dynamic reconfiguration uniformly?

To preserve consistency in the presence of a failure or reconfiguration, we introduce a general set of conditions that guarantee the correctness of recovery and dynamic reconfiguration of applications that need not be serializable or linearizable. These conditions are based on a basic definition of interactive consistency. Existing recovery techniques, including those that exploit application-specific semantics, satisfy these conditions.

We present algorithms for automatically checking these conditions from application behaviors specified in a hierarchical finite-state machine model. These correctness conditions, defined as properties of finite-state machine graphs, enable us to improve recovery efficiency by exploiting the permutation and substitution of operations allowed by the behavior specification. They also permit combinations of different types of recovery methods to be used in a recovery.

We have separated the policy information for maintaining consistency from the mechanism that implements them. We exploit this in an implementation of a uniform set of control mechanisms that use these information to maintain consistency in the presence of concurrency, failure, and reconfiguration.

Acknowledgements

I would like to thank my advisor, Prof. Stuart A. Friedberg, for the many constructive advices and insights that helped shape and direct this research. I very much appreciate his patience and effort in checking countless revisions of the concepts and presentation. He has always been a refreshing spring of knowledge that has strengthened my understanding of distributed systems and sharpened my research skills.

I would also like to thank the other members of my committee: Prof. Mike Carey, Miron Livny, Marvin Solomon, and Parmesh Ramanathan. In particular, I like to thank Mike Carey for introducing me to the problems and approaches to concurrency control and recovery in advanced database applications; and Marvin Solomon for the helpful discussions on the model and synchronization problems.

Finally, but most importantly, I would like to thank my wife Kim Choon for her love, support and encouragement, and my children, Joanna, Sharon, and Isaac for their vivacity, great sense of humor, and the diversions. Kim Choon deserves a big round of applause for enduring these years in a student housing and raising those children single-handedly. It's about time things change for the better. I thank God for faithfully sustaining and enriching us through the years and for this opportunity to examine a minuscule portion of the vast knowledge in this remarkably ordered universe.

Table of Contents

Table of Figures

Table of Algorithms

Chapter 1

Introduction

In many distributed applications, processes synchronize with one another in a complex way and execute for a long period of time. Furthermore, these applications may involve frequent recovery from exceptional conditions as well as regularly scheduled reconfiguration or installation of new modules. Distributed applications with these characteristics include computer-aided automated manufacturing [19,23,26,41,73], multi-transaction database activities [27,83], network service applications [74,82], and process control [76].

A major problem in these applications is maintaining consistency in the presence of concurrency, failure and reconfiguration. Atomic transactions are commonly used to simplify the management of concurrency and failure by preserving serializability and failure atomicity. Each transaction can be viewed as a sequential execution whose intermediate results are not visible to other concurrent transactions. The drawback of preserving the serializability and failure atomicity properties is that they restrict the types of synchronization that can be specified. The mismatch between the properties of atomic transactions and the characteristics of general distributed systems is discussed in [13]. Instead of making incremental extensions to transactions, this thesis explores a new approach based on a general synchronization model of process interaction. It allows software designers to specify general interactive behavior. We introduce a novel set of correctness conditions for ensuring consistency that do not require applications to be serializable or atomic. These conditions allow semantics of applications to be exploited to enhance concurrency and the control of recovery and dynamic reconfiguration. Existing recovery techniques, including those that exploit application-specific semantics, satisfy these conditions for correctness of recovery.

In this thesis, we concentrate on applications such as automated manufacturing where distributed computerized control systems are increasing in number and complexity [23,26,41]. This work is also applicable to other application domains. Consider an automated manufacturing system that requires coordination of several workstations and material handling robots to produce a family of products from streams of raw material. Systems like this pose many challenging problems. First, concurrent

operations may be non-serializable. Operations of the workstations and robot movement must be synchronized in complex ways to avoid interference and enforce cooperation. Second, backward recovery may be impossible if failure involves non-recoverable physical effects; forward recovery may be the only option. Failure may occur in various components: machine tools, equipment, robots, computer control system, etc. In a factory environment, failure is more a routine than an exception. Third, the configuration of these systems often changes over time. From an economic standpoint, these machines should maintain continuous operation as much as possible. For manufacturing facilities to maintain the competitive edge, it is important to provide the capability for automated manufacturing applications to be dynamically reconfigured to meet the rapid fluctuation in market demands for a range of existing and new products. Configuration change may also result from regularly scheduled maintenance and from periodic upgrades because of improvement in technology or design. Conventional synchronization and recovery schemes, as well as existing automated manufacturing software, provide poor support for reconfiguration.

Consider the following example of a manufacturing cell (Fig. 1.1) that consists of a cutter, two milling machines, an assembler, and a robot. A workpiece is first cut, then each piece is milled according

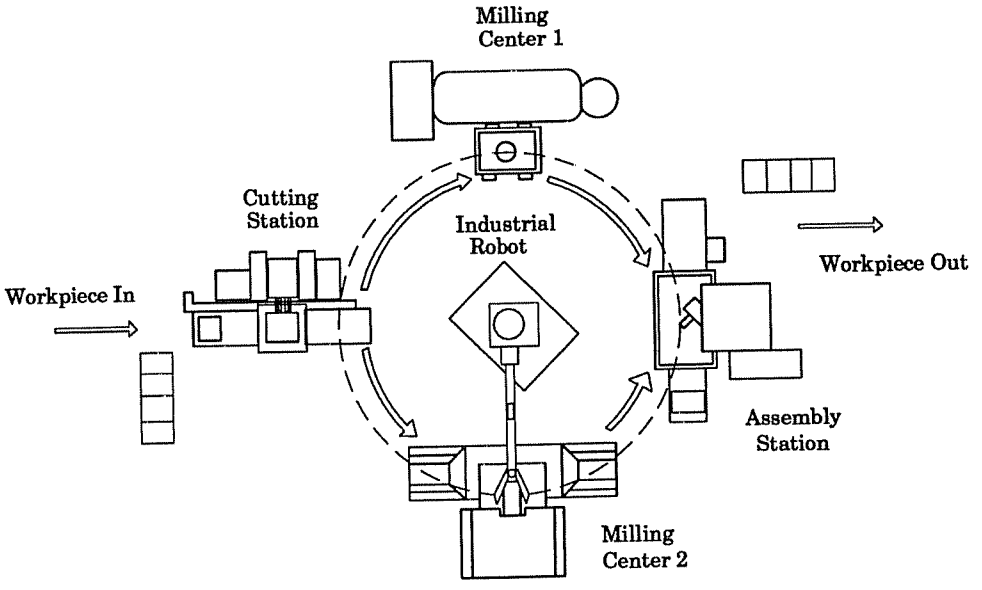


Figure 1.1. A Manufacturing Cell

to different specifications, and finally the pieces are assembled together. The robot is responsible for transporting the workpieces from one machine to another, constrained by the timing of the operations and conditions of the machines. Each machine operates for a long period in a cyclic manner, i.e. repeatedly waiting then performing its function. This example will illustrate our synchronization, recovery, and reconfiguration goals. First, we need to synchronize the cyclic operation of the individual machines to obtain correct behavior. For example, the cutter must have finished before the robot tries to grab a workpiece, and the milling machine must be idle before the robot tries to deliver one. Next, we must provide for the inevitable failures or exceptions that will arise during an operation. If a tool breaks during a milling operation, we must recover the group of machines to an acceptable state, which may mean discarding the otherwise unaffected piece being processed by the second milling machine. Finally, and critically in the automated manufacturing domain, the programs controlling each machine must be replaceable without shutting down the whole group of machines. Such program changes are common for many reasons: improvements in production processes, engineering design changes, or quick switches to producing a different part in a family of related parts, etc. Today's automated manufacturing software is poor at reconfiguration without downtime. Approaches in [65, 85] require the manufacturing system to be stopped for newly generated software to be installed and initiated. They do not address the problem of maintaining consistency of other components that interact with the changed components. All the changes just mentioned must be managed in the context of a larger factory, where our example cell is just one part of a production system and the workpiece is a part of a larger job.

Our approach to solve these problems has the following important characteristics. First, it automatically determines those processes that are unaffected by the failure or reconfiguration and allows them to continue their normal operations. For example, a machine in a manufacturing cell can be upgraded or replaced without shutting down the entire group of machines. In the presence of failure and reconfiguration, the system maintains consistency by automatically analyzing how the application behaves under various conditions and the dependency between operations. Designers are relieved of the burden of checking for consistency. Second, processes need not be prevented from interacting with others within well-defined boundaries (e.g. begin and commit transaction). Furthermore, in the presence of failure or reconfiguration, processes may be recovered to some intermediate execution points. Third, synchronization, recovery, and dynamic reconfiguration is controlled in a unified way based on an open

model that allows combination of different recovery or reconfiguration techniques to be implemented.

1.1. Fundamental Problems and Thesis Goals

A fundamental problem in managing these general, reliable and reconfigurable distributed systems is maintaining consistency in the presence of failure and reconfiguration. A good solution to this problem should possess three important properties: (1) it should allow general interaction patterns among concurrent processes, (2) it should check for consistency automatically, and (3) it should exploit semantics of the application to enhance recovery and affect the fewest processes. This thesis explores a novel solution with all these properties.

Existing reliability techniques do not possess all these properties. Techniques based on message-logging and checkpointing, either the pessimistic or optimistic approach, do not allow semantics of the applications to be used to improve message logging, checkpointing, and recovery. Furthermore, they often involve high overhead because of the cost of message logging to stable storage as well as cost of correlating independent message logs of distributed processes.

Atomic transactions simplify the problem of preserving consistency by allowing concurrency and failure (or abort) of other transactions to be ignored when checking for consistency of a particular transaction. This is made possible since every concurrent transaction execution is equivalent to some serial execution and every committed serial transaction is consistent. Unfortunately, this property is often inappropriate for application with irregular interactions. The key problem can be illustrated as follows.

Consider two concurrent activities, A and B, representing the following sequences of operations:

$$A = <a_1, a_2, ..., a_n>$$
$$B = <b_1, b_2, ..., b_m>$$

Suppose the actual interleaving of the execution of the operations is S:

$$S = <a_1, b_1, b_2, a_2, ..., a_n, b_3, ..., b_m>$$

If the properties of the operations b_1 and b_2 permit them to commute with all the operations a_i, where $1 < i \leq n$, then S is equivalent to a serialized sequence S':

$$S' = <a_1, a_2, ..., a_n, b_1, b_2, ..., b_m>$$

However, in many general distributed applications, b_1 may need to be synchronized after a_1 and a_2 after b_2. In such cases, A and B cannot be serialized and it would be inappropriate to define A and B as transactions (or sub-transactions of a nested transaction) since transactions (and sub-transactions) must be serializable. One solution to this problem is defining each individual sub-operation a_i (and b_j) as a transaction. This would however increase the overhead of beginning and ending transactions. Furthermore, some committed sub-transactions may be affected by a failure and the top-level transaction may need to be rolled back entirely. Unless the system analyzes the dependency of other sub-transactions on the recovered ones, which existing recovery techniques do not, then all sub-transactions which *may* interact with the recovered ones must also be recovered. Another solution to this problem is to define cooperating (nested) transactions with A and B as sub-transactions whose intermediate results are visible to each other. However, failure of a sub-transaction, say A, would require rollback of both sub-transactions and may cause large amount of computation and physical work of B to be wasted. A better solution, explored in this thesis, is to recover to some intermediate state that is consistent, e.g. after a_1 is executed, without incurring overhead to ensure recoverability of every operation. This approach does not require either serializability or failure atomicity to be preserved.

The main goal of this thesis is to define and maintain consistency among distributed processes in the presence of failure and reconfiguration, without requiring them to be serialized with one another. They may synchronize with one another in complex ways and execute for a long period of time. We are interested in reducing wasted computation and physical work during concurrent execution, failure recovery and dynamic reconfiguration. To achieve this, we define two sub-goals. The first sub-goal is to define the general conditions for preserving consistency of distributed processes in the presence of concurrency, failure and reconfiguration. They should allow more general interaction patterns and permit higher concurrency than approaches that enforce serializability and failure atomicity. The second sub-goal is to explore a way to specify distributed applications so that their semantics can be easily exploited for automatic checking of all properties necessary for maintaining consistency. For example, dependencies in the above example may be checked automatically. The conditions should allow an application to recover to an intermediate state that is consistent when a failure occurs, thus avoiding destruction of some of the previous work. A secondary goal of this thesis is to show that a common set of control mechanisms can be used to manage synchronization, recovery, and dynamic reconfiguration in a

unified way, whereby different techniques for recovery and dynamic reconfiguration can be easily incorporated.

1.2. Summary of Approach and Results

Our approach exploits the semantics of the application defined by a behavior specification. It is general in the sense that it allows irregular interactions among the processes. This sub-section presents an overview of this approach and the main results.

We use a partial-order model because it permits more general behavior of distributed processes than those requiring sequentiality which restrict the pattern of interactions. Although sequential models can define partial order by permitting two orderings, say <a,b> and <b,a> of non-conflicting atomic operations a and b, this definition depends on which operations are considered to be atomic[1]. It is more appropriate to reason about operations that execute for some duration and analyze their interactions. Furthermore, sequential models cannot easily define partial orders based on condition synchronization and event triggers, where a process may wait for some condition to be satisfied or an event to occur. Arbitrary partial ordering of operations may be possible.

We first define consistency based on a partial-order model of concurrency. We use a fundamental notion of consistency that is preserved when every process uses only information or conditions, established by another process, that are not removed by exceptional events, such as failure or recovery. Based on this notion of consistency, we introduce a general set of conditions that guarantee the correctness of recovery and dynamic reconfiguration. Existing concurrency and recovery correctness conditions based on some notion of sequentiality [10,51] preserve consistency in a conservative way by isolating intermediate results of atomic operations. This is inappropriate in many distributed applications where intermediate results of a process may be visible to other processes and their operations may not be serialized.

In our model, software designers specify the behavior of applications with a hierarchical finite-state machine that permits direct automatic checking of the conditions for preserving consistency of applications. We present an algorithm that automatically determines dependencies between pairs of

[1] If sub-operations of a and b interact with each other, their sub-operations must be made atomic although a and b are not. In practice, this may not be possible, e.g. when failure modes of a or b may affect their completed sub-operations.

operations from a behavior specified in the model. These dependencies are used by a second algorithm that automatically checks the correctness conditions and finds a correct sequence of operations for recovering an application from a failure. Checking the fundamental consistency conditions directly from the specified behavior of applications permits us to improve recovery efficiency by exploiting the permutation and substitution of operations allowed by the application semantics. Recovery of a failed process may not affect another process if the other does not actually depend on the failed process (though there may be apparent dependencies). Other approaches that allow general and irregular interaction patterns, such as message-logging and checkpointing [39, 79], do not exploit semantics of the application to improve recovery in the presence of failure and aborts. By analyzing the behavior of applications, we can also improve efficiency by checkpointing only states that would be most important for recovery purposes.

We extend the above correctness conditions appropriately to preserve consistency during dynamic reconfiguration where the behavior of an application may be changed. We present an algorithm for finding a correct sequence of reconfiguration operations in which these correctness conditions are automatically checked. Intermediate behavior of the application during reconfiguration is derived from the current and desired configurations and is used for controlling normal executions of processes unaffected by the reconfiguration.

The behavior of the application specified by the designer and the results of the analyses of correct recovery sequences are used to control the execution of the application. Since the conditions for preserving consistency can be checked automatically solely from the behavior specification, we can separate the common control mechanism from the (policy) information that maintains the behavior and consistency of distributed applications. The advantage of this over mechanisms with built-in policies is that we can then control synchronization, recovery and dynamic reconfiguration in a uniform way. This separation also enables different techniques of recovery and dynamic reconfiguration to be specified by changing the policy that affects the selection of recovery and reconfiguration operations without any change to the control mechanism. It also simplifies reasoning about various techniques for restoring consistency.

1.3. Scope and Assumptions

In this thesis, we restrict our scope to applications with finite-state behavior to simplify automated checking of dependency and consistency. The finite-state machine model is useful in many distributed applications such as computer control of automated manufacturing [23, 67]. Systems with infinite state behavior, such as unbounded FIFO channels and other systems described in [70], will require an extension to the analysis described here. However, the definition of consistency and the conditions for preserving consistency are applicable to infinite as well as finite-state systems.

Most applications in our target environments permit modular design. Each module can be designed to involve only a small set of local operations. The interaction of a group of modules is defined in their parent module which then presents only an abstraction of the group operations to a higher parent module. With hierarchical design, we can limit the size of the finite-state machine describing the behavior of each module. This assumption is reasonable in automated manufacturing where control is clustered in groups at each level in the factory hierarchy: workstation, manufacturing cell, shop floor, factory, and corporation [41]. Applications can be modularly specified whereby only abstracted behavior of each module is visible to higher level modules. Each module should contain few processes whose collective behavior can be specified by a reasonably small finite-state machine. The behavior of a module that is visible to other modules is an abstraction of its internal behavior. We assume that this abstraction represents a behavior with a substantially smaller finite-state machine than that of the internal behavior.

We focus mainly on the issue of maintaining state consistency of distributed applications in the presence of concurrent execution, failure and dynamic reconfiguration. We concentrate on the framework and the algorithms for defining and maintaining consistency during failure recovery and dynamic reconfiguration. We will not concentrate on other aspects of designing distributed systems, such as linguistic support and efficient implementation of monitoring and control mechanisms. We will not address the issue of maintaining type consistency during dynamic reconfiguration.

We assume that communications and the execution control managers are reliable. Failures of communication and the manager can be handled by known solutions, such as using replication and election. By avoiding these issues, we simplify our discussion on the novel aspect of this work.

1.4. Plan of the Thesis

Maintaining consistency is the key issue in three major inter-related aspects of distributed systems: synchronization, failure recovery and dynamic reconfiguration. In Chapter 2, we summarize the approaches of other work on each of these aspects. In discussing other work on synchronization, we concentrate on work related to failure recovery and reconfiguration. We also mention work that provides a uniform framework for managing different recovery techniques with our wider goal of combining the management of these three aspects under a uniform framework.

In Chapter 3, we introduce our approach by first defining a more fundamental notion of consistency than those based on sequentiality. This notion of consistency captures the partial-order semantics of concurrent execution. We show the advantage of this over other notions of consistency, particularly serializability and linearizability.

We choose an appropriate model of computation and specification – a hierarchical finite-state machine model – that will enable us to exploit this notion of consistency. In Chapter 4, we discuss how we can model the behavior of applications, including synchronization, failure and recovery. (The model for dynamic reconfiguration is discussed in Chapter 10.) We also present the analysis of the specification, which includes detection of various synchronization problems. This analysis plays an important role in defining certain failures and managing recovery. In Chapter 5, we describe how we can provide a uniform mechanism for controlling synchronization, recovery and reconfiguration using policy encapsulated in the specification defined in this model. We discuss how a specification is compiled and analyzed. The results of analysis are then used by a consistency control manager to control synchronization, recovery and dynamic reconfiguration.

While the finite-state machine model allows us to specify the behavior of application, it does not allow us to reason about the actual execution. In Chapter 6, we introduce the model of execution sequences that is crucial for analyzing consistency in the presence of failure and reconfiguration. We describe how the execution sequence model is related to the FSM behavior specification as well as failure and recovery events.

Chapter 7 discusses the correctness conditions in terms of the finite-state machine model. We also present recovery algorithms that check for these correctness conditions and exploit the semantics of

applications derived directly from their specifications. In Chapter 8, we discuss implementation issues of recovery techniques that will enable these conditions to be satisfied as well as improve efficiency. Various recovery techniques can be implemented as different recovery policies that are separated from the control mechanisms.

In Chapters 9 and 10, we generalize our scope to include dynamic reconfiguration. In Chapter 9, we first augment our model to deal with dynamic reconfiguration. We then discuss the framework and mechanics of dynamic reconfiguration that will allow processes unaffected by the reconfiguration to continue executing normally. As in failure recovery, consistency must be maintained during reconfiguration. This requires generalizing the correctness conditions to such cases where the configuration or behavior of the application may be changed. In Chapter 10, we discuss the implementation of the mechanisms and the policies for managing dynamic reconfiguration.

In Chapter 11, we summarize our results, the significance of our contributions, and some of our experiences. We then discuss some possible further work.

Chapter 2

Related Work

In this chapter, we summarize related work in each of the three important aspects of distributed systems: synchronization, failure recovery and dynamic reconfiguration. We compare them with our approach and focus mainly on the capability and ease of preserving consistency. We also discuss other work that provides uniform support for different recovery techniques and compare with our goal of uniformly supporting these three aspects of distributed system.

2.1. Synchronization

Several state-transition approaches to specifying distributed applications have been used in the past. Lam and Shankar [50] use a finite state machine model for specifying and verifying communication protocols from a user's description of the properties. Clarke, et. al. [21], use another approach for automatic verification of concurrent systems using temporal logic [59] specifications provided by the users. CCS [62] is suitable for detecting deadlock but not starvation or livelock without requiring users to specify these properties using modal logic [84]. SPANNER [1-3] allows detection of deadlocks but requires users to specify the liveness properties in terms of finite state machines to verify them. Our approach differs from these approaches in that we provide automatic verification of limited but useful properties, such as freedom from deadlock and possible starvation or livelock, without requiring users to provide the specifications of the properties, and in providing runtime support for failure recovery and system modification. A similar approach for verifying liveness has been used by Gouda, et. al. [29,30], although they did not discuss the general case where only a subset of the machines is involved and did not allow the problems to be avoided dynamically.

Our basic interpretation of finite-state machines is similar to [52] in that states are clean-points (instantaneous) and transitions represent operations requiring some time. This contrasts with the traditional interpretation of finite-state machines where machines may be in a state for a duration of time (possibly executing some operations) and transitions are instantaneous. However, our model contains additional features that enable software desigers to characterize the behavior of applications in

the presence of failure, recovery and dynamic reconfiguration. We also allow hierarchical composition of finite-state machines.

Other models such as Petri nets [68] have been successfully used to model distributed systems. Petri nets have operational semantics and behavior specifications that cannot be easily used for checking consistency. We choose the finite-state machine model because it enables us to analyze consistency in the presence of failure automatically. Furthermore, we can describe hierarchies of finite-state machines using abstraction and composition more easily than with Petri nets. The ability to express abstraction and composition also distinguishes our work from path expression [5, 17].

2.2. Failure Recovery

There are two major classes of techniques to reliable distributed systems: *physical redundancy*, implementing multiple copies of hardware or software; and *time redundancy*, re-executing operations of failed and affected processes.

An example of physical redundancy is replicated servers. Despite the failure of some servers, the service will still be provided as long as at least one server is functional. ISIS/Meta [12, 60] provides mechanisms and an environment for monitoring and controlling fault-tolerant distributed applications using replicated servers. In [75], a state machine model has been used for designing reliable distributed systems based on replication. There are two main problems with the replication approach. First, accesses to replicated servers need to be synchronized correctly to maintain consistency. Second, the cost of replicating very large objects can be prohibitive.

Techniques based on *time redundancy* include transactions and various forms of checkpointing. Physical redundancy and time redundancy methods are orthogonal to each other; time redundancy is usually implemented using some physical redundancy, such as logs and checkpoints. Here we will only discuss approaches based on time redundancy.

There is a spectrum of reliable distributed systems that preserve consistency in the presence of failure. At one extreme are systems that allow general synchronization but have less efficient recovery management while at the other extreme are those based on atomic transactions with simpler recovery management but have restrictive concurrency control. A good system should allow general synchronization and at the same time provide efficient recovery.

A common approach to reliable distributed systems is rollback recovery based on message-logging and checkpointing [39,44,72,79]. Interactions through messages are used to determine a consistent recovery line; that is, a set of checkpoint states of each process. Determining a consistent recovery line solely from interaction information is too conservative because some interactions may not be crucial to the execution of some processes. For instance, most messages from sensor probes do not exceed some threshold value and do not affect the operations of the receiving process. There is then no actual dependency between the process controlling the sensors and the receiving process, although there is an apparent dependency from analyzing the messages. The drawback of not determining actual dependency is that this technique is inefficient and causes more processes to recover than necessary. Whether an interaction is crucial can only be determined from the semantics of the application and not from the interaction history alone. The cost of logging messages to stable storage and the analysis required to merge independent message logs can be prohibitive, especially for distributed applications involving many messages. Other variations [42,43,71] of this recovery technique also do not exploit semantics of the application to detect actual dependence.

Another common approach to reliable distributed systems is based on atomic transactions [22,56,57,77,78] that preserve the following properties: all-or-nothing, consistency, isolation, and durability [32]. The *all-or-nothing* property guarantees that either the transaction completes or it has no effect even though it may have failed. This property is sometimes called *recoverability* [31] which allows uncommitted transactions to be aborted without invalidating the semantics of committed ones. The *consistency* property ensures that when the system starts in a consistent state, a transaction will bring it to another consistent state. The *isolation* property ensures that within the boundary of a transaction there is no interaction with other transactions. *Durability* guarantees that the effects of a committed transaction are not lost by a failure. It is debatable whether all these properties are essential or merely convenient for designing reliable distributed systems. The all-or-nothing and isolation properties are too strong for the class of systems in which we are interested. We would like to be able to recover to intermediate execution points of a process while allowing it to interact with others. Isolation is enforced in transactions to simplify recovery but it restricts the range of permissible synchronization. Even in implementations of transactions, recovery methods have been known to affect concurrency control [6,31,88]. In more general applications, we believe the right approach is to allow general

synchronization and yet provide appropriate mechanisms to control recovery.

While some work [7,28,35,77,87] extends the traditional transaction concept by exploiting semantic knowledge to provide more concurrency, we have started from a general synchronization environment and made it reliable. There is a limitation on extending transactions because all transactions must eventually satisfy the serializability or recoverability criteria. This is inappropriate for many distributed applications that usually involve non-serializable and non-recoverable operations as will be discussed in Section 3.4. Atomic abstract data types [77,87] require transactions to be serializable after non-dependent operations are commuted. In atomic abstract data types, as in ACTA [20], the burden of analyzing commutativity (or dependency) between operations is placed on the programmers. Optimistic approaches [35,49] also require committed transactions to be serializable. Cooperative transactions [7] extend basic nested transactions [63], but still require the partial order of lower level nested transactions to be equivalent to a total order of operations invoked by subtransactions of the cooperating transaction. Although a protocol, i.e. a set of rules that restricts admissible execution, may allow nonserializable execution, it is unclear how recovery from a failure can be done efficiently.

In contrast, our approach does not require concurrent processes to be serializable. They may involve non-commutatable interactions. When failure occurs, processes may recover to some intermediate execution points, i.e. the all-or-nothing property need not be preserved. Consistency is preserved by automatically checking for all operations dependent on those recovered operations and recovering them.

In our model, we exploit FSM specifications of the behavior of applications to increase concurrency and improve efficiency of recovery. We check for dependency by analyzing properties of the FSM specifications. Furthermore, the FSM model allows us to generalize failure recovery to processes with non-sequential behavior that is visible to other processes. In contrast, only sequential visible behavior[1] is permitted in transactions. We achieve this flexibility for handling reliability in a general synchronization environment without much compromise on modularity. The behavior of each process can be programmed independently without considering how other concurrent processes might be

[1] Any non-sequential behavior of a transaction is internal and is not observable by external processes. This is true even in "non-serializable" transactions [46,87] if we commute non-dependent operations.

affected. Allowable interactions between processes are defined separately in a composite specification for the group of processes. We allow groups of processes to be hierarchically composed. The specification of a group of processes can be used for composing higher level groups where individual process behavior is hidden at higher levels. As will be discussed in Chapter 4, a composite specification has the power of predicate path expressions [5, 17] or serializability constraints to synchronize independently programmed processes.

2.3. Dynamic Reconfiguration

Durra [8] and HPC [55] deal primarily with the problem of structural reconfiguration, where groups of modules can be recomposed via links, but did not deal with the problem of maintaining state consistency. HPC provides the mechanisms for users to define abstraction and composition of processes. This thesis supplements systems like HPC and Durra by preserving state information during exceptional operations such as dynamic reconfiguration and recovery.

Conic [47, 48, 58] differs from HPC and Durra by managing state consistency correctly during dynamic reconfiguration. It allows user to create and dynamically interconnect modules through well-defined interfaces. Since it is based on transaction concepts, dynamic reconfiguration is done only at "quiescent" states of modules. We avoid this limitation by allowing users to reconfigure modules at arbitrary legal states as long as appropriate recovery and reconfiguration transitions are defined.

Bloom [16] built a dynamic reconfiguration facility over the transaction mechanism of Argus. It provides a framework for analyzing the legality of replacing objects and managing states of replaced objects. This work is however limited in the sense that the behavior of the new configuration can only be the same as (or a superset of) the behavior of the old configuration. The analytical framework is not appropriate for cases where the behavior of the new configuration differs from that of the old configuration, which is common in flexible manufacturing systems. Our work also allows more general synchronization and interaction between objects than client-server transactions.

Other aspects of dynamic reconfiguration has been addressed in other work and are omitted here. Dynamic reconfiguration, implemented in Polylith [69], addressed the practical problem of capturing and restoring states although they did not address the issues of maintaining state consistency. In [33], Hailpern and Kaiser mainly address the problem of maintaining type consistency during dynamic

reconfiguration. In this paper, we will not address those aspects of reconfiguration that relate to safety, protection, and heterogeneity.

2.4. Uniform Support Framework

A support framework for distributed applications should have two important characteristics. First it should manage normal concurrent operations as well as failure recovery and dynamic reconfiguration in a uniform way, through common control mechanisms. Second, it should allow different implementations and techniques of recovery and reconfiguration to be used easily, possibly in combination. The framework should be responsible for maintaining consistency and making it transparent to the software designer.

Two existing systems that provide a uniform framework are based on transactions and have the restrictions discussed in Section 2.2. The first is the Argus transaction mechanism [56,57] with the extension of dynamic reconfiguration mechanism by Bloom [16]. Argus, however, does not provide support for incorporating new recovery techniques, such as compensation used in forward recovery. Furthermore, the reconfiguration mechanism has the limitations discussed in Section 2.3. Conic [47,48] gives only a partial framework; providing support for dynamic reconfiguration but little support for recovery management. We have separated the control mechanism from the recovery and reconfiguration policies that ensure preservation of consistency. This allows uniform control of the normal execution as well as different techniques for failure recovery and reconfiguration.

Other work has concentrated on allowing designers to customize a limited set of application-specific techniques only for recovery management. Several researchers [15,34,57,78] support customizable recovery techniques by separating the underlying mechanisms from the policies that are associated with various recovery techniques and options. These systems provide the basic mechanisms for supporting transactions, e.g. transaction creation and commit, logging, and recovery management. They differ in the level at which the recovery management primitives are exposed. Black [15] proposed an approach of exposing the mechanism for implementing and selecting abstractions that characterize transactions, such as atomicity, consistency, isolation and persistence. QuickSilver [34] exposes a set of primitives that is at a lower level than most other systems, such as Camelot [78] and Argus [57], and permits servers to implement their own recoverable storage and log recovery. While these systems are

based on concepts of transactions, we include other recovery techniques, such as compensation and exception, as well as reconfiguration operations in distributed applications with general and non-serializable inter-process interactions.

Chapter 3

Maintaining Consistency

In this chapter, we define a fundamental notion of consistency in concurrent processes. This definition of consistency is more general than that based on serializability or linearizability. It also differs from message-based consistency by eliminating some apparent dependences due to coincidental message timing.

3.1. Aspects of Consistency

We distinguish between two different aspects of consistency. In distributed systems, the consistency of the state of a process includes both internal consistency and interactive consistency.

Internal consistency denotes conformance to user-defined predicates on information manipulated by processes when each executes in the absence of other concurrent processes. An example is conformance to data integrity specifications. The process manipulating a set of data must leave it in conformance with the data integrity specifications if it originally satisfies them. The programmer must ensure that all processes always preserve internal consistency, even in the absence of other concurrent processes.

Interactive consistency[1] is the condition where a process may depend on an interaction with other processes only if all processes involved in the interaction are in a state subsequent to the interaction. For example, to preserve interactive consistency in message-based systems, a process may receive or process a message only if the sender of the message has not been recovered to a state before sending that message. For the purpose of this definition, an interaction may range from a synchronization event signaling some implicit precondition to a full exchange of arbitrary data to be processed. Dependence on an interaction means the interaction is still relevant to a process's future behavior. We need to consider only information relevant to synchronization and to the level of abstraction in which the process and interaction are defined. Our notion of interactive consistency is at a higher semantic level than the

[1] We use the term *interactive consistency* in a different technical sense from other literature [53, 80]. We chose to use these words because they accurately describe our intended concept. Throughout this thesis, we will use the term in the sense defined here.

notion of global state consistency described in [18,39,44], which is defined at the level of message communication abstraction. By examining the specified behavior of an application, we can determine if a process is semantically dependent on information or conditions established by another process.

Ensuring internal consistency is the responsibility of the programmer but ensuring interactive consistency can be the responsibility of the system. In this chapter, we will examine different ways in which consistency of distributed processes can be preserved. In the remainder of the thesis, we will use the term "consistency" to mean interactive consistency and assume the programmer has ensured that every program is internally consistent.

3.2. Serializability

A simple criterion for preserving interactive consistency is to enforce sequential ordering of the operations. A weaker (and more efficient) criterion than strict sequential ordering is to ensure that the interleaved execution of operations is equivalent to a sequentially ordered execution. *Sequential consistency* is defined in [51] as, "The result of an execution is the same as if the operations had been executed in the order specified by the (sequential) program." (While this notion of consistency originated in the context of multiprocessors, it is also applicable to distributed systems.) This notion of sequential consistency does not place restriction on the time interval in which operations may be moved. It is thus weaker than serializability and linearizability discussed below.

In much work on databases and distributed system, a similar criterion used for transactions (sequences of operations) is *serializability* [9, 10,31,66]. Two sequences of transactions are equivalent if the dependence relation between every pair of transactions in both sequences are identical [24,66]. A transaction T_i is dependent on T_j if T_j writes a data item whose value is then read by T_i. In general, the notion of dependence between operations is derived from the history log and the conflict table: an operation X is dependent on Y if and only if Y is recorded before X in the history log and X and Y conflict with each other. From the actual interleaved sequence of operations, an equivalent sequential ordering can be derived by commuting operations that are not dependent on one another.

Serializability maintains consistency in the presence of concurrency but not in the presence of failure and abort. Transactions must be made recoverable by ensuring that states of committed transactions can be restored and intermediate results of uncommitted transactions can be removed. The

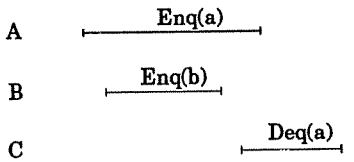
serializability property of transactions simplifies the mechanism for failure atomicity because the intermediate results of uncommitted transactions are not visible to other transactions.

The major disadvantage of preserving serializability is that it is too conservative, allowing only interactions that will result in some equivalent serialization of processes. For example, processes that involve non-serializable interaction (Section 1.2) cannot be implemented as transactions. Many research projects have concentrated on improving concurrency by commuting operations using application semantics or abstract data type specifications [28,77,86]. Even in these approaches, commuted operations must eventually satisfy the serializability criteria.

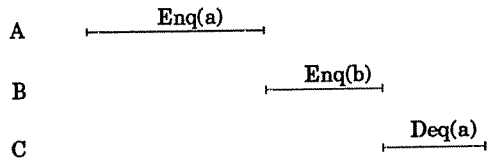
3.3. Linearizability

The linearizability criterion removes some of the restrictions of the serializability criteria by allowing concurrent executions of operations that access shared data objects as long as the results do not violate the semantics of the abstract data types [36,37]. It provides the illusion that each operation appears to take effect instantaneously at some point in time in the interval in which it was executed.

For example, consider three processes *A*, *B* and *C* in Figure 3.1(a), where time flows from left to right and a horizontal line shows the duration of the operation. `Enq(a)` (`Deq(a)`) is an operation that



(a) Linearizable Executions



(b) Equivalent Sequential Executions

Figure 3.1. Operations of a FIFO queue

inserts (removes) an item `a` into (from) a particular FIFO queue. Although the three operations `Enq(a)`, `Enq(b)`, and `Deq(a)` overlap in time, they are linearizable (though not serializable) because their concurrent execution is equivalent to a legal sequential history (Figure 3.1(b)) which does not violate the semantics of a FIFO queue. At each instant of time, there is a set of possible values of the objects that represents a set of possible legal sequential executions. For instance, after invoking `Enq(b)` (but before completing it), the set of possible values of the queue is `{[], [a], [b], [a,b], [b,a]}`. A dequeue operation that returns an `a` is thus legal since it assumes one of the sequential executions. To ensure linearizability, enqueue and dequeue operations must be implemented using special techniques and operators, e.g. atomic swap operations.

As mentioned in [37], linearizability is applicable only for maintaining consistency of concurrent execution and not failure recovery since there is no counterpart of failure atomicity. The traditional techniques for concurrency control and recovery of transactions would involve excessive overhead because in linearizable applications each operation on shared data is viewed as a transaction.

3.4. Partial-Order Semantics

Both serializability and linearizabilty simplify the analysis of concurrent executions by transforming problems in the concurrent domain into simpler problems in the sequential domain. Despite permitting more concurrent execution, linearizability still restricts the interaction pattern of processes because concurrent operation executions must all preserve the linear semantics of the shared objects. In applications where such transformation is not possible (or would cause inefficiency in execution or recovery), a more appropriate approach is to reason about the problems in the concurrent domain with partial-order semantics and to use smarter analytical techniques. In the following discussion, we compare this approach with linearizability although the discussion applies also to serializability.

We first illustrate a non-linearizable behavior by a pattern of interaction. Then we show that this interaction pattern exists in many distributed applications. Consider the executions of three processes A, B, and C (Figure 3.2) in which the sequence of operations is restricted by the information flow shown by dotted arrows. The operation executions are not linearizable because b initiated c before a but b sees c's result after it has seen and used a's result.

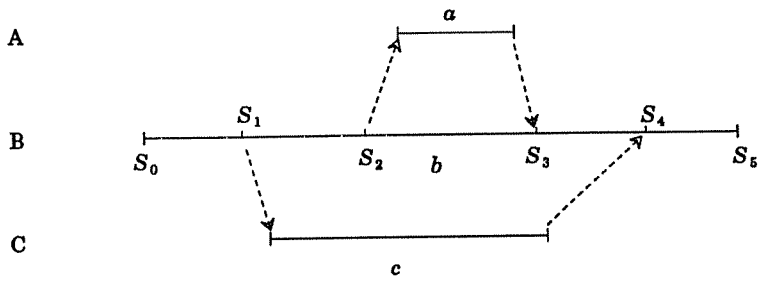


Figure 3.2. Non-Linearizable Executions.

The above example of a non-linearizable behavior can be found in many distributed applications. For example, processes A, B, and C may represent those that control operations in a manufacturing cell (Figure 1.1) of a factory application. Processes A and C control the operations of milling machines 1 and 2 respectively. Process B controls the overall cell operation for the workpiece and ensures that the workpieces are correctly cut, milled and assembled. It includes control of the cutter and assembler operations, although these may be managed by different processes. Operations in one milling machine may affect the concurrent operations in the other milling machine, since they are part of the final product. When an operation of milling machine 1 fails, process A may need to be aborted if the workpiece cannot be repaired. If process B can be restarted at state S_2 then process C can continue normally since it is independent of the effect of the restart and is thus unaffected by A's failure. Otherwise, if B must be recovered to state S_0, then process C must be recovered since it is dependent on B.

Another example of a non-linearizable behavior in manufacturing applications is the assembly of workpieces by multiple cooperating robots. Figure 3.3 shows the precedence graph [38] of the various assembly operations (picking and placing) on the following workpieces: sideplates, levers, shafts, locking pins. For instance, operations E and F must be completed before operation H begins. Depending on the precedence relation, some operations may be done simultaneously, e.g. operation H and L can be executed simultaneously. When an operation fails, e.g. while placing the locking pin at position 12, it may damage another workpiece, e.g. the spacer placed at position 05. This may affect other concurrent operations and require appropriate recovery, e.g. operation H must be aborted and operation E must be re-executed. The overall operation that controls the assembly of the workpieces (which may also control

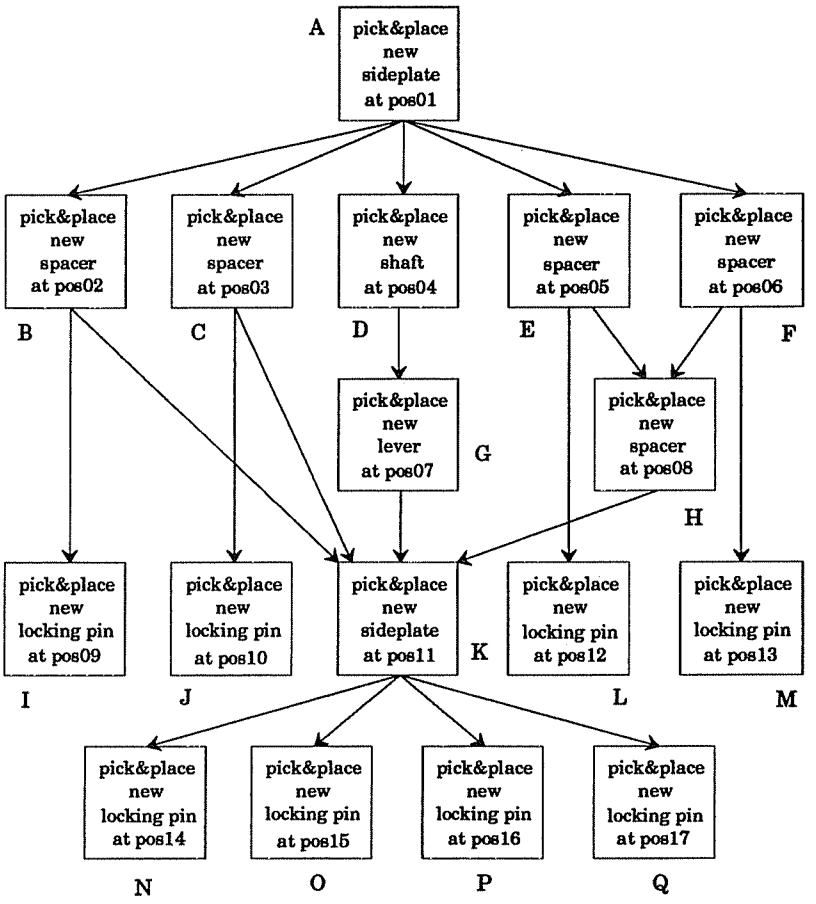


Figure 3.3. Precedence Graph of An Assembly. (after Figure 6.7 of [38])

operations A and K) is not linearizable with either the concurrent operations H or E.

Yet another example is a manufacturing assembly system that involves several manufacturing cells in a factory floor. Figure 3.4 shows the precedence diagram of the assembly system [73] with multiple assembly cells. The nodes represent operations and the directed edges represent the precedence relations between operations. Nodes are partitioned into groups of operations to be executed in each robotized assembly cell: R_1, R_2, R_3 and R_4. As in the previous example, failure of an operation in one cell may affect other concurrent operations executed in another cell.

The pattern of interaction can also be found in another class of applications: multi-transaction database activities. For example, the processing of a purchase order (i.e. an activity) in some company

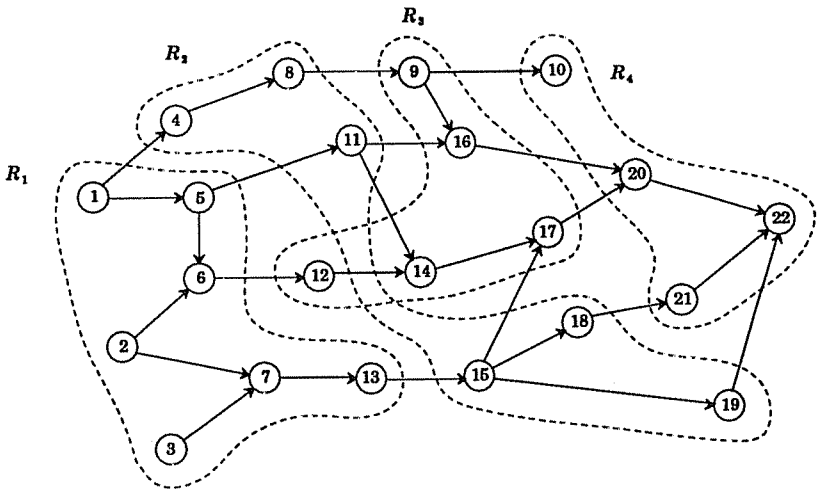


Figure 3.4. Precedence Graph of An Assembly System with Multiple Cells (after Figure 10.4/b of [73])

involves a collection of steps: phone call, enter order, billing, inventory, shipping [27]. The activity starts with a phone call to order certain merchandise. The order is entered into the database and the system then initiates concurrent processing of billing and inventory. Shipping is initiated only when both billing and inventory processing are completed. The overall operation for processing a purchase order (which may include operations "enter order" and "shipping") cannot be linearized with the billing and inventory operations without degrading performance due to loss of concurrency. A failure in inventory processing would affect the processing of billing, e.g. abort billing the customer when there is insufficient merchandise in the inventory.

In the above applications, operations like *b* in Figure 3.2 must be treated as a unit when analyzing for recovery because the effects of the sub-operations at the beginning of *b* can still be destroyed before *b* is completed. Even if some applications can be made linearizable by splitting *b* into smaller operations, linearizability would depend on the granularity of the operations and designers would need to analyze the application carefully to select the appropriate granularity. The major disadvantage of a smaller granularity is the high overhead of beginning and committing transactions. A straightforward but unsatisfactory solution to reduce this overhead is by defining these sub-operations as cooperating transactions in a nested transaction model [7], where intermediate results of cooperating transactions are visible to each other. When a sub-transaction fails, all cooperating transactions within the parent

transaction must be aborted, resulting in wasted computation and physical work.

In our approach, we allow operations to recover to an intermediate execution state without excessive overhead of maintaining checkpoints at beginning of each sub-operation. We use the semantics of the applications to select only important states to be saved for recovery purposes. Recovering to an intermediate state requires analysis of actual dependencies from the partial-order semantics of the distributed application. We can determine actual dependency more accurately than in transaction management which can only determine dependency conservatively based on conflict tables, defining either commutativity [87] or recoverability [6]. Furthermore, by analyzing the application semantics, we can also preserve consistency for non-rollback recovery operations, such as corrective operations, compensation and reset.

The above examples show the importance of dealing with the partial-order semantics of many distributed applications. A specification of a distributed application can define its partial-order semantics by defining the states at which operations can execute concurrently and those at which they must execute serially. In other words, we allow designers to define general synchronization rather than restricting applications to serializable or linearizable behaviors. Since we no longer depend on a notion of sequential execution for preserving consistency, we need to consider a more general set of conditions that will preserve interactive consistency in applications with general synchronization. Here, we will first examine the problem of preserving consistency during recovery management. In Chapter 10, we will generalize it to dynamic reconfiguration.

There are three fundamental conditions that must be satisfied to ensure the consistency of a recovery state. First, recovery operations obey synchronization constraints. Since we allow recovery operations to execute concurrently with operations of unaffected processes, we must ensure that recovery operations and normal operations are synchronized appropriately. Second, the state recovered to must preserve the specified behavior of the application (or some of its behavior) had the failure not occurred. The transactional approach preserves the behavior by maintaining a history log of the actual execution and allowing only re-ordering of commutable operations. On the other hand, we do not need to maintain a history log, but rather preserve the behavior by analyzing the application semantics and ensuring that the behavior after recovery is equivalent to the behavior before failure (or some of its previous behavior). Third, operations that are dependent on conditions (or information) established by a recovered operation

must themselves be recovered. The third condition preserves consistency because when an operation is recovered, no external process should be able to perceive it had ever been executed before. Sequential behavior of individual processes and the synchronization constraints among processes define the potential behavior of the application. We can exploit the behavior specification of the application in checking for actual dependency among operations to maintain consistency of the application during concurrent execution, failure recovery, and dynamic reconfiguration. This more accurate dependency analysis allows higher concurrency and more efficient recovery than dependency analysis from conflict tables in transactions.

In Chapter 8, we will define these conditions more precisely in terms of our finite-state machine model. We will also describe how these conditions can be checked automatically from the behavior specification of the application.

These conditions allow flexibility in enforcing consistency during recovery. In atomic transactions, a "virtual" block structure is required to enforce isolation and recoverability. The isolation property restricts interaction among uncommitted transactions. Unlike transactions, our conditions for consistent recovery do not restrict the synchronization mechanism. While transactions use an avoidance strategy, we use a detection and recovery strategy. Instead of using block structures and enforcing isolation and recoverability, we deal with operation dependencies by detecting all operations dependent on an aborted process and forcing them to recover. This is a generalization of the *restorability* property [64] where all operations that depend (transitively) on a recovered operation must themselves be recovered. This definition does not assume a particular recovery method, i.e. recovery methods include not only rollback recovery but also forward compensating and corrective recovery. It can be satisfied without resorting to a strict block structure as imposed by transactions.

Chapter 4

The Behavior Model

We model distributed processes as finite state machines (FSM's). Each FSM operates independently but may interact with others. To help designers specify the behavior of groups of processes, we propose a hierarchical FSM model that includes mechanisms for abstraction and composition. This hierarchical model enables us to control the runtime behavior of processes in a modular manner. It also enables us to analyze specified behaviors of applications, e.g. for preserving consistency. Furthermore, this model avoids three problems that make the traditional FSM model inadequate for representing a large group of independent processes. First, the traditional model is a flat representation. The behavior of a group of processes is modeled by a product of their FSM's. As the number of processes increases, the product machine grows extremely large and becomes more difficult to analyze and understand. Second, the traditional model does not model concurrent operations very well. Third, a large number of transitions must be used to explicitly represent some simple actions, such as interrupts and failures, because these actions may occur in any state.

The hierarchical FSM model described below overcomes these shortcomings. The abstraction of a composite machine limits the scope of analysis and projects away uninteresting or illegal states. This makes the system scalable. We may model concurrent operations by an appropriate set of constraints or by means of composite machine transitions that represent sets of concurrent transitions of the component machines. The programmer may specify failure states that may be entered from a set of normal states without explicitly specified transitions.

4.1. Basic Definitions

We allow software designers to break a complex system into manageable pieces by specifying independent basic modules and a set of constraints controlling their interactions. We use a Cartesian product of the specified individual modules to verify the feasibility and correctness of their composition. A group of modules may be composed with other groups by higher level composite specifications. Thus, designers can hierarchically structure their distributed applications. We now discuss the basic

definitions of the model and illustrate them in Figure 4.1.

Basic Machines.

We model the behavior of each process (or resource) by a FSM, known as a *basic machine* (or basic FSM). Basic machines are the independent lowest-level modules. Each basic machine consists of a pre-defined set of states and set of transitions from state to state. A *basic machine state* represents a clean, internally consistent state of the process. We do not intend an FSM to model all possible program states; only those states representing important points in the process execution, such as synchronization points, consistent checkpoints, a unit of work, etc. The programmer provides the many-to-one mapping from program states to states in the FSM's. A *basic transition* represents a procedure or an action, often a physical operation, that moves a machine from one basic state to another. Basic transitions take a finite time to complete and are non-atomic, i.e. failure may occur in the midst of them. Our interpretation of finite-state machines is similar to Lamport's model [52] in that states are clean-points and transitions represent operations requiring some time. Software designers specify basic machines purely in term of their own sequential operations without describing synchronization with other basic machines. Each basic machine executes independently but may interact with others.

Product Machines.

When several basic machines interact with each other in a group, we model their group behavior by a *product machine* that is formed by taking the Cartesian product of the basic machines executing in the group. Product machines are automatically generated to assist in the analysis of composite machines (discussed below). A *product state* is a tuple of concurrent states of the component basic machines. There is a product state for every combination of the basic machine states, some of which may be forbidden in practice. A *product transition* moves the product machine from one product state to another[1]. Several product transitions can execute concurrently at a product state S if *every* permutation of those transitions maps to a legal path from S. When two product transitions execute concurrently, the product state that the application enters depends on which transition completes first. Each product transition

[1] This does not necessarily imply that product transitions are sequentialized. In Section 4.6, we describe an equivalent asynchronous representation for specifying concurrent execution. Although we discuss various concepts in terms of the synchronous representation, the discussions are also applicable to the asynchronous representation.

maps to one basic transition. There is a one-to-many correspondence from basic transitions to product transitions. We use the term "basic transition" in the context of the product machine graph to mean the set of *all* product transitions that map to the same basic transition.

Composite Machines.

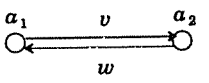
Software designers specify a *composite machine* by specifying its composite states, composite transitions, and a set of constraints on the behavior of a group of basic machines. Composite machines are abstractions maintained by the runtime environment and do not correspond to actual processes. A *composite state* is an arbitrarily specified set of product states. The composite states in a composite machine must be pairwise disjoint. A composite machine is in a composite state A when either its basic machines together are in a product state belonging to A or the basic machines are in some transitions between product states comprising the composite state A. As with basic states, a composite state represents a consistent state of the group of modules. Internally, the composite machine may be in some transition between the product states, but to an external observer, it will always appear to be in the well-defined composite state. A *composite transition* does not necessarily correspond to a particular product transition, but instead, it may represent a set of concurrent product transitions, a sequence of product transitions, or a combination of both.

If a composite machine is in composite state A, it remains in that state as long as product transitions take it to another (or the same) product state in A. The composite machine *leaves* state A when a product transition takes it from a product state in the set A to one that is not. The composite machine *enters* composite state B when a product transition takes it from a product state not in B to one that is in B. A composite transition $A \rightarrow B$ moves from a composite state A to a composite state B and maps to a subgraph such that all states and transitions in the subgraph are in some path that leaves a product state in A and enters a product state in B without leaving or entering any composite state except through the first and last product transitions in the path. We define a *legal state (transition)* as a product state (transition) that is either contained in a composite machine state or a composite transition, and is not explicitly disallowed by a synchronization constraint specified by a software designer. A *restricted product machine* is a product machine that consists of only legal states and transitions. The effect is that software designers can erase (i.e. forbid) any set of product states and product transitions to

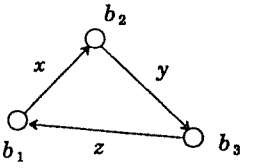
express arbitrary synchronization constraints.

Example

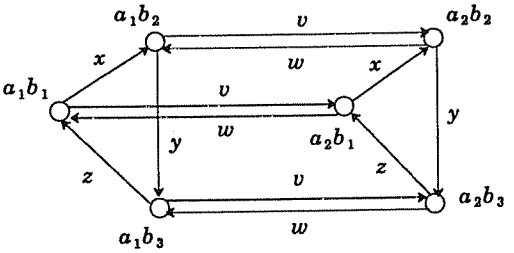
Figure 4.1 shows an example of two basic machines given by a software designer, the product machine generated from them, a composite machine also given by a programmer and the restricted product machine. Basic machine A consists of states a_1 and a_2, and transitions v and w. Basic machine B consists of states b_1, b_2 and b_3, and transitions x, y and z. The product machine of A and B is shown in Figure 4.1(c). The composite machine consists of the states c_1 and c_2, and a transition u from c_1 to c_2. The software designer defines the composite state c_1 (c_2) to contain a product state a_1b_1 (a_2b_3). The



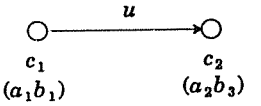
(a) Basic Machine A



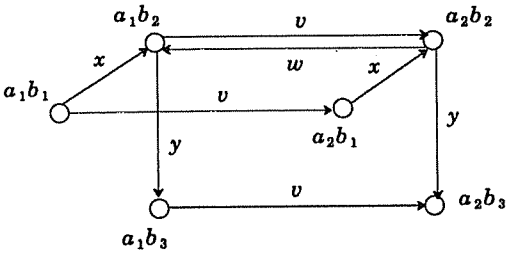
(b) Basic Machine B



(c) Product Machine



(d) Composite Machine



(e) Restricted Product Machine

Figure 4.1. Examples of Machines

transition $c_1 \rightarrow c_2$ can then be automatically mapped to the graph shown in Figure 4.1(e) which, in this example, is also the restricted product machine with all illegal states and transitions eliminated. Since there is no composite transition *into* $c_1 = \{a_1 b_1\}$, all transitions into $a_1 b_1$ in the product machine are eliminated. Similarly, since there is no composite transition *out* of $c_2 = \{a_2 b_3\}$, all transitions out of $a_2 b_3$ are eliminated.

4.2. Modeling Synchronization

We provide several forms of synchronization constraints in addition to the composite transition constraint described above. Composite transitions are useful for defining abstraction.

Preventing physical interference between machines is a common problem in automated manufacturing. Specification writers may use a second form of constraint to explicitly prohibit the product states (transitions) that represent interference between machines. In general, forbidden states or transitions define conflicts or exclusive states and prevent inappropriate behavior. There may be *unsafe* product states that are legal states but inevitably lead to some illegal state. Unsafe states are automatically detected. Such information is useful to programmers for correctly specifying composite machines and to the consistency control manager (described in Section 5.2) in controlling execution. We define a *region* as a set of connected product machine states. We may identify *safe regions* as those that do not contain any unsafe or illegal states.

Coordinating the operations of various autonomous physical equipment is also a common problem in automated manufacturing. A third form of constraints allows programmers to specify mandatory transitions that ensure that before entering a product state, a set of basic machines are required to start from some specified states. Mandatory transitions enforce cooperation between machines.

4.3. Specification of Composite FSM

The programmer defines a composite machine by specifying the composite states, composite transitions, and constraints on states and transitions. In practice, the low-level constraint expressions discussed in this section may be generated from constraints expressed in higher-level structured language constructs that ensure ease and correctness of programming distributed applications.

Composite machine states are specified using first-order logical expressions over basic machine states in the following form:

$$S: e(s_1, s_2, \ldots, s_n)$$

S is the name of the composite state that contains all the product states that satisfy the logical expression $e(s_1, s_2, \ldots, s_n)$ on the basic states $s_1, s_2, \ldots, s_n$.

Composite transitions are defined using composite states in the following form:

$$T: (S_1 \rightarrow S_2)$$

T is the name of the composite transition that moves from composite state S_1 to S_2.

Constraints on the legality of product states and transitions can be specified in first-order logical expressions on basic machine states and transitions over a specified scope. The scope of a constraint is the set of composite transitions originating from a specified composite state. State constraints are specified in the following form:

$$S: f(s_1, s_2, \ldots, s_n)$$

S is a composite state where the state constraint applies to all paths from S to next (unspecified) composite states. If S is *, the state constraint applies globally to all product states. f is a function on the basic machine states $s_1, s_2, \ldots, s_n$. The effect of a state constraint is that if S was the last composite state entered, *no* product state satisfying $f(s_1, s_2, \ldots, s_n)$ may be entered until another composite state is entered.

Transition constraints are specified as either forbidden transitions or mandatory transitions. Forbidden transition constraints have the following form:

$$S: (\alpha \rightarrow \beta)$$

where S is a composite state as described above, $\alpha = f(s_1, s_2, \ldots, s_n)$ and $\beta = g(t_1, t_2, \ldots, t_m)$. f and g are first-order logical expressions on basic machine states. The effect of a forbidden transition constraint is that if S was the last composite state entered, *none* of the product transitions in $\alpha \rightarrow \beta$ may be taken until another composite state is entered.

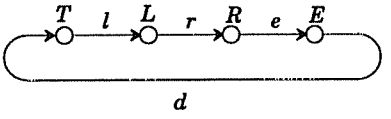
Intuitively, a mandatory transition constraint ensures the sequence of execution of transitions by forcing one product state to be entered before another. Mandatory transition constraints are specified in the following form:

$$S : \mu(\alpha \rightarrow \beta)$$

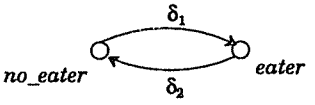
where $\alpha = f(s_1, s_2, ..., s_n)$ and $\beta = g(t_1, t_2, ..., t_m)$. To properly specify the start and end state of each process involved, basic machines mentioned in β should be in the set of machines mentioned in α. The mandatory transition constraints allow programmers to enforce the requirement that certain transitions must be taken before a certain product state is entered. The constraint ensures that for the application to enter β, the following three conditions must be satisfied when S has been entered and before another composite state is entered: (1) α must have been entered since the previous time β was entered, (2) along the path since exiting α, the basic machines mentioned in α and β must pass only through basic states in α and β, and (3) the path from α to β contains no cycle involving machines specified in α.

Examples of Composite FSM Specifications

To illustrate how composite machines are specified, consider the dining philosopher problem where *N* philosophers are seated in a circular table and each philosopher repeatedly cycles through the states *T* (thinking), *L* (has left fork), *R* (has right fork), and *E* (eaten). The operations entering each of the above state are *d* (replace forks), *l* (grab left fork), *r* (grab right fork), and *e* (eating), respectively. Figure 4.2(a) shows the basic machine representing a philosopher. A philosopher shares a fork each with his left and



(a) Basic Machine of Each Philosopher



(b) Composite Machine of N Philosophers

Figure 4.2. Basic and Composite Machines of Dining Philosophers

right neighbors. No two philosophers can pick up the same fork simultaneously[2].

The composite machine consists of N philosopher basic machines where the composite states are expressed in terms of concurrent states of each philosopher, $P_i.\alpha$, where $1 \le i \le N$ and $\alpha \in \{T,L,R,E\}$. The composite states, *no_eater* and *eater*, are defined as follows.

$$\begin{array}{ll} \textit{no\_eater}: & \forall \mathrm{i}\ (1 \le i \le N \Rightarrow P_i.T \vee P_i.L) \\ \textit{eater}: & \exists \mathrm{i}\ (1 \le i \le N \wedge (P_i.R \vee P_i.E)) \end{array}$$

The composite transitions, δ_1 and δ_2, are defined as follows:

$$\begin{array}{ll} \delta_1: & \textit{no\_eater} \dashrightarrow \textit{eater} \\ \delta_2: & \textit{eater} \rightarrow \textit{no\_eater} \end{array}$$

Figure 4.2(b) shows the composite machine of N philosophers with its composite transitions. When a philosopher has entered the state R, he has acquired both forks and can take the transition e that corresponds to the eating operation.

A constraint on states reachable from all composite states is as follows. (* means the constraint applies to all composite states.)

$$*: \quad \neg\, \exists \mathrm{i}\ (P_i.L \wedge P_{left(i)}.R)$$

This invariant disallows neighboring philosophers from picking up the same fork. It is intrinsic to the synchronization problem. (In Section 4.7 and 5.3, we discuss the algorithms used to detect the possibility of deadlock or starvation in this example.)

To illustrate behavior involving complex interaction that cannot be sequentialized, we describe an example of a manufacturing application from [67]. There are two die-stamping machines and a conveyor-based workpiece-replacement mechanism (Figure 4.3). By assigning appropriate values to its actuators, each die-stamping machine stamps a workpiece held between two stops at the conveyor. The workpiece in each machine rests on different section of the same conveyor and must be restrained by stops controlled by their corresponding actuators. The single conveyor is used to replace stamped workpieces with unstamped workpieces in each machine. To introduce workpieces to be stamped (and

[2] For clearer illustration, we simplify the problem by fixing the order of picking up the left fork before the right; we can extend this to a more general solution by simply specifying alternative transitions from T that allow a philosopher to grab a right fork before a left one. Similarly, we can specify different transitions for putting down each fork. However, we use a simpler example for clarity in illustrating important concepts without losing generality.

removing stamped workpieces), the stops and the conveyor must be synchronized in an appropriate way.

Our model allows the designer to specify the control operation of each die-stamping machine individually as shown by the basic machines in Figure 4.4(a) and (b). The basic FSM for controlling DIE 1 consists of four states: an idle state (I_1), a loading state (L_1), a ready state (R_1), and a terminated state (T_1). The transition s_1 represents operations to start the stamping cycle and the transition q_1 represents those that end the cycle. In the transition c_1, the raising and lowering of the stops are coordinated with the conveyor movement to remove stamped workpieces and introduce unstamped workpieces. The transition d_1 represents the operations that extend the die and retract it after some delay. The FSM for DIE 2 is similar. To synchronize the operations of both DIE 1 and DIE 2, the designer need only disallow the product state, L_1L_2[3], where both control process are in the loading state and the conveyor will be subject to conflicting motion. The resulting restricted product machine is shown

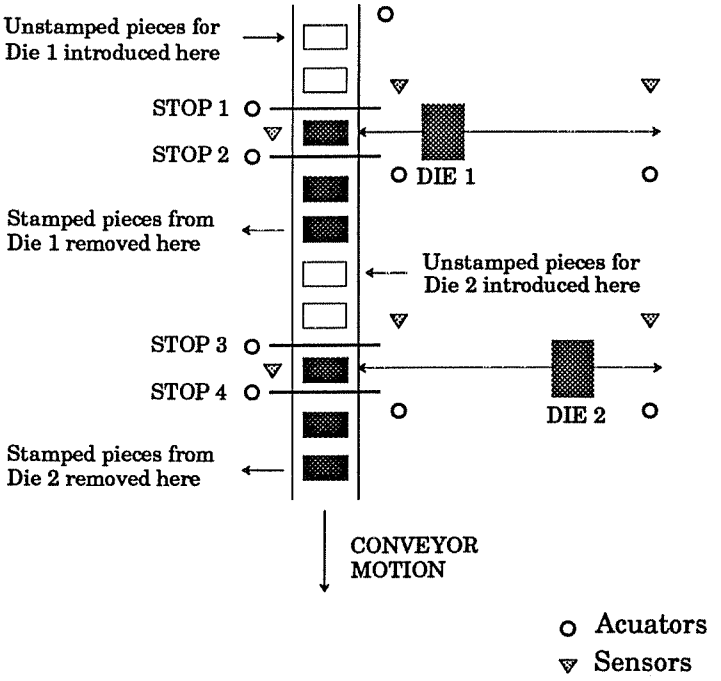
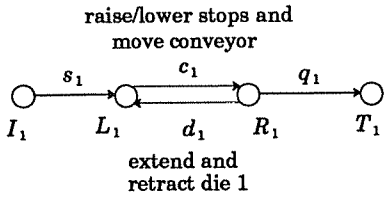
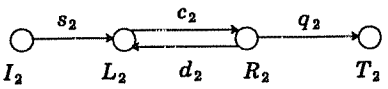


Figure 4.3. A Manufacturing Application consisting of two stamping machines and a conveyor system

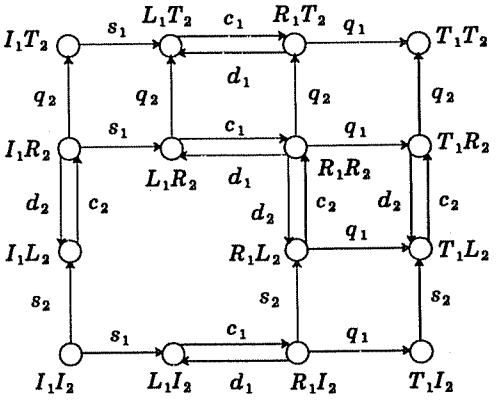
[3] We really want to disallow the concurrent execution of c_1 and c_2, which can only be specified in the asynchronous model (Section 4.6). Since we are using a synchronous model here, disallowing the product state L_1L_2 effectively achieve the same effect.



(a) Basic FSM for Controlling DIE 1



(b) Basic FSM for Controlling DIE 2



(c) Restricted Product FSM

Figure 4.4. FSM Specification of the Manufacturing Application

in Figure 4.4(c). In contrast to the approach used in [67], our approach does not require the designer to specify the complete global flow control of the operations and is less susceptible to human error.

This example also illustrates how we can express any arbitrary interaction that can be expressed in path expressions. Using path expressions, all sequential and non-interfering operations of each die-stamping machine must be explicitly specified, whereas in our model, the designer only needs to disallow interferences.

4.4. Modeling Failure and Recovery

In a factory environment, machines frequently break down, often with external real-world side-effects on the interrupted piece of work. This requires the system to use not only conventional rollback or backward recovery but also more general forward recovery defined for each specific application.

In the preceding, we described a normal model, with normal states and transitions, which characterizes the behavior of applications during normal execution, i.e. in the absence of failure. We now augment the FSM model to characterize behavior involving failure and recovery. In discussing the failure and recovery model, as in the synchronization model, we consider two levels. The first level concerns failure and recovery of each individual basic machine. The second deals with groups of basic machines where their synchronization and consistency constraints can determine the allowable or necessary sequence of recovery transitions.

Failure

To the states of the normal model, we add a set of *failure states*. We consider only fail-stop failures. Software designers may define the failure states for each individual basic machine (in addition to some system default failure states). When considering a group of basic machines, a failure product state is a product state that contains at least one failure basic state. For example, Figure 4.5 shows the failure states, $\Phi_1 R_2$ and $R_1 \Phi_2$, in the manufacturing application. $\Phi_1 R_2$ ($R_1 \Phi_2$) may represent a failure while removing or introducing a workpiece during the execution of transition c_1 (c_2). We distinguish external failure, e.g. a breakdown or an unspecified behavior of an equipment or device controlled by a process, from process failure, e.g. an abort by an end-user, a program crash, prevention from progress by other processes and deadlock. (Methods for analyzing process failure both statically and dynamically are described in Section 4.7 and Section 5.3.) There may be other classes of failure, but we only deal with these. When the consistency control manager (described in Section 5.2) detects an external or process failure, it forces the process into a failure state.

Recovery Transitions

To the normal model, we also add *recovery transitions* from failure states to normal states. Recovery transitions are distinct from normal transitions and represent operations that must be implemented either by the basic machine designer or the consistency control manager. During recovery,

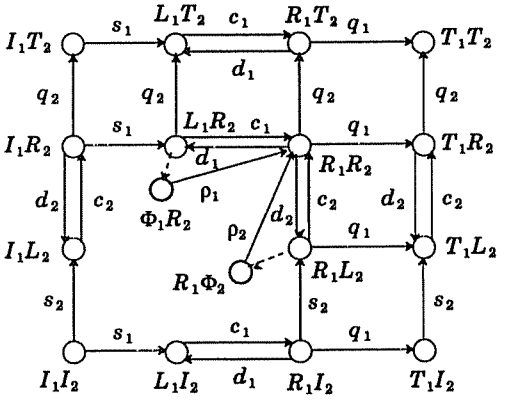


Figure 4.5. Examples of Failure States and Recovery Transitions

the consistency control manager forces a failed or affected process to take a recovery transition. For example, Figure 4.5 shows the recovery transitions, ρ_1 and ρ_2, where ρ_1 (ρ_2) may represent a recovery action that removes the stamped workpiece and introduces a new workpiece between the stops at DIE 1 (DIE 2).

We distinguish between two classes of recovery transitions: reinstate and non-reinstate recovery. We define a *reinstate* recovery transition as any operation that eliminates the effects of a transition execution. Two examples of reinstate recovery are restore and undo. Each reinstate recovery transition is associated with the normal transition (or sequence of normal transitions) that it reinstates. *Non-reinstate* recovery transitions include operations that correct for errors and also operations like *reset* that simply move the process to another (possibly initial) state.

The designer defines failure states and recovery transitions for each individual basic machine independently. Reinstate recovery transitions defined by the designers should affect only the normal transitions of the same basic machine. Recovery operations may conflict with other normal operations. Synchronization constraints can control recovery transitions in the same way as normal transitions. For a group of basic machines, we can generate the appropriate *recovery path*, defined as a sequence of recovery transitions from a failure state to a normal product state called a *recovery state*. A recovery path may contain recovery transitions from more than one basic machine. In Chapter 7, we will discuss the conditions that must be satisfied for the recovery path to preserve consistent behavior of the affected processes.

4.5. Hierarchies of Composite Machines

We allow machines to be hierarchically composed. Each composite machine may be used as a basic machine by a higher level consistency control manager (discussed in Section 5.2). where individual basic machine states and transitions are hidden from composite machines at higher levels. Figure 4.6 shows a two-level hierarchy of composite machines. Only level A composite states and transitions are visible to the level B composite machines; lower-level basic machine states and transitions are transparent to the level B composite machines. Composite machines at level B view composite transitions (states) at level A as basic transitions (states) although they may represent sequences of lower-level basic transitions (sets of lower-level basic states).

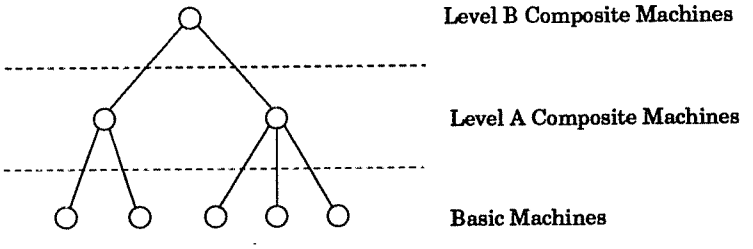


Figure 4.6. A Hierarchy of Composite Machines

The capability to specify hierarchical composite machines assists application programmers in building large and complex distributed systems by grouping together smaller modules at each level. Each composite machine defines a limited scope in which the behavior of a group of machines can be independently analyzed and controlled. We restrict the scope of composite machines at each level to enable analysis and control of hierarchical composite machines to be manageable.

4.6. Synchronous and Asynchronous Representations

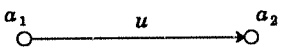
Only basic and composite machines are visible to the designer; product machines are an internal representation solely for analysis and use by the consistency control manager and are hidden from the designer. Although we use a synchronous representation to simplify discussion in this thesis, the system may use an internal asynchronous representation, particularly for restricted product machines. In the synchronous representation, all states are clean points, transitions take some (substantial) finite time, and when a machine leaves a state it is generally not known which outgoing transition it is executing until it reaches the next state. The properties useful to people are different from those useful to programs. The synchronous representation is easier for programmers to reason about synchronization and recovery because its clean-point states allow programmers to describe the state of each machine unambiguously. Internally, product machines may be taken from the *asynchronous representation*. In the asynchronous representation, states can be either clean-point states (corresponding to states in the synchronous machine), or operations (corresponding to transitions in the synchronous machine). All transitions are instantaneous moves between states. We refer to a machine in the asynchronous (synchronous) representation as a asynchronous (synchronous) machine and the states and transitions of asynchronous (synchronous) machines as asynchronous (synchronous) states and transitions.

From a synchronous composite machine specification, we can generate the asynchronous restricted product machine that is used by the consistency control manager for dynamic analyses and control purposes. The asynchronous representation is more suitable than the synchronous for the actual implementation of the consistency control manager environment for three reasons. First, it captures the concurrency of basic transitions with an asynchronous product state that contains asynchronous basic states representing overlapping basic transitions. Second, it models the state of uncertainty where several basic transitions may be taken when a machine leaves a (synchronous) clean-point state. Third, it simplifies handling of asynchronous events because an asynchronous machine is always in a well-defined state.

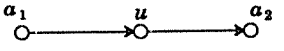
The synchronous representation can be easily converted to the asynchronous representation using the following algorithm. For each synchronous state create a corresponding asynchronous clean-point state with the same label. For the set of synchronous transitions $\{T_1, ..., T_N\}$ leaving synchronous state A, create one asynchronous operation state for each non-empty subset of $\{T_1, ..., T_N\}$. Create an asynchronous transition from A to the operation state $\{T_1, ..., T_N\}$, create an asynchronous transition from each singleton state $\{T_i\}$ to the asynchronous clean-point state corresponding to the target of the synchronous transition T_i, and create asynchronous transitions from each asynchronous operation state with set S to asynchronous operation states with proper subsets of S. Figure 4.7(a) (Figure 4.7(c)) shows a simple synchronous machine and Figure 4.7(b) (Figure 4.7(d)) shows its asynchronous representation.

The asynchronous product machine in Figure 4.7(e) illustrates the use of the asynchronous representation to capture concurrency of basic transitions. An asynchronous product machine is produced by first converting each synchronous basic machines to asynchronous basic machines and then taking a Cartesian product of these asynchronous basic machines. The asynchronous product state uv represents a concurrent execution of synchronous transitions u and v. If the asynchronous product machine were produced by first taking the product and then converting to the asynchronous representation, the concurrent transition state uv would not be generated. Hence the order of conversion and product is important.

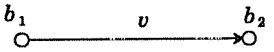
The basic machine in Figure 4.8 illustrates the use of the asynchronous representation to handle uncertainty during transition. When the synchronous machine leaves state A, we may not know for a substantial time which transition it is taking. The asynchronous representation allows us to capture all



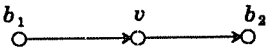
(a) Synchronous Basic Machine A



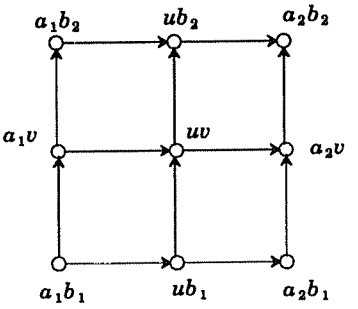
(b) Asynchronous Basic Machine A



(c) Synchronous Basic Machine B



(d) Asynchronous Basic Machine B



(e) Asynchronous Product Machine

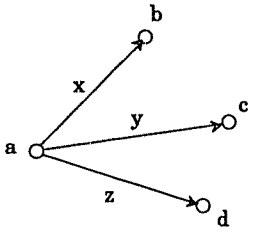
Figure 4.7. Asynchronous Product Machine

the information available about the machine's commitment to some future state. For example, when the state $\{x,y\}$ is entered, there is a commitment to reach either b or c, and synchronization conflicts based on the possibility of reaching d can be disregarded.

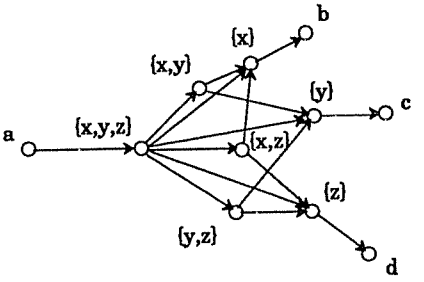
The number of asynchronous transitions may be reduced by creating transitions between operation states only when their sets of synchronous transitions differ by one. Figure 4.8(c) shows the reduced asynchronous machine. At runtime, elimination of several destinations (e.g. a move from $\{x,y,z\}$ to $\{x\}$) must be mapped to serial elimination of one destination at a time.

4.7. Detecting Synchronization Problems

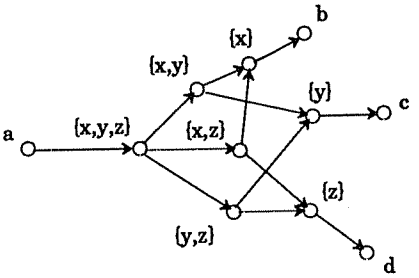
Synchronization constraints may cause certain types of failure, such as deadlock, starvation and livelock. Here we discuss how these synchronization problems are detected using a graph-theoretic approach where properties of composite machines are inferred from properties of their restricted product machine graphs. In what follows, we first introduce the notion of reachability within each composite transition. Then we outline the algorithms for detecting synchronization problems using the complete



(a) A Synchronous Machine



(b) Equivalent Asynchronous Machine



(c) Reduced Asynchronous Machine

Figure 4.8. Synchronous and Asynchronous Representations of a Basic Machine with Multiple Outgoing Transitions

restricted product machine graph. We also discuss some strategies for preventing these problems statically. Techniques for detecting and recovering from these problems dynamically will be discussed in the next chapter.

4.7.1. Feasibility of Composite Transitions

The purpose of analyzing each composite transition individually is (1) to report to the software designer if any composite transition is not feasible, and (2) to assist the consistency control manager (described in the next chapter) in controlling executions of basic machines. An important property to check is whether a product state in a composite state is reachable from a product state in another composite state. We first define two basic concepts that will help us check for reachability. The *basin* of a product state P is the set of product states that *allow* the composite machine to enter P without entering a composite state that does not contain P. (We may leave or remain in a composite state.) The *well* of a product state P is the set of product states that *force* the composite machine to enter P without

entering a composite state that does not contain P. The basin of a composite state A consists of all product states in A and the union of the basins of all the product states in A. The well of a composite state A consists of the union of the wells of all the product states in A and also product states that belong to only basins of states in A but not to a basin of a product state not in A. Basins and wells provide useful restrictions on the product graph for analysis and runtime control. A legal transition must remain within the basin of a composite state at the head of some composite transition. A transition that leaves the permitted basins can be removed from the product graph. A request to make an illegal transition at run time will not be granted.

Using the notion of basin and well, we define the *feasibility* of composite transitions and *reachability* of composite states as follows.

> *Definition 1*: A composite transition $A \rightarrow B$ is *feasible* if and only if *some* product state in A is in the basin of *some* product state in B.

> *Definition 2*: A composite state B is *reachable* from a composite state A if and only if there is a feasible composite transition $A \rightarrow B$.

We adopt this definition of reachability since it is the weakest of the useful variations and it gives software designers flexibility in defining composite transitions and states. A much stronger definition is: a composite state B is reachable if *every* product state of composite state A is in the basin of *every* product state in B for all composite transitions $A \rightarrow B$. We however choose the less restrictive definition.

4.7.2. Deadlock

A deadlock is a situation in which each process is waiting for a resource held by the next process in a cycle. Our definition of deadlock is based on reachability in the restricted product graph. First, consider the case where all basic machines are deadlocked. This is represented by a product state D from which none of the basic machines can make any basic transition. In more general cases, only a subset of the basic machines are deadlocked. We use the following definition instead.

> *Definition 3*: A product state s is a *deadlock* state for a set P of basic machines if *no* path from s involves any transition representing an action of any basic machine in P.

There may be other basic machines that are not deadlocked and are free to make normal transitions within the set of deadlock product states (or deadlock region).

Static Deadlock Analysis

Before discussing the algorithm to find deadlock regions, we introduce the notion of inhibition from *leaving* a state. A basic machine M is permanently inhibited from leaving a (non-terminal) basic state s if the application executes in a strongly connected region R of the restricted product graph with the following properties: (1) M can only be at state s in R, (2) M does not have a transition leaving a product state in R, and (3) there is no path involving other machines to another region that contains transitions of M.

The algorithm to find deadlock regions identifies strongly connected components (SCC) in the restricted product graph in which one or more basic machines are inhibited from leaving their state in that region. A deadlock SCC need not be a sink SCC since the paths from one deadlock SCC to another may not contain any transition of the deadlock basic machines. The outline of the algorithm is as follows.

(1) Identify all SCC, $C_1, ..., C_N$, in the restricted product graph G using Tarjan's algorithm [81]. (For this algorithm, each node not in any strongly connected component forms a singleton SCC.) Generate a strong reduction graph G' as follow: nodes of G' are SCC's of G and an arc $C_i \rightarrow C_j$ exists in G' if and only if there is a transition $x \rightarrow y$ such that $x \in C_i$ and $y \in C_j$.

(2) Label each C_i with the set S_i of basic machines that have a single (non-terminal) state in the region. (S_i may be empty.)

(3) From each node C_j in G' after all its out-going arcs are traversed, for each in-coming arc $C_i \rightarrow C_j$ update S_i to $S_i \cap S_j$. (Sinks in G' do not have out-going arcs and are computed first.)

When the algorithm terminates, deadlocked regions are those SCC's with non-empty labels. The labels indicate those basic machines that are deadlocked within each region.

Our model provides a simple and declarative way to prevent deadlock. If static analysis identifies deadlock regions, the software designer can add additional constraints to the composite specification to prohibit them. (We remark that separation of synchronization (policy) from basic machine specification allows deadlock to be prevented in this way.) Upon recompilation, the problem areas will be removed from the restricted product graph. Traditionally, software designers select a resource allocation policy or make some adjustment to an algorithm to statically prevent deadlock.

4.7.3. Starvation

Traditionally, starvation is defined as a situation where a process continues to be denied access to a resource although that resource is being allocated to other processes. To define starvation, we first introduce the notion of *significant* basic machine states, which are states where useful some application-specific works are completed, e.g. a philosopher eating after picking both forks. Significant basic states are explicitly specified by software designers. Each basic machine should enter its significant states infinitely often when unconstrained, but may be suspended by the manager when executing as a member of a composite machine. A composite state is significant if it contains at least one significant basic machine state or a software designer explicitly designates it to be significant. We define starvation in terms of properties of the restricted product machine graph:

> *Definition 4*: A product state s is a possible *starvation* state for a set P of basic machines if s is in a strongly connected region R (with some infinite path from s) containing no significant state of any basic machine in P *and* there is at least a path from R to another region with significant states of all basic machines in P such that all transitions leaving R do not belong to any basic machine in P.

Actual starvation depends on the behavior of the machines able to cause a transition out of the possible starvation region. A starved basic machine must be able to reach a state where useful work is done in the absence of operations of other basic machines that prevent it from leaving the starvation region. We assume the manager implements a fair queuing policy for request from basic machines to make transitions (discussed in Section 5.2). This fair queuing policy eliminates one cause of starvation.

Static Starvation Analysis

First, we introduce the notion of local inhibition from *entering* a basic machine state. A basic machine M can be locally inhibited from entering its basic state s in a strongly connected region R if (1) s is not in any product state in R, (2) there is no transition (or path), belonging to M, leaving a product state in R to a product state containing s, and (3) there is a path involving transitions of basic machines other than M to another region that contains product state containing s. If M is locally inhibited from entering a set of its significant states in a region R, then M might be starved in R.

The algorithm to find potential starvation regions identifies strongly connected components in the restricted product graph where one or more basic machines are locally inhibited from entering their significant states. We need to analyze one basic machine at a time because there may be a starvation region of a basic machine M within a strongly connected component of the restricted product graph containing an M's significant state.

First, label all states with an empty set. Then, for each basic machine M, unmark all states and transitions and do the following:

(1) Mark product states containing M's significant states and all transitions to and from these states. Propagate backward from these states along M's transitions only, marking the states and transitions along the path. Mark all transitions to and from these newly marked states.

(2) Using only unmarked states and transitions, identify all SCC's, C_1, ..., C_N, using Tarjan's algorithm.

(3) Starting from each product state with M's significant state, propagate backward (ignoring marks) to all reachable restricted product states. For each C_i that is reachable, add M to the label of each state in C_i.

The potential starvation regions are those SCC's containing states with non-empty labels. (Potential starvation regions of different basic machines may overlap.) The labels indicate the basic machine that may be starved.

Static starvation prevention requires that software designers modify the specification so that starved processes may break the cycle followed by the processes that are getting the resources, i.e. introduce or allow transitions of the starved machines out of the potential starvation region. However, it is not always easy to modify specifications to break all potential starvation cycles. In Section 5.3, we will discuss how starvation is dynamically detected and recovered.

4.7.4. Livelock

Traditionally, livelock is defined as a situation in which each process repeatedly decides whether to access some resources (or perform some activity) but fails to reach a decision while continuing to consume computational resources in making those decisions. We define livelock in terms of properties of the restricted product graph:

Definition 5: A product state s is a possible *livelock* state for a set P of basic machines if it is a possible starvation state except that each basic machine B in P has a transition leaving the strongly connected region R and leading to another region containing a significant state of B.

Livelock is a non-deterministic property, as is starvation, contingent on the scheduling of the operation of each livelock machine. While the composite machine cycles indefinitely within a non-significant region, it merely performs subsidiary (synchronization) operations of the livelocked machines and none of their significant work. Livelock differs from starvation in that all basic machines involved in a livelock can break the livelock by selecting different operations or suspending some of their operations, whereas starvation can only be stopped by some basic machine (or the manager) that is not starved.

Static Livelock Analysis

The algorithm to find potential livelock regions is similar to that for finding starvation regions. It identifies cyclic regions in the restricted product graph where a *set* of basic machines is locally inhibited from entering their significant state and where they can make infinite number of transitions in cycles. Again, we analyze the general case where other basic machines not involved in the livelock are free to operate normally in the livelock region. The outline of the algorithm which uses the restricted product graph is as follows. (We do not consider self-loop transitions around non-significant states.)

(1) First, label all states with an empty set. Then, for each basic machine M:

  (a) Mark all transitions belonging to M except those adjacent to a product state with a significant state of M.

  (b) Using only marked edges, find all SCC's, C_1, ..., C_N, using Tarjan's algorithm.

  (c) Starting from each significant state of M, propagate backward to all reachable restricted product states. For each C_i that is reachable, add M to the label of each state in C_i.

(2) Using only states with non-empty labels, find all SCC's, D_1, ..., D_N, using Tarjan's algorithm. Relabel each D_i with the union of all its states' labels. For each D_i containing only one machine in its labels, set the label to empty.

(3) For each basic machine M do:

  (a) Starting from each of its significant states, propagate backward along M's transition until

we reach a deadend or a D_i containing M in its labels. Mark the first state(s) in each D_i reached.

(b) For each marked state, if there is no transition of any machine in its label leaving it and entering another state with the same label, then remove M from all state labels in D_i, otherwise mark M in the label.

(4) For each D_i with unmarked M in its label, remove M from it.

(5) For each D_i and for each machine M in its label, if there is no state in D_i with a disallowed M's transition leaving it, then remove M from all state labels in D_i.

The possible livelock regions are those SCC's with non-empty labels. The labels indicate those basic machine that may potentially engage in a livelock.

As in starvation, it is often not easy to prevent livelock statically by modifying the composite machine specification. In Section 5.3, we discuss the dynamic prevention of livelock.

Chapter 5

Uniform Framework: Architecture

We use the model just described in Chapter 4 to statically specify and analyze the behavior of distributed applications separately from the behaviors of their component processes. We can also use the model to dynamically enforce the specified behavior and provide automated failure recovery as well as dynamic reconfiguration of running applications. The runtime mechanism that performs these task is independent of the policies software designers may specify through the model. The well-known separation of policy and mechanism [40,54] has not been previously applied to synchronization, reliability and reconfiguration of distributed systems.

Figure 5.1 shows an overview of how the model is used, particularly, how basic and composite machine specifications are compiled statically and how a consistency control manager provides the common control mechanism. While Section 4.7 describes the algorithms for static analysis of synchronization problems, Section 5.3 describes dynamic detection and recovery of synchronization problems. Chapters 6 to 10 describe support for failure recovery and dynamic reconfiguration through

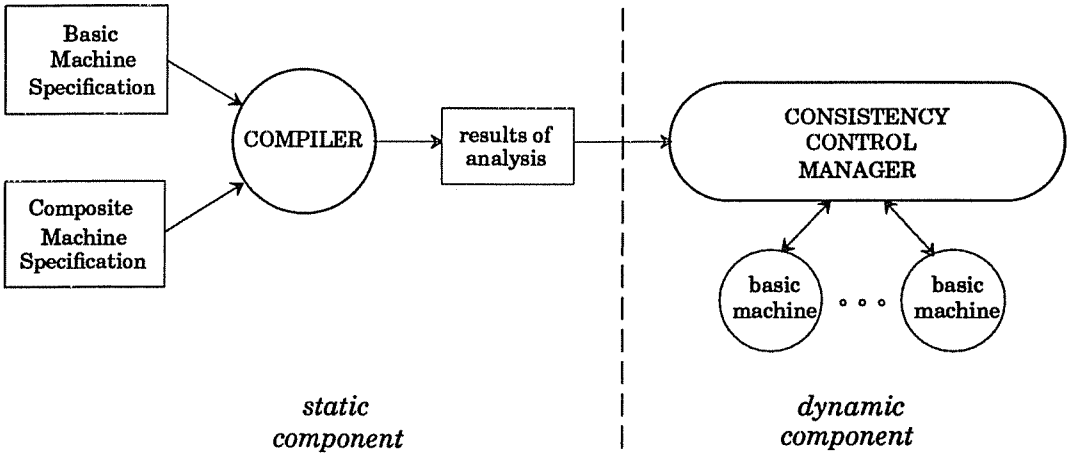


Figure 5.1. Overview of Compiler and Consistency Control Manager

automatic analysis of dependencies and selection of recovery states and reconfiguration paths.

As a first step, the programmer generates basic machine descriptions and a composite machine specification. These specifications are given to a compiler for static analysis, resulting in a report of regions in the graph that result in deadlock or possible starvation or livelock. The programmer may add constraints to the specifications to avoid these synchronization problems. When a satisfactory behavior is specified, the compiler produces a restricted product machine graph (with information on reachability, potential starvation and livelock) and a set of recovery paths from any (unrestricted) product state to the restricted product machine. After compilation, processes are initiated for the basic machines, and the manager is started to monitor them. The manager uses the compiler output to control synchronization and recovery of the basic machines. In the following subsections, we discuss the compiler and manager in greater detail, illustrating some of the features with examples from the dining philosopher problem.

5.1. Composite FSM Compiler

The composite FSM compiler accepts programmer specifications of basic and composite machines (Figure 5.2). It first makes several diagnostic checks:

- syntactic correctness
- overall graph connectivity of each machine
- feasibility of each composite transition

Invariant constraints specified on product states and transitions are checked to verify that they are consistent with each other. Overlaps between different constraints can be detected by checking every state in the finite-state product machine. The composite machine specification should also be consistent with the model as defined in Section 4.1. For example, the specification violates the model definition if any two composite states overlap. Complete graph connectivity is important since some product machine states will be unreachable if there are disjoint connected components in the restricted product machine graph. In addition, all specified composite transitions must be feasible, i.e. there must be some product states in the destination composite state that are reachable from at least one product state in the start composite state. Problems are reported to the programmer for correction.

If the input is acceptable, the compiler then applies the algorithms described in Section 4.7 to identify:

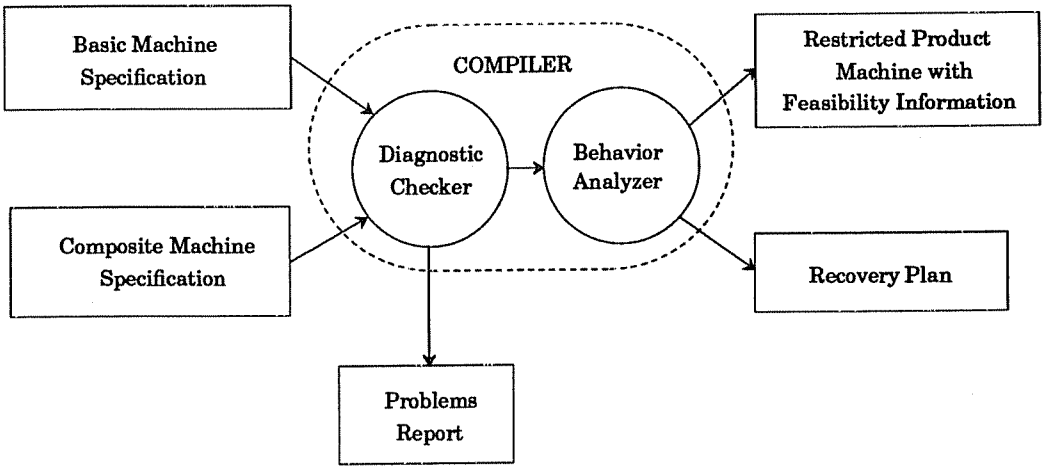


Figure 5.2. Composite FSM Compiler

- sets of legal product transitions for each composite state and transition
- deadlocked product states
- potential livelocked and starved product states

The compiler determines regions in the restricted product machine that lead to deadlock or possible livelock or starvation. Each product state and transition is labeled to indicate its legality, the composite states that are reachable, the composite states it belongs to (if any), and the composite states that inhibit it. Product states are also labeled if they are in a region that may deadlock, potentially livelock, or cause starvation.

Figure 5.3 shows an example of the restricted product machine of the dining philosopher problem with two philosophers. Each basic machine should be able to visit its eating (E) state infinitely often. If the composite machine enters the state (L,L), then there is a deadlock since there is no other product state to which it can move. To prevent deadlock without using some specific algorithm like waiters or a dining room ticket, the programmer may simply disallow deadlock states by adding the following state constraint to the specification in Section 4.3.

$$*: \quad \neg\forall i(P_i.L)$$

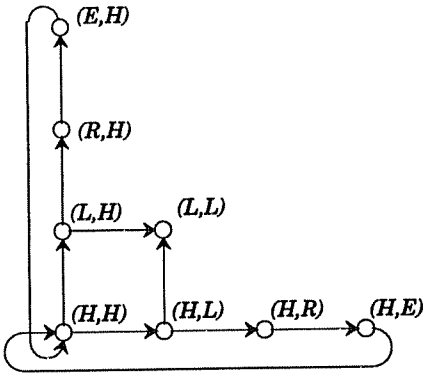


Figure 5.3. Restricted Product Machine of Two-Dining Philosopher Problem

Once the compiler identifies all the illegal, deadlocked, and potentially livelocked or starved product states, it may provide means to recover from them. In addition to these problem states, other unacceptable states from which the composite machine needs to be recovered include failure of a basic machine, unconditional reset or abort by the end-user, erroneous or unspecified basic transition, and failure of a basic machine to cooperate with the manager. The compiler computes a path through the product machine (including failure states) from each illegal and problem state to acceptable states in the restricted product graph. This set of paths is the recovery plan. These paths can include recovery actions (such as basic machine reset) that are not available during normal operation. The recovery plan is used by the manager whenever it must force the composite machine into an acceptable state.

Figure 5.4 shows an example of a recovery plan for the dining philosopher problem with two philosophers. The programmer specifies the failure states for individual basic machines: Φ_1 for philosopher 1 and Φ_2 for philosopher 2. The compiler then generates the failure product states (Φ_1,H), (Φ_1,L), (H,Φ_2) and (L,Φ_2). (Φ_1,H) is entered when the basic machine P_1 fails after the composite machine has left the states (L,H), (R,H), or (E,H). The recovery transition from (Φ_1,H) to the product state (H,H) represents the recovery action of P_1. There are similar recovery plan for failure states (Φ_1,L), (H,Φ_2) and (L,Φ_2).

5.2. Consistency Control Manager

After compilation and analysis, a consistency control manager is created for each composite machine. The manager controls scheduling, enforces synchronization control, implements recovery,

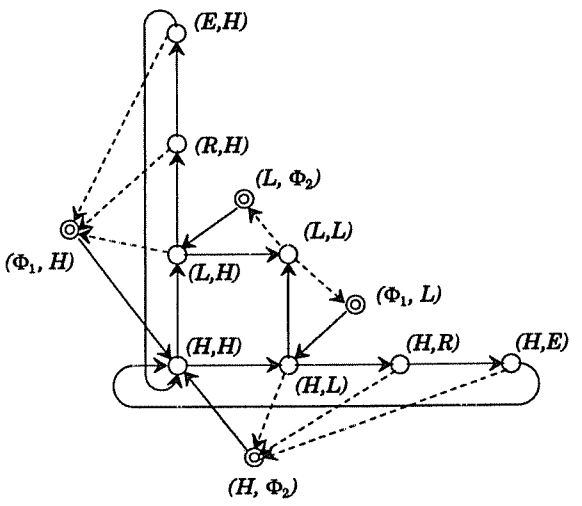


Figure 5.4. Recovery Plan of Two-Dining Philosopher Problem

manages reconfiguration, and takes the role of a basic machine when the composite is used in a higher-level group. Managers are not customized for each composite machine. Instead, the separation of policy and mechanism lets them be driven by the output of the compiler's behavior analyzer.

Figure 5.5 shows how a manager interacts with the basic machines and a higher level manager. There is a communication channel between each basic machine and the manager and this interface has only a few features. The manager communicates with the basic machines through messages. There are two modes of interactions – autonomous and exception. In autonomous mode, a basic machine requests permission to make a transition and the manager responds by granting or denying the transition depending on the synchronization constraints. The manager can also query the current state or transition of a basic machine. It can force basic machines into exception mode when failure occurs. In exception mode, the manager forces the basic machine to follow specific transitions. In both modes, a basic machine reports when it has finished a transition and reached a state. When the basic machines are recovered from the failure, the manager forces them back into autonomous mode[1].

Basic machines and managers may interact with an external semantic information facility that maintains information that is not easily represented by a finite-state machine model, such as counters,

[1] It is possible to improve the fault-tolerance of the centralized consistency control manager through various replication techniques [14, 75]. However, we will not focus on this aspect of the work here.

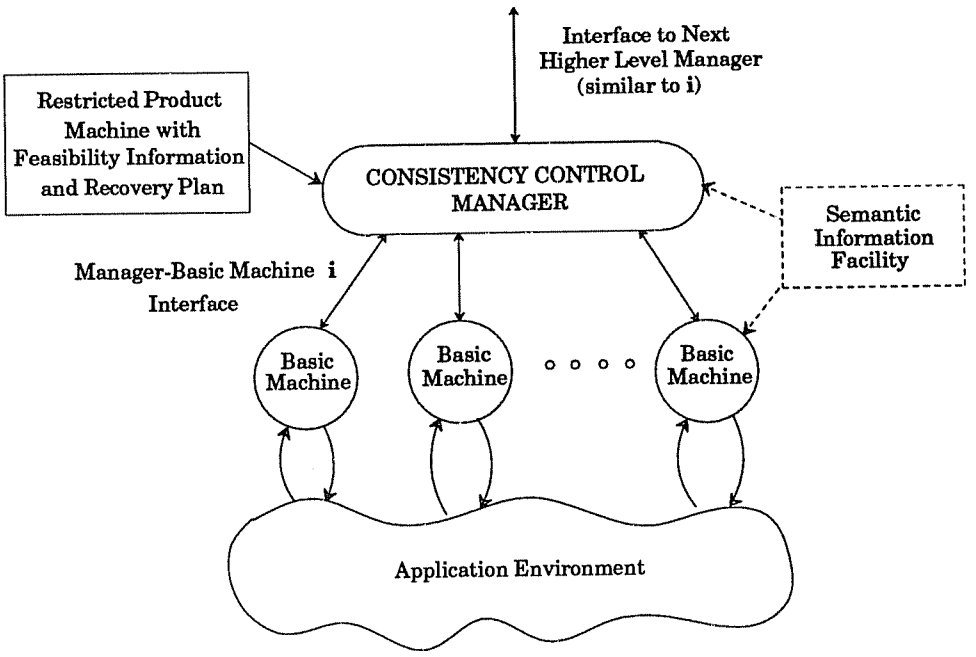


Figure 5.5. Consistency Control Manager

timers, and arbitrary predicates. We will not explore this facility further in this thesis.

Synchronization

The manager controls synchronization by repeatedly listening and serving requests from all basic machines. When each basic machine is instantiated, it first calls `announce` to announce its initial state. To request permission to move to a destination state, it will then call `request_permit` (implemented as a remote procedure call to the manager). Subsequently, each basic machine always calls `request_permit` before making a transition.

The manager examines the restricted product machine graph to determine if the requested transition violates synchronization constraints or will lead to some problems. A transition is acceptable if it will not enter an illegal, unsafe or problem state. To improve concurrency, two or more product transitions are permitted to execute concurrently at a product state S if there is a legal path from S for *every* permutation of those transitions. If acceptable, the manager authorizes the transition by returning `GRANTED` to the pending `request_permit` call. Otherwise, it returns `HOLD`. The basic machine may then decide to make an alternative transition. However, if the basic machines decides to wait for the

transition to become acceptable, it calls `block_request`. The manager then puts the request in a pending list and returns `GRANTED` when the transition becomes acceptable, i.e. after other basic machines have made some transitions. (The manager always authorizes the oldest request that is currently acceptable, eliminating one cause of starvation.) When a basic machine has reached its destination state, it may call `announce` to announce its new state. Instead of calling `announce`, it may simply make another call to `request_permit` and pass its current state and a requested destination state as the parameters. The manager is thus notified of the completion of the basic machine's previous transition. Figure 5.6 shows the pseudocode of the manager for serving basic requests from basic machines.

Consider the two-philosopher problem, where the composite machine is in state (H,R) and basic machine P_1 requests permission to make a transition to state L. The manager consults the restricted product machine graph (Figure 5.3) that indicates product state (L,R) is illegal. The manager returns

```
/* Consistency control manager interface for basic request */

entry request_permit(current_state, destination_state) {
    if caller's request is acceptable
        return GRANTED;
    else
        return HOLD;
    announce(current_state);
}

entry block_request(current_state, destination_state) {
    if caller's request is acceptable
        return GRANTED;
    else
        put request on pending list;
    announce(current_state);
}

entry announce(current_state) {
    scan pending list {
        if caller's current state enables a pending request
            return GRANTED to pending requester;
    }
}
```

Figure 5.6. Pseudocodes for Serving Basic Requests for Permission

HOLD to inform P_1 to wait. P_1 may then request to block until it receives permission. Meanwhile, P_2 makes transitions to E and H, taking the composite machine to state (H,H). At that point, the manager returns GRANTED to allow P_1 to make the transition to L. If deadlock were not eliminated statically, the manager would be responsible for *dynamic avoidance* of deadlock (discussed in Section 5.3). Suppose the composite machine is in state (H,L), and P_1 requests the transition to state L. From the restricted product machine graph, the manager knows that leads to the deadlock state (L,L). The manager avoids deadlock by informing P_1 to hold until P_2 is done eating.

There are two ways in which the number of message can be reduced. First, a basic machine may present to the manager a list of alternative states in order of decreasing preference by calling the function `select_permit`. The manager returns the index of the first state in the list that the basic machine is allowed to execute. The function `select_permit` avoids the additional messages required for repeatedly requesting permission to execute different transitions after the previous request is disallowed. Second, a basic machine may call `select_path` to present a list of paths in decreasing order of preference. Each path represents a request to reserve a sequence of transitions in the paths. If one of these paths exists in the restricted product machine and is compatible with other concurrently executing transitions, then permission will be granted to the basic machine. The function `select_path` avoids messages required to request permission to execute transitions in sequence. Figure 5.7 shows the pseudocode of the manager for serving batched requests from basic machines.

A basic machine may receive other unexpected responses from the manager while waiting for a response from the manager after making a request through `request_permit`, `block_request`, `select_permit`, or `select_path`. Furthermore, a basic machine may also receive unexpected messages from the manager when it is idle or when it is executing a long operation without interacting with the manager for an extended time period. (For each basic machine, the system may create a separate thread that waits for unexpected messages from the manager.) There are three types of unexpected messages from the manager. First, a basic machine may receive an `exception` message (through a `force_mode` call) when the manager wants to force the basic machine into an exception mode to recover from a failure. The basic machine then goes into an exception mode and waits for further instruction from the manager. The manager then sends an instruction to execute a mandatory transition. The basic machines executes the mandatory transition and discards its previous request for permission, if any.

```
/* Consistency control manager interface for batched request */

entry select_permit(current_state, state_list, number_of_states) {
    for i from 1 to number_of_states {
        if transition to state_list[i] is acceptable
            return i;
        else
            return HOLD;
    }
    announce(current_state);
}

entry select_path(current_state, path_list, number_of_paths) {
    for i from 1 to number_of_paths {
        if all transitions in path_list[i] are acceptable
            return i;
        else
            return HOLD;
    }
    announce(current_state);
}
```

Figure 5.7. Pseudocodes for Serving Requests for Batched Permission

Second, the manager may sometimes query the status of the basic machine. The basic machine then answers the query and continue to wait for the previously expected response if there is a pending request. Third, the manager may send a `checkpoint` message to force the basic machine to checkpoint itself. The basic machine then completes the checkpoint, returns the result to the manager and waits for response to its previous request, if any.

If static analysis had not been done, the manager must analyze the restricted product machine graph dynamically to determine the safety of transitions. The manager can construct an operating sub-graph consisting of all reachable states within a fixed distance of the current product state. The greater the distance, the better the analytical results, at the expense of higher computational cost. The manager can analyze the operating sub-graph for potential synchronization problems using the same algorithms as the compiler would apply to the complete product graph. There is a tradeoff between static and dynamic analysis. For large restricted product graphs, static analysis is time consuming and only a

small part of the results may be used in the actual execution. Selective analysis at run time may be more efficient in such cases. However, dynamic analysis slows down the manager. Furthermore, some problem regions may not be detected if the operating sub-graph is not large enough.

Recovery

If the composite machine violates its concurrency and sequencing constraints for any reason, the manager initiates exception mode. Examples of such violations include failure of a basic machine action, erroneous or unspecified basic transition, failure of a basic machine to cooperate with the manager, or dynamic detection of a synchronization problem (e.g., deadlock). The manager may also be forced into exception mode by a manager of a higher-level composite.

Since the consistency control manager monitors the state of each basic machine during normal execution, it knows the normal product state entered just before the failure. From the recovery plans in the restricted product machine, it determines the appropriate recovery path. It then forces each affected process to recover by taking a mandatory transition. Each basic machine then performs the mandated recovery operations and notifies the manager on successful completion. When all affected basic machines have successfully completed, the manager sets all basic machines back to autonomous mode. Details of the recovery mechanisms are described in Section 8.1.

Dynamic Reconfiguration

Software designers or system administrators may initiate dynamic reconfiguration through the consistency control manager. The old restricted product machine is replaced by a desired restricted product machine. From the current product state, the manager determines the appropriate reconfiguration path. It then forces the affected basic machines into exception mode and instructs them to take the mandatory transitions (either recovery or reconfiguration transitions). When the manager is notified of successful completion of these transitions, it then sets these basic machines back to autonomous mode. The restricted product machine representing the new configuration is then used for controlling all future executions. Details of the mechanisms are described in Section 10.1.

5.3. Dynamic Synchronization Problems Detection and Recovery

In the Section 4.7, we discussed how synchronization problems are detected statically. If these problems are not prevented statically, they must be detected dynamically and recovered appropriately.

Deadlock Avoidance

When the application is executed, the composite machine may still contain deadlock regions. The consistency control manager is then responsible for dynamic deadlock avoidance by suspending basic machines if their autonomously requested transitions will enter a deadlock region. Other basic machines may later take the composite machine to another region so that it will be safe again for the suspended machines to make their requested transitions. This is similar to the traditional approach where the operating system delays resource allocation to avoid deadlock dynamically. We however provide an automatic and integrated mechanism.

Deadlock Detection and Recovery

Deadlock prevention or avoidance might not be implemented if the designer decided complete static analysis was too costly. In that case, the manager must dynamically analyze the current operating sub-graph (which may not be large enough to detect some deadlock regions). The composite machine may enter a deadlock region before the manager detects it. To recover from deadlock, the manager forces all affected basic machines into a failure mode. The manager then determines the failure state of the composite machine. Recovery of deadlock is done by forcing one or more victims to make mandatory transitions and recover to a state where other deadlocked machines are able to continue with their pending transitions. The recovery state selected should involve the minimum undo operations. After the recovery state is reached, the manager resumes the normal mode of operation.

Starvation Detection and Recovery

The consistency control manager performs dynamic starvation detection and recovery by monitoring the number of cycles the composite machine spends within a starvation region. A composite machine may enter a starvation region without any immediate problem. Only when the composite machine cycles in the region indefinitely *while an inhibited machine has pending request* will starvation occur. When the number of transitions spent within a region exceeds a threshold t, the manager

suspends the active basic machines. (A default value of t is pre-defined although a software designer may define an more appropriate value of t.) Their suspension may be delayed until they reach a state (automatically identified by static analysis) with transitions from the starvation region to another region in which a starved machine may enter its significant state.

Consider the dining philosopher problem with five philosophers and an additional condition that a philosopher can pick up its left fork only when neither of its neighbors has picked up its right fork and is ready to eat. Figure 5.8 shows a subgraph of the restricted product graph showing some transitions of only three of the philosophers with their states indicated in the product state as (P_1, P_2, P_3, ...). The cyclic region represented by bold edges represents the operating region where philosopher 2 can be starved by a conspiracy by his neighboring philosophers 1 and 3. In order to prevent starvation, the manager will monitor the number of times that philosophers 1 and 3 have repeated the significant eating state. If that exceeds the threshold value, the manager will suspend one of them, e.g. philosopher 1 when

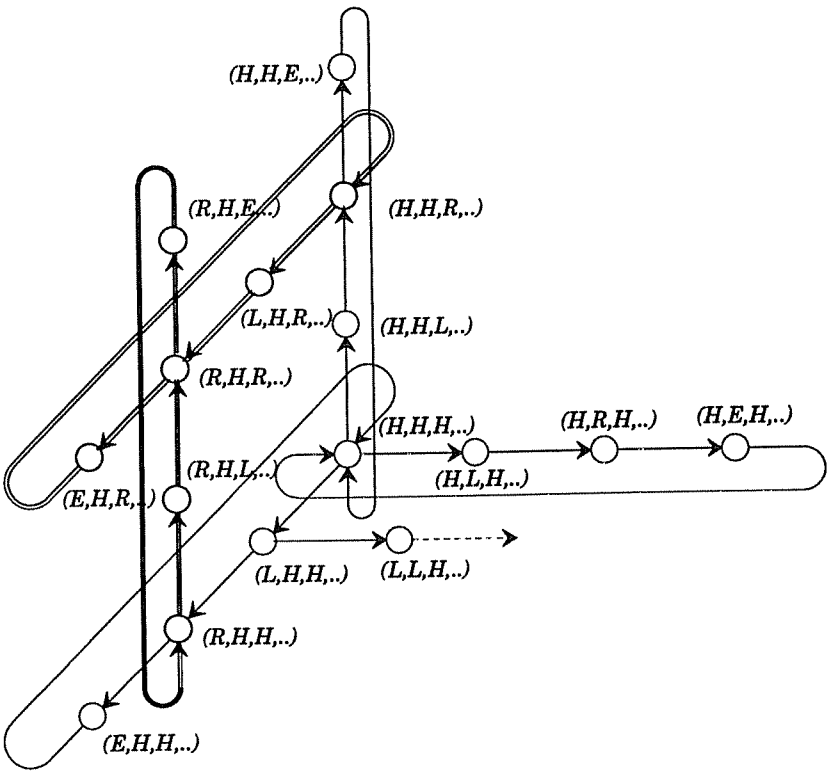


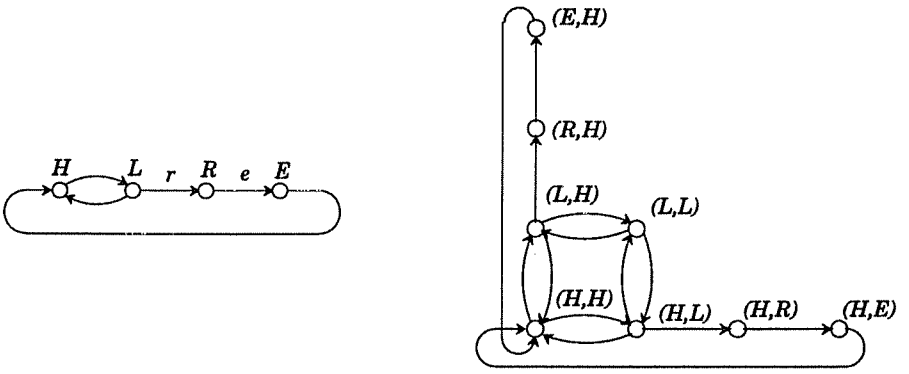
Figure 5.8. Example of a Starvation Region

it reaches the state *H*. Any move by philosopher 2 will then take the composite machine out of the starvation region.

Livelock Detection and Recovery

The consistency control manager performs dynamic livelock detection and recovery by monitoring the number of non-significant cycles the basic machines make in the livelock region and change the schedule for granting permission, e.g. suspending one machine at a product state where another livelocked basic machine may make a transition out of the region.

To illustrate livelock, consider the dining philosopher problem where a philosopher will put down his left fork if he cannot access his right fork immediately after picking up the left. Figure 5.9(a) shows the modified basic machine of a philosopher and Figure 5.9(b) shows the restricted product machine with two philosophers. The analysis algorithm identifies states (*H*,*H*), (*L*,*H*), (*L*,*L*), and (*H*,*L*) as a potential livelock region. During execution, if the manager detects that composite machine cycles in that region a specified number of times, it can suspend philosopher 1 when it reaches state *H* (identified automatically by static analysis). This will allow philosopher 2 to pick up the left and then the right forks. Philosopher 1's request will be granted the next time it is legal.



(a) Modified Philosopher Basic Machine (b) Modified Restricted Product Graph

Figure 5.9. Modified Two-Dining Philosopher Problem

Chapter 6

The Execution Model

A FSM specification in the behavior model defines an allowable behavior, i.e. the set of transition sequences that can be executed repeatedly. While the behavior model is concerned with the sequences of operations that can or cannot be executed, it does not contain information on their actual execution by the application. The analysis of failure and recovery requires information on operation executions, particularly information on interrupted and past executions. The behavior model deals with *operation behavior* while the execution model deals with the effects of *operation executions*. An operation behavior can be thought of as a *type* that specifies the set of operation sequences that can be executed repeatedly, whereas an operation execution refers to an *instance* of such execution.

To analyze the effects of failure and the correctness of recovery, we need information about actual executions that may not be found in the FSM abstraction. However, global states are usually large, and most of the state is not relevant to the analysis. We use two principles to identify the relevant features of an execution. First, we assume the software designer has explicitly specified all important restrictions on behavior. When two transitions have no mutual constraints, they are treated as independent operations during recovery, as well as normal execution. On the other hand, transitions with a common constraint may be treated as dependent, where recovery of one may require recovery of another. Second, we assume that all information about a global state needed for a correct recovery is captured by the corresponding execution sequence. This requires the software designer to explicitly distinguish as FSM states and transitions any global states or operations that could affect failure recovery correctness.

We use execution sequences only for reasoning about the properties of augmented FSM's that will guarantee correctness of recovery. Once we have identified these properties, we do not need execution sequences for analyzing a particular application. Instead we use the behavior FSM specification for analyzing the correctness of recovery using a suffix of the execution sequence. This approach contrasts with most other works [9,31,45,64] that use a history log for analyzing and verifying the correctness of recovery of individual applications.

In this chapter, we describe the notion of execution sequences, equivalence between execution sequences, how they are modified by normal and recovery operations and how they are used to analyze recovery from a failure. We discuss how consistency is preserved despite modifications of an execution sequence during recovery. Dependencies between pairs of operations are analyzed from the behavior specification of the application to ensure that modified execution sequences preserve consistency.

6.1. Execution Sequences

A transition *execution* is an *instance* of executing a transition in a FSM specification. An *execution sequence* is a sequence of transition executions in order of completion (as viewed by the consistency control manager). It represents a function that returns the current global state when applied to an initial global state. An execution sequence may contain transition executions of several processes.

An execution sequence maps to a path in the behavior of the application defined by the restricted product graph. Two execution sequences are equivalent if the corresponding paths in the restricted product graph are equivalent. We use the notations $<S_i, S_j>$ and $<S_i..S_k..S_j>$ to denote paths from a state S_i to a state S_j in the restricted product machine. The first form denotes *any* path from S_i to S_j. In the second form, we require the path pass through the intermediate state S_k. We can similarly use transitions instead of states to define paths.

We define *path equivalence* as: every path in $<S_o, S_d>$ is equivalent in effect since they all move the application from the original state S_o to the destination state S_d. All operations following S_d will not depend on which path in $<S_o, S_d>$ is used since the behavior of an FSM depends solely on its current state and does not depend on how it arrives at that state. The principle of path equivalence implies both the principle of *substitution* of one operation for another and the principle of *permutation* (*reordering*) of a sequence of operations. By the principle of substitution, two different set of operations of a basic machine with identical source and destination states are equivalent. By the principle of permutation, two paths that have identical source and destination states but contain different ordering of the same transitions are equivalent. For example, consider three product transitions a, b, and c. Two distinct and legal paths $<a,b,c>$ and $<c,b,a>$, that exist between the product states S_o and S_d, are equivalent although a and b do not commute (similarly b and c). On the other hand, all operations that commute can always be permuted. Thus, permutativity property subsumes commutativity property that is described in [87].

The notion of execution sequence is similar to history in [31,45,64,87]. We however use execution sequences only for reasoning about the properties of the augmented FSM's that will guarantee correctness of recovery. The main difference between the two approaches is that an FSM is an abstract behavior model that includes both general synchronization constraints and dependency relations whereas analysis using history logs requires additional information from conflict tables to define mutual exclusion and commutativity of operations.

6.2. Modification of Execution Sequences

Execution sequences may be modified by various actions during normal execution, failure and recovery. The following axioms define how execution sequences are affected by these actions.

(A1) If an execution sequence maps to a legal path in the FSM specification, then the execution sequence is legal.

(A2) During normal execution, an execution sequence can only be appended with new operation executions.

(A3) When a failure occurs, the system always records an execution sequence up to the last product state before the failure.

(A4) Reinstate recovery operations may delete operations from an execution sequence, not limited to the tail of the sequence.

(A5) Non-reinstate recovery operations only append to execution sequences and do not delete any previous operation execution.

Axiom A1 is justified by the normal interpretation of FSM's that an execution sequence is *legal* according to the FSM specification if and only if it is accepted by a FSM specification (we assume every state is an accepting state). Since normal execution progresses only in the forward direction, axiom A2 is justified. The consistency control manager (Chapter 5) will have a record of a valid execution sequence before the failure, justifying axiom A3. In practice, the size of an execution sequence can be limited by setting it to empty whenever a firewall state (Section 7.1) is reached.

While normal executions only append to an execution sequence, recovery operations may modify an execution sequence if they involve reinstate operations (axiom A4). The set of erased executions is empty

when a recovery is complete and normal operations resume. When reinstate recovery operations are used during a recovery the set of erased executions is non-empty. After a reinstate recovery erases some operation executions of a process, all other processes should perceive the erased operations of the first process as if they were never executed. A recovery involving only non-reinstate operations does not modify an execution sequence (Axiom 5) since non-reinstate operations move the applications to some future states without affecting previous executions.

Since recovery operations may erase operations from an execution sequence, consistency of the global state may be affected. In the following sub-sections, we first define a notion of dependence between pairs of operations and then describe how interactive consistency can be preserved.

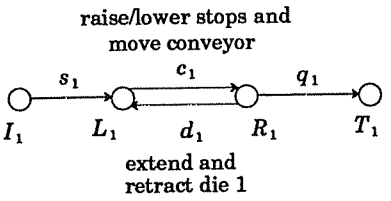
6.3. Analysis of Dependence from Behavior Specification

Synchronization constraints defined by software designers may restrict the order in which basic transitions can take place. This sequencing constitutes control dependencies among the processes. To describe these dependencies, we define two relations on basic transitions: "can precede" ($\rightarrow$) and "necessarily precede" ($\triangleright$). These relations are defined in a scope $[S_i, S_j]$, a subgraph of the restricted product graph, with source S_i, sink S_j, and free of cycles involving both S_i and S_j, where such cycles are broken by removing outedges (in those cycles) from S_j. To simplify the discussion, we refer to the start and end states as singleton states – we can easily extend the analysis to sets of states.

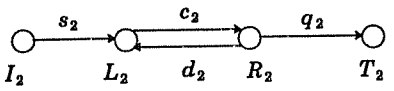
We first define two predicates: *Occur* on basic transitions and *Legal* on paths. $[S_i, S_j]$ $Occur(A)$ is true if basic transition A occurs in a path in the scope $[S_i, S_j]$. $Legal(<S_i,S_j>)$ is true if $<S_i, S_j>$ is a legal path in the restricted product machine graph.

$[S_i,S_j]\, A \rightarrow B$ denotes that the basic transition A *can* precede B in the scope $[S_i, S_j]$, i.e. $\exists S_k(Legal(<S_i..S_k..S_j>) \wedge [S_i,S_k]Occur(A) \wedge [S_k,S_j]Occur(B))$. The incidental precedence relation "$\rightarrow$" does not require the two transitions to be immediately adjacent. For example, in Figure 6.1(c), $[I_1I_2, R_1L_2]$ $d_1 \rightarrow s_2$ is true. We use the notation $[S_i,S_k]\, A \not\rightarrow B$ to denote $\neg([S_i,S_k]A \rightarrow B)$.

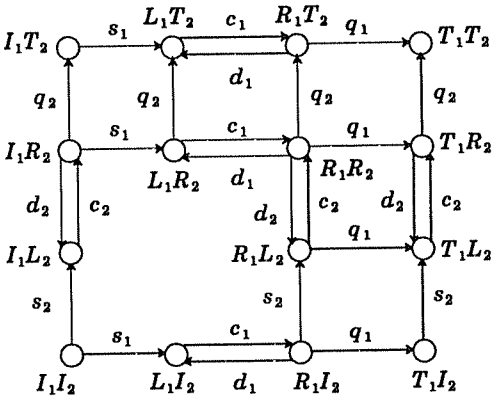
$[S_i,S_j]\, A \triangleright B$ denotes that the basic transition B occurs in $[S_i, S_j]$ and the basic transition A must always occur before it along every path from S_i to B, i.e. $[S_i,S_j]Occur(B) \wedge \neg\exists S_k(Legal(<S_i..S_k..S_j>) \wedge [S_i,S_k](B \rightarrow A \wedge A \not\rightarrow B))$. We say A *necessarily precedes* B if the relation $A \triangleright B$ holds. The $\triangleright$ relation holds between two basic transitions if and only if the second must be synchronized



(a) Basic FSM for Controlling DIE 1



(b) Basic FSM for Controlling DIE 2



(c) Restricted Product FSM

Figure 6.1. FSM specification of the Manufacturing Application

after the first in the subgraph $[S_i, S_j]$, justifying the following axiom.

(A6) If $[S_i, S_j]\ A \triangleright B$ holds, then B *depends* on the conditions (or information) established by A in the scope $[S_i, S_j]$.

This dependence relation encapsulates all forms of control dependence including those that results from mutual exclusion mechanisms, conditional synchronization and triggers. As discussed earlier, software designers explicitly specify these important restriction on the behavior of the application. For example, in Figure 6.1(c), in the scope $[I_1I_2, R_1L_2]$, the transition s_2 depends on c_1 since enforcing the mutual exclusion requirement of the application results in the relation $[I_1I_2, R_1L_2]\ c_1 \triangleright s_2$ being true.

The notion of dependence in [45, 64, 87] is based not only on the ordering in a history log (which can be incidental) but also on a separate synchronization arbitrator, such as a conflict table. For example, if a transaction T_1's write to data x is recorded before T_2's read of data x in the history log and write and read operations are incompatible in the conflict table, then T_2 is dependent on T_1. In comparison, we combine the two types of information in the restricted product machine graph and hence we can extract dependence information solely from the graph. The restricted product machine graph also contains more complex synchronization constraints that cannot be expressed in conflict tables, e.g. conditional synchronization and triggers. While conflict tables prevent any pair of operations from executing concurrently, it cannot enforce sequencing of operations. In a restricted product machine, sequencing can be enforced by removing paths that would allow them to be executed in a different order. The

limitation of the FSM approach is the additional cost involved in analyzing the restricted product graph to determine the dependencies between transitions. In Section 7.4, we discuss the algorithms for detecting dependencies and the time complexity of the algorithms.

Another difference in our notion of dependence is that a larger scope may have fewer dependencies than a small one because of the greater number of alternative equivalent paths. This is possible if a dependent process can reach a state (in a larger scope) regardless of whether the transitions it depends (in the smaller scope) had occurred. For example, in Figure 6.1(c), in the scope $[I_1I_2,R_1L_2]$, the relation $c_1 \triangleright s_2$ holds, but not in the larger scope $[I_1I_2,R_1R_2]$.

6.4. Effects of Modifying Execution Sequences on Consistency

From the definition of interactive consistency in Section 3.1, consistency is preserve if information established by a process that enable another process to execute some operations is not erased by some recovery operations. This notion of consistency is similar to those in [18,39,44]. To analyze consistency, we must rely on the actual executions of the processes represented by an execution sequence. We cannot analyze consistency solely from the behavior of the application represented by a restricted product machine because although the behavior defines the allowable transitions it does not provide information on the actual execution.

Modifying execution sequences by appending executions does not affect consistency if the executions conform to the allowable transitions of the FSM (Axiom A1). However, reinstate recovery operations that erase executions from the execution sequence may remove information on which another process depends, e.g. values of condition variables. Suppose an operation execution of a process establishes a precondition that enables another process to execute some operations. If the operation execution of the first process is subsequently erased by a recovery operation, then the precondition established earlier is also erased. The executions of the second process that depend on the precondition are then no longer valid since it is based on an information that no longer exists.

To determine if an execution may depend on information established by another process, we use the notion of dependence described in the previous section. Dependencies between operations are determined from the restricted product machine of the application. We define an execution of a basic transition A to be *invalidated* in a scope $[S_i, S_j]$ if (1) the execution of A has been erased by a recovery operation (e.g.

rolled back), or (2) it is dependent on an invalidated execution of basic transition B, such that $[S_i, S_j]$ $B \triangleright A$. We can only use properties of the FSM model conservatively to define how transition executions are invalidated. When a basic transition execution is erased, we conservatively mark as invalid all previous execution of the product transitions (in the scope $[S_i, S_j]$) that map to the basic transition. The second clause in the definition of invalidation is conservative in that A may be dependent on another execution of B that is not invalidated. We need to assume this since we do not use a history log of the actual execution. (We can certainly improve on this initial simple approach by using a short history log for resolving any uncertainty of which product transitions were erased by the recovery action. In many applications, the additional gain in recovery efficiency may not be worth the additional cost of maintaining a history log.) Thus, execution invalidation can be determined from a dependence analysis of the behavior of the application.

We can now redefine interactive consistency using the notion of equivalence between execution sequences and execution invalidation. Interactive consistency is preserved at a product state S_j if S_j results from an execution sequence (or one equivalent to it) that does not contain invalidated operation executions in a scope $[S_i,S_j]$, where S_i is a known consistent product state. Since there is no operation that is dependent on the erased executions, then the state of each process at the product states S_j does not depend on information established by the erased executions.

This definition of interactive consistency is used for defining the correctness of recovery described in the next chapter. In Section 7.2.3, we will show that if all operation executions in an execution sequence are not dependent on executions erased by some recovery operations, then we can find another equivalent sequence such that the erased executions occur after those non-dependent executions. If no such equivalent execution sequence can be found and dependent operations are not erased during a recovery, then the recovery state is inconsistent.

While we use execution sequences to reason about the conditions for correctness of recovery, we do not use them to analyze the recovery of a particular application. Instead we only use the abstract FSM specification to derive equivalent execution sequences and to analyze transition execution invalidation (based on necessary precedence) for ensuring that a particular recovery is correct.

Chapter 7

Recovery Using Behavior Specification

In recovery management, we are primarily concerned with how to automate an efficient recovery of a group of processes, which may interact in arbitrarily complex ways, from the failure of one or more of these processes. Because failure of a process may cause inconsistency, the goal of recovery is to place all affected processes in a globally consistent state again where they may continue with their normal execution. The recovery state selected must be a legal product state in the FSM specification. It must also preserve interactive consistency. In addition, actions taken to reach the recovery state must obey the behavior constraints of the FSM.

In this chapter, we first define the three conditions in terms of the behavior and execution models. We then describe a method for computing dependencies efficiently from the FSM specification. We finally present the algorithms for detecting dependencies and computing a recovery state.

7.1. Basic Definitions and Notations

Historical basic states are those reached in the actual execution of a process. Historical product states are those collectively reached by the group of processes. We assume the existence of at least one consistent (firewall) historical state S_p of the restricted product machine, e.g. a designer-defined initial state or a global checkpoint state. S_p serves as a firewall during recovery because an application may never be required to recover to states that precede S_p. Figure 7.1 shows a consistent historical state S_p in a sub-graph of a restricted product machine.

The point of failure S_c is the last normal state entered before a failure occurs. As described in Chapter 5, the consistency control manager monitors all states entered by each basic machine. The manager can then determine the state S_c after a failure, although it need not keep the complete state information of processes it monitors.

From a particular S_c, we can determine a restricted set of potential future states, $\mathcal{F}$, reachable from S_c. The path from S_c to any of these future states may not enter S_p. In this discussion, let S_f be a state in $\mathcal{F}$. Figure 7.1 shows some future states in a restricted product machine. S_f is similar to a future final

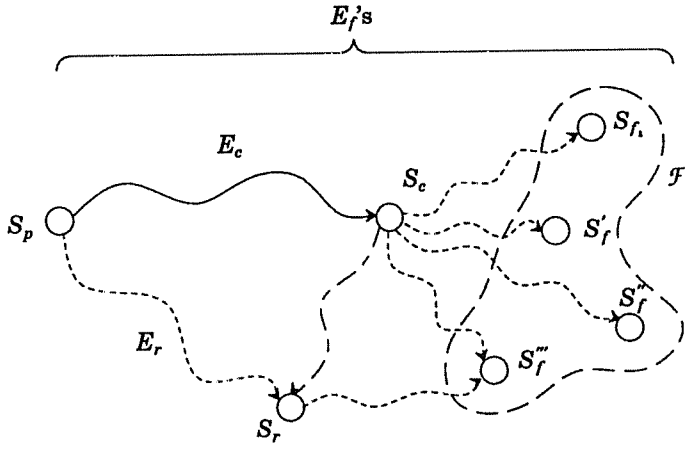


Figure 7.1. A Sub-Graph of a Restricted Product Machine

state defined in [61] with two exceptions. First, we consider not just the final states but any state in any path from S_c to the final states. The set of future states may be restricted to significant states, i.e. states where useful application-specific work has been performed. Significant states are explicitly defined by the designer. Second, we consider states reachable from S_c instead of from the initial state.

We refer to S_r as the recovery (normal) state that is reached by a recovery path from S_c. The recovery path consists of first the instantaneous transition to a failure state followed by a sequence of recovery transitions. We include transitions to the failure state in a recovery path to simplify the discussion. S_r may be anywhere in the restricted product machine, so long as there exists a legal recovery path that can reach it.

7.2. Correctness of Recovery

The main task of failure recovery is to find an appropriate recovery path through the restricted product machine that preserves consistency invariants. We do not rely on correctness conditions that transform problems in a concurrent domain into simpler problems in the sequential domain, such as correctness conditions based on serializability and linearizability. Instead we reason about correctness in the concurrent domain with partial-order semantics.

Using the behavior model, we now define more precisely the three conditions, described at the end of Section 3.4, that ensure recovery states are consistent. The recovery of finite-state processes is correct if the following three conditions are satisfied,

(1) there must be legal path from the point of failure S_c to the recovery state S_r (*legality* condition),

(2) the recovery state S_r is in a legal path from a consistent historical state S_p to a future state S_f (*continuity* condition), and

(3) there is a legal path from the consistent historical state S_p to the recovery state S_r that contains no reinstated operation or dependent on a reinstated operation (*dependency* condition).

The legality condition ensures that recovery operations obey synchronization constraints. The continuity condition ensures that the behavior at the recovery state preserves some of the behavior before the failure. It prevents the problem of recovering to a state that is not reachable from a historical state. In Section 8.2, we will discuss how it can be strengthened for specific recovery policies. The dependency condition ensures that all operations dependent on recovered operations are themselves recovered. The following sub-sections discuss these correctness conditions in more detail.

7.2.1. Synchronization Consistency

The first condition ensures that actions during recovery are subject to the same sequencing and synchronization constraints as normal activity. For example, in a manufacturing cell, when we recover a robot arm to a specific location, the arm movement should not interfere with the normal operation of another equipment or robot. This is enforced by requiring a legal path from S_c to S_r.

If there is a legal path $<S_c, S_r>$ from a failure state S_c to a recovery state S_r, then the recovery operations in $<S_c, S_r>$ obey synchronization constraints. Since legal paths contain only legal transitions that are not removed by the synchronization constraints (Section 4.2) on the normal and recovery transitions, then all recovery operations in the legal path $<S_c,S_r>$ obey the synchronization constraints.

7.2.2. Continuity

The second condition ensures that the behavior after recovery is related to the behavior before a failure in an appropriate way. This is the concept of *continuity* that can be captured by legal paths in the restricted product graph. To show that the continuity condition preserves some of the behavior before a failure, we first introduce some basic concepts and notations.

Let E_c be an actual execution sequence up to the point of failure. Since processes never recover to a product state before S_p, we assume E_c begins at S_p. E_c maps to a legal path in the restricted product graph from S_p to S_c. We define an E_f as E_c concatenated with a legal path to some future state. Since only the operations in E_f before S_c are really executed, E_f is a "projected" execution sequence. We define the behavior of a group of processes at a particular product state as a set of sequences of transitions that they *can* execute beginning at that product state. For each product state, the legal set of transition sequences can be derived from the restricted product graph. The behavior before the failure (at S_c) is a set of legal paths to the states in $\mathcal{F}$. Since we know the actual execution sequence E_c, the behavior at S_p contains the set of E_f's. If a recovery results in an execution sequence (possibly projected) that is a prefix of an E_f, then some of the behavior before the failure is preserved. For a recovery state S_r, E_r denotes a sequence E_f (or one equivalent to E_f) delimited by S_p and S_r. (Although S_r may be in a path that also leads to a state not in $\mathcal{F}$, some future states of S_r are in $\mathcal{F}$.) Figure 7.1 illustrates the execution sequences E_c and E_r and a set of E_f's.

Theorem 1.

If the recovery state S_r is in a legal path from a consistent historical state S_p to some future state S_f, then some of the behavior of the processes before the failure is preserved.

Proof.

We consider the behavior of the processes at S_p that contains the set of E_f's. The behavior at S_p includes the behavior before the failure (at S_c) which is a set of legal paths to those states in $\mathcal{F}$. If S_r is in a legal path from S_p to some S_f, then some future states of S_r are also future states of S_c. Since the set of sequences of transitions $<S_p..S_r..S_f>$ is equivalent to some E_f, then some of the behavior of the processes before the failure is preserved. □

We can improve recovery by using execution sequences that are equivalent to E_f, whenever possible, and decreasing the length of the suffix $E_f - E_r$. We can exploit the principle of substitution discussed in Section 6.1 to improve efficiency of recovery by replacing a set of operations that was executed with another set of operation that have fewer dependent operations. The principle of permutativity can be exploited by using operation ordering that will avoid recovering other processes.

To simplify the mechanical detection of consistent recovery state, we modify the continuity condition as follows: there is a legal path from S_r to a future state S_f. Continuation of behavior is preserved partly by the modified continuity condition that ensures there is a legal path $<S_r,S_f>$, and partly by the dependency condition that ensure there is a legal path $<S_p,S_r>$.

7.2.3. Recovery Involving Dependency

The dependency condition is based on the definition of interactive consistency in Section 6.4, specifically applied to recovery. Recovery operations may affect the execution sequence and consequently the consistency of the global state. To avoid inconsistency, the dependency condition ensures there is at least one legal path $<S_p,S_r>$ that does not contain any invalidated transition. However, there may be several other paths from S_p to S_r that contains invalidated transitions. Theorem 2 states how interactive consistency can be preserved in the presence of failure and recovery operations. We first define a *proper execution sequence* E_r as a prefix of an execution sequence E_f (or one equivalent to it) that contains no reinstated operations or operations dependent on them.

Theorem 2.

A recovery state preserves interactive consistency if and only if there exists a proper execution sequence to the recovery state.

Proof.

We prove the "if" part by considering an equivalent and proper execution sequence E_r, resulting from a recovery. Then the effects of the recovery only eliminate executions that happen after E_r. Since all transitions in E_r happen before the eliminated executions, they can be executed without being dependent on conditions or information from those eliminated executions. Thus, by the definition of interactive consistency (Section 6.4), the recovery state is consistent.

To prove the "only if" part, we note that a recovery satisfies the continuity condition and thus the recovery state is on a path equivalent to some execution sequence E_f. If the recovery state S_r preserves interactive consistency, then by definition of interactive consistency, none of the operations from the historical state S_p to S_r is eliminated or depends on conditions established by eliminated operations. Thus $<S_p,S_r>$ is a proper execution sequence. □

We can restate the dependency condition using the $\triangleright$ relation as follows. Let A_M be a transition execution of a basic machine M that is reinstated and B_N be any transition execution of another basic machine $N(N \neq M)$. We use the notation $[S_i, S_j]$ $(A \not\triangleright B)$ to denote $\neg([S_i, S_j]$ (A $\triangleright$ B)). The dependency condition can be rewritten as follows: $\forall M \forall N \forall A_M \forall B_N$ $(A_M \in <S_p , S_f> \wedge B_N \in <S_p, S_r> \Rightarrow [S_p, S_c]\, A_M \not\triangleright B_N)$. From the definition of the $\triangleright$ relation, we observe that if $[S_p, S_f]$ $(A_M \not\triangleright B_N)$, then $\neg$ $[S_p, S_f]$ $Occur(B) \vee \exists S_k(Legal(<S_p..S_k..S_f>) \wedge [S_p,S_f]$ $(B_N \rightarrow A_M \wedge A_M \not\rightarrow B_N))$. This means that a basic transition B_N can precede an occurrence of A_M in a path $<S_p..S_k..S_f>$. We can thus find a recovery state S_r such that $<S_p,S_r>$ contains B_N but not A_M. The dependency condition ensures the global state is consistent after a recovery by disallowing unerased executions to be dependent on erased executions.

7.3. An Algebraic Method for Computing Dependency

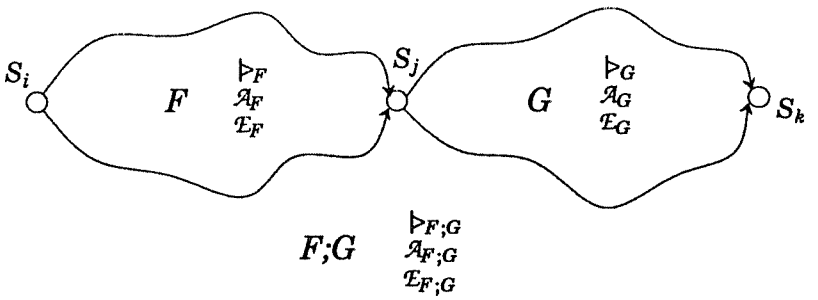
We now discuss a method for computing $\triangleright$ relations in a given scope. We also describe how to transform a subgraph into an equivalent one that gives the same $\triangleright$ results. In particular, we are interested in transforming cyclic subgraphs into acyclic subgraphs in order to compute $\triangleright$ efficiently.

7.3.1. Computing $\triangleright$ of Subgraphs in Series and Parallel

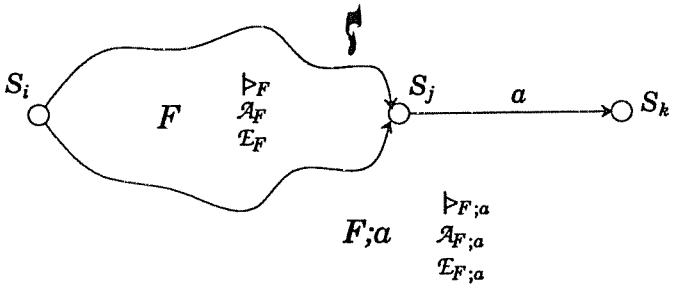
At each node and transition, we record three important sets: $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$. $\mathcal{A}$ is a set of transitions that occur in all the paths from the source S_i of the scope to that node or transition. $\mathcal{E}$ is a set of transitions that occur in at least one path from S_i to that node or transition. (Since $\mathcal{A} \subseteq \mathcal{E}$, only one set needs to be maintained, with appropriate markings to indicate those members that belong to $\mathcal{A}$.) For each transition T, we maintain a set $T\triangleright$ of those transitions that depend on T, i.e. U is in $T\triangleright$ at state S_j if $[S_i,S_j](T \triangleright U)$. $\triangleright$ is a set of $T\triangleright$ for each $T \in \mathcal{E}$. (In the following discussions, the scope may be omitted if it includes the entire subgraph under consideration.)

Recording these sets at a node means that the information are true for all paths from the source to that node while recording them at a transition means that they are true for all paths to the tail of the transition appended with that transition. At the source of the scope S_i, $\mathcal{A} = \{\}$, $\mathcal{E} = \{\}$ and $\triangleright = \{\}$. We next consider how the sets $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ of two subgraphs G_1 and G_2 can be combined in series and parallel.

The expression $F;G$ denotes that a subgraph F is merged in *series* with subgraph G, as Figure 7.2(a) shows. The sets $\mathcal{A}_{F;G}$, $\mathcal{E}_{F;G}$ and $T\triangleright_{F;G}$ of $F;G$ is computed as follows.



(a) Two subgraphs in series



(b) A subgraph in series with a transition

Figure 7.2. Merging in Series

$$\mathcal{A}_{F;G} = \mathcal{A}_F \cup \mathcal{A}_G$$

$$\mathcal{E}_{F;G} = \mathcal{E}_F \cup \mathcal{E}_G$$

$$TH_{F;G} = \begin{cases} T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F) & \text{if } T \in \mathcal{A}_F \\ \{\} & \text{if } T \notin \mathcal{A}_F \end{cases}$$

$$T\triangleright_{F;G} = TH_{F;G} \cup [T\triangleright_F - \mathcal{E}_G] \cup [T\triangleright_G \cap (T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))]$$

$$\triangleright_{F;G} = \{T\triangleright_{F;G} \mid T \in \mathcal{E}_{F;G}\}$$

Figure 7.2(b) shows a special case of a series merge where a subgraph F is merged in series with a transition a, resulting in $F;a$. From the above set of formulas, we can derive for this special case the formulas for computing $\mathcal{A}_{F;a}$, $\mathcal{E}_{F;a}$ and $T\triangleright_{F;a}$ of $F;a$ as follows.

$$\mathcal{A}_{F;a} = \mathcal{A}_F \cup \{a\}$$

$$\mathcal{E}_{F;a} = \mathcal{E}_F \cup \{a\}$$

$$T\triangleright_{F;a} = \begin{cases} T\triangleright_F & \text{if } T \in \mathcal{A}_F \wedge a \in \mathcal{E}_F \\ T\triangleright_F \cup \{a\} & \text{if } T \in \mathcal{A}_F \wedge a \notin \mathcal{E}_F \\ T\triangleright_F - \{a\} & \text{if } T \notin \mathcal{A}_F \end{cases}$$

$$\triangleright_{F;a} = \{T\triangleright_{F;a} \mid T \in \mathcal{E}_{F;a}\}$$

The expression $F \| G$ denotes that a subgraph F is merged in *parallel* with subgraph G as Figure 7.3 shows. F and G have common source S_i and sink S_j. The sets $\mathcal{A}_{F\|G}$, $\mathcal{E}_{F\|G}$ and $T\triangleright_{F\|G}$ of $F \| G$ is computed as follows.

$$\mathcal{A}_{F\|G} = \mathcal{A}_F \cap \mathcal{A}_G$$

$$\mathcal{E}_{F\|G} = \mathcal{E}_F \cup \mathcal{E}_G$$

$$T\triangleright_{F\|G} = [T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F)] \cap [T\triangleright_G \cup (\mathcal{E}_F - \mathcal{E}_G)]$$

$$= [T\triangleright_F - \mathcal{E}_G] \cup [T\triangleright_G \cap (T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))]$$

$$\triangleright_{F\|G} = \{T\triangleright_{F\|G} \mid T \in \mathcal{E}_{F\|G}\}$$

We derive $T\triangleright_{F\|G}$ in the parallel merge above by expansion and elimination, i.e. $[T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F)] \cap [T\triangleright_G \cup (\mathcal{E}_F - \mathcal{E}_G)]$ = $[(\mathcal{E}_F - \mathcal{E}_G) \cap (T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))] \cup [T\triangleright_G \cap (T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))]$ = $[T\triangleright_F - \mathcal{E}_G] \cup [T\triangleright_G \cap (T\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))]$. Thus, $T\triangleright_{F\|G}$ in the parallel merge is identical to $T\triangleright_{F;G} - TH_{F;G}$ in the series merge.

In graph composition expressions where parentheses are omitted, ";" and "||" are evaluated from left to right, e.g. $F;G \| H;I = ((F;G) \| H);I$.

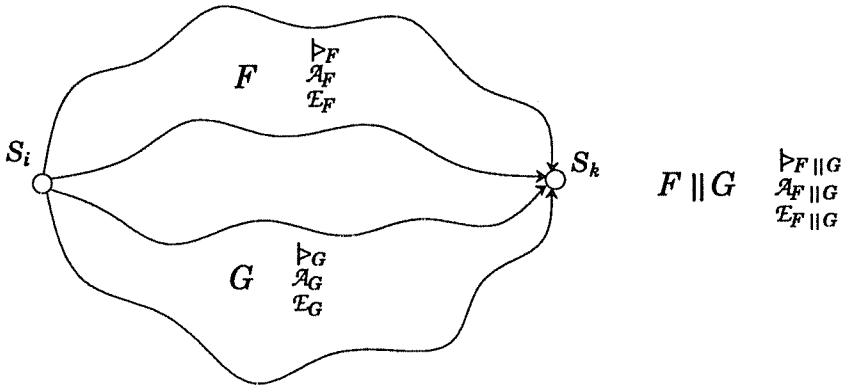


Figure 7.3. Merging in Parallel

7.3.2. Graph Transformation that Guarantees $\triangleright$-Equivalence

The above formulas allow us to compute the sets $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ of series and parallel combinations of subgraphs where the resulting graph contains no cycle. To compute those sets for cyclic graphs, we first transform cyclic graphs into acyclic graphs that give the same results for $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$. We first define a *source* of a strongly connected component (SCC) as a node with incoming transitions (defined as *input transitions*) that do not belong to the SCC. A *sink* of an SCC is a node with outgoing transitions (defined as *output transitions*) that do not belong to the SCC. Two graphs are $\triangleright$*–equivalent* if they produce the same results for $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ at each corresponding sink, given identical values for $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ at the input transitions. The cyclic to acyclic graph transformation makes use of several important properties of $\triangleright$ in SCC's. The $\triangleright$'s of nodes in an SCC depend on the source nodes but not the sink nodes. In any SCC, $\triangleright$ is identical at all its nodes regardless of the number of sources.

Theorem 3.

$\triangleright$ at all nodes of a strongly connected component are identical.

Proof.

Let $a \triangleright b$ at one node X in a strongly connected component, where a and b are transitions. Assume that $a \not\triangleright b$ at another node Y. There are two cases at Y: either $b \notin \mathcal{E}_Y$ or there is some path containing b whose prefix does not contain a. The first case cannot occur as $\mathcal{E}$ is (obviously) identical at all nodes in a strongly connected component. For the second case, take that path, append to it any path to X and conclude that $a \not\triangleright b$ at X. Therefore, by contradiction $\triangleright$ at all nodes of a strongly connected component are identical. □

Since by Theorem 3, $\triangleright$ at all nodes of an SCC are identical, we introduce a virtual sentinel node Σ that is reachable from every node of the SCC and contains the $\triangleright$ of the SCC. We will first discuss SCC's with only one source and later extend the results to SCC's with more than one source.

Transforming Strongly Connected Components with One Source

We transform a cyclic graph into a $\triangleright$-equivalent acyclic graph using the following three steps.

(1) Transform the SCC, using depth-first search, into a spanning tree with additional back edges, forward edges and cross edges. The source of the SCC is the root of the spanning tree.

(2) Create a sentinel node Σ. Modify all back edges to point to Σ instead of an ancestor node. (We omit forward and cross edges for now and discuss them later in the section.) We will show that $\triangleright_\Sigma$ is the $\triangleright$ of the SCC.

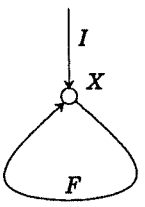
(3) Duplicate the resulting graph and create an ε transition from Σ to the root of the duplicated subgraph. (We may omit ε and collapse its head and tail nodes.) Input transitions to the source and all output transitions at the sinks of the SCC are also duplicated. Remove all output transitions in the original subgraph, the duplicated sentinel node Σ′ and all edges to Σ′.

We now show that the computation of $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ at the sinks of the transformed graph is equivalent to that of the original graph. The duplication in the third step is to propagate the computed sets $\mathcal{A}_\Sigma$, $\mathcal{E}_\Sigma$ and $\triangleright_\Sigma$ of the SCC to all nodes of the duplicated subgraph, particularly the sinks.

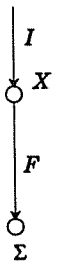
We first consider a strongly connected component with a simple cycle consisting of only one back edge, one source and one sink where the source and sink are identical. For example, Figure 7.4(a) shows a strongly connected component that consists of a single acyclic path F from X back to itself, where X is both the source and sink. A subgraph I merges in series at node X through an input transition. (Output transitions are omited from the figure.) We show that Figure 7.4(b) is $\triangleright$-equivalent to Figure 7.4(a).

Lemma 1.

Let a subgraph I merge in series at node X with a strongly connected component that consists of a single acyclic path F from X back to itself. At node X, $\triangleright_{I\,||(I;F)} = \triangleright_{I;F}$, $\mathcal{E}_{I\,||(I;F)} = \mathcal{E}_{I;F}$ and $\mathcal{A}_{I\,||(I;F)} = \mathcal{A}_I$.



(a) A strongly connected component



(b) $\triangleright$-equivalent graph

Figure 7.4. A Simple SCC with a Single Source

Proof.

We can compute $\mathcal{A}_F$, $\mathcal{E}_F$ and $\triangleright_F$ independently. When we merge $\mathcal{A}_I$ and $\mathcal{A}_{I;F}$ in parallel at X, we get $\mathcal{A}_I \cap \mathcal{A}_{I;F} = \mathcal{A}_I$, since $\mathcal{A}_I \subseteq \mathcal{A}_{I;F}$. Merging $\mathcal{E}_I$ and $\mathcal{E}_{I;F}$ in parallel at X, we get $\mathcal{E}_I \cup \mathcal{E}_{I;F} = \mathcal{E}_{I;F}$, since $\mathcal{E}_I \subseteq \mathcal{E}_{I;F}$. Next, we prove that $\triangleright_{I||(I;F)} = \triangleright_{I;F}$. We show this by proving $T\triangleright_{I||(I;F)} = T\triangleright_{I;F}$ for each $T \in \mathcal{E}_{I||(I;F)}$.

$$\begin{aligned} T\triangleright_{I||(I;F)} &= [T\triangleright_I \cup (\mathcal{E}_{I;F} - \mathcal{E}_I)] \cap [T\triangleright_{I;F} \cup (\mathcal{E}_I - \mathcal{E}_{I;F})] \\ &= [T\triangleright_I \cup (\mathcal{E}_F - \mathcal{E}_I)] \cap T\triangleright_{I;F} \\ &= T\triangleright_{I;F} \end{aligned}$$

We can derive the third line from the second because $T\triangleright_{I;F} \subseteq [T\triangleright_I \cup (\mathcal{E}_F - \mathcal{E}_I)]$. To show that this relation is true, we need only consider four cases for each transition $t \in \mathcal{E}_I \cup \mathcal{E}_F$. First, if $t \notin \mathcal{E}_F$ the relation is true because t will not increase the size of $T\triangleright_I$. Second, if $t \notin \mathcal{E}_I$, the relation is true because $T\triangleright_{I;F} \subseteq [T\triangleright_I \cup \mathcal{E}_F]$. Third, if $t \in \mathcal{E}_I$, $t \in \mathcal{E}_F$ and $t \in T\triangleright_I$, then the relation is true whether $t \in T\triangleright_{I;F}$ or $t \notin T\triangleright_{I;F}$. Fourth, if $t \in \mathcal{E}_I$ and $t \in \mathcal{E}_F$, but $t \notin T\triangleright_I$, then $t \notin T\triangleright_{I;F}$, in which case the relation is true. □

From Lemma 1 and Theorem 3, the set $\triangleright$ of each node in a simple cycle (involving only one back edge) is always equal to $\triangleright_{I;F}$ where I is the subgraph in series with the cycle at a single node X, and F is the path from X to itself. The set $\mathcal{E}$ at each node is always equal to $\mathcal{E}_{I;F}$. The set $\mathcal{A}_X$ is always equal to $\mathcal{A}_I$. $\mathcal{A}$ may be different at each node in the cycle but for each node N in the cycle, $\mathcal{A}_N$ is always the same regardless of the number of times the cycle is traversed. Thus, each of the three sets, $\triangleright$, $\mathcal{E}$ and $\mathcal{A}$ of the simple cycle can be computed from the $\triangleright$-equivalent graph shown in Figure 7.4(b) and is independent of the number of times the cycle is traversed.

In the remainder of our discussion, we use the notation $\triangleright_F \mathrel{O} \triangleright_G$ to mean $T\triangleright_F \mathrel{O} T\triangleright_G$ for each $T \in \mathcal{E}_F \cup \mathcal{E}_G$ where O may be any of the following operators: $\subseteq$, $\supseteq$, $\cup$, $\cap$, $=$, or $-$.

Lemma 2.

Let a graph contain a simple cycle with a back edge c to a node V. Next modify the graph such that c points instead to X, an ancestor of V. Then $\triangleright_C$ in both graphs are equivalent, where $\triangleright_C$ is the $\triangleright$ computed at c.

Proof.

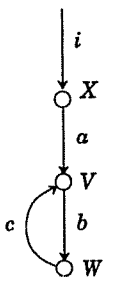
Consider the graph with a simple cycle in Figure 7.5(a) and a modified graph with a moved back edge shown in Figure 7.5(b). In both cases, from Lemma 1, $\mathcal{E}$ computed at c are identical. Similarly, $\mathcal{A}$ are identical for each corresponding node. In the case shown in Figure 7.5(a), $\triangleright_C = \triangleright_{(I;a)\|(I;a;b;c)} = \triangleright_{I;a;b;c}$ (Lemma 1), where I is the subgraph traversed before entering X through i. In the case shown in Figure 7.5(b), $\triangleright_C = \triangleright_{I\|(I;a;b;c)} = \triangleright_{I;a;b;c}$. Thus, $\triangleright_C$ is identical in both cases. ($\triangleright_X$ may be different since b and c do not affect $\triangleright_X$ in Figure 7.5(a)). □

Lemma 3.

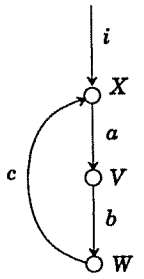
Let $F;G;F$ be a series merge of the subgraphs shown in Figure 7.6. Then $\mathcal{E}_{(F;G);F} = \mathcal{E}_{F;G}$, $\mathcal{A}_{(F;G);F} = \mathcal{A}_{F;G}$ and $\triangleright_{(F;G);F} = \triangleright_{F;G}$.

Proof.

From the series merge formula, we derive $\mathcal{E}_{(F;G);F} = (\mathcal{E}_F \cup \mathcal{E}_G) \cup \mathcal{E}_F = \mathcal{E}_F \cup \mathcal{E}_G = \mathcal{E}_{F;G}$. Similarly, $\mathcal{A}_{(F;G);F} = (\mathcal{A}_F \cup \mathcal{A}_G) \cup \mathcal{A}_F = \mathcal{A}_{F;G}$. Next, we prove that $\triangleright_{(F;G);F} = \triangleright_{F;G}$. First we show that $\triangleright_F \cap \triangleright_{F;G} \subseteq \triangleright_{F;G}$. Consider first the case for each transition $t \in \mathcal{A}_F$. Then, from the series formula, $\triangleright_{F;G} = [\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F)] \cup [\triangleright_F - \mathcal{E}_G] \cup [\triangleright_G \cap (\triangleright_F \cup (\mathcal{E}_G - \mathcal{E}_F))]$. Thus, $\triangleright_F \cap \triangleright_{F;G} = \triangleright_F \cup [\triangleright_F - \mathcal{E}_G] \cup [\triangleright_G \cap \triangleright_F] \subseteq$



(a) A strongly connected component



(b) Modified Graph

Figure 7.5. Moving the Head of a Back Edge to An Ancestor Node

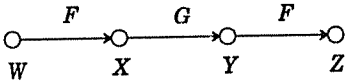


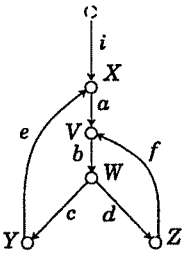
Figure 7.6. Merging Repeated Subgraphs in Series

$\triangleright_{F;G}$. Similarly, for the second case where $t \notin \mathcal{A}_F$, $\not\triangleright_F \cap \triangleright_{F;G} = [\triangleright_F - \mathcal{E}_G] \cup [\triangleright_G \cap \triangleright_F] \subseteq \triangleright_{F;G}$. Therefore, $\triangleright_F \cap \triangleright_{F;G} \subseteq \triangleright_{F;G}$ in both cases. Finally, $\triangleright_{(F;G);F} = \triangleright_{F;G} \cup [\triangleright_{F;G} - \mathcal{E}_F] \cup [\triangleright_F \cap \triangleright_{F;G}] = \triangleright_{F;G}$. □

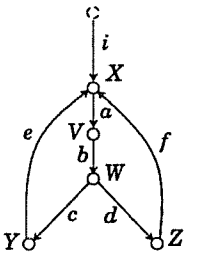
Using both Lemma 1 and Lemma 2, we can show that even if some back edges enter a descendent of the source X, i.e. overlapping cycles, the graph with all back edges redirected to Σ will still allow us to compute $\triangleright$ of the SCC correctly at Σ. For example, Figure 7.1(a) shows a strongly connected component containing a single source X. A descendent W of X contains two branches where one branch has a back edge to X and the other has a back edge to a node V in the path from X to W. There is no cross or forward edge between the branches. We show that the graph, Figure 7.7(c), resulting from the transformation steps is $\triangleright$-equivalent graph to the original graph in Figure 7.7(a). Using Lemma 3, we can then show that $\mathcal{E}$ and $\triangleright$ at the sinks of the duplicated subgraph are identical to $\mathcal{E}_\Sigma$ and $\triangleright_\Sigma$ respectively.

Theorem 4.

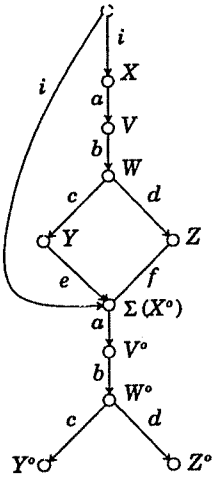
Let a strongly connected component contain a single source and two back edges, one entering the



(a) A strongly connected component



(b) Modified Graph



(c) $\triangleright$-equivalent graph

Figure 7.7. An SCC with Overlapping Cycles

source and another entering a descendent of the source. Then, the three transformation steps produce a $\triangleright$-equivalent graph. $\mathcal{A}$ can be computed by ignoring all the back edges. Parallel merge of $\mathcal{E}$ and $\triangleright$ at head of the back edges in the original graph is equivalent to parallel merge of $\mathcal{E}$ and $\triangleright$ at the sentinel Σ. $\mathcal{E}$ and $\triangleright$ at all duplicated nodes are identical to $\mathcal{E}_\Sigma$ and $\triangleright_\Sigma$ respectively.

Proof.

Consider the subgraph in Figure 7.7(a). $\triangleright_F$ (i.e. $\triangleright$ computed at f) may modify the value of $\triangleright_E$ (i.e. $\triangleright$ computed at e). From Lemma 2, identical $\triangleright_F$ can be computed by redirecting f to point to X instead of V (Figure 7.7(b)). Furthermore, the results of $\triangleright_F$ are merged at X through c and e. After redirecting the back edge f to X, we can transform the graph into an equivalent graph shown in Figure 7.7(c) where all the back edges enter Σ. Since there is no cross edge between the branches, cycles within one branch do not affect another branch. From Lemma 1, $\mathcal{E}_X$ in the original graph (Figure 7.7(a) is identical to $\mathcal{E}_\Sigma$ (Figure 7.7(c)). Also, $\mathcal{A}_X = \mathcal{A}_I$ (where I is the graph traversed before entering X) and is independent of the back edges. The proof that $\triangleright_\Sigma$ is equivalent to $\triangleright_{\Sigma||I}$ follows directly from Lemma 1; parallel merge of $\triangleright_\Sigma$ with $\triangleright_I$ at X does not change the value of $\triangleright_\Sigma$, regardless of the number of times X is entered. By Theorem 3, $\triangleright$ at all nodes in the strongly connected component is identical to $\triangleright_\Sigma$. By Lemma 3, $\mathcal{E}_{V^o} = \mathcal{E}_\Sigma$ and $\triangleright_{V^o} = \triangleright_\Sigma$. Similarly for Y^o and Z^o. By Lemma 1, $\mathcal{A}_\Sigma = \mathcal{A}_I$ and $\mathcal{A}$ at each duplicated node can be computed by ignoring the backedges. □

Transforming Strongly Connected Components with Multiple Sources

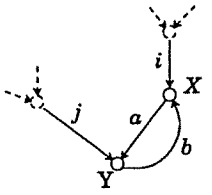
Next we consider strongly connected components with more than one source. The main problem with multiple sources is that $\triangleright$ of input transitions to each source may mutually affect one another. To account for this mutual effect, we add an additional step between step 2 and 3 of the procedure for transforming the cyclic graph with one source:

(2′) Replicate the resulting graph and collapse the root of the replicated subgraph with Σ. Input transitions at all the sources are also replicated (but not the output transitions at the sinks). The sink of the replica is Σ'.

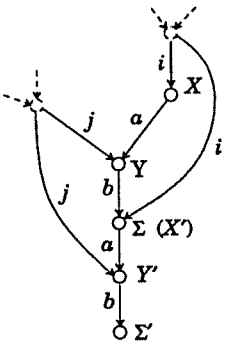
Step 3 of the transformation then duplicates only the original graph, as before, but with an ε transition from Σ' to the root of the duplicated graph. Output transitions exist only in the duplicated graph created by step 3. We will show that the transformed graph is $\triangleright$-equivalent to the original graph by first considering the case with two sources and then the more general case with more than two sources. In the first case, we consider an example of a strongly connected component containing two sources X and Y and edges shown in Figure 7.8(a). Using transformation step 2′ (and omitting step 3 for now), we generate a modified graph shown in Figure 7.8(b). We show that $\triangleright_{\Sigma'}$ of the graph in Figure 7.8(b) is identical to $\triangleright$ of the strongly connected component. We exclude cross (and forward) edges in Lemma 4 and Theorem 5 and later show how cross (and forward) edges may be handled.

Lemma 4.

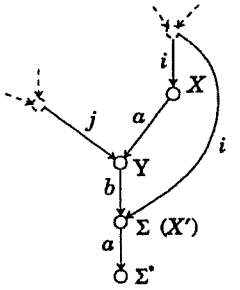
Let a strongly connected component contain two sources. Using transformation step 2′ (and omitting step 3), then $\mathcal{E}$ and $\triangleright$ at the sentinel node Σ' of the resulting graph is identical to $\mathcal{E}$ and $\triangleright$



(a) An SCC with two sources



(b) Modified graph for computing $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$



(c) Reduced graph

Figure 7.8. An SCC with Two Sources

of the SCC. The set $\mathcal{A}$ at each node is equivalent to that computed in the replicated graph with Σ as the source.

Proof.

Consider the SCC in Figure 7.8(a). Let the original subgraph from X to Y (i.e. a) be F and the subgraph from Y to X (i.e. b) be G. We can compute $\mathcal{A}_F$, $\mathcal{E}_F$ and $\triangleright_F$ (also $\mathcal{A}_G$, $\mathcal{E}_G$ and $\triangleright_G$) independently. Let I (J) denote the subgraph traversed, including i (j), before entering X (Y). At node X of the graph in Figure 7.8(a), merging $\mathcal{E}_I$ and $\mathcal{E}_{((I;F)\|J);G}$ in parallel, we get $\mathcal{E}_I \cup \mathcal{E}_{(I;F)\|J;G} = \mathcal{E}_{(I;F)\|J;G}$. This is identical to $\mathcal{E}_\Sigma$. At node Σ in Figure 7.8(c), we derive $\mathcal{A}_\Sigma = \mathcal{A}_{((I;F)\|J;G)\|I} = [((\mathcal{A}_I \cup \mathcal{A}_F) \cap \mathcal{A}_J) \cup \mathcal{A}_G] \cap \mathcal{A}_I = [((\mathcal{A}_I \cup \mathcal{A}_F) \cap \mathcal{A}_J) \cap \mathcal{A}_I] \cup (\mathcal{A}_G \cap \mathcal{A}_I) = (\mathcal{A}_J \cup \mathcal{A}_G) \cap \mathcal{A}_I$ which is identical to $\mathcal{A}_X$ of the original graph in Figure 7.8(a). Similarly, at node Y', $\mathcal{A}_{Y'} = \mathcal{A}_{I;F\|J;G\|I;F\|J} = \mathcal{A}_{(I;F)\|J} = \mathcal{A}_Y$.

To compute $\triangleright$, we must account for the effects of $\mathcal{A}_I$ and $\mathcal{A}_J$ on all the transitions in the strongly connected component. Let $P = ((((I;F)\|J);G)\|I;F$. We now prove that $\triangleright_{(((P\|J);G)\|I);F} = \triangleright_{(P\|J);G} = \triangleright_P = \triangleright_\Sigma$ which is the $\triangleright$ of the SCC; traversing the SCC more times will not change the result of $\triangleright_P$. We observe that $\triangleright_P \subseteq (\triangleright_J \cup (\mathcal{E}_P - \mathcal{E}_J))$ because if a transition t is such that $t \in \mathcal{E}_P$, $t \in \mathcal{E}_J$ and $t \notin \triangleright_J$, then $t \notin \triangleright_P$. Thus, $\triangleright_{P\|J} = [\triangleright_P \cup (\mathcal{E}_J - \mathcal{E}_P)] \cap [\triangleright_J \cup (\mathcal{E}_P - \mathcal{E}_J)] = \triangleright_P \cap [\triangleright_J \cup (\mathcal{E}_P - \mathcal{E}_J)] = \triangleright_P$. Similarly,

$$\begin{aligned}
\triangleright_{P;G} &= \triangleright_P \cup [(\triangleright_P \cup (\mathcal{E}_G - \mathcal{E}_P)) \cap (\triangleright_G \cup (\mathcal{E}_P - \mathcal{E}_G))] \\
&= \triangleright_P \cup [\triangleright_P \cap (\triangleright_G \cup (\mathcal{E}_P - \mathcal{E}_G))] \\
&= \triangleright_P.
\end{aligned}$$

$$\triangleright_{P\|I} = [\triangleright_P \cup (\mathcal{E}_I - \mathcal{E}_P)] \cap [\triangleright_I \cup (\mathcal{E}_P - \mathcal{E}_I)] = \triangleright_P.$$

$$\triangleright_{P;F} = \triangleright_P \cup [(\triangleright_P \cup (\mathcal{E}_F - \mathcal{E}_P)) \cap (\triangleright_F \cup (\mathcal{E}_P - \mathcal{E}_F))] = \triangleright_P.$$

Therefore $\triangleright_{(((P\|J);G)\|I);F} = \triangleright_{(P\|I);F} = \triangleright_P = \triangleright_\Sigma$; i.e. P represents the minimum number of cycles the SCC must be traversed and traversing the SCC more times will not change the value of $\triangleright$ of the SCC. □

From Lemma 4, we note two important properties for any SCC C with multiple sources. First, $\mathcal{E}_C$ must always include $\mathcal{E}_I$ of each input transition i at each source. For example, in Figure 7.8(a), the set $\mathcal{E}$ of the SCC is $\mathcal{E}_{((I;F)\|J);G}$ which includes $\mathcal{E}_I$ and $\mathcal{E}_J$. Second, we must ensure that the $\triangleright$-equivalent graph

G for computing $\triangleright_C$ is such that every input precedes every transition in C. In other words, for each transition t in the SCC, there must be a subgraph $F;T$ of G, where t is in T and i is in F for each input transition i of the SCC. (If $G = F;T$, we say F precedes T.) For example, in Figure 7.8(a), the $\triangleright$ of the SCC is $\triangleright_{((((I;F)\|J);G)\|I;F}$ where both I and J precede both F and G.

Lemma 5.

Let H be a graph that is $\triangleright$-equivalent to a strongly connected component C with multiple sources. Then for each transition t in C, there is a subgraph $F;T$ of H such that t is in T and i is in F for each input transition i of C.

Proof.

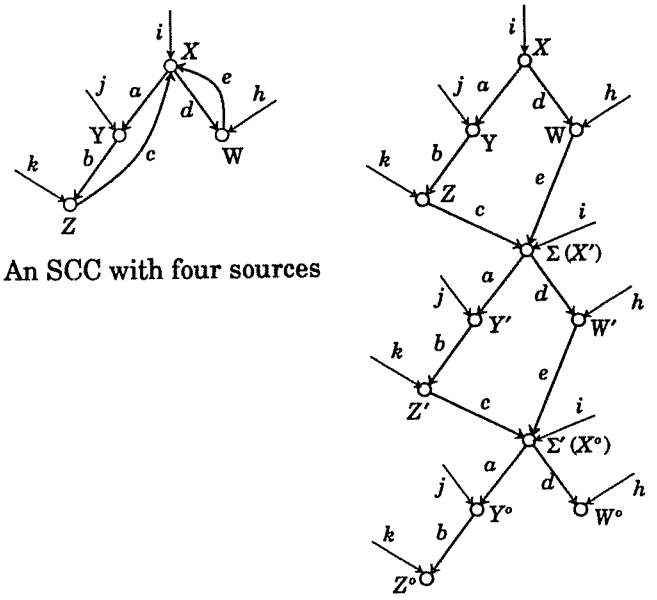
We prove this by contradiction. Consider the SCC shown in Figure 7.8(a) where I, J, F and G are as defined in Lemma 4. Suppose we can use a subgraph from X to Σ for computing the $\triangleright$ of the SCC. Let $Q = (((I;F)\|J);G)\|I$ where there is no subgraph containing both I and J that precedes F. Then $\triangleright_\Sigma = \triangleright_Q$. But $\triangleright_{Q;F}$ may remove some transitions in $\triangleright_Q$ because $\mathcal{A}_{X'} = (\mathcal{A}_J \cup \mathcal{A}_G) \cap \mathcal{A}_I \subseteq \mathcal{A}_I$. Since by Theorem 3 that states all nodes in a SCC must have identical $\triangleright$, then $\triangleright_\Sigma$ is not the $\triangleright$ of the SCC. □

For SCC's with two sources, it is sufficient to replicate only the spanning tree with one source as root minus the subgraph with the second source as root. Figure 7.8(c) is a reduced graph of Figure 7.8(a) where $\triangleright_{\Sigma'} = \triangleright_{\Sigma'}$. We can reduce the replicated subgraph because the computation of $\triangleright$ at the subtree with the second source as root would have included the $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ values from incoming transitions at the first source.

By replicating using step 2′, we can transform SCC's with more than two sources into a $\triangleright$-equivalent graph. We consider a strongly connected component C (Figure 7.9(a)) that contains N sources where $N > 1$. We now show that a graph (Figure 7.9(b)) generated using the transformation steps with the additional replication step is $\triangleright$-equivalent to the original graph.

Theorem 5.

Let a strongly connected component C contain N sources where $N > 1$. Then the graph generated using the transformation steps with the additional replication step is $\triangleright$-equivalent to C. The set $\mathcal{A}$ at each node is equivalent to that computed in the replicated graph with Σ as the source. The sets



(a) An SCC with four sources

(b) A ▷-equivalent graph

Figure 7.9. An SCC with More Than Two Sources

$\mathcal{E}$ and ▷ of the SCC is equivalent to $\mathcal{E}$ and ▷ at the sentinel node Σ', respectively. $\mathcal{E}$ and ▷ at all duplicated nodes are identical to $\mathcal{E}_{\Sigma'}$ and $\triangleright_{\Sigma'}$ respectively.

Proof.

Consider, without loss of generality, the SCC in Figure 7.9(a) (there may be more than two branches). Using the transformation steps with the additional replication step, a ▷-equivalent graph shown in Figure 7.9(b) is generated. From Lemma 4, $\mathcal{E}_{\Sigma'}$ is equivalent to the $\mathcal{E}$ of the SCC since $\mathcal{E}_{\Sigma'}$ includes all transitions in the SCC and $\mathcal{E}$ from replicated input transitions do not add new transitions to $\mathcal{E}_{\Sigma'}$. Also using a proof similar to that of Lemma 4, $\mathcal{A}$ of each node is equivalent to that computed at the corresponding replicated node. To compute ▷, we consider two different cases: (1) sources that are on the same branch, e.g. Y and Z, and (2) sources that are on different branches, e.g. W and Y. We show that the transformed graph is ▷-equivalent to the original graph for both cases. Case 1 is similar to Lemma 4. In case 2, we must account for the effects of the sources in one branch on the sources in another branch. Let $P = (((A \| J);B) \| K);C$ and $Q = (D \| H);E$ where A to E represent the subgraphs that contain the transitions a to e respectively and H to K represent the subgraphs traversed before entering W to Z respectively. $(I;P) \| (I;Q)$

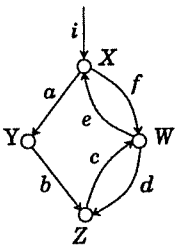
then contains all the input subgraphs to the SCC. Since $\triangleright_{\Sigma'} = \triangleright_{I \| [(I;P)\|(I;Q);P] \| [(I;P)\|(I;Q);Q]}$, then $(I;P) \| (I;Q)$ precedes both P and Q. From Lemma 4 and 5, $\triangleright_{\Sigma'}$ is the $\triangleright$ of the SCC. From Lemma 4, $\mathcal{E}_{Y^o} = \mathcal{E}_{\Sigma'}$ and $\triangleright_{Y^o} = \triangleright_{\Sigma'}$. Similarly, for W^o and Z^o. □

We now discuss how cross and forward edges in SCC's can be handled using the replication rule of transformation step 2′ for SCC's with multiple sources. (We will use "cross edge" here to mean both cross and forward edges.) A cross edge affects the computation of $\triangleright$ in a SCC C only if the cross edge enters a SCC C' within C that does not contain the tail (or its ancestor) of the cross edge. If C' contains the tail (or its ancestor) of the cross edge, the $\triangleright$ of the cross edge does not affect $\triangleright_{C'}$ since by Theorem 3, $\triangleright_{C'}$ is identical for all nodes of C'. If the tail (or its ancestor) of the cross edge is not in C', we replicate C' using the replication rule of transformation step 2′. Back edges within C' are changed to point to the corresponding node in the replica of C' while the back edges of the replica of C' points to the sentinel node Σ. We can omit replicating the part of C' that is a subtree rooted at the head of the cross edge since the computation of $\triangleright$ at that subtree includes the $\triangleright$ of the cross edge. The above transformation step for cross edges is performed after the transformation steps 1 and 2. Using the new graph, we then perform step 2′ and 3.

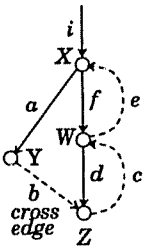
Figure 7.10(a) shows an SCC with a cross edge b (shown in Figure 7.10(b)) that enters a SCC C' involving nodes W and Z (though another spanning tree may be constructed with f as a forward edge, the results are the same). We replicate C' by replicating the node W and setting the back edge c to point to the replicated node W'. The edges d and e are then set to point to Σ. The resulting graph for computing $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ of the SCC is shown in Figure 7.10(c). Note that replicating the entire SCC instead of C' does not resolve the problem of a cross edge because $\mathcal{A}_W$ of that replica still contains f while $\mathcal{A}_{W'}$ in Figure 7.10(c) does not.

7.4. Computing Recovery Paths Automatically

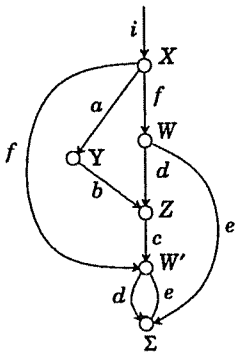
We can compute recovery paths automatically from the restricted product graph (with failure states and recovery transitions) based on the correctness conditions described in Section 7.2. We first describe an algorithm for detecting dependencies between pairs of transitions within a specified scope, using the method described in the previous subsection. For detecting dependencies, we use the graph consisting of only normal transitions and states and ignore recovery transitions and failure states.



(a) An SCC with cross edge



(b) A Spanning Tree



(c) Modified graph for computing $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$

Figure 7.10. An SCC with a Cross Edge

However, for computing recovery path, we include recovery transitions and failure states.

7.4.1. Algorithm for Detecting Dependency

Given a scope, we wish to determine from the restricted product machine graph whether there is a dependency between any two transitions. We proceed by extracting the subgraph defined by the scope using the Algorithm 7.1 described below. Then we transform the subgraph into an acyclic one and apply a diffusion method for computing dependencies at each node using the formulas discussed in the previous sub-section.

As defined in Section 6.3, a scope $[S_i,S_j]$ is a subgraph of the restricted product machine graph with source S_i, sink S_j and free of cycles involving both S_i and S_j. The function *extract_scope* takes three input parameters S_i, S_j and R. The graph G is first set to *NULL* and all nodes in the original graph R are marked "new". The function then uses the procedure *df_search* to derive the subgraph G of the scope $[S_i,S_j]$ by traversing the original graph from S_i in depth-first order. As it traverses the graph, it

Given : Nodes S_i and S_j, and graph R.

Compute : G, the scope in R with source S_i, sink S_j, and free of cycles involving both S_i and S_j.

Method :

```
1:  function extract_scope(Si,Sj,R) {
2:      set G to NULL;
3:      for all node u in R mark u "new";
4:      df_search(Si,Sj,R);
5:      return G;
6:  }

7:  procedure df_search(v,w,R) {
8:      mark v "old";
9:      for each transition <v,x> in R {
10:         if (w is reachable from x) {
11:             add transition <v,x> to G;
12:             if (x is marked "new") {
13:                 mark x "old";
14:                 add x in G;
15:                 df_search (x,w,R);
16:             }
17:         }
18:     }
19: }
```

Algorithm 7.1. Extract subgraph G defined by the scope $[v,w]$

adds all nodes that can reach S_j and all traversed transitions to these nodes. It makes use of a function, *reachable* (x,w), that checks if w is reachable from x. (The compiler of the restricted product machine pre-computes for each node a set of reachable nodes.) Since each node and transition in the scope $[S_i,S_j]$ is traversed only once, the time complexity of *df_search* is $O(n_s + t_s)$, where n_s (t_s) is the number of nodes (transitions) in the scope $[S_i,S_j]$ of the restricted product machine graph. The space complexity is also $O(n_s + t_s)$.

Since a subgraph G of a scope may contain cycles, we need to transform G into an acyclic $\triangleright$-equivalent graph. This algorithm is based on the algorithm for computing strongly connected components described in [4]. The main procedure *transform* performs a depth-first search through G and transforms strongly connected components into acyclic subgraphs as they are found. Strongly connected components are detected using depth-first number k, low link number L and a stack *STACK*

as described in [4]. At each node v, we add two stacks $v.B$ that records all back edges originating at the descendents of v and $v.C$ that records all cross and forward edges originating at the descendents of v. At lines 13 to 16, where $w.k < v.k$, we determine if <v,w> is a back edge or a cross edge by maintaining a current path from the root to v. If w is in the path, $<v,w>$ is a back edge and is pushed into the stack $v.B$. Otherwise, it is a cross edge and is pushed into the stack $v.C$. The current path from the root is omitted from the procedure *transform* for clarity. If $w.k > v.k$ and w is on the *STACK*, then $<v,w>$ is a forward edge and is also pushed into the stack $v.C$. The time complexity of this part of the algorithm is $O(n_s + t_s)$, similar to the algorithm for detecting strongly connected components.

When a strongly connected component is found at line 22, we begin reconstructing it by first creating a sentinel node Σ_v and changing the head of all the back edges to point to Σ_v (line 28). By Theorem 4, the computation of $\triangleright$ at Σ_v is equivalent to that computed using the original back edges (in the absence of cross or forward edges). For each cross or forward edge $<x,y>$, we replicate (in lines 30-35) the largest subgraph with a back edge from a node that is a descendent of the y (or y itself) such that the head of the back edge is not an ancestor of x (or x itself). All incoming edges to this replica are also replicated. Back edges from the descendents of y are then modified to point to the corresponding node in the replicated subgraph. the sentinel node Σ. By Theorem 5, the computation of $\triangleright$ at the sentinel node of the replicated subgraph is equivalent to that in the subgraph with multiple sources where there is an incoming transition (through cross or forward edge) as well as from the root of the spanning tree.

The time complexity of this part (lines 27-35) of the algorithm is $O(t_k + t_g(n_f + t_f))$, where t_k (t_g) is the total number of back edges (cross edges) of all the strongly connected components in the scope and n_f (t_f) is the total number of nodes (transitions) of the subgraph entered by the cross edges. Since n_f is a fraction of n_s and t_k, t_g and t_f are also fractions of t_s, then the worst-case time complexity is $O((n_s + t_s)t_s)$.

The modified acyclic graph of the original strongly connected component is finally replicated in lines 37 and 38 to account for the effects of multiple incoming transitions to the strongly connected component (Theorem 5). The acyclic graph is again replicated in lines 39 and 40 to propagate the computed values of $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ of the SCC to all the nodes in the SCC. The time complexity for this part of the algorithm is $O(n_s + t_s)$. By summing the three parts, the worst-case time complexity of Algorithm 7.2 is $O(n_s+t_s) + O((n_s + t_s)t_s) + O(n_s + t_s) = O((n_s + t_s)t_s)$. Since the space required is proportional to the size of the scope, the space complexity is $O(n_s + t_s)$.

```
1:  procedure transform(v) {
2:      mark v "old"; i ← i + 1; v.k ← i; v.L ← i; push v into STACK;
3:      for each valid transition <v,w> {     /* where w is not marked "removed" */
4:          if w is marked "new" {
5:              transform(w);
6:              v.L ← MIN(v.L,w.L);
7:              v.B ← APPEND(v.B,w.B);
8:              v.C ← APPEND(v.B,w.C);
9:          }
10:         else {
11:             if w.k < v.k and w is on STACK {
12:                 v.L ← MIN(w.k,v.L);
13:                 if IS_BACKEDGE(v,w)
14:                     push <v,w > into v.B;
15:                 else if IS_CROSSEDGE(v,w)
16:                     push <v,w > into v.C;
17:             }
18:             else if w is on STACK
19:                 push <v,w > into v.C;      /* <v,w> is a forward edge */
20:         }
21:     }
22:     if v.L = v.k {
23:         repeat
24:             pop x from top of STACK and mark x "removed";
25:         until x = v;
26:         create a new sentinel node Σ_v;
27:         for each backedge <a,b> in v.B
28:             replace <a,b> with <a,Σ_v>;
29:         for each crossedge <x,y> in v.C {
30:             get a backedge <a,b> in y.B with minimum b.k
31:                 where b is not an ancestor of x;
32:             replicate subgraph with b as source and Σ_v as sink
33:                 and all input transitions to that subgraph;
34:             for each backedge <a,b> in y.B
35:                 replace <a,Σ_v> with <a,b'>;      /* b' is a replica of b */
36:         }
37:         G' ← replicate subgraph from v to Σ_v and all input transitions;
38:         create an ε transition from Σ_v to v';      /* v' is a replica of v in G'*/
39:         G^o ← replicate subgraph from v to Σ_v and all input and output transitions;
40:         create an ε transition from Σ'_v to v^o;      /* v^o is a replica of v in G^o*/
41:         remove all outgoing transitions from the original subgraph;
42:     }
43: }
```

Algorithm 7.2. Transforms a Graph into a $\triangleright$-equivalent Acyclic Graph

The entire algorithm for generating $\triangleright$-equivalent acyclic graph is given in Algorithm 7.3 that uses the procedure *transform*.

Given : A directed graph $G = (V,E)$ and a source S_i;

Compute : A $\triangleright$-equivalent acyclic graph;

Method :

```
1:  procedure make_acyclic(G,S_i) {
2:      i ← 0;
3:      for all node v in G do
4:          mark v "new" and initialize v.B and v.C to empty;
5:      initialize STACK to empty;
6:      transform(S_i);
7:  }
```

Algorithm 7.3. Algorithm for Generating $\triangleright$-equivalent Acyclic Graphs

Given the transformed $\triangleright$-equivalent acyclic graph, we use a centralized diffusion method in Algorithm 7.4 for computing the $\triangleright$ relations between pairs of transitions in the scope. Each node and transition of the $\triangleright$-equivalent graph is visited only once. At each node and transition, the $\triangleright$ relations of pairs of transitions traversed so far and other important information ($\mathcal{A}$ and $\mathcal{E}$) are computed and stored.

The algorithm begins with the source of the scope S_i which has empty $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$. The sets $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ of all outgoing transitions from a node u are computed (at lines 13-16) using the formulas for series merge discussed in Section 7.3. We use the notation $\mathcal{A}_{[u,v]}$ (similarly $\mathcal{E}_{[u,v]}$ and $\triangleright_{[u,v]}$) to denote the value of $\mathcal{A}(\mathcal{E}, \triangleright)$ computed at transition $<u,v>$ for the subgraph from S_i to $<u,v>$. Let n_a (t_a) be the number of nodes (transitions) in the $\triangleright$-equivalent graph. Let the total number of basic transitions in the scope be $t_b = \sum_{i=1}^{N} t_i$, where t_i is the number of transitions of basic machine i in the scope and N is the number of basic machines. Then t_b is of order $t_s^{\frac{1}{N}}$. During series merge, the running time of union operations on sets with the maximum size of t_b is $O(t_b)$. There is only a union operation in each of the series merges $\mathcal{A}_{u;<u,v>}$ and $\mathcal{E}_{u;<u,v>}$ while there are t_b union (or difference) operations in $\triangleright_{u;<u,v>}$. Since each transition is merged once, the worst-case time complexity of the algorithm involving the series merges is $O(t_a t_b) + O(t_a t_b) + O(t_a t_b^2) = O(t_a t_b^2)$.

Given : A subgraph G defined by the scope $[S_i,S_j]$.

Compute : $\triangleright$ at each node and transition of G.

Method :

```
1:  function compute_dependency(G,S_i,S_j) {
2:      copy G into G';
3:      make_acyclic(G',S_i');
4:      for each node u in G' do
5:          u.count ← number of incoming transitions;
6:      initialize Q to contain S_i' only;
7:      while Q is not empty {
8:          remove u from head of Q;
9:          /* do the following for n incoming transition to u */
10:         A_u ← ||_{i=1}^{n} A_[w_i,u];
11:         E_u ← ||_{i=1}^{n} E_[w_i,u];
12:         ▷_u ← ||_{i=1}^{n} ▷_[w_i,u];
13:         for each transition <u,v> in G' {
14:             A_[u,v] ← A_u;<u,v>;
15:             E_[u,v] ← E_u;<u,v>;
16:             ▷_[u,v] ← ▷_u;<u,v>;
17:             decrement(v.count);
18:             if v.count = 0
19:                 insert v at the tail of Q;
20:         }
21:     }
22:     return ▷_S_j';
23: }
```

Algorithm 7.4. Using a Centralized Diffusion Method to Compute $\triangleright$ at Each Node and Transition

When all incoming transitions to a node v have been computed, v is placed at the tail of the queue Q. When v is removed from the head of Q, the parallel merges $\triangleright$, $\mathcal{A}$ and $\mathcal{E}$ of all incoming transitions at v are computed (lines 10-12). The notation $\|_{i=1}^{n} \mathcal{A}_{[w_i,u]}$ is used to mean a parallel merge of $\mathcal{A}_{[w_i,u]}$ at all incoming transitions from w_i to u for i=1 to n where n is the number of incoming transitions. We use the notation $\|_{i=1}^{n} \mathcal{E}_{[w_i,u]}$ and $\|_{i=1}^{n} \triangleright_{[w_i,u]}$ in similar ways. For each pair of transitions, there is only a union (intersection) operation in each parallel merge $\|_{i=1}^{n} \mathcal{A}_{[w_i,u]}$ ($\|_{i=1}^{n} \mathcal{E}_{[w_i,u]}$) while there are two union operations and an intersection operation for each transition in $\|_{i=1}^{n} \triangleright_{[w_i,u]}$. Since each transition is involved in a parallel merge only once, the worst-case time complexity of the algorithm involving the parallel merge is $O(t_a t_b) + O(t_a t_b) + O(t_a t_b^2) = O(t_a t_b^2)$.

The overall worst-case time complexity of the entire Algorithm 7.4, including running time of *make_acylic* analyzed earlier, is then $O(t_a t_b^2) + O((n_s + t_s)t_s) = O(n_s + (t_s + t_b^2)t_s)$. At each node and transition, the space required (omitting *make_acyclic*) for recording the sets $\mathcal{A}$, $\mathcal{E}$ and $\triangleright$ is of order $2t_b + t_b^2$. Then, the space complexity of the algorithm is $O((n_a+t_a)(2t_b + t_b^2)) = O((n_s+t_s)t_b^2)$.

The algorithm is guaranteed to terminate because the $\triangleright$-equivalent subgraph G' of the scope is acyclic and the number of incoming and outgoing transitions at each node is finite. When the algorithm terminates, the set $\triangleright$ at the sink S_j of the scope contains $T\triangleright_G$ for each transition $T \in G$. Each $T\triangleright_G$ is a set of transitions U in G that satisfy the relation $[S_i,S_j]\ T \triangleright U$. That is, each transition U is dependent on T in the scope $[S_i,S_j]$.

7.4.2. Algorithm for Computing Recovery Paths

A recovery path is a sequence of recovery operations that will move the application to a recovery state. Our objective here is to find a correct recovery path when one or more processes fail. The correctness conditions are satisfied by an algorithm that iteratively searches for a recovery path $<S_c,S_r>$ and a path $<S_p,S_r>$ that contains no invalidated transitions. It also ensures that there is a legal path $<S_r,S_f>$. We remark that this algorithm does not require a history log, i.e. an execution sequence. As discussed in Section 6.4, the notion of execution sequence is used only for reasoning about the correctness conditions for recovery.

For a given set of failed processes, all legal recovery paths involving recovery transitions of only the failed processes can be obtained from the restricted product machine. We need only consider recovery paths within the scope $[S_p,S_f]$ since we assume there exists a firewall recovery state S_p. That is, the algorithm will return the recovery state S_p in the worst case.

The basic strategy for computing a correct recovery path is given in Figure 7.11. It first sets the variable U to the set of transitions invalidated by a recovery path (which may not satisfy the dependency condition) for the failed processes. It then finds a new recovery path that recovers all transitions in U. To recover transitions in U, the recovery transitions may use corrective, compensating and other special recovery operations, in addition to reinstate recovery operations. Since the new recovery path may reinstate transitions not in U, all new reinstated transitions must be added to U. Next, U is set to the closure of all transitions dependent on the transitions in U. If the new recovery path covers all

```
U ← transitions invalidated by a partially-correct recovery path for the
        failed processes.
do {
    find a new recovery path R covering U;
    add new transitions invalidated by R to U;
    U ← closure of U ▷;
} until R covers U;
return R;
```

Figure 7.11. Outline of Strategy for Computing a Recovery Path

transitions in U, the algorithm returns the recovery path since it satisfies all three correctness conditions. Otherwise, it repeat the above steps. The algorithm will always terminate since there is a state S_p to which all processes can recover, U increases monotonically and the size of the scope is finite.

We first describe Algorithm 7.5 that finds a recovery path that recovers the given set of invalidated transitions. First, the algorithm traverses the graph G breadth-first from S_i to S_j along only transitions

Given : A point of failure S_c, a scope $[S_i,S_j]$ and its subgraph G with invalidated transitions marked.

Compute : The shortest partially correct recovery path that recovers the *given* invalidated transitions.

Method :

```
1:  function shortest_potential_recovery_path(Sc,Si,Sj,G) {
2:      unmark all states in G;
3:
4:      traverse G breadth-first from Si along non-invalid transitions
5:          until Sj is reached, marking those traversed states;
6:
7:      traverse G breadth-first from Sc along only recovery transitions {
8:          if a marked state Sr is reached
9:              return the recovery path <Sc,Sr>;
10:     }
11: }
```

Algorithm 7.5. Find the Shortest Recovery Path that Recovers a Given Set of Invalidated Transitions

that are not invalidated and mark those states traversed. It then traverses G breadth-first from the point of failure S_c along only recovery transitions. When a marked state is reached, it returns the recovery path. Let t_r (n_r) be the number of recovery transitions (failure states). Let $t_t = t_s + t_r$ and $n_t = n_s + n_r$. Since the algorithm traverses each normal and recovery transition and node only once, the time complexity is $O(n_t + t_t)$. The space required is also $O(n_t + t_t)$.

The algorithm finds the shortest recovery path since the breadth-first traversal along recovery paths is done after the traversal along valid transitions from S_p. Although the recovery path returned by the algorithm recovers all the given invalidated transitions, it may further reinstate additional transitions. It is thus the responsibility of the main program to ensure that any further reinstated transitions are recovered by another recovery path found in the next iteration.

Algorithm 7.6 uses the basic strategy in Figure 7.11 to compute a recovery path that satisfies the three conditions of correctness. First, the algorithm pre-computes the dependency between processes in the scope $[S_p,S_f]$ and store it in $\triangleright_G$. It then unmarks all transitions in the scope. Next, from the given set of failed processes, it finds a recovery path Q for the failed process without satisfying the dependency condition, i.e. it may invalidate the operation executions of other processes. Using the set $\triangleright_G$, it then finds and marks as "invalid" (lines 7-10) all transitions invalidated by each of the transition execution removed by Q. It then finds a new recovery path that recovers all invalidated transitions. The new recovery path itself may removed more transition executions and cause further transition invalidation. The search for another recovery path is repeated until the algorithm finds a recovery path that recovers all invalidated transitions.

Theorem 6.

Algorithm 7.6 returns a correct recovery path that satisfies the correctness conditions.

Proof.

The legality condition is satisfied since the function *shortest_potential_recovery_path* searches only legal recovery paths in the given scope of the restricted product machine. The modified continuity condition is satisfied because within the scope in which the correct S_r is searched, every state may reach the state S_f. The algorithm is guaranteed to exit from the *repeat* loop (lines 12 – 20) for two reasons: (1) every process has at least a recovery path to S_p and (2) the number of transitions in

Given : A set of failed processes, a restricted product machine graph R, a point of failure S_c and a scope $[S_p,S_f]$.

Compute : A recovery path of the group of processes.

Method :

```
1:  G ← extract_scope(S_p,S_f,R);
2:  ⊳_G ← compute_dependency(G,S_p,S_f);
3:  unmark all transitions in G;
4:  Q ← shortest partially correct recovery path to recover only the failed processes;
5:          /* the recovery path need not satisfy the dependency condition and */
6:          /* need only be within the scope [S_p,S_f] */
7:  U ← all transitions removed by Q;   /* U is a set of invalidated transitions */
8:  for each transition t ∈ U
9:      U ← U ∪ t ⊳_G;        /* t⊳_G = {x | t ⊳_G x} */
10: mark all basic transitions in U "invalid";
11:
12: repeat {
13:     R ← shortest_potential_recovery_path(S_c,S_p,S_f,G);
14:     DIFF ← {all transition removed by R} − U;
15:     V ← DIFF;       /* V is a set of new invalidated transitions */
16:     for each transition t ∈ DIFF
17:         V ← V ∪ t⊳_G;
18:     mark all basic transitions in (V − U) "invalid";
19:     U ← U ∪ V;      /* add new transitions invalidated by R to U */
20: } until there is a path from S_p to S_r that contains no invalid transition;
21: return R;
```

Algorithm 7.6. Compute a Recovery Path for the Failure of More than One Process

the scope $[S_p,S_f]$ is finite and the set of invalidated transitions maintained in U is monotonically increasing since the recovery path returned by the function *shortest_potential_recovery_path* in line 13 will always cover the set U from the previous iteration (although it may add more invalidated transitions). The dependency condition is satisfied because on exiting the *repeat* loop, the path $<S_p,S_r>$ does not contain transitions that are invalidated by a recovery transition in $<S_c,S_r>$. □

The running time of the initial part of the algorithm (lines 3-10) in the worst case is of order $t_s + t_b^2$. In the second part of the algorithm (lines 12-20), each normal transition in the scope can be invalidated only once. When new transitions are invalidated in each iteration, the function *shortest_potential_recovery_path* is executed. In the worst case, one transition is invalidated at each

iteration and the time complexity of this part of the algorithm is $O(t_s(n_t+t_t))$. Hence, the worst-case time complexity of the entire algorithm, including that for extracting the scope, computing dependencies, and both parts of this algorithm is $O(n_s + t_s) + O(n_s + (t_s + t_b^2)t_s) + O(t_s) + O(t_b^2) + O(t_s(n_t+t_t))$. The number of basic transitions t_b is of the order $t_s^{\frac{1}{N}}$ (where N is the number of basic machines) and the number of normal transitions t_s (similarly n_s) in the scope is a fraction of t_t (n_t) which includes recovery transitions (failure states) in the scope. Then, the worst-case time complexity is $O((n_t + t_t)t_s)$. By similarly considering each component, the upper bound of the space required by Algorithm 7.6 is $O(n_s+t_s) + O((n_s+t_s)t_b^2) + O(t_s) + O(n_t+t_t)$ which is $O((n_s+t_s)t_b^2)$.

The algorithm does not necessarily find the shortest recovery path, although its heuristics will find the shortest recovery path in a local search region. To find the shortest recovery path would require checking all possible recovery paths that will recover the failed processes and finding one that involves the fewest additional invalidated transitions. This is not done because of the potentially high computational cost.

Chapter 8

Uniform Framework: Recovery

The correctness conditions of recovery, discussed in the previous chapter, can be analyzed from the restricted product machine graph. The analysis is independent of the mechanism that controls the execution of the application. Thus we can separate recovery mechanisms from recovery policies. We will first discuss the mechanisms for recovery management and then describe different recovery policies that may be used without violating the correctness conditions. We also compare our approach with existing recovery techniques.

8.1. Mechanisms

The recovery mechanisms discussed here are part of the mechanisms for controlling consistency during normal execution, recovery and reconfiguration of distributed application as discussed in Section 5.2. We will first describe the basic interface and interaction between the basic machine and the consistency control manager needed to recover from a failure. Then we describe in more detail some important mechanisms and discuss related implementation issues.

8.1.1. Basic Mechanisms for Failure Recovery

When a basic machine detects a failure, it may notify the consistency control manager of the failure immediately by calling `announce` with a failure state as its parameter. However, a process may fail without informing the manager of the failure. When the manager has not received any request from a process for an extended time period, the manager may query the process status by calling the function `query_status`.

If a basic machine announces a failure or returns a failure status to the query, the manager forces the basic machine from an autonomous mode to an exception mode by calling the function `force_mode` with the mode `EXCEPTION`. If a basic machine is unable to respond to the query, the manager then initiates a restart of the basic machine and forces the new basic machine into an exception mode. From the current states of each basic machine monitored by the manager during normal execution, the

manager determines the normal product state just before the failure.

There are two approaches for determining the appropriate failure state: centralized and distributed. In the centralized approach, the manager queries a damage assessor for the failure state that the failed basic machine must enter. In the distributed approach, the basic machine queries the damage assessor and requests a transition to the failure state (which is immediately granted). The advantage of the centralized approach is that the manager can analyze the restricted product machine to make a better selection of an appropriate failure state than the failed basic machine. The advantage of the distributed approach is that the responsibility of determining failure states is distributed among failed basic machines that remain autonomous in deciding the appropriate failure states.

A damage assessor is a process that assesses the extent of the damage due to a failure and determines which recovery operations can repair the damage. (The failure state entered restricts the recovery operations that can be executed.) The assessor may obtain the damage information from a human operator or from an automatic damage assessment system. For example, a damaged workpiece in a factory workstation may be assessed to be repairable and allowed to continue with its normal operation after the repair. The selected failure state would allow the appropriate rework operations. In the worst case, the assessor may give a conservative (default) damage assessment that does not require human intervention: discard all affected computation and all external work that may be damaged.

After the manager has determined the appropriate failure state, it then executes Algorithm 7.6 to compute a correct recovery path. Each basic machine with recovery transitions in the recovery path will then be forced into exception mode. As each basic machine halts, it returns the status `halt` and announces its current state. The manager then calls `mandatory_transition` to force the basic machines through the recovery path.

On receiving a `mandatory_transition` notification from the manager, the basic machine tries to comply and responds by returning the status `OKAY` if the mandatory transition succeeds, and `FAIL` otherwise. If the response is `FAIL`, the manager may attempt alternate recovery paths or report the failure to the system administrator. After restoring the application to a recovery state, the manager resumes normal operation by calling `force_mode` to set all basic machines back to `AUTONOMOUS` mode and then grants permission for outstanding legal transitions. However, pending requests from basic

machines that are forced to recover during the recovery are automatically cancelled.

To illustrate, consider the recovery plan of the dining philosopher problem in Figure 8.1. Suppose philosopher P_1 is currently in state H and has requested permission from the manager to make the transition to state L. Meanwhile, P_2 fails during transition from state L to R. The damage assessor may determine that P_2 be moved to Φ_2, i.e. the failure product state is (H,Φ_2). At that point, all normal transitions are suspended. The manager then instructs P_2 to initiate recovery action to the state H. After P_2 is restored to state H, the manager returns GRANTED to P_1 to move to state L.

8.1.2. Checkpointing Consistent Historical States

As described Section 7.1, we assume the existence of at least one consistent (firewall) historical state S_p. A state S_p can be created using known techniques for checkpointing global states of the processes, either conservatively [44] or optimistically [79]. S_p may move with the normal execution resulting in a reduction in the recovery scope and the length of the recovery path. Conversely, we do not require every product state entered to be checkpointed. There are two approaches to selecting the appropriate product states to be checkpointed: (1) supplied by software designers and (2) automatic analysis of the restricted product graph. We discuss both approaches below.

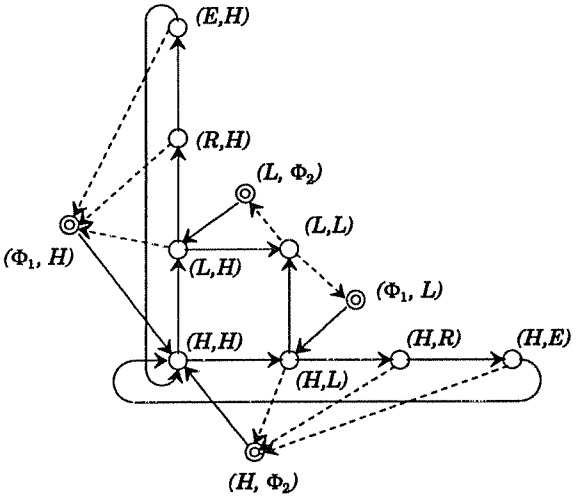


Figure 8.1. Recovery Plan of Two-Dining Philosopher Problem

In the first approach, a software designer may explicitly define the *initial* basic state(s) as checkpoint states when specifying a basic machine. He may also specify some intermediate states in each basic machine as checkpoint states. In a restricted product machine, an initial (checkpoint) product state is one in which all its component basic states are initial (checkpoints). A consistent historical state S_p is either an initial product state or a checkpoint product state. During normal execution, checkpointing is done automatically by each basic machine when it reaches the pre-defined checkpoint state. The manager needs not be involved in checkpointing of states. Software designers may implement different methods for checkpointing each basic machine as long as it can always recover to its checkpointed state. Checkpoint methods include dumping its entire execution state to stable storage and maintaining write-ahead logs of the changes in conjunction with less frequent checkpoints. The advantage of this approach is that checkpointing is performed independently of the manager.

The second approach requires the manager to analyze the restricted product graph to determine the appropriate product states to checkpoint. The advantage of this approach is that more appropriate checkpoint states (that will reduce the number of checkpoints) can be determined by analyzing the restricted product machine. First, we make the restricted product graph acyclic by traversing it breadth-first and avoid visiting any vertex more than once. (Cross and forward edges may exist.) Then we find *dominators* of the restricted product graph. A dominator is a product state that must be entered to reach a set of product states. If the dominated set of product states is large, many paths (to the dominated states) in the restricted product graph would pass through the dominator. When a failure occurs during some execution in those paths, then the dominator can be used as the firewall state. Thus, we can reduce the number of checkpoint states by checkpointing dominators with large set of dominated product states.

When checkpoint states are determined by the manager, the manager must enforce checkpointing by forcing each basic machine to take checkpoint at the appropriate states. When a basic machine requests permission to make a transition, the manager will check if its current state is a checkpoint state. If so, it will call the function `checkpoint` to inform the basic machine to checkpoint its current state. As in the first approach, different checkpointing methods may be defined at each basic machine by a software designer. When the basic machine returns a message indicating a successful checkpoint, the manager continues to serve the previous request of the basic machine.

Cascaded rollback is avoided by specifying appropriate consistent historical states at important execution points. When a failure occurs after a consistent historical state S_p, operations before S_p need not be recovered. While existing approaches [32, 43, 72] to avoiding domino effect of cascaded abort are based on recovery blocks, we analyze the semantics of the applications encoded in the restricted product graph and determine the appropriate consistent historical states, e.g. using dominator states, in which all processes take checkpoints. The advantage of our approach is that cascaded rollback can be avoided even in applications designed with arbitrary interaction that does not require strict block structures.

8.2. Recovery Policies

The conditions for correct recovery defined in Chapter 7 must be satisfied by all types of recovery methods. However, software designers may still select some continuity options. Both backward and forward recoveries are possible. Here we show how these recovery policies are related to the correctness conditions in some traditional recovery techniques.

8.2.1. Backward Recovery

Backward recovery requires that a recovery state be on a path from S_p to S_c, i.e. $S_f = S_c$. Our definition of backward recovery provides opportunities for optimizing recovery. Any equivalent legal path $<S_p, S_r>$ may replace the actual execution sequence. For example, the path containing the fewest invalidated operation executions may be chosen. Since equivalent paths and dependencies between operations can be derived solely from the restricted product machine, detailed history logs need not be maintained,

To illustrate backward recovery involving dependency, we consider a modified example of the manufacturing application described in Section 4.3. The only change we make is to require the conveyor to transport each workpiece stamped by DIE 1 to be stamped again by DIE 2. The region between DIE 1 and DIE 2 acts as a buffer for intermediate workpieces. DIE 2 can start only if there is a workpiece in the buffer. Figure 8.2 shows the basic machine representing the behavior of this buffer. The restricted product machine of DIE 1, DIE 2 and the buffer is shown in Figure 8.3. At the state R_1I_2E, when a new workpiece is placed in the buffer, the application will be moved to the state R_1I_2N by the increment transition i.

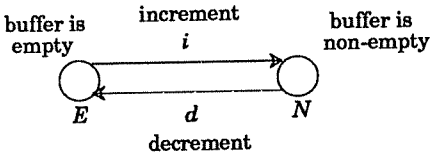


Figure 8.2. Basic Machine of Buffer for Intermediate Workpieces

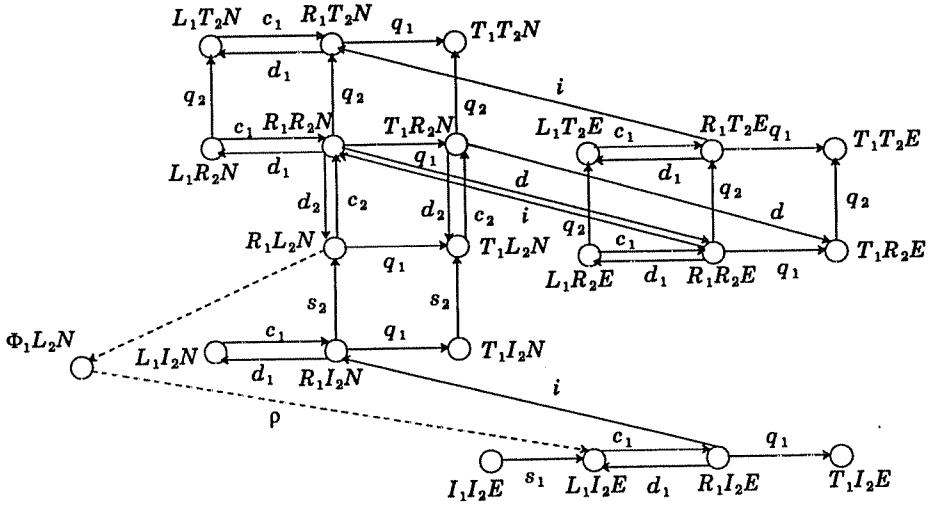


Figure 8.3. Restricted Product Machine of Manufacturing Application

Suppose the buffer is initially empty (I_1I_2E). DIE 1 then completes stamping a workpiece and places it in the buffer (R_1I_2N). DIE 2 then initiates the placement (s_2 and c_2) of the intermediate workpiece between its stops. Suppose the workpiece then drops off the conveyor in transit to STOP 4. This is considered a failure that would invalidate the placement (i.e. by c_1) of the intermediate workpiece on the buffer, requiring c_1 to be erased from the execution sequence. The failure state is Φ_1L_2N. From the restricted product graph we can determine the dependent transitions (in the scope $[I_1I_2E, R_1L_2N]$), i and s_2, which also need to be recovered, i.e. $[I_1I_2E, R_1L_2N]$ $c_1 \triangleright i$ and $[I_1I_2E, R_1L_2N]$ $c_1 \triangleright s_2$. A recovery path $<\Phi_1L_2N, L_1I_2E>$, denoted by ρ, may be composed of three reinstate recovery transitions: the first to the state R_1I_2N, the second to R_1I_2E, and the third to L_1I_2E.

Consider another example where dependency is irrelevant. Let the application be at the state R_1R_2N, when q_1 is first executed followed by d. Then while q_2 is being executed, some external operation cause the effect of q_1 to be destroyed. However, there is a path $<d,q_2,q_1>$ that is equivalent to the actual sequence $<q_1,d,q_2>$ executed. Using this equivalent path, we can then allow q_2 to continue

normally while q_1 is recovered.

8.2.2. Forward Recovery

The continuity condition can be satisfied in two ways. First, the recovery state may be in a path from S_c to S_f, i.e. the recovery sequence may include operations that correct or repair the failure and place the application in a state that makes some forward progress since the failure. Second, the recovery state may be in a path from S_p to S_f (but not S_c), i.e. the recovery transition may represent operations that compensate for some previous operation execution. We provide flexibility in managing recovery by allowing each process to use different types of recovery, e.g. reinstate or non-reinstate operations. As long as the system keeps track of the operations invalidated by reinstate recoveries, it can ensure that recovery is correct.

We illustrate a forward recovery involving non-reinstate recovery by considering the manufacturing application shown in Figure 8.3. Consider a failure that occurs as described in Section 8.2.1. Suppose the transition c_1 has a non-reinstate recovery in which the dropped workpiece is repositioned correctly in the conveyor (by a robot or human). This is represented by a recovery transition that will move the control process of DIE 1 to the state R_1. The transition c_2 may then proceed normally.

8.2.3. Selecting the Recovery Scope

The flexibility to select recovery scope may allow us to reduce the number of dependent operations and recover fewer processes. As discussed in Section 6.3, a larger scope may have fewer dependencies than a small one. We can thus avoid recovering some transitions.

To reduce the amount of recovery needed, we can find and compare correct recovery paths for increasingly larger scopes. The search begins with the scope $[S_p, S_c]$ where S_p is the most recent consistent historical state and S_c is the point of failure. There are two ways in which we can enlarge the scope. We can use increasingly earlier historical states before the current S_p and we can use increasingly more distant future states S_f reachable from S_c. The future states considered are those significant states where useful application-specific work is done. The damage assessor may provide hints about the appropriate future states to use since it has information on the failure state and the necessary

recovery operations of the failed basic machine.

We may iteratively enlarge scope by alternating the direction of expansion. At each iteration, we determine if enlarging the scope has improved the recovery. Using Algorithm 7.6 on different scopes we may automatically select a recovery path that may be either of backward or forward recovery type. Either type of recovery path may contain combinations of both reinstate or non-reinstate recovery operations.

In most situations, the best recovery state (for a given S_f) is one that has the shortest path $<S_r, S_f>$, since the recovery involves the minimum amount of reinstatement of previous execution and achieves the most forward progress. However, other metrics of evaluating recovery state may be used. In manufacturing control, the cost of an operation is proportional with the cost of the external workpiece (or equipment) affected, rather than the computational cost. Consider a failure in a manufacturing process involving damage to a workpiece. A reinstate recovery operation may discard the workpiece and a non-reinstate recovery operation may repair the workpiece. When the cost of the workpiece exceeds the cost of a repair operation, a forward recovery path will be selected.

8.3. Comparison with Other Recovery Techniques

The advantage of our system is that it allows software designers to specify arbitrary synchronization constraints without requiring processes to be serializable or linearizable. Furthermore, recovery can be improved by exploiting permutations and substitutions of operations (Section 6.1) allowed by the behavior of an application specified with partial-order semantics. The disadvantage of our system is the additional cost of finding a correct recovery state from the restricted product graph that has a worst-case running time of $O((n_t + t_t)t_s)$, where t_t (n_t) is the number of normal and recovery transitions (states) in the scope and t_s is the number of normal transitions in the same scope. With an appropriate selection of checkpoint states, the size of the scope can be reduced. Furthermore, applications can be designed using hierarchical FSM structures to reduce the size of the restricted product graph at each level.

We only compare our system with other recovery techniques in the context of finite-state processes. To model applications with infinite state behavior, such as unbounded FIFO channels, would require augmentations to our system where subgraphs of the restricted product machine are incrementally

generated at runtime and recovery paths computed from the subgraphs. However, the FSM model is useful for many distributed applications such as automated manufacturing [23, 67].

In the following, we compare the correctness conditions of existing backward and forward recovery techniques with our correctness conditions. In Section 8.3.1 and 8.3.2, we show that two important existing backward recovery methods [10, 72, 77, 86], satisfy the conditions for correctness of backward recovery. In Section 8.3.3 and 8.3.4, we describe how some existing forward recovery methods satisfy the correctness conditions for recovery.

8.3.1. Restore and Undo

Traditional methods of transactional recovery based on restore and undo [10, 64, 88] satisfy the correctness condition for backward recovery defined in Section 8.2.1. Restore can be represented by a recovery transition that leaves the current basic state and enters a previous basic state that may be of arbitrary distance away. Undo can be represented by a recovery transition that leaves the end basic state of one normal transition and enters the start basic state of that transition. A sequence of undos in the reverse order of the normal operation execution is equivalent in effect to a restore operation. To satisfy the correctness conditions for backward recovery, the system finds a recovery state S_r, whereby the recovery path $<S_c, S_r>$ represent a sequence of undo or restore operations. The legality condition for recovery is satisfied by the enforcement of synchronization constraints. The continuity condition is satisfied since recovery involves only undo and restore that are associated with normal transitions that would reach the point of failure, S_c, from S_r. In transactional systems, restore and undo are performed only on uncommitted transactions. By the isolation property, the intermediate results of a transaction are not visible to other transactions. This implies no operation of a transaction is dependent on an operation of another uncommitted transaction, satisfying the dependence condition.

8.3.2. Atomic Abstract Data Types

Various works [28, 77, 86] have exploited semantic information to increase the level of concurrency among transactions and optimize failure recovery. We will take Weihl's work [86, 87] as a representative one. In [87], applications are implemented in terms of abstract data types and semantics of abstract data types are used to define two notions of commutativity between sequences of transitions. In the following

definitions, the notation $T(s) = \perp$ denotes that the sequence of transitions T is not defined in the state s. Two sequences of transitions, R and S, of a state machine "*commute forward* if, for every state s in which R and S are defined, $R(S(s)) = S(R(s))$ and R(S(s)) $\neq \perp$." On the other hand, "R and S *commute backward* if $R(S(s)) = (S(R(s))$ for every state s." In our model, we can represent an abstract data type as a basic machine. The restricted product machine, generated from these basic machines and the synchronization constraints, determines the operations that can execute concurrently and those that cannot. We represent forward commutativity of R and S, by the existence of two legal paths <s..$R(s)$..$S(R(s))$> and <s..$S(s)$..$R(S(s))$> from every state s with legal outgoing transitions R or S, to a legal state $R(S(s))$ (which is the same product state as $S(R(s))$). Backward commutativity of R and S is represented by the existence of legal paths <s..$R(s)$..$S(R(s))$> and <s..$S(s)$..$R(S(s))$> from every state s that can reach the legal state $R(S(s))$. Suppose R must be recovered. If S commutes with R such that S occurs in a legal path $<S_p, S_r>$, then S is not invalidated and need not be recovered, satisfying the dependence condition. (The legality and continuity conditions are satisfied in the same way as described in Section 8.3.1.)

Our model permits more flexibility in recovery since it considers sequences equivalence with respect to two specific product states and does not require the sequences to be commutable (or equivalent) for *every* state, either with legal outgoing transitions (forward commutativity) or that can reach some destination state (backward commutativity). As a result more commutativity is allowed on a case-by-case basis. Furthermore, as discussed in Section 6.3, it allows the use of equivalent permutations even when the transitions do not commute pair-wise.

8.3.3. Exception Handling

One form of forward recovery is exception handling. Exceptions may be treated as transitions to some failure states. When the system detects a failure, an exception can be raised triggering a procedure that may assess the failure and take corrective actions. For example, when a tool in the machining center breaks, a procedure for handling this failure may take two steps. First, the system determines the extent of the damage on the workpiece. If backward rollback recovery is inappropriate, e.g. operations that involve external side-effects, forward corrective recovery must be used. Second, the system initiates the recovery operations, such as repair operations if possible or rejection of the

workpiece. The legality and continuity conditions are satisfied by the appropriate recovery operations that move the application to a future state. The dependence condition is trivially satisfied.

8.3.4. Compensation

Compensation is a form of forward recovery which uses operations that eliminate or reduce the effects of another (compensated-for) operation which cannot be rolled back. However, operations that are compensated for may have affected other (dependent) operations, e.g. triggered or prevented these operations. These dependent operations must either be similarly compensated or determined to be semantically unaffected by the compensation. For example, in an assembly of workpieces, suppose workpiece X is first placed at one position and workpiece Y must then be placed adjacent to it. If X must later be repositioned (e.g. to compensate for some error), then Y which is dependent on X must also be repositioned. In our model, the system automatically determines the actual dependence between operations and finds the sequence of recovery operations to a consistent state.

A restricted form of compensation is compensating transactions defined in [45]. It is restrictive in the sense that compensating transactions must be serializable with other transactions. Compensating transactions are recovery operations that undo the results established by transactions that have been committed and whose results may have been made visible to other (dependent) transactions. In this discussion, we use the notion of "dependence" defined in [45]: a transaction A is dependent on B if A reads data written by B. The behavior of compensating transactions is guided by three basic constraints: (1) they undo all the effects of the compensated-for committed transactions, (2) all other transactions must be serializable with respect to the compensating transaction, and (3) based on some consistency predicates, they do not affect the results established by the dependent transactions.

We can satisfy the first constraint in our model by requiring that the compensating recovery transition returns to the start basic state of the transition representing the compensated-for transaction. The second constraint is satisfied by specifying serializable synchronization constraints on compensating recovery transitions. In [45], the third constraint is satisfied by ensuring that the compensating and dependent transactions commute (or R-commute with respect to some relation R on database states). We can satisfy the third constraint by specifying the appropriate synchronization constraints on product states and transitions based on some relation R on product states. The relation R defines a set of product

states S which are equivalent with respect to R. In the restricted product FSM, a weaker form of commutativity can be represented by the existence of equivalent paths from a product state to any state in the set S. If dependent transactions happen before the compensated-for transaction, reinstating the compensated-for transactions by the compensating transactions will not invalidate the dependent (by their definition) transactions. We can thus find a recovery state that satisfies the dependence condition for correctness of recovery.

In [45], compensation takes place when no dependent transaction is affected, with respect to some relation R. In contrast, our model enables the system to detect dependent transactions that are not commutable with the compensated-for transaction and force a recovery on those transactions.

Chapter 9

Dynamic Reconfiguration

Informally, a configuration is a set of processes with specific allowed behaviors. Reconfiguration is the process of changing the current configuration to a new one, which may involve changing the set of processes or their allowed interaction. Dynamic reconfiguration is one during which part of the application may continue to execute. The general procedure for dynamic reconfiguration is simple: The system administrator supplies the desired configuration and any constraints on transient behavior that are not implicit in the current or desired configurations. From this, we can automatically generate a sequence of reconfiguration operations.

Some of the challenges are to choose a reconfiguration sequence that obeys all consistency constraints, avoids deadlock and allows old operations to continue when possible. Because of the complex interaction between different possible behaviors and the consistency requirements, it is important to provide tools for managing dynamic reconfiguration that are simple to use. Software designers and system administrators should be able to specify the required change without the need to analyze for inconsistency and synchronization problems themselves.

We have the following design goals to make dynamic reconfiguration mechanisms efficient and general. First, processes should be allowed to interact arbitrarily without unnecessary restrictions. Existing work on dynamic reconfiguration [11, 16, 47] is based on transactions as a model of process interaction. As discussed in Chapter 3 and in [13], transactions restrict the way processes can synchronize among one another. Second, we should not mandate quiescence of all affected processes before dynamic reconfiguration can begin. The approach in [47] requires all affected processes to be in quiescent states before a reconfiguration can begin. Some operations that take a long time to complete will delay a reconfiguration unnecessarily. Furthermore, reconfiguration operations may take some time to complete. Third, a new configuration should be unconstrained by the properties of the old configuration. Bloom pioneered a work [16] dealing only with a special case of the dynamic reconfiguration problem where the new configuration must be a superset of the old configuration. Her legality conditions for dynamic reconfiguration (or replacement) do not allow the removal of any behavior

in the old configuration. In some distributed applications, such as automated manufacturing, the new configuration may be partially or entirely different from the old configuration.

9.1. Motivating Problem

Consider a manufacturing scenario where an urgent demand for a new product requires modifying the manufacturing application of Section 8.2.1. The current configuration of the application consists of a shared conveyor that transports each workpiece to the first pair of stops to be stamped by DIE 1 and later to the second pair of stops to be stamped by DIE 2. The restricted product graph of this configuration is shown in Figure 8.3. The new configuration requires Die 1 to be replaced by a drilling machine. Figure 9.1 shows the basic machine representing the process controlling the drilling machine. In this new configuration, each workpiece is drilled at the first pair of stops and then transported along the conveyor to be stamped by Die 2 at the second pair of stops. The restricted product graph of the new configuration is shown in Figure 9.2.

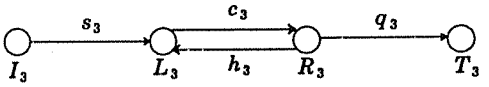


Figure 9.1. Basic Machine Representing the Drill

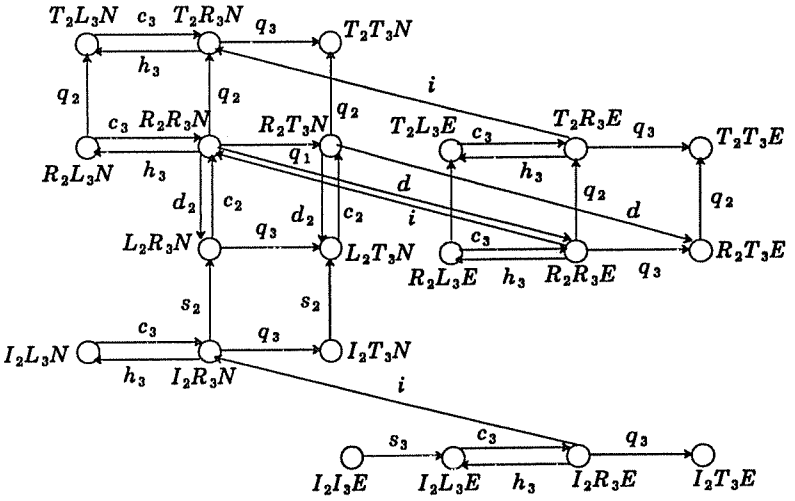


Figure 9.2. Restricted Product Machine of the New Configuration

Our goal is to allow some operations in the old configuration, e.g. stamping by Die 2, to continue while reconfiguration begins, thus improving concurrency during reconfiguration. A major problem is ensuring that the state of DIE 2 is consistent when DIE 1 is being removed and replaced by the drilling machine. For example, removal of DIE 1 may require the removal of some partially stamped workpiece on the conveyor to avoid interference. The removal of workpieces may affect the DIE 2 process if it is in a state ready to stamp on the removed workpieces.

The procedure for dynamic reconfiguration in this manufacturing scenario is as follows. The system administrator specifies the new configuration shown in Figure 9.2. From the current and desired configurations, the system may select a reconfiguration sequence that first removes DIE 1 and then adds the drill. Suppose the removal of DIE 1 requires removal of some workpieces in the buffer. Then DIE 2 may need to be recovered to a state consistent with the modified state of the buffer. Thus an automatic selection of reconfiguration sequence involves checking for consistency in addition to finding a reconfiguration sequence that obeys synchronization constraints. The selected reconfiguration sequence may include recovery operations.

9.2. Model of Reconfiguration

The reconfiguration model is the behavior model of Chapter 4 augmented by special reconfiguration transitions. We represent a *configuration* by the restricted product machine of a set of processes. A *reconfiguration transition* always moves from a product state in one configuration to a product state in another configuration. There are two classes of primitive reconfiguration transitions: (1) *process-changing transitions* that add or remove processes from the current configuration and (2) *constraint-changing transitions* that add or delete normal and recovery transitions (constraints). Process-changing transitions change the dimensions of a restricted product machine, while constraint-changing transitions modify the allowed behavior of a fixed set of processes. A reconfiguration transition that removes a process before it reaches a significant (goal) state also reinstates all transition executions of the removed process. The intermediate results of the removed process should be eliminated since some of its state information may not be made permanent (e.g. through checkpoint) leading to potential internal inconsistency.

Changing an application from one configuration to another may require a set of reconfiguration transitions that is either supplied by a software designer or derived automatically from the current and desired configurations given by the software designer. Every state in the current configuration is the source of a graph sinked at a state in the desired configuration. This graph is a subgraph of the Cartesian product of reconfiguration transitions. (We assume that in practice only a few reconfiguration transitions are required to complete a reconfiguration.) There is a sequence of reconfiguration transitions for each permutation of the transitions.

A reconfiguration sequence maps a state in the current configuration to a state in the new configuration in the following ways. If the sequence contains only constraint-changing transitions that add or delete constraints, then the product state entered (which may be illegal) must be the same as that at the beginning of the sequence. If the sequence contains a transition that adds (or removes) basic machine B, then the product state entered must be the product state at the beginning of the sequence with the state of B added (removed). The state of an added basic machine may be initial or transferred from a removed basic machine. If a basic machine is added (removed), there is a one-to-many (many-to-one) mapping from a product state in the current configuration to states in the new configuration. In the case of adding basic machines, the one-to-many mapping may further be restricted by explicit designer-provided maps. For example, if a state is transferred from a removed process to a replacement, a software designer provides an explicit mapping from states of the removed process to corresponding states of the added process.

During reconfiguration, processes may be added and there may be more concurrent processes than either the current or target configuration contains. These concurrent processes may interfere with one another, although there is no constraint restricting them in either the current or target configuration. (Software designers need not specify constraints on processes that do not execute concurrently.) If interference is possible, the system administrator must supply additional synchronization constraints on their behavior that are not implicit in either the current or target configuration. Furthermore, since reconfiguration transitions move between configurations, additional synchronization constraints on these transitions may need to be specified to prevent interference between reconfiguration transitions and normal transitions. All these additional synchronization constraints described above are called *transient constraints* (Figure 9.3). Transient constraints apply only during dynamic reconfiguration and

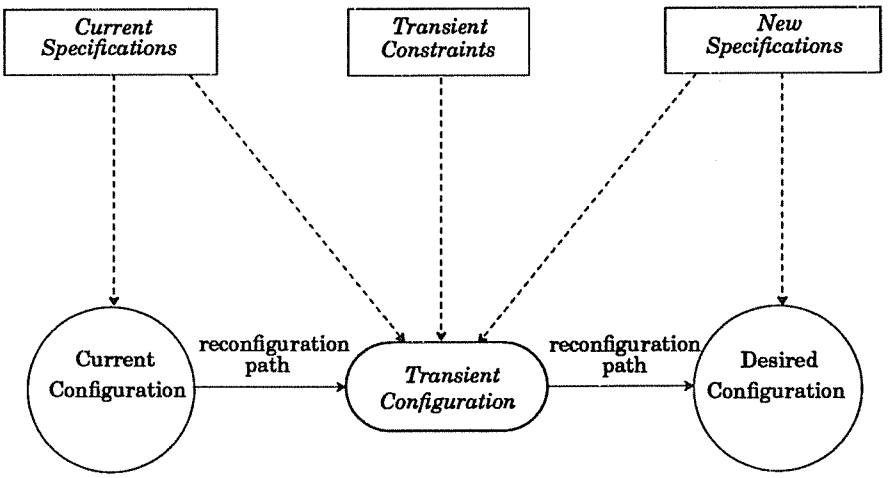


Figure 9.3. Stages of Dynamic Reconfiguration

remove disallowed reconfiguration transitions. For example, in the manufacturing application (Section 9.1), to prevent reconfiguration beginning while DIE 1 is in state L_1, a transient constraint may remove all reconfiguration transitions leaving product states in which DIE 1 is in state L_1.

The introduction of new constraints may conflict with existing ones. A constraint may remove a product state (transition) from a product machine, require a product state (transition) be in the machine, or not care if a product state (transition) is legal or not. Two constraints conflict with each other if one prohibits a product state (transition) while the other requires that feature be legal. Software designers are responsible for ensuring (with the help of a compiler diagnostic checker described in Section 5.1) that constraints in the new configuration are mutually consistent. Furthermore, they are also responsible for ensuring that transient constraints do not conflict with constraints in both the current and new configurations. Reconfiguration from an arbitrary configuration to another completely arbitrary configuration, possibly with conflicting constraints, may be done in several stages through a series of *transient configurations*. Transient constraints may be grouped for each stage to ensure that each group do not conflict with constraints of its adjacent configurations.

To analyze reconfiguration, we define a *reconfiguration graph* as the current and new configurations and the subgraph of reconfiguration transitions between the configurations. Transient constraints remove disallowed transitions from the subgraph of reconfiguration transitions. We also remove all transitions (and nodes) that are not in some path from a legal product state in the current

configuration to a legal state in the new configuration.

A sequence of only reconfiguration transitions may not be adequate to complete a change from one configuration to another. We may have to insert recovery transitions in the sequence to maintain or restore consistency with the changing set of constraints. Such a sequence of mixed recovery and reconfiguration transitions is a *reconfiguration path.* Any reconfiguration from a current to a desired configuration can be done in one reconfiguration path. However, a reconfiguration path may be broken into two consecutive reconfiguration paths.

In this (synchronous) model, normal transitions are allowed only between reconfiguration paths (which may be short). However, the complete alteration of one configuration to another may involve many alternating normal and reconfiguration paths, with transient configurations between reconfiguration paths as Figure 9.4 shows. Normal execution remains within one configuration and reconfiguration paths move between them (using recovery transitions to prepare processes for the next reconfiguration transition). However, normal transitions of processes not affected by a reconfiguration may execute concurrently with the reconfiguration transition. As with concurrent execution of normal transitions (described in Section 4.6), a normal transition may execute concurrently with a reconfiguration transition at a product state S if there is a legal path from S for each ordering of the normal and reconfiguration transitions.

To illustrate, consider the reconfiguration of the manufacturing application described in the previous section. The reconfiguration transitions are "remove DIE 1" and "add drill". The first reconfiguration path may contain the "remove DIE 1" transition (and other recovery transitions), while the second reconfiguration path may contain the "add drill" transitions. The transient configuration

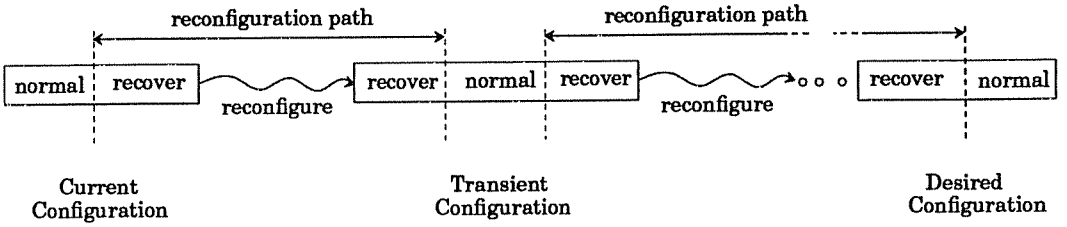


Figure 9.4. Alternating Normal and Reconfiguration Paths

after the removal of DIE 1 is a restricted product machine composed of the basic machine of DIE 2 and the buffer. DIE 2 may continue with its normal operation under the constraints of the transient configuration. A transient constraint may be specified to prevent the drill from being added while DIE 1 is in place.

In the following, we introduce some basic definitions and concepts before discussing how consistency is maintained during reconfiguration.

9.3. Basic Definitions

We define some important product states in the current and new configurations. As in Section 7.1, S_p is a consistent (firewall) historical state. S_c is the last product state entered before a reconfiguration is initiated. If reconfiguration cannot begin at S_c because of some constraints, then some recovery operations are needed to force some processes to specific states where a reconfiguration operation can begin. S_r is a recovery state reached by a recovery path from S_c (Figure 9.5).

A sequence of reconfiguration transitions then moves from S_r (or S_c if recovery operations are not required) in the current configuration to a mapped state S_r' (S_c') in the new configuration. Since reconfiguration transitions change the configuration of the application, other product states, such as S_p and S_c, in the current configuration may also be similarly mapped to S_p' and S_c' in the new configuration.

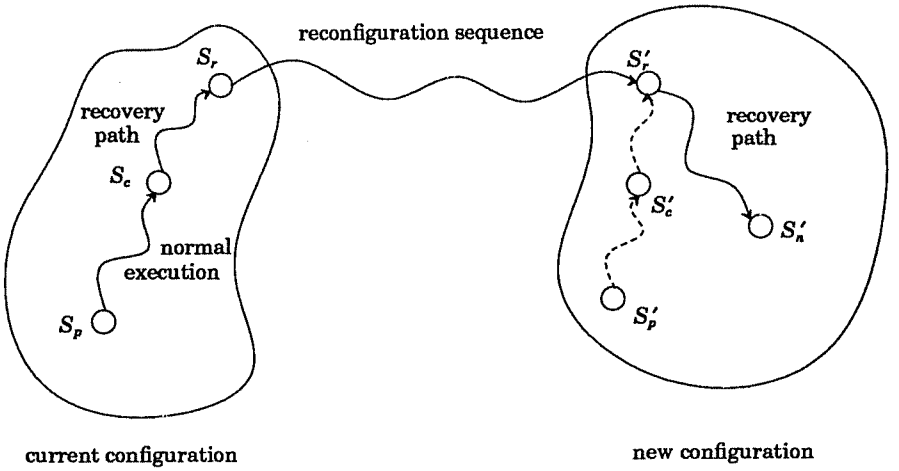


Figure 9.5. A Reconfiguration Path from a Current to a New Configuration

However, the path $<S_p', S_c'>$ (or $<S_c', S_r'>$) may not be legal in the new configuration.

If further recovery operations are needed to restore consistency, then S_n' is the final state reached by a second recovery path from S_r' (or S_c' if no recovery transition is executed in the current configuration). If no recovery operation is required, S_r' (S_c') is the final state reached by the reconfiguration path.

9.4. Correctness of Dynamic Reconfiguration

Reconfiguring directly to S_c' may cause inconsistency in an application. To handle this problem, we modify the correctness conditions for recovery (Section 7.2). The continuity condition is not applicable here because the configuration is changed and the previous behavior of the application need not be preserved. The legality and dependency conditions are modified to account for the configuration change:

(1) (legality condition) There is a legal reconfiguration path from a product state S_c in the current configuration to a product state S_n' in the new configuration.

(2) (dependency condition) There is a legal reconfiguration path from a historical consistent state S_p to the final reconfiguration state S_n' that contains no invalidated transition.

The legality condition ensures that reconfiguration operations obey synchronization constraints. The dependency condition ensures that removal of transition executions by a reconfiguration path also recovers any transition that depends on them. The main modification we make is the definition of transition invalidation (Section 6.4) to account for the dynamic change in the configuration of an application. A reinstated transition executed in the current configuration also invalidates (conservatively) all product transitions in the new configuration that map to that transition. For dependent invalidation, we use the dependence relation of the configuration where the product transitions are located. We may consider dependencies between pairs of transitions in the new configuration as well as in the old configuration.

9.4.1. Conforming to Synchronization Constraints

Reconfiguration operations do not belong to any configuration and are not subjected to constraints in either the current or new configuration; they need only conform to transient constraints. A legal reconfiguration path obeys synchronization and transient constraints since transitions that violate them

are removed from the reconfiguration graph. For example, a transient constraint supplied by a software designer may forbid a reconfiguration transition T to begin at a particular product state S. Then, a transition T leaving the state S is removed and cannot be included in any legal path.

9.4.2. Maintaining Interactive Consistency

The dependency condition ensures consistency is preserved by checking the dependency between operations of processes that may execute in different configurations. The analysis is more complicated than in failure recovery because dependence relations may change across configurations. There are two ways in which dependence relations may change: (1) a dependence relation that exists in the old configuration is removed from the new, and (2) a dependence relation that does not exist in the old configuration is introduced in the new. We will show that the dependency condition will preserve interactive consistency in all cases, although in some cases, it is stricter than necessary.

There are two ways in which a dependence relation can be removed: (a) by removing the synchronization constraints and (b) by removing a set of processes.

When a synchronization constraint is removed, the dependence relations in the new configuration replace those in the current configuration and may be used for determining consistency. An invalidated operation in the current configuration may not be invalidated in the new configuration and need not be recovered. For example, consider a reconfiguration path $<S_c..S_r..S_r'..S_n'>$ and two transitions x and y in a path $<S_p,S_r>$ such that x is reinstated by the recovery path $<S_c,S_r>$. Suppose, in the current configuration, $[S_p,S_r](x \vdash y)$ but in the new configuration $[S_p',S_r'](x \nvdash y)$. If there is a legal sequence of reconfiguration transitions from S_p to S_p', then y is not invalidated in $<S_p..S_p'..S_n'>$ and need not be reinstated. The dependency condition is conservative in that it requires a legal path from S_p to S_p' or that some state S_k in $<S_p,S_r>$ has a legal path to S_k' in the new configuration such that $<S_p,S_k>$ and $<S_k',S_n'>$ do not contain any invalidated transition. (This requirement is conservative because reconfiguration transitions do not change basic machine states and there is a map from each product state in the current configuration to some product state in the new configuration regardless of the existence of a legal path.) An example of removing a synchronization constraint is replacing exclusive access to a shared object with non-exclusive access. An operation, dependent on a reinstated operation that accessed the shared object in the current configuration, may not be required to recover if there is no

dependency in the new configuration.

When a reconfiguration involves removal of processes, two cases are possible: (i) a removed (dependent) process is dependent on another process with reinstated operations and (ii) another process is dependent on the removed (supporting) process that has reinstated operations. (The other process in each case is not removed.) If the dependence relation in the new configuration does not cause operations to be invalidated, consistency need not be restored before removal of dependent processes. Consider a transition y of a removed process P that moves from S_i to S_c and a transition x in a path $<S_p,S_i>$ such that x is reinstated by the recovery path $<S_c,S_r>$. Suppose, in the current configuration $[S_p,S_r](x \triangleright y)$ is true. Since basic states of P are removed from product states of the new configuration, S_i and S_c both map to S_c' in the new configuration. All transitions of P are also removed from the new configuration. If there is a legal sequence of reconfiguration transitions $<S_p,S_p'>$ (or $<S_k,S_k'>$ for some S_k in $<S_p,S_i>$), then there is a legal path $<S_p..S_p'..S_c'..S_r'>$ that does not contain invalidated transitions of P.

When new dependence relations are introduced in the new configuration, the previous execution is unaffected and need not be reinstated if the dependence relation does not exist in the old configuration. For example, transition x is reinstated by the recovery path $<S_c,S_r>$. In the new configuration $[S_p',S_r'](x \triangleright y)$, although in the current configuration $[S_p,S_r](x \not\triangleright y)$. There is thus a legal path $<S_p..S_r..S_r'..S_n'>$ that does not contain invalidated transitions. Appropriate transient constraints may be specified if more control over this situation is desired.

9.5. Computing Reconfiguration Paths and Transient Configurations

The conditions for dynamic reconfiguration enable us to automatically compute correct reconfiguration paths. To permit more concurrency during reconfiguration, reconfiguration paths may also be broken into shorter paths and transient configurations may be generated automatically from these paths.

Software designers or system administrators specify the set of reconfiguration operations required to change the application from the current to a new configuration. Additional synchronization (transient) constraints to be enforced during reconfiguration may also be specified. We adopt this approach in preference to a declarative approach where software designers or system administrators simply specify a new configuration and the system derives the set of reconfiguration transitions automatically. Although

the declarative approach might give better selection of reconfiguration transitions, it incurs substantial cost in analyzing and comparing the current and new configurations.

9.5.1. Algorithm for Computing Reconfiguration Paths

Our objective here is to find a correct reconfiguration path from a graph generated from the current and new configurations, reconfiguration transitions and transient constraints. To compute reconfiguration paths from a current product state, we may need to use recovery transitions to force an application into a state where legal reconfiguration may be started. Although recovery transitions may cause inconsistency in the current configuration, consistency may be restored by further recovery in the new configuration.

First, a new configuration must be generated by applying each of reconfiguration transitions to the current specification using a function called *generate_configuration*. The function scans each of the reconfiguration transitions and performs the operations to add (or remove) a basic machine or add (or remove) a synchronization constraint. Since basic machines are independent, the order of applying these transitions is irrelevant. At the completion of the scan, the result is a new set of basic machines and constraints. A new restricted product machine is then generated by the compiler (Section 5.1).

Next, we generate a reconfiguration graph consisting of the current and new configuration and a subgraph of reconfiguration transitions permitted by transient constraints. We derive a subgraph by taking a Cartesian product of reconfiguration transitions from each legal state in the current restricted product machine to its mapped legal state in the new restricted product machine. Transient constraints are then applied and disallowed reconfiguration transitions are removed. The resulting reconfiguration graph is then used to compute a reconfiguration path.

To compute a legal reconfiguration path that satisfies the conditions for reconfiguration, we modify Algorithm 7.6 with appropriate changes for reconfiguration shown in Algorithm 9.1. The three major differences between the algorithms are as follows. First, two sets of dependency are computed, one for each configuration. Second, there is no failure and recovery operations are used only to move the application to a product state where reconfiguration can begin. Third, the dependency condition is checked on a reconfiguration path that moves from one configuration to another.

Given : A graph H consisting of a current and a new restricted product machine graph R and R' respectively and legal reconfiguration transitions between them; the current state S_c and the scope boundaries S_p and S_f, in R.

Compute : A correct reconfiguration path.

Method :

```
1:  G ← extract_scope(S_p,S_f,R);
2:  G' ← extract_scope(S'_p,S'_f,R');
3:  ⊳_G ← compute_dependency(G,S_p,S_f);
4:  ⊳_G' ← compute_dependency(G',S'_p,S'_f);
5:  unmark all transitions in H;
6:  Q ← Find a legal recovery path <S_c,S_r> such that there is a sequence of
          reconfiguration transitions from S_r to S'_r;
7:  U ← all transition reinstated by Q;   /* U is a set of invalidated transitions */
8:  U' ← U;
9:  for each transition t ∈ U
10:       U ← U ∪ t ⊳_G;        /* t⊳_G = {x | t ⊳_G x} */
11: for each transition t ∈ U'
12:       U' ← U' ∪ t ⊳_G';      /* t⊳_G' = {x | t ⊳_G' x} */
13: mark all basic transitions t ∈ U in G "invalid";
14: mark all basic transitions t ∈ U' in G' "invalid";
15:
16: repeat {
17:       P ← shortest_potential_recovery_path(S'_r,S_p,S'_f,H);
18:       DIFF ← {all transition removed by P} − U';
19:       V ← DIFF;       /* V is a set of new invalidated transitions */
20:       for each transition t ∈ DIFF
21:            V ← V ∪ t⊳_G';
22:       mark all basic transitions in (V − U') "invalid";
23:       U' ← U' ∪ V;       /* add new transitions invalidated by P to U' */
24: } until there is a path <S_p,S'_n> that contains no invalid transition;
25: return <S_c..S_r..S'_r..S'_n>;
```

Algorithm 9.1. Compute a Reconfiguration Path for a Given Set of Reconfiguration Transitions

The scope $[S_p,S_f]$ for searching a reconfiguration path is either supplied by the software designer or pre-defined by the system. We assume there exists some legal reconfiguration from the selected firewall state S_p in the current configuration. In the worst case, the initial product state of the current configuration is used since any reconfiguration can always begin at that state. The mapped states S'_p and S'_f are derived by applying the set of reconfiguration transitions on S_p and S_f respectively.

In lines 1 to 4, a set of dependencies is computed for each configuration and is later used to determine invalidation of transitions executed in each configuration. Unlike Algorithm 7.6, there is no

failure. Instead, the purpose of line 6 is to find a state (which may be S_c) from which there is a legal reconfiguration sequence to the new configuration. Recovery path Q moves the application to the state S_r before reconfiguration begins. In this algorithm, we use the first recovery path that can be found by a breadth-first search from S_c. Consistency can always be restored by a second recovery path in the new configuration since in the worst-case there is always a recovery to S_p'. (As noted above, we may have to choose a suitable S_p.) We will not try to find optimal recovery paths because of the potentially high computational cost. The algorithm then determines all transitions invalidated by Q in each configuration by checking dependencies in that configuration (lines 7 -- 14). The purpose of lines 16 to 24 is to find a second recovery path $<S_r',S_n'>$ in the new configuration such that there is a legal reconfiguration path $<S_p,S_n'>$, from a consistent (firewall) historical state S_p in the current configuration to a final reconfiguration state S_n' in the new configuration with no invalidated transitions.

Theorem 6.

Algorithm 9.1 computes a reconfiguration path that satisfies the correctness conditions for reconfiguration.

Proof.

The legality condition is satisfied since the recovery path $<S_c,S_r>$ contains only legal recovery transitions, the reconfiguration transition sequence $<S_r,S_r'>$ is legal and the function *shortest_potential_recovery_path* searches only legal recovery paths in the new configuration. The algorithm is guaranteed to exit from the loop in lines 16 – 24 for two reasons. First, every process has at least a recovery path to S_p' and there is always a sequence of legal reconfiguration transitions from S_p to S_p'. Second, the number of transitions in the scope $[S_r',S_f']$ is finite and the set of invalidated transitions in U is monotonically increasing. On exiting the loop, the dependency condition is satisfied because there is a reconfiguration path $<S_p,S_n'>$ that does not contain any transition invalidated by either recovery path $<S_c,S_r>$ or $<S_r,S_n'>$. □

The worst-case time complexity for extracting scopes in lines 1 and 2 is $O(n_s + t_s + n_s' + t_s')$, where n_s (t_s) is the number of nodes (transitions) in the scope $[S_p,S_f]$ and n_s' (t_s') is the number of nodes (transitions) in the scope $[S_p',S_f']$. The worst-case time complexity for computing dependencies in both configurations (lines 3 and 4) is $O((n_s + (t_s + t_b^2)t_s) + (n_s' + (t_s' + t_b'^2)t_s'))$, where t_b (t_b') is the number of basic

transitions in the scope $[S_p,S_f]$ ($[S'_p,S'_f]$). The running time for finding the recovery path $<S_c,S_r>$ in the worst case is of order t_t. (Finding an appropriate S_r only requires that there is a legal reconfiguration transition leaving S_r because any transition that does not lead to a legal state in the new configuration is removed from the reconfiguration graph H.) The worst-case running time for invalidating transitions in lines 7 to 14 is of order $t_b^2 + t_b'^2$. Finally, the worst-case time complexity of finding the second recovery path in lines 16 to 24 is $O((n_c + t_c)t'_s)$, where $t_c = t_s + t'_t + t_m$ and $n_c = n_s + n'_t + n_m$ such that t_m (n_m) is the number of reconfiguration transitions (nodes) between the scopes in the current and new configurations. After adding and simplifying, the worst-case time complexity of the Algorithm 9.1 is $O((n_c + t_c)t'_s)$. By similarly considering each component of Algorithm 9.1 individually and adding their space requirements, the upper bound on the total required space is $O(n_s + t_s + n'_s + t'_s) + O((n_s + t_s)t_b^2 + (n'_s + t'_s)t_b'^2) + O(t_t) + O(n_c + t_c)$ which simplifies to $O((n_s + t_s)t_b^2 + (n'_s + t'_s)t_b'^2 + n_m + t_m)$.

9.5.2. Automatic Generation of Transient Configurations

For reconfigurations that take a long time to complete, it may be beneficial to break the reconfiguration path into smaller segments with transient configurations between them. Unaffected processes may execute in a transient configuration while reconfiguration is incomplete. The cost of splitting reconfiguration paths is the additional recovery within transient configurations.

When a reconfiguration path is split, the transient configuration between the two segments should include the following constraints involving any basic machine in the transient configuration: (1) the synchronization constraints from both the current and new configurations, and (2) the transient constraints. We conservatively include all synchronization constraints from both configurations to prevent interference among the processes, although some condition synchronization constraints may be unnecessary if a process involved in those constraints is removed.

Since constraints from different configurations are combined, there may be conflicts among the constraints. Transient configurations with conflicting constraints are illegal and corresponding splits in the reconfiguration path are disallowed.

Consider a reconfiguration path split into two segments. A basic machine must be added in or after the reconfiguration sequence that adds all constraints involving it. (A constraint involves a basic

machine B if the constraint specification contains a state or transition of B.) A basic machine must be removed in or before a sequence that removes any constraints involving it. This guarantees that machines are always properly synchronized. To avoid conflicting constraints in a transient configuration, the algorithm must also ensure that all constraints introduced in the first segments do not conflict with those in the current configuration.

Suppose we need to reconfigure from an arbitrary configuration to another completely arbitrary configuration whose constraints may conflict with those of the first configuration. Since each configuration should not contain conflicting constraints, the conflicting constraints in the current configuration should be removed (in the first segment) before those in the desired configuration are added (in the second segment).

Algorithm 9.2 partitions a set of reconfiguration transitions into two. It takes two arguments: T_R, a

---

Given : A set of reconfiguration transitions T_R and a set of transient constraints K_T.

Compute : Two partitions of reconfiguration sequences T_R.

Method :

procedure partition (T_R, K_T, T_1, T_2) {

(1) If t in T_R adds basic machine m, let C_m be t and all transitions in T_R that add constraints involving m.

(2) Let T_M be the set of transitions in T_R that add constraints which conflict with the current configuration. Repeat until T_M is unchanged:
For all t in T_M, if t adds a constraint involving machine m, add C_m to T_M.

(3) Generate a graph by taking a Cartesian product of transitions in T_R. Remove transitions from the graph that are disallowed by transient constraints in K_T.

(4) Do the following two searches simultaneously (alternately). From the source, traverse forward breadth-first along only transitions *not* in T_M. From the sink, traverse backward breadth-first along *any* transitions. Stop when the two traversals reach a common node X. Set T_1 to a set of transitions traversed from the source to X and set T_2 to a set of transitions traversed backward from the sink to X. If the two traversals stop without meeting at any node, set T_1 to T_R and set T_2 to empty.

}

Algorithm 9.2. Split a Set of Reconfiguration Transitions

set of reconfiguration transitions, and K_T, a set of transient constraints; and generates two sets of reconfiguration transitions T_1 and T_2 such that $T_R = T_1 \cup T_2$ and $T_1 \cap T_2 = \{\}$. Step 3 generates a graph with one source and one sink. By analyzing each step of the algorithm, we then compute the worst-case time complexity to be $O(t_p + t_q) + O((n_o + t_o)t_q) + O(2^{t_p+t_q}) + O(2^{t_p+t_q})$, where t_p (t_q) is the number of process-changing (constraint-changing) reconfiguration transitions and t_o (n_o) is the number of transitions (states) in the current configuration. By simplifying, the worst-case time complexity is $O((n_o + t_o)t_q + 2^{t_p+t_q})$. The upper space bound of the algorithm is $O((n_o + t_o + t_p + t_q) + 2^{t_p+t_q})$. In practice, the total number of reconfiguration transitions, t_p+t_q, is small.

After partitioning T_R into T_1 and T_2, it may still be beneficial to partition T_1 and T_2 further. We use a function *exceed_split_threshold* to determine if the benefit of splitting is greater than the cost of splitting. The default heuristic used is to split if the length of reconfiguration transitions is greater than the length of additional recovery paths required in the transient configuration between the partitions. Software designers may define other heuristics that attach weights (representing computational time or external cost) on reconfiguration and recovery transitions. The function *exceed_split_threshold* will always return false if there is only one process-changing transition in T_R. In Algorithm 9.3, a set of reconfiguration transitions is repeatedly partitioned when it is cost-effective to split. (The list RS is initially empty.) The time complexity of the algorithm is dominated by the running time of Algorithm 9.2 that is executed on successively smaller set of reconfiguration transitions. In the worst case, there are $\log(t_p+t_q)$ iterations and at each iteration, i, Algorithm 9.2 is executed 2^i times using $(t_p + t_q)/2^i$ reconfiguration transitions each time. The worst-case time complexity of Algorithm 9.3 is computed by summing these execution times, i.e.

$$\sum_{i=0}^{\log(t_p+t_q)} 2^i \times O((n_o + t_o)t_q / 2^i + 2^{(t_p+t_q)/2^i}).$$

Since the set of reconfiguration transitions is largest when $i = 0$ and there are $\log(t_p+t_q)$ iterations, we can then approximate this as $O(((n_o + t_o)t_q + 2^{t_p+t_q})\log(t_p + t_q))$. In both Algorithm 9.2 and 9.3, we do not attempt to find an optimal partitioning or list of partitions because the computational cost may be high.

From the list of reconfiguration partitions $T_1, T_2..T_N$, we can generate a sequence of transient configurations. For example, with a set of transitions T_1, transient constraints K_T and the current configuration, a transient configuration is generated using the function *generate_configuration* described

Given : A set of reconfiguration transitions T_R and a set of transient constraints K_T.

Compute : A list of reconfiguration sequences RS.

Method :

```
1:  procedure generate_reconfiguration_list (T_R, K_T, RS) {
2:    if exceed_split_threshold (T_R) {
3:        partition(T_R,K_T,T_1,T_2);
4:        if T_2 is not empty {
5:            generate_reconfiguration_list (T_1,K_T,RS);
6:            generate_reconfiguration_list (T_2,K_T,RS);
7:        }
8:        else
9:            append T_1 to RS;
10:   }
11:   else
12:       append T_R to RS;
13: }
```

Algorithm 9.3. Compute a List of Reconfiguration Partitions for a Given Set of Reconfiguration Transitions

in Section 9.5.1. The complete reconfiguration graph is then generated with sequences of reconfiguration transitions (as permitted by K_T) between the current, transient and new configurations.

The reconfiguration graph that is statically generated by a compiler is then supplied to the consistency control manager. Given the current product state, the manager then computes correct reconfiguration paths between any two configurations using Algorithm 9.1. In each transient configuration, a reconfiguration path is computed at runtime based on the actual product state reached.

Chapter 10

Uniform Framework: Dynamic Reconfiguration

As in failure recovery, correct dynamic reconfiguration is determined by analyzing a graph. The reconfiguration graph is generated from the current and new configurations, the set of reconfiguration transitions and the transient constraints. Since the graph prescribes neither an implementation nor a specific choice of reconfiguration path, we can separate mechanisms and policies from correctness, and from each other. Here we will discuss possible mechanisms and policies in turn, then compare them with other dynamic reconfiguration techniques.

10.1. Reconfiguration Mechanisms

To reconfigure an application, a system administrator supplies a set of reconfiguration transitions and transient constraints to a compiler that generates a reconfiguration graph (described in Section 9.2). The consistency control manager applies Algorithm 9.1 to the reconfiguration graph to compute a reconfiguration path from the current product state. The manager determines the affected basic machines involved in both the partial recovery path in the current configuration and the reconfiguration transitions. It then forces all affected basic machines from autonomous mode to exception mode using the function `force_mode`. When all basic machines have moved to exception mode successfully, the manager calls `mandatory_transition` to force the basic machines through the partial recovery path to a product state where reconfiguration transitions may begin.

Reconfiguration transitions in the reconfiguration path are then executed by a (supervisor) process that controls basic machines and has the privileges for killing and starting basic machines. This role could be assumed by the manager itself. Constraint-changing reconfiguration transitions are used by the compiler to generate the new (or transient) restricted product machine for controlling synchronization. At runtime, these transitions require no further action. A process-changing reconfiguration transition may be implemented by combining several of the following system-provided primitive mechanisms: save (restore) states, and create (remove) and execute basic machines. (Software designers may implement other mechanisms to suit particular applications, e.g. establishing and disconnecting communication

links and saving and restoring undelivered messages.)

(1) *Save state*: The saved state includes the internal state of the process and state information stored on behalf of the process, e.g. opened file descriptors and undelivered messages in each open communication link. The command for saving states is:

```
save (basic_machine_id, state_location);
```

A `basic_machine_id` also contains the execution location. When a process is created, the system assigns it a unique `basic_machine_id`. The saved state will be stored at `state_location` which need not be where the basic machine executes. This command is primarily to record state when one basic machine is to be substituted for another. (It may also be used to save state in a backup processor to enable recovery from a processor failure.)

(2) *Restore state*: If a basic machine is being replaced, the saved state is restored to a new basic machine. States may be restored at the previous execution or another location. The command for restoring state is:

```
restore (basic_machine_id, state_location, transformation_function);
```

A saved state at `state_location` is transferred to the memory image of a new process `basic_machine_id`. (Directory service may be needed to locate saved states of basic machines.) If data structures of the state need to be modified to conform to the implementation of the new basic machine, a `transformation_function` is supplied by the designer.

(3) *Remove basic machine*: Basic machines are terminated, i.e. native process termination primitives. The command is simply:

```
remove (basic_machine_id);
```

The command also removes the basic machine from the restricted product machine used by the manager.

(4) *Create basic machine*: The command for creating basic machines is:

```
basic_machine_id = create ();
```

This command creates a new basic machine in an initial state. If the new basic machine replaces a previously removed basic machine, a `restore` command may later be used to restore previously saved state. The command also generates a new restricted product machine for the manager.

(5) *Execute basic machine*: After a basic machine is created (and possibly a saved state restored), the basic machine is executed using the following command:

```
execute (basic_machine_id);
```

After successfully executing the reconfiguration transitions, a state in a transient or new configuration is reached. The manager then calls `mandatory_transition` to force basic machines through the second recovery path in the transient or new configuration. If additional basic machines are involved in this recovery path, the manager must force them into exception mode first. If the desired configuration is reached, the reconfiguration is complete and the manager calls `force_mode` to force the basic machines back into autonomous mode. If a transient configuration is reached, the manager computes the next reconfiguration path from the current product state in the transient configuration (where other unaffected processes may have moved to new basic states). On computing a new reconfiguration path, the manager again forces the basic machines through the recovery paths and reconfiguration transitions. In transient configurations, we adopt a conservative rule that basic machines in exception mode are not forced back into autonomous mode since they may be required to take more recovery or reconfiguration transitions. However, if it is determined that a basic machine will not be required to take any more of these transitions, then the basic machine may be forced back into autonomous mode where it may then resume normal operation.

10.2. Reconfiguration Policies

Section 9.4 describes the conditions for correctness of all types of reconfiguration that maintain interactive consistency. However, software designers may still select appropriate policies to preserve the semantics of an application. Policies may be selected in the following four areas: (1) a set of reconfiguration transitions, (2) transient constraints, (3) criteria for partitioning reconfiguration transitions, and (4) scope for recovery transitions required for maintaining consistency during reconfiguration.

The policy for selecting a new configuration is the responsibility of either a software designer or application-specific adaptation software. For example, by analyzing a failure, the adaptation software may automatically select a new configuration that avoids the failed component and uses an alternate software or processor [11]. However, here we will not address the problem of selecting a new

configuration from an analysis of failure. Different types of reconfiguration may be specified by designers or adaption software by selecting an appropriate set of reconfiguration transitions. For instance, replacement of a basic machine with another requires reconfiguration transitions that save (restore) process states and create (remove) basic machines. On the other hand, restructuring a group of processes into one with a different behavior may not involve reconfiguration transitions that save or restore process states.

Transient constraints are useful for preventing interference among reconfiguration and normal transitions as well as specifying policies by disallowing reconfiguration at certain product states based on the semantics of the application. For example, a basic machine state may indicate the equipment it controls is in active operation. Reconfiguration at this state should be disallowed. Transient constraints are also useful for defining conventions for adding and removing processes. For example, in the dining philosopher problem, the following convention may be specified to ensure that there are correct number of forks and philosophers in the new configuration: A philosopher can only be removed when it possesses exactly one (left) fork and a philosopher can only be added if it possesses exactly one (left) fork. We can implement this convention by simply specifying transient constraints that only allows a reconfiguration transition to remove (add) a philosopher when the philosopher is (will be) in a state where it possesses a left fork.

The policy for partitioning an end-to-end reconfiguration path from a current to a desired configuration may be determined by how we consider the benefits and costs of splitting. Splitting reconfiguration paths allows greater concurrency among unaffected processes since they may execute at transient configurations between reconfiguration paths. However, splitting a reconfiguration path may require additional recovery operations to restore consistency at the transient configuration. In Algorithm 9.3, the function *exceed_split_threshold* defines the criteria for splitting reconfiguration paths. Those criteria may be re-defined using heuristics based on the application semantics that attach weights to reconfiguration and recovery transitions. For example, a reconfiguration transition requiring an extended period of time to complete may be given a larger weight. A resulting split would then allow some processes to execute while the reconfiguration is in progress.

An appropriate scope may be selected for the recovery operations that bracket a reconfiguration sequence. Policies for selecting a scope are similar to those described in Section 8.2.

Different policies may be selected to implement different types of dynamic reconfiguration. In the following, we discuss three common types of dynamic reconfigurations: replacement, restructuring, and relocation. The three types differ mainly in the selection of the set of reconfiguration transitions and transient constraints. These types of reconfiguration may be used in any combinations. A software designer may implement the failure recovery of a module by replacing it with an alternate module that executes in a different processor. This reconfiguration combines replacement with relocation.

10.2.1. Replacement

Software modules often undergo changes to improve performance, remove bugs, or add new functions. A new version of a basic machine may dynamically replace another basic machine. The conditions for legal dynamic replacement discussed in [16] are as follows. First, the behavior of a replacing basic machine must preserve the behavior of the replaced machine, i.e. the replacing basic machine must allow any sequence of transitions that is allowed by the replaced machine. However, new sequences of transitions may be allowed by the replacing machine. Second, the state S_n' at which the replacing machine begins executing must correspond to the state S_c at which the replaced basic machine stopped, i.e. the set of transition sequences from S_c should be a subset of the set of transition sequences from S_n'. Software designers should provide a mapping from states in the replaced basic to those states in the replacing basic machines with the above properties. In addition to the above policy, our system also allow other policies, such as permitting replacing basic machines to delete some of the function of the replaced basic machine. However, the designer must decide on how such change may be handled. For instance, clients to a modified server with a deleted function may be notified through an error message and must modify their execution accordingly.

Dynamic replacement requires four steps. (Transient constraints may be specified to ensure particular ordering of reconfiguration transitions.) First, a reconfiguration transition may be specified to save the current state of a basic machine. Another reconfiguration transition then kills the replaced basic machine. Next, a basic machine is created and the saved state restored. A final reconfiguration transition executes the new basic machine.

The state of the replaced basic machine that is saved may either be a current, past or future state. To save a past or future state, an appropriate recovery path in the current configuration must be

executed. In a recovery to a past state, some transition executions may be reinstated and transitions dependent on these reinstated executions must be rolled back as well.

10.2.2. Relocation

In relocation, a process that executes in a particular processor is migrated to a different processor. Relocation is a special case of replacement where there is no change in individual or collective behavior. i.e. the current and new configurations are identical. The reconfiguration transitions used are similar to those in replacement: saving a state, halting the basic machine, restoring a state and starting up the basic machine. However, relocation differs from replacement in two ways. First, the replacing basic machine is started up on a different processor which involves a transfer of state information from the existing processor to the target processor. Second, since there is no change in the behavior of the application, reconfiguration may begin at any state and it is not necessary to recover before relocation. No transient constraint is needed to restrict the state in which the relocation may begin.

However, when relocation is used to recover from a processor failure, previously saved states stored in other functional (backup) processors may be used for restoring the state of the relocated basic machine. This case may require recovery to reestablish consistency by selecting an appropriate reconfiguration path.

10.2.3. Restructuring

An application may be restructured by adding (removing) basic machines and modifying the synchronization constraints. Restructuring differs from replacement and relocation in that states of removed basic machines need not be saved. New basic machines are started up in their initial state. However, recovery operations that are necessary before removing a basic machine may reinstate some executions and thus require some recovery operations to maintain consistency. An example of restructuring is the reconfiguration described in Section 9.1 where a die-stamping machine in a manufacturing cell is replaced by a drill. When removing the die-stamping machine some workpiece may need to be removed which may require recovery of the second die-stamping machine.

If a reconfiguration transition changes only synchronization constraints, the resulting new configuration contains the same basic machines but differs in the restricted product graph that controls

their interaction. We may reconfigure without halting the basic machines if the new constraints allow the concurrent executions permitted by the old configuration.

10.3. Comparison with Other Dynamic Reconfiguration Techniques

The two main advantages of our system are: (1) we allow dynamic reconfiguration of applications that may interact in complex ways and (2) we allow greater concurrency, than other systems [16,58], by breaking reconfiguration paths into segments to allow unaffected processes execute at transient configurations between segments. (A reconfiguration path is partitioned only if the cost is outweighed by the benefits of higher concurrency resulting from partitioning.) However, compared to those transactional systems, we incur the additional cost of finding a correct reconfiguration path. Since the cost depends on the size of the scope for reconfiguration and recovery, the cost can be reduced by hierarchical design of an application and the selection of checkpoint states that reduces the size of a scope. In the following, we compare our approach to other techniques only for applications with finite-state processes.

10.3.1. Module Replacement in Argus

Bloom [16] introduced a framework for checking legality of dynamic module replacement based on the transaction mechanisms of Argus [56,57]. Processes interact entirely through the client-server model. The transaction mechanisms provided by Argus preserve interactive consistency. For example, a sequence of transitions representing a replacement is implemented as an atomic transaction. We can implement a replacement transaction by specifying appropriate recovery transitions that allow states before the transaction to be saved and restored if the replacement fails. Recovery from a failure during a reconfiguration is handled in a similar way as failure recovery during normal execution where the recover transitions will restore the application to a saved state in the current configuration. We may impose Bloom's conditions for correctness of replacement described in Section 10.2.1 as follows. The first condition can be satisfied in our system by implementing appropriate policies for selecting new configurations. To preserve the behavior of the replaced module, the state after the reconfiguration must be able to reach a set of future states in the new configuration that map to the future states in the replaced configuration. This mapping is supplied by a software designer. The second condition is satisfied by defining transient constraints that allow only reconfiguration transitions reaching a state in

the new configuration that maps to a state in the replaced configuration.

While our system allows policies restricting new configuration as defined by Bloom, we also allow policies that permit a new configuration that behaves differently from its predecessor. This is useful in manufacturing applications where machine cells may be reconfigured with different behavior for making a new product. Neither the reconfiguration mechanisms nor the application need be implemented as transactions since consistency of states in the new configuration is preserved by analyzing dependencies automatically.

10.3.2. Conic

Dynamic reconfiguration in Conic [47,58] is also based on transactions. It requires each process that may communicate with those being reconfigured to be at a quiescent state before reconfiguration can begin. A quiescent state is one in which no process sends any message or expects a response from a process being removed. In our system, we can impose the policy that only allows reconfiguration when affected processes are at quiescent states by specifying transient constraints that disallow a sequence of reconfiguration transitions to leave a product state S if either or both the following conditions are true: (1) some transition of a removed process must be executed after S as a result of some transition execution of another process before S, and (2) some transitions of another process must be executed after S as a result of some transition execution of a remove process before S. In other words, processes must not be synchronized with the process being removed at the state where reconfiguration begins. The above constraint can be specified using the "necessary precede" relation $\triangleright$. Another way to restrict reconfiguration only at quiescent states is to allow designers to specify sets of states representing quiescent communication behavior. Transient constraints only allow reconfiguration to begin at these states.

The main problem with permitting reconfiguration only at quiescent states is that the system may need to wait some time for processes to reach a quiescent state through normal execution. We can overcome this problem by forcing processes to recover to states where reconfiguration may begin. Recovery of those processes being removed does not waste any computation.

10.3.3. Others

Polylith [69] addressed the problem of capturing and restoring state but ignored the problem of maintaining consistency of those states. They limit the amount of state information that needs to be saved and restored by defining specialized *encode* and *decode* functions. The *encode* function uses application semantics to determine the relevant state information to save, such as stack data structures. Other variables such as loop counters need not be stored because they restrict reconfiguration only at certain states, e.g. those with the same loop counter value. This is similar to the approach taken by Conic of restricting reconfiguration to quiescent states. We can similarly use transient constraints to allow only reconfiguration at states that contain certain variable values. In our system, software designers may use new specialized operations for saving and restoring limited application-specific states in place of the more general commands for saving and restoring states supplied by the system. Software designers must specify the states at which specialized save (restore) operations may begin (end).

Unlike the systems discussed so far, Durra [8] and HPC [25,55] do not deal with the problem of transferring states but rather concentrate on the problem of structural reconfiguration where groups of modules can be recomposed via modification of communication links. In our system, a software designer may use the mechanism for removing and establishing communication links in conjunction with reconfiguration transitions that create and remove basic machines. Changes in abstraction and composition of processes in HPC can be implemented in our system by defining new configurations using the hierarchical finite-state machine model.

Chapter 11

Conclusions

11.1. Summary

We make distributed applications reliable through automatic analyses of their behavior. These applications may contain processes that synchronize in complex ways. We avoid both the restrictive requirement that applications be serializable or atomic and the high cost of message-logging and dataflow analyses. This thesis explored automatic recovery and dynamic reconfiguration in an approach that exploits the semantics of application behavior specified as hierarchical finite-state machines to preserve consistency and reduce wasted computation and external work.

The hierarchical finite-state machine model is an appropriate one for computing recovery and reconfiguration paths automatically. It allows us to use graph-theoretic methods to analyze the partial-order semantics of the application behavior, compute dependencies, and ensure correct recovery and reconfiguration. Because the correctness conditions for recovery – legality, continuity, and dependency – do not require atomicity or total ordering of transitions (or sequences of transitions), we can allow recovery to intermediate states with some operations in progress or incomplete. Extending these conditions to dynamic reconfiguration allows us to compute a consistency-preserving reconfiguration path starting from any current state, not just quiescent states.

By integrating all information for controlling synchronization, recovery and reconfiguration in a single model that does not prescribe an implementation, we designed a common framework with a uniform set of control mechanisms for maintaining consistency. Policies affecting the selection of appropriate recovery and reconfiguration paths may be changed for specific applications without modifying the control mechanisms.

11.2. Contributions

Our primary contribution is a system of automatic recovery management suitable for applications, such as automated manufacturing systems, where conventional approaches using atomic transactions

are unsuitable. Our system allows complex, non-atomic interactions between long-lived, repetitive, or cyclic processes without requiring designer-specified commutativity of operations or a limited range of synchronization and sequencing constraints. Although some approaches also permit complex synchronization constraints, such as message-logging and checkpointing [39, 79], they do not exploit the semantics of applications to improve recovery.

We took interactive consistency, rather than serial execution, as the model for correct execution. This allows recovery and reconfiguration of applications with partial-order semantics: a larger group than serializable or linearizable applications.

As a by-product of this recovery system, we developed a method of extracting dependencies (necessary precedence) between processes automatically from the behavior specification, avoiding reliance on conflict tables or dataflow analyses. Our definition of dependency is based on the properties of the restricted product machine that may contain complex synchronization constraints that cannot be expressed in conflict tables, e.g. condition synchronization and event triggers. We also introduced a notion of equivalence between subgraphs with respect to dependencies and developed a method for computing equivalent acyclic subgraphs.

The conditions for recovery and reconfiguration enable us to improve efficiency by allowing permuting and substituting transitions whenever permitted by the behavior specification. By analyzing specific product states in the (potential) behavior of an application, our method computes more accurate dependencies than other work on transactions [28, 77, 87] which specifies only commutativity of operations for all product states. Using more accurate dependencies reduces the "cascade" effect during recovery, thus reducing wastage of computation.

We presented an algorithm for computing necessary precedence automatically from a restricted product machine. Besides computing accurate dependencies, this algorithm frees software designers from analyzing dependencies and commutativity between operations as in other work [20, 77, 87].

To check for some causes of failure, we presented algorithms for detecting synchronization problems such as deadlock, possible livelock and starvation. We also presented an algorithm for computing recovery paths that preserve consistency. These algorithms have a reasonable worst-case running time, proportional to the square of the size of a recovery scope, that can be made small with

proper design.

Our recovery management system extends to consistent dynamic reconfiguration and allows a graceful transition from a current to a new configuration. During reconfiguration, designer-specified transient constraints control reconfiguration transitions as well as conflicts between processes from both configurations. We presented an algorithm to compute a correct reconfiguration path that preserves consistency. Using a related algorithm, we can break a reconfiguration path into segments to allow greater concurrency of unaffected processes at transient configurations between segments. Our approach does not require processes to be quiescent before reconfiguration as in other systems [58, 69]. Furthermore, we allow applications to interact in complex ways, whereas other systems [16, 58] only permit reconfiguration of applications based on transactions.

11.3. Future Work

There are many areas of future research that will enhance the work presented here, e.g. specification environment, analysis of more synchronization problems, linguistic support and tools, compilation and performance tuning of implementations. However, in the following we discuss future work of greater significance.

In this thesis, we have focused primarily on distributed applications with finite-state behavior that is common in areas such as automated manufacturing. An important future task is to extend this work to systems with infinite-state behavior, such as database systems and communication systems with unbounded FIFO channels. This requires augmentations to our system where such applications can be specified and dependencies can be analyzed in a small subgraph of the restricted product machine. We must investigate how to generate appropriate subgraphs of recovery scopes incrementally at runtime from the specification of basic machines and synchronization constraints. For example, the recovery scope moves with the current execution state. As a new firewall state is entered, the previous subgraph may be discarded and a new subgraph generated. To compute a recovery (or reconfiguration) path, we need only analyze dependencies and preserve behavior continuity in a recovery scope.

Decentralizing the consistency control manager is another problem area. Although the manager as described here is centralized, we can make the control of processes more efficient and robust by decentralizing the manager. To avoid some redundancy of control information and distributing

dependency analysis, a restricted product machine may be factored so that each manager controls basic machines in each factor. We need to investigate how factorization can be computed and how dependencies and recovery (reconfiguration) paths can be computed in parallel.

To improve recovery and reconfiguration further, we may refine the algorithms so that they compute optimal recovery and reconfiguration paths. The main issue is to avoid enumerating all possible paths while selecting one that involves the lowest cost according to some metrics, possibly by pre-processing the graph to allow direct computation of optimal paths.

Other areas that need to be examined more fully are issues related to design of "hierarchical" finite-state machines. First, sharing of basic machines by more than one composite machine is important for capturing sharing of equipment in manufacturing cells. We must investigate how to specify and control sharing of basic machines as well as their effects on recovery and reconfiguration path computation. Second, a hierarchical model might further simplify our consistency control by exploiting property inheritance, delegation and reflection.

Finally, an important long-term future task is the implementation of a general distributed computing environment that supports reliable and dynamically reconfigurable distributed applications based on the work here similarly to the way Camelot (Encina) [78] and Argus [56,57] support reliability in general distributed applications based on transactions.

Appendix A

A Glossary of Terms

The following terms are given specific technical meanings in the thesis. Some terms may be used in a different technical sense in the literature. The section number in parenthesis indicates where the term is defined.

Basic Machine: A finite-state machine representing the behavior of a process. (§4.1)

Basic State: A clean, internally consistent state of a process. (§4.1)

Basic Transition: A procedure or an action, often a physical operation, that moves a basic machine from one basic state to another. Basic transitions take a finite time to complete and are non-atomic. (§4.1)

Basin: The basin of a product state P is the set of product states that *allow* the composite machine to enter P without entering a different composite state. (§4.7.1)

Composite Machine: A finite-state machine representing the behavior of a group of processes. Software designers specify a composite machine by specifying its composite states, composite transitions, and a set of constraints on the basic machines. (§4.1)

Composite State: An arbitrary set of product states specified by a software designer. (§4.1)

Composite Transition: A transition that moves from one composite state to another and may represent a set of concurrent product transitions, a sequence of product transitions, or a combination of both. (§4.1)

Configuration: The specified allowed behavior of a group of processes represented by its restricted product machine. (§9.2)

Consistency Control Manager: A supervisor that controls the scheduling of a composite machine, enforces synchronization control, implements recovery, manages dynamic reconfiguration, and takes the

role of a basic machine when the composite machine is used in a higher-level group. (§5.2)

Execution Sequence: A sequence of transition executions (possibly of several processes) in order of completion. (§6.1)

Failure Basic State: A basic state that is entered when a process fails. Software designers specify the failure basic state entered from a set of normal basic states. (§4.4)

Failure Product State: A product state that contains at least one failure basic state. (§4.4)

Internal Consistency: A condition where a process that manipulate some information, in the absence of other concurrent processes, conforms to user-defined predicates on those information. (§3.1)

Interactive Consistency: A condition where a process may depend on an interaction with other processes only if all processes involved in the interaction are in a state subsequent to the interaction. (§3.1)

Legal State (Transition): A product state (transition) that is either contained in a composite state or a composite transition, and is not explicitly disallowed by a synchronization constraint specified by a software designer. (§4.1)

Mandatory Precedence: The mandatory (or necessary) precedence relation $\triangleright$ is defined as follows: $[S_i,S_j]\, A \triangleright B$ is true if and only if the basic transition B occurs in the scope $[S_i,S_j]$ and a basic transition A always occur before it along every path from S_i to B. (§6.3)

Non-reinstate Recovery: Operations that correct for errors and also operations, such as reset, that simply move a process to another (possibly initial) state. (§4.4)

Path Equivalence: Every path in $<S_o, S_d>$ is equivalent in effect since they all move the application from the original state S_o to the destination state S_d. (§6.1)

Product Machine: A Cartesian product of a set of basic machines executing concurrently in a group. It represents all possible behavior of the group of processes. (§4.1)

Product State: A tuple of concurrent states of the component basic machines. (§4.1)

Product Transition: A transition that moves from a product state to another. (§4.1)

Reconfiguration Path: A sequence of mixed recovery and reconfiguration transitions. (§9.2)

Reconfiguration Transition: A transition that moves from a product state in one configuration to a product state in another configuration. It may include operations that add or remove processes and operations that add or remove synchronization constraints. (§9.2)

Recovery Transition: A transition that moves from a failure state to a normal state. (§4.4)

Recovery State: A normal product state that is entered from a failure product state through a recovery path. (§4.4)

Recovery Path: A sequence of recovery transitions from a failure state to a normal state. (§4.4)

Reinstate Recovery: Operations that eliminate the effect of a transition execution, e.g. restore and undo. (§4.4)

Restricted Product Machine: A product machine that consists of only legal states and transitions. (§4.1)

Transient Configuration: An intermediate configuration between two reconfiguration paths that lead to a final desired configuration. Normal transitions may be executed within a transient configuration. (§9.2)

Transient Constraints: Additional synchronization constraints, specified by a software designer, that prevent interference during reconfiguration among normal and reconfiguration transitions. (§9.2)

Well: The well of a product state P is the set of product states that *force* the composite machine to enter P without entering another composite state. (§4.7.1)

Appendix B

Process-Manager Interface Commands

This appendix summarizes the interface commands for communication between processes and the consistency control manager. There are two types of commands: process commands, those that are used by the processes, and manager commands, those used by the manager. The usage of these commands for controlling synchronization, recovery, and dynamic reconfiguration were discussed in Chapters 5, 8 and 10.

B.1. Process Commands

```
code announce (current_state)
string current_state;
```

A process uses `announce` to identify its current state to the manager. The command `announce` is used only when a process will be idle at a state. When the process intends to make another transition, it instead identifies its current state by calling a command for requesting permission, e.g. `request_permit`. The manager returns the code `OKAY` if the state reported is accepted and `ERROR` if it was found to be inconsistent.

```
permit_code block_request (current_state, destination_state)
string current_state, destination_state;
```

A process may use `block_request` instead of `request_permit` to request permission to make a transition when it is willing to block if the transition is unacceptable. The first string indicates the current state of the process and the second the state to which it intends to proceed. If the current state is inconsistent, the manager returns `STATE_INCONSISTENCY` immediately. If the transition is currently unacceptable, the manager puts the request in a pending list. When the transition later becomes acceptable, the manager notifies the process by returning the code `GRANTED`.

```
permit_code request_permit (current_state, destination_state)
string current_state, destination_state;
```

A process uses request_permit to request permission from the manager to make a transition. The first string indicates the current state of the process and the second the state to which it intends to proceed. If the request is acceptable, the manager returns the code GRANTED otherwise it returns HOLD. If the current state is inconsistent, the manager returns STATE_INCONSISTENCY.

```
code select_path (current_state, path_list, number_of_paths)
string current_state;
path_list_type path_list;
integer number_of_paths;
```

A process may use select_path to specify a list of alternative paths it wants to execute, in order of decreasing preference. The manager returns the index of the first path that contains all transitions that are acceptable, if any. Otherwise, it returns HOLD. Both select_path and select_permit may be used to reduce the messages between a process and the manager.

```
code select_permit (current_state, state_list, number_of_states)
string current_state;
state_list_type state_list;
integer number_of_states;
```

select_permit allows a process to specify a list of alternative states to which it like to move, in order of decreasing preference. The manager returns the index of the first state in the list that is acceptable, if any. If none of the states is acceptable, it returns HOLD.

B.2. Manager Commands

```
code force_mode (process_id, mode)
integer process_id;
mode_type mode;
```

The manager uses force_mode to force a process, identified by process_id, into one of two modes, EXCEPTION and AUTONOMOUS. When a manager forces a process into EXCEPTION mode, there are several possibilities. If the process is executing a transition, it aborts and enters an exception mode at its previous state. If suspended, the process cancels the transition request and enters exception mode at its current state. If it is at a state or has failed, the process enters an exception mode at the current state.

When in an exception mode, a process may not request a transition. After the manager forces it back to AUTONOMOUS mode, e.g. after a successful recovery from failure, the process may then resume requesting transition.

Once it enters an exception (or autonomous) mode, the process responds with a code OKAY. However, it returns FAILED if it cannot enter an exception (or autonomous) mode for some reasons.

```
code mandatory_transition (process_id, destination_state)
integer process_id;
string destination_state;
```

When a process is in an exception mode, the manager may use mandatory_transition to force it to make a mandatory transition. The manager supplies the destination state to which the process must make a transition. If the transition is not defined in the process's legal set of (exception) transitions, it returns ILLEGAL. Otherwise, the process will execute the transition and on successful completion, it returns OKAY. If it fails to make the transition, it returns FAILED.

```
status_code query_status (process_id, timeout)
integer process_id;
timer_value timeout;
```

The manager uses query_status to query the status of a process identified by process_id. The process may return one of the following status:

Status	Description
SUSPENDED	Blocked awaiting a granted transition
IN_TRANSITION	Executing a transition
AT_STATE	At a state in autonomous mode
EXCEPTION	At a state in exception mode
FAILED	Unable to proceed due to an internal failure.

The timeout value indicates the period of time the manager waits for a response for the process. If the timer expires before any response is received, the process is then marked as failed.

References

[1] Aggarwal, S., D. Barbara, and K. Z. Meth, "SPANNER – A Tool for the Specification, Analysis, and Evaluation of Protocols," *IEEE Transactions on Software Engineering* **SE-13**(12) pp. 1218-1237 (December 1987).

[2] Aggarwal, S., D. Barbara, and K. Z. Meth, "A Software Environment for the Specification and Analysis of Problems in Coordination and Concurrency," *IEEE Transactions on Software Engineering* **SE-14**(5) pp. 280-289 (March 1988).

[3] Aggarwal, S., C. Courcoubetis, and P. Wolper, "Adding Liveness Properties to coupled Finite-State Machines," *ACM Transactions on Programming Languages and Systems* **12**(2) pp. 303-339 (April 1990).

[4] Aho, Alfred V., John E. Hopcroft, and Jeffery D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley (1974).

[5] Andler, S., "Predicate Path Expression," *Proceedings of the 6th ACM Symposium on Programming Languages*, pp. 226-236 (January 1979).

[6] Badrinath, B. R. and Krithi Ramamritham, "Semantic-based Concurrency Control: Beyond Commutativity," *Proceedings of the Third International Conference on Data Engineering*, pp. 304-311 (1987).

[7] Bancilhon, Francois, Won Kim, and Henry F. Korth, "A Model of CAD Transactions," *Proceedings of the 11th International Conference on Very Large Data Bases*, pp. 25-33 (1985).

[8] Barbacci, M. R., C. B. Weinstock, and J. M. Wing, "Programming at the Processor-Memory-Switch Level," *Proc. 10th Int. Conf. on Software Engineering*, p. 1928 (April 1988).

[9] Beeri, Catriel, Philip A. Bernstein, and Nathan Goodman, "A Model for Concurrency in Nested Transactions Systems," *Journal of the ACM* **36**(2) pp. 230-269 (April 1989).

[10] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley Publishing Company (1987).

[11] Bihari, Thomas A. and Karsten Schwan, "Dynamic Adaptation of Real-Time Software," *ACM Transactions on Computer Systems* **9**(2) pp. 143-174 (May, 1991).

[12] Birman, Kenneth and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *Proc. 11th ACM Symposium on Operating Systems Principles*, pp. 123-138 (November 1987).

[13] Birman, Kenneth P., "Maintaining Consistency in Distributed Systems," Technical Report TR91-1240, Department of Computer Sciences, Cornell University, Upson Hall, Ithaca, NY 14853 (November 12, 1991).

[14] Birman, Kenneth P. and Thomas A. Joseph, "Exploiting Replication in Distributed Systems," pp. 319-367 in *Distributed Systems, Sape Mullender (ed.)*, ACM Press, Addison-Wesley Publishing Company (1989).

[15] Black, Andrew P., "Understanding Transactions in the Operating System Context," *Operating Systems Review* **25**(1) pp. 73-76 (January 1991).

[16] Bloom, T., "Dynamic Module Replacement in a Distributed System," Technical Report MIT/LCS/TR-303, MIT Laboratory for Computer Science (March 1983).

[17] Campbell, R. H. and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," pp. 89-102 in *Lecture Notes in Computer Science*, Springer-Verlag (1974).

[18] Chandy, K. Mani and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems* **3**(1) pp. 63-75 (February 1985).

[19] Choobineh, Fred and Rajan Suri, (eds.), *Flexible Manufacturing Systems: Current Issues and Models,* Industrial Engineering and Manufacturing Press, Institute of Industrial Engineers, Atlanta, Georgia (1986).

[20] Chrysanthis, Panayiotis K. and Krithi Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," *Proceedings of the ACM-SIGMOD Conference on Management of Data,* pp. 194-203 (May 1990).

[21] Clarke, E. M., E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems* **8**(2) pp. 244-263 (April 1986).

[22] Detlefs, David L., Maurice P. Herlihy, and Jeannette M. Wing, "Inheritence of Synchronization and Recovery Properties in Avalon/c++," *Computer* **21**(12) pp. 57-69 (December 1988).

[23] Duffie, Neil A., Ramesh Chitturi, and Jong-I Mou, "Fault-tolerant Heterachical Control of Heterogeneous Manufacturing System Entities," *Journal of Manufacturing Systems* **7**(4) pp. 315-328 (1988).

[24] Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM* **19**(11) pp. 624-633 (November 1976).

[25] Friedberg, Stuart Arthur, "Hierarchical Process Composition: Dynamic Maintenance of Structure in a Distributed Environment," Technical Report 294, Ph.D. Thesis, Department of Computer Sciences, University of Rochester (1988).

[26] Fussell, Paul A., P. K. Wright, and David Bourne, "A Design of a Controller as a Component of a Robotic Manufacturing System," *Journal of Manufacturing Systems* **3**(1) pp. 1-11 (1984).

[27] Garcia-Molina, Hector, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem, "Coordinating Multi-Transaction Activities," CS-TR-247-90, Computer Sciences Department, Princeton University (February 1990).

[28] Garcia-Molina, Hector and K. Salem, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems* **8**(2) pp. 186-213 (June 1983).

[29] Gouda, M. G. and C. K. Chang, "A Technique for Proving Liveness of Communicating Finite State Machines with Examples," *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* pp. 38-49 (August 1984).

[30] Gouda, M. G., C. H. Chow, and S. S. Lam, "On The Decidability of Livelock Detection in Networks of Communicating Finite State Machines," *Proceedings of IFIP Workshop on Protocol Specification, Testing, and Verification* **IV** pp. 47-56 Elsevier Science Publishers B. V., (1985).

[31] Hadzilacos, Vassos, "A Theory of Reliability in Database Systems," *Journal of the ACM* **35**(1) pp. 121-145 (January 1988).

[32] Haerder, Theo and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *Computing Surveys* **15**(4) pp. 287-317 (December 1983).

[33] Hailpern, Brent and Gail E. Kaiser, "Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data," *Proc. 11th International Conference on Distributed Computing Systems,* pp. 73-80 (May, 1991).

[34] Haskin, Roger, Yoni Malachi, Wayne Sawdon, and Gregory Chan, "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems* **6**(1) pp. 82-108 (February 1988).

[35] Herlihy, Maurice, "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Transactions on Database Systems* **15**(1) pp. 96-124 (March 1990).

[36] Herlihy, Maurice P. and Jeannette M. Wing, "Axioms for Concurrent Objects," *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pp. 12-26 (January 1987).

[37] Herlihy, Maurice P. and Jeannette M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems* **12**(3) pp. 463-492 (July 1990).

[38] Horman, Kluas, "Programming of the Robot Cell," in *Robot Technology and Applications, edited by Ulrich Rembold*, Marcel Dekker, Inc., New York, New York (1990).

[39] Johnson, David B. and Willy Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pp. 171-181 (August 15-17, 1988).

[40] Jones, Anita K. and William A. Wulf, "Towards the Design of Secure Systems," *Software – Practice and Experience* **5** pp. 321-336 (1975).

[41] Jones, Albert T. and Charles R. McLean, "A Proposed Hierarchical Control Model for Automated Manufacturing Systems," *Journal of Manufacturing Systems* **5**(1) pp. 15-25 (1986).

[42] Kim, K. H., "Approach to Mechanization of the Conversation Scheme Based on Monitor," *IEEE Transactions on Software Engineering* **SE-8**(3) pp. 189-197 (May 1982).

[43] Kim, K. H., "Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation," *IEEE Transactions on Software Engineering* 14(6) pp. 810-821 (June 1988).

[44] Koo, Richard and Sam Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering* **SE-13**(1) pp. 23-31 (January 1987).

[45] Korth, Henry F., Eliezer Levy, and Abraham Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 95-106 (August 1990).

[46] Korth, Henry F. and Gregory D. Speegle, "Formal Model of Correctness Without Serializability," *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pp. 379-386 (September 1988).

[47] Kramer, Jeff and Jeff Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering* **16**(11) pp. 1293-1306 (November 1990).

[48] Kramer, J. and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transaction on Software Engineering* **SE-11**(4) pp. 424-436 (April 1985).

[49] Kung, H. T. and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* **6**(2) pp. 213-226 (June 1981).

[50] Lam, Simon S. and A. Udaya Shankar, "A Relational Notation for State Transition Systems," *IEEE Transaction on Software Engineering* **16**(7)(July 1990).

[51] Lamport, Leslie, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers* **C-28**(9) pp. 690-691 (September 1979).

[52] Lamport, Leslie, "Specifying Concurrent Program Modules," *ACM Transactions on Programming Languages and Systems* **5**(2) pp. 190-222 (April 1983).

[53] Lamport, Leslie, Robert Shostak, and Marshall Pease, "The Byzantine Generals Problem," *ACM Transaction on Programming Languages and Systems* 4(3) pp. 382-401 (July 1982).

[54] Lampson, Butler W., "Protection," *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pp. 437-443 Princeton University,, (March 1971). Also in ACM Operating Systems Review, 8, 1, January 1974, p. 18-24

[55] LeBlanc, T. J. and S. A. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," *Proc. 5th Int. Conf. on Distributed Computing Systems*, pp. 26-34 (May 1985).

[56] Liskov, B., D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," *Proceedings of the 11th Symposium on Operating Systems*, pp. 111-122 (November 1987).

[57] Liskov, B. and R. W. Scheifler, "Guardians and actions: Linguistic Support for Robust , Distributed Programs," *ACM Transaction on Programming Languages and Systems* 5(3) pp. 381-404 (July 1983).

[58] Magee, J., J. Kramer, and M. Sloman, "Constructing Distributed Systems in Conic," *IEEE Transactions on Software Engineering* **15**(6) pp. 663-675 (June 1989).

[59] Manna, Z. and A. Pnueli, "Verification of Concurrent Programs: A Temporal Proof System," pp. 163-255 in *Foundations of Computer Science IV, Distributed Systems: Part 2 (J.W. DeBakker and J. Van Leuwen, eds.)*, Center for Mathematics and Computer Science (CWI), Amsterdam (1983).

[60] Marzullo, Keith, Robert Cooper, Mark Wood, and Kenneth Birman, "Tools for Distributed Application Management ," TR 90-1136, Department of Computer Science, Cornell University (July 1990).

[61] Mili, Ali, "Towards a Theory of Forward Error Recovery," *IEEE Transactions on Software Engineering* **SE-11**(8) pp. 735-748 (August 1985).

[62] Milner, Robin, *A Calculus of Communicating Systems*, Springer Verlay, New York (1980).

[63] Moss, J. Eliot B., "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, MIT (April 1981).

[64] Moss, J. E. B., N. D. Griffeth, and M. H. Graham, "Abstraction in Recovery Management ," *Proceedings of the 1986 ACM SIGMOD Conference*, pp. 72-83 (May 28-30, 1986).

[65] Naylor, A. and R. A. Volz, "Design of Integrated Manufacturing System Control Software ," *IEEE Transaction on Systems, Man, and Cybernetics* **SMC-17**(6) pp. 881-897 (November/December 1987).

[66] Papadimitriou, Christos H., "The Serializability of Concurrent Database Updates," *Journal of the Association for Computing Machinery* **26**(4) pp. 631-653 (October 1979).

[67] Pathak, Dhiraj K. and Bruce H. Krogh, "Concurrent Operation Specification Language, COSL, for Low-Level Manufacturing Control," *Computer in Industry* **12**(2) pp. 107-122 (May 1989).

[68] Peterson, James L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall (1981).

[69] Purtilo, James M. and Christine R. Hofmeiser, "Dynamic Reconfiguration of Distributed Programs," *Proc. 11th International Conference on Distributed Computing Systems*, pp. 560-571 (May, 1991).

[70] Ramadge, Peter J. G. and W. Murray Wonham, "The Control of Discrete Event Systems," *Proceedings of the IEEE* **77**(1) pp. 81-98 (January 1989).

[71] Randell, Brian, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

[72] Randell, B., P. A. Lee, and P. C. Treleavan, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

[73] Ranky, Paul G., *Computer Integrated Manufacturing,* Prentice-Hall International, UK, Ltd., Great Britain (1986).

[74] Rodeheffer, Thomas L. and Michael D. Schroeder, "Automatic Reconfiguration in Autonet," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 183-197 (October 13-16, 1991).

[75] Schneider, Fred B., "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys* **22**(4) pp. 299-319 (December 1990).

[76] Schoeffler, James D., "Distributed Computer Systems for Industrial Process Control," *IEEE Computer* **17**(2) pp. 11-18 (February 1984).

[77] Schwarz, Peter M. and Alfred Z. Spector, "Synchronizing Shared Abstract Types," *ACM Transactions on Computer Systems* **2**(3) pp. 223-250 (August 1984).

[78] Spector, Alfred Z., Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Bloch, "Camelot: A Distributed Transaction Facility for Mach amd the Internet – An Interim Report," Technical Report CMU-CS-87-129 , Department of Computer Science, Carnegie Mellon University (June 17, 1987).

[79] Strom, Robert E. and Shuala Yemini, "Optimistic Recovery in Distributed Systems," *ACM Transactions on Computer Systems* **3**(3) pp. 204-226 (August 1985).

[80] Strong, Ray, "Interactive Consistency," pp. 125-137 in *Dependability of Resilient Computers, T. Anderson (ed.)*, BSP Professional Book (1989).

[81] Tarjan, Robert, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.* 1(2) pp. 146-160 (June 1972).

[82] Twidle, K. and M. Sloman, "Domain Based Configuration and Name Management for Distributed Systems," *Proc. Workshop on Future Trends of Distributed Computing Systems in 1990's*, IEEE Computer Society, (September 1988).

[83] Wachter, Helmut, "ConTract: A Means for Improving Reliability in Distributed Computing," *Proceedings of COMPCON Spring 1991*, (February 1991).

[84] Walker, D. J., "Automated Analysis of Mutual Exclusion Algorithms using CCS," ECS-LFCS-89-91, Department of Computer Science, University of Edinburgh (August 1989).

[85] Weck, M. and E. Kohen, "Configurable Control Software for FMS," pp. 437-445 in *Software for Discrete Manufacturing: Proceedings of the 6th International IFIP/IFAC Conference of Software for Discrete Manufacturing,* , Paris, France (June 1985).

[86] Weihl, William E., "Specification and Implementation of Atomic Data Types," Ph.D. dissertation, MIT/LCS/TR-314, Massachusetts Institute of Technology, Cambridge (March 1984).

[87] Weihl, William E., "Commutativity-based Concurrency Control for Abstract Data Types," *IEEE Transactions on Computers* **C-37**(12) pp. 1488-1505 (December 1988).

[88] Weihl, William E., "The Impact of Recovery on Concurrency Control," *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 259-269 (March 29-31, 1989).