This is to certify that the
dissertation entitled

ASSURANCE OF ADAPTATION IN DISTRIBUTED SYSTEMS

presented by

Karunkumar N. Biyani

has been accepted towards fulfillment
of the requirements for the

Doctoral degree in Computer Science and Engineering

g S Kulkarni

Major Professor's Signature

12/03/07

Date

ASSU

ASSURANCE OF ADAPTATION IN DISTRIBUTED SYSTEMS

By

Karunkumar N. Biyani

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2007

ABSTRACT

ASSURANCE OF ADAPTATION IN DISTRIBUTED SYSTEMS

By

Karunkumar N. Biyani

Software systems need to adapt due to changing requirements or changing environment conditions. For long-running and safety-critical applications it is highly desirable to adapt the system without completely stopping the system. In the case of distributed systems, adaptation often requires changes to multiple processes. Typically, such adaptation is performed by dynamically adding or removing components from multiple processes. As a result, during adaptation, the system may consist of both changed and unchanged processes, causing the *old* and the *new* components to overlap. This overlapping of components during adaptation may induce cross-component communication, which may lead to unpredictable and/or undesirable behavior during adaptation.

In order to gain confidence in adaptation in distributed systems, in this disseration, we address the assurance requirements at various stages of adaptation development: (i) modeling and verification of adaptation, (ii) testing of adaptation, (iii) design of components involved in adaptation, and (iv) design of a framework that supports adaptation.

In this dissertation, we describe an approach based on *adaptation lattices* to model and verify adaptation. Specifically, we present *transitional-invariant lattices* and *transitional-faultspan lattices* to verify the correctness of adaptation in absence and presence of faults, respectively.

Furtherm

namely, *mix*

changed proc

time and com

lenges involve

approach can

We also d

how the existi

can be extend

We also d

One aspect o

not only the

The design

the compon

integrates t

is separate

Furthermore, we identify the issues that arise in an important class of adaptation, namely, *mixed-mode adaptation*. Mixed-mode adaptation allows the changed and the unchanged processes to interact during adaptation, thereby, minimizing service interruption time and communication overhead. In this dissertation, we identify and address the challenges involved in mixed-mode adaptation. Specifically, we show how the adaptation lattice approach can be used in the case of mixed-mode adaptation.

We also discuss an approach for testing adaptation in distributed systems. We show how the existing approaches based on predicate detection for testing distributed systems can be extended for testing adaptation.

We also describe *component family* design to build a library of adaptive components. One aspect of the design is to build an *adaptation-verified* library of components in which not only the components but also the adaptations between the components are verified. The design applies the principle of separation of concerns to separate *adapt-active* parts of the components from their core functionality. Furthermore, the component family design integrates the framework that performs adaptation while ensuring that the adaptation logic is separate from the core functionality of components as well as the application.

To my parents and wife for their love and sacrifices

First of al
motivation an
and guidance
this dissertati
with me and u
I want to t
Dr. Philip K
comments an
Science and
want to thank
is truly a wo
Linda Moore
Computer Sc
I want to
tory. Specifi
hesh) Arumu
Ji Zhang and
and support.
father Mr. N
Rathi and Ri
loving wife K

ACKNOWLEDGMENTS

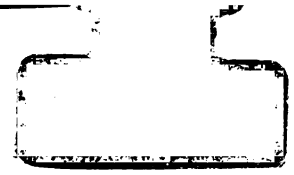
First of all I want to thank my loving Lord, Krishna, and my teacher for the inspiration, motivation and support to pursue things the right way. I am very thankful to my advisor and guidance committee chairperson, Dr. Sandeep S. Kulkarni, without whose guidance this dissertation would have never completed. I specifically thank him for being patient with me and understanding my limitations. He is a wonderful person and a great mentor.

I want to thank the other members of my guidance committee, Dr. Betty H. C. Cheng, Dr. Philip K. McKinley, Dr. Laura Dillon and Dr. Jonathan I. Hall, for their helpful comments and advise over the years. I also want to thank the faculty of the Computer Science and Engineering Department at the Michigan State University. In particular, I want to thank Dr. Abdol Esfahanian, whose teachings had a great influence on me. He is truly a wonderful teacher, who makes learning easy and fun. I also want to thank Mrs. Linda Moore, Mrs. Debbie Kruch, Mrs. Norma Teague and the rest of the staff of the Computer Science and Engineering Department.

I want to thank my colleagues in Software Engineering and Network Systems Laboratory. Specifically, I want to thank: Bruhadeshwar (Bru) Bezawada, Umamaheswaran (Mahesh) Arumugam, Ali Ebnenasir, Borzoo Bonakdarpour, Farshad Samimi, Masoud Sadjadi, Ji Zhang and Limin Wang. I want to thank my friends and family for their encouragement and support. No words of gratitude can ever sum up the contribution of my family: my father Mr. Nandkishore Biyani, my mother Mrs. Santosh Biyani, and my sisters Kavita Rathi and Rinku Malpani, towards my life. And last but not least, my special thanks to my loving wife Kavita Biyani for all her love and sacrifices.

TABLE OF CONTENTS

LIST OF FIGURES



LIST OF TABLES

Chapter 1

Introduction

Evolution is vital in all software systems and has been studied for decades in the context of software maintenance and software upgrade. It is a well known fact in software engineering that programs undergo several changes during their lifetimes. These changes are usually performed for fixing bugs or adding new functionality that the users of the systems did not anticipate initially at the time of original specifications.

Furthermore, software systems have become more and more pervasive with easy access to personal computers, smartphones, and cellular and wireless networks. The challenges in building today's software systems include providing seamless service to user requirements in such heterogeneous operating conditions as: device failures, transitions across wired and wireless environments, high packet loss in wireless networks, byzantine behaviors, and security attacks. In order to meet these challenges, software systems must be able to adapt to environment conditions that may not be foreseen at the time of software development.

In general, software systems need to adapt (change) in response to one or more of the following reasons: (i) discovery of errors/bugs, (ii) change in requirements, and (iii)

change in environment. In a traditional approach, the change is usually performed by stopping the currently running program and then installing the new version of the program. However, stopping the system to perform the change is undesirable for a variety of reasons, such as: (i) it may be simply inconvenient for users; (ii) it may lead to monetary loss, for example, in the case of banking and e-commerce systems; or (iii) it may be unsafe, for example, in the case of safety critical systems such as air traffic control systems. Moreover, in systems that need to adapt in response to frequent or transient changes in environment, stopping the system for upgrade may not even be an option. Clearly, in all these cases, it is highly desirable to perform the change while the system continues to operate. This type of change is commonly referred to as *dynamic adaptation*. A software system that supports dynamic adaptation is known as *adaptive software*.

There has been a growing interest in building adaptive softwares. An increasing number of systems are now being developed with some built-in mechanisms for adaptation that allow for change to be done without completely stopping the system. Adaptive software techniques (*e.g.*, [1–10]) allow the system to modify its own functional or non-functional behavior (*e.g.*, its fault-tolerance, quality of service or security requirements). These modifications include reconfiguration of some parameters, or addition, removal, or replacement of application code. A survey in [11] presents various tools and techniques in building adaptive software. Additionally, Buckley *et al.* [12] has given a taxonomy in the context of *when*, *how*, *where*, and *what*, of software change.

Numerous works in adaptation have either focused on single-process systems or in distributed systems where changes to processes are independent of each other. However, comparatively fewer works have addressed adaptation in distributed systems where changes to

multiple processes need synchronization. Moreover, behavioral verification during adaptation in distributed systems that require changes to multiple processes has not been adequately addressed. In the next section, we discuss some of the issues in adaptation in distributed systems.

1.1 Adaptation in Distributed Systems

In distributed systems, multiple processes need to be changed during adaptation. In such adaptations, changes to multiple processes need to be synchronized and interactions between changed and unchanged processes need to be controlled. We refer to the system before adaptation as the *old program* and to the system after adaptation as the *new program*. A process before it is modified is considered as a part of the old program, and after modification it is considered as a part of the new program. We now give the classification of adaptation in distributed systems.

1.1.1 Classification of Adaptation

We classify adaptation in distributed systems as: (i) *overlap adaptation* - when the old program and the new program overlap during adaptation, and (ii) *non-overlap adaptation* - when the old program and the new program are not present in the system simultaneously during adaptation.

Furthermore, we classify overlap adaptation into three main categories: (i) *mixed-mode* adaptation, (ii) *quiescence* adaptation, and (iii) *parallel* adaptation. In the case of quiescence adaptation, which is the most common approach for adaptation in distributed sys-

tems, there is no communication allowed between the old program and the new program. Consequently, during adaptation, changed and unchanged processes exist in the system simultaneously, but the processes are modified in such a way that the changed process and the unchanged process do not communicate with each other. In contrast, in case of mixed-mode adaptation, changed processes and unchanged processes are allowed to communicate with each other. In the case of parallel adaptation, each node has both the changed process and the unchanged process, but communication between changed processes and unchanged processes is not allowed; the changed (respectively, unchanged) process at one node can communicate with the changed (respectively, unchanged) process at another node.

To gain assurance in adaptation, formal specification and verification of adaptation is crucial. In the context of adaptive distributed systems, there are three aspects of verification: (i) verifying the system before adaptation, (ii) verifying the system during adaptation, and (iii) verifying the system after adaptation. While existing verification techniques can be used to verify the system before and the system after adaptation, such techniques cannot be applied directly to verify the system during adaptation. This is because during adaptation the system is changing whereas existing work assumes that the system remains unchanged. Especially, in the case of distributed systems, during adaptation the system exhibits *overlapping behavior* that is not well specified.

Numerous techniques have been proposed to address various issues in formalizing adaptation. A survey in [13] discusses various approaches based on graphs, process algebras, logic and other formalisms used to specify adaptive systems. Most of the approaches [1, 2, 4, 5, 7, 14–20] focus on design and implementation of adaptive systems. Other approaches [21–24] address the issue of verifying adaptation. The approaches in [21–23]

focus on offline adaptation, whereas the approach in [24] focuses on online adaptation of a single process system (that can also be extended to distributed systems that communicate via RPC). However, none of these approaches explicitly focus on the behavior of the system during adaptation in distributed systems.

Furthermore, mixed-mode adaptation has not received much attention, as it is normally considered difficult. Most existing approaches avoid dealing with mixed-mode behavior during adaptation by employing non-overlap, quiescence or parallel adaptation. Quiescence adaptation behaves as if the adaptation is performed by changing all processes at the same logical time. There is overhead in performing quiescence adaptation as a large number of messages are required to enforce synchrony among processes. In contrast, mixed-mode adaptation gives better performance in terms of service interruption time and communication overhead. Nevertheless, to develop mixed-mode adaptation correctly involves a lot of challenges.

In the rest of this chapter, we first discuss advantages of mixed-mode adaptation in Section 1.1.2. Then, in Section 1.1.3, we identify challenges in adaptation that arise or are exaggerated due to overlapping behavior of mixed-mode adaptation. Finally, in Section 1.2, we discuss the contributions of this research.

1.1.2 Advantages of Mixed-Mode Adaptation

We expect mixed-mode adaptation to offer the following two main advantages compared to other types of adaptation:

- **Reduced service interruption.** Since individual processes need not block while waiting for other processes during adaptation, the service interruption time is reduced. This is especially important when adaptation occurs frequently (in response to changes in the environment).

- **Low communication overhead.** Since mixed-mode adaptation allows the old program and the new program to interact during adaptation, the synchronization required among processes during adaptation is reduced, thereby reducing the communication overhead. This is especially important in systems that are operating on limited-power or resources.

We validate these advantages in this dissertation using a case study.

1.1.3 Challenges in Mixed-Mode Adaptation

The challenges in mixed-mode adaptation arise because the behavior of the old program and the new program overlap during adaptation. These challenges also occur in other forms of adaptation; here we discuss them in the context of mixed-mode adaptation.

- **Consistency.** All updates to individual processes must ensure consistency of the whole system. In mixed-mode adaptation some interactions between the old program and the new program during adaptation may not be *acceptable*. Only mixed-mode interactions that are acceptable should be allowed to occur. For example, if process X requires a service from process Y, then updating Y before X should be allowed only if new process Y' can handle the service requests from old process X (we use notation Y' to denote process Y after it has been changed).

- **State-Transfer.** To ensure proper mixed-mode operation, each process will need to preserve the state during adaptation. Specifically, if component C_1 is replaced by component C_2, then the state of component C_1 at each process needs to be transferred to component C_2. The efficiency of state-transfer mechanisms is particularly important in the case of mixed-mode adaptation compared to other forms of adaptation.

- **Assurance.** To provide assurance guarantees for mixed-mode adaptation requires a formal way to specify and verify the adaptation. In the case of mixed-mode adaptation, the changes cannot be specified in terms of system structure because the behavior of the old program and the new program overlap during adaptation. The first challenge is in specifying the overlapping behavior. Next, there are challenges in verification and testing of mixed-mode adaptation due to overlapping behavior.

- **Reuse.** There is a large amount of work done in adaptation. An approach for mixed-mode adaptation should reuse existing adaptation techniques whenever possible.

1.2 Thesis

Based on the above motivation, in this dissertation we propose the following thesis. Our goal is to develop an approach for assurance of adaptation that applies to both quiescence and mixed-mode adaptations.

The *lattice-based modeling* helps in verifying and testing of adaptive behavior in distributed systems.

To defend this, we make the following contributions in this dissertation:

1. **Modeling and specification of adaptive behavior**. We present the concept of an *adaptation lattice* to use for modeling and specifying the adaptive behavior. The adaptation lattice approach identifies the atomic adaptations, and the behaviors of the intermediate programs that occur during adaptation.

2. **Verification of adaptive behavior**. We present the concept of a *transitional-invariant lattice* to use for verifying the correctness of the system during adaptation. The transitional-invariant lattice approach ensures that safety is satisfied during adaptation and that adaptation eventually terminates.

3. **Fault-tolerance during adaptation**. We present the concept of a *transitional-faultspan lattice* to use for verifying the fault-tolerance properties of the system during adaptation. The approach can be used to verify different types of fault-tolerance during adaptation. Also, faults considered during adaptation can be different from the faults that the system is subjected to before or after adaptation.

4. **Tradeoffs in adaptation**. In order to assist the adaptation developer, we identify tradeoffs in adaptation that are useful while designing adaptation in various contexts. We show how concurrency during adaptation leads to increased verification complexity. We also show that increased concurrency during adaptation can also increase the communication overhead, which may not be desirable in certain systems.

5. **Testing of adaptation**. We use predicate detection techniques to test adaptation in distributed systems. We identify two classes of predicates for testing adaptation: (i)

adaptation-transient, and (ii) *adaptation-stable*. We introduce the notion of *adaptation vector* to identify states of the intermediate programs during adaptation.

6. **Component family design**. We describe the design of a *component family* to support adaptation. The component family design integrates aspects related to decision-making, adaptation logic, and component functionality, while maintaining a strict separation of different concerns. Specifically, the design separates the *adapt-active* parts of the component from the core functionality of the component. Moreover, the design separates the adaptation logic from the component functionality, thereby, simplifying the task of verifying adaptation.

Organization of the dissertation. The remainder of this dissertation is organized as follows: We first discuss the background and related work in adaptation in Chapter 2. In Chapter 3, we describe the adaptation lattice approach to model adaptation and adaptive systems. In Chapter 4, we present transitional-invariant lattice to verify adaptation in the absence of faults and discuss a case study based on message communication application to demonstrate its use. In Chapter 5, we present transitional-faultspan lattice to verify adaptation in the presence of faults and discuss a case study to demonstrate its use. In Chapter 6, we present a case study of mixed-mode adaptation where we show performance advantage of mixed-mode adaptation. We also discuss how the adaptation lattice approach can be used to verify mixed-mode adaptation. In Chapter 8, we describe the testing of adaptation using predicate detection techniques. In Chapter 9, we describe the component family design to build an adaptation-verified library of components. Finally, we discuss conclusions and future work in Chapter 10.

Chapter 2

Background and Related Work

Adaptation has been studied in various contexts, such as software change, reconfiguration, upgrade or update, change on fly, program modification, software evolution, *etc.* In this chapter, we first give an overview of some taxonomies of software adaptation and discuss the areas in adaptation that are focus of this dissertation. Then, we review some of the related works.

2.1 Adaptation Taxonomy

Several works [11–13, 25–27] have presented taxonomies of adaptation to categorize different adaptation mechanisms. In this section we discuss some of them.

One of the early works in classifying adaptation was done by Lientz and Swanson [25] in context of software maintenance. They proposed a topology that distinguishes between *perfective*, *adaptive* and *corrective* maintenance activities. This taxonomy was further refined by Chapin *et al.* [26], where they classify software evolution and software

maintenance into 12 different types: evaluative, consultive, training, updative, reformative, adaptive, performance, preventive, groomative, enhancive, corrective and reductive. In essence, the works in [25] and [26] categorize the adaptation activities on the basis of their *purpose* (*i.e.*, the *why* of software change). In our work, the reason why a software needs to adapt is irrelevant.

In [12], Buckley *et al.* gave a taxonomy of adaptation that focuses on the *how*, *when*, *where*, and *what*, of software change. This taxonomy is based on the characteristics of adaptation mechanisms and the factors that influence these mechanisms. The adaptation mechanisms refer to the software tools and algorithms used to perform the adaptation. However, this taxonomy does not consider formalisms used in adaptation. Regardless, the taxonomy is very useful in identifying and classifying various adaptation scenarios, and in categorizing and comparing different adaptation mechanisms. In our work, we focus on dynamic adaptation, *i.e.* the one performed at run-time (*when* of adaptation).

In [11], McKinley *et al.* classified adaptation into two categories, namely, *parameter adaptation* and *compositional adaptation*. Parameter adaptation modifies program variables that determine behavior. In contrast, compositional adaptation exchanges algorithmic or structural system components with another. Compositional adaptation provides more flexibility in supporting a variety of changes compared to simple tuning of program variables in case of parameter adaptation. However, compositional adaptation involves a lot more challenge in design, implementation and verification. Although our approach in this dissertation focuses on compositional adaptation, it can also be applied in the case of parameter adaptation. Specifically, we consider addition, removal, or replacement of *components* [28] in this dissertation. A component implements part of the desired behavior

of the program. Our notion of component is different from the popular usage of the term in the development community, where a component and a class (or object) (an artifact in object-oriented programming) are considered one and the same. According to our definition, a component can consist of more than one class, and may even be deployed at multiple processes.

Dynamically changing software is challenging in terms of correctness, robustness, and efficiency. Formal specification and verification is important in order to gain assurance in adaptive software. A variety of formal specification languages have been developed to gain a better understanding of the foundations of software change. Bradbury *et al.* in [13] has given a survey of 14 formal specification approaches based on graphs, process algebras, logic, and other formalisms. Graph-based approaches [14–16] use graph rewriting rules to specify dynamism. Approaches in [17, 18, 29, 30] use a variety of process algebras such as Calculus of Communicating Systems (CCS), Communicating Sequential Processes (CSP), and π-calculus. Architectural Description Language (ADL) based approaches [19, 20, 31] model programs as components and connectors, and adaptation as reconfiguration of connections. Generally, these approaches have focused on specifying the design of adaptive software and changes are specified in terms of the system structure. However, the approaches are inadequate in specifying the adaptive behavior. Approaches in [17, 18, 29, 30] are a few exceptions that have used process algebras to specify the behavior of adaptive programs. However, these approaches suffer from the following limitations: (i) adaptation-specific behavior of the program is not distinctly separated from the non-adaptive behavior of the program, (ii) the approaches are appropriate to specify changes in client-server applications, but it is not clear how it would apply to protocol changes in group

communication application, (iii) state-transfer is not specified explicitly, so it is not clear if the new program behavior has to start in the initial state or can start in some arbitrary state, (iv) the approaches are inadequate in specifying mixed-mode behavior during adaptation, and (v) the approaches use specific type of formalisms that may potentially limit widespread use and any extensions to include different types of adaptations. In our work, we address formal specification and verification of the behavior of system during adaptation.

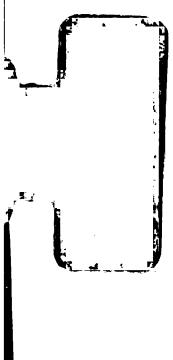
2.2 Related Work

In this section, we briefly review some of the previous work in the context of verifying adaptation.

2.2.1 DYMOS

Lee presented one of the early systems to support dynamic updating called DYMOS (Dynamic Modification System) [32]. It supports a single programmer modifying a module-based program dynamically (that is, without stopping its execution). In DYMOS, the programmer modifies and recompiles the source code of procedures and modules that need to be replaced. The programmer then requests the system to change the current core image to incorporate new code and data. New object code is inserted by a dynamic modification process that is executed in parallel with other user processes.

DYMOS supports program written in StarMod language [33]. In the DYMOS environment each updateable program is associated with a command interpreter, a source code management system, a StarMod compiler and a run-time environment. The system allows



individual procedures of a program to be changed.

In the context of ensuring correctness, Lee gave a procedure for partitioning change into sequence of smaller changes. The decomposition is done in such a way that the program behaves "acceptably" after each change. The decomposition approach helps in specifying behavior between changes where the program is operating with some reduced functionality. DYMOS does not address changes in distributed programs, however, it is important as it is one of the early systems that studied dynamic adaptation with particular concern for correctness. It gave some basic theoretical contributions in the context of correctness of adaptation.

2.2.2 CONIC

Conic [9, 34, 35] is a distributed programming system that supports dynamic reconfiguration of programs. A program in Conic consists of a number of processes that communicate with each other using well-defined entry and exit ports. Conic modules (processes) do not communicate by naming each other but by naming the ports. Thus, reconfiguration can be done by creating instances of modules and linking entry ports of these modules with exit ports of other instances.

The reconfiguration is done by providing a *configuration change specification*. The specification specifies the creation and deletion of modules and links. The *configuration manager* translates the change specification into commands to the operating system to execute the reconfiguration operation.

In [9], Kramer and Magee gave a formal basis for dynamic reconfiguration. They

specify change as structural change, in terms of component creation/deletion and connection/disconnection. In their model, interactions between processes are considered as *transactions*. A transaction is an exchange of information between two nodes, initiated by one of the nodes. A node is said to be in *passive* state if it is not currently engaged in a transaction that it initiated and it will not initiate new transactions. A node is *quiescent* if it is in passive state, is not currently engaged in servicing a transaction, and no transactions have been or will be initiated by other nodes which require service from this node. It is claimed that a dynamic reconfiguration will leave the system in a "consistent" state if all the involved processes are quiescent at the time of reconfiguration.

The main limitation of using node quiescence for adaptation is that it leads to excessive blocking during reconfiguration. Moreover, a large number of messages are required in synchronizing all nodes to reach quiescence state. Furthermore, local states of nodes are not considered in determining node quiescence. As a result, if local states of nodes are not consistent at the time of adaptation (reconfiguration), then the new version of the program will start in an inconsistent state.

2.2.3 ARGUS

Argus [36] is a programming language for building reliable distributed applications. It is based on the CLU programming language and provides support for atomic transactions and crash recovery. An application in Argus consists of a set of servers called *guardians*, that communicate with each other using remote procedure calls.

Dynamic reconfiguration in Argus is described in [37]. Similar to Conic, the connec-

tions between guardians can be rerouted dynamically. Guardians are made quiescent before replacement. To reach quiescent state the guardian can either abort running transactions or wait for them to complete. Argus suffers from the same limitations as Conic. Additionally, the replacement system requires Argus crash recovery facilities in order to work properly, which may make it difficult for use in other systems.

2.2.4 Online Software Version Change

Gupta and Jalote [24, 38] presented a framework for modeling changes to running programs and use it to study the *validity* of an on-line change. In their notion of validity, a change is valid if some time after the change, the process reaches a reachable state of the new program version. Thus, there is a "transition period" following a change, after which the the system behaves like a new program. However, it is not clear what behavior is acceptable during the transition period.

In their work, they consider different programming language styles, including imperative languages without procedures, imperative languages with procedures, and object-oriented languages. Their work focuses on change in sequential programs. They also discuss how their approach can be extended to distributed programs where only one process is changed. For the case where multiple processes are affected due to change, they consider a remote procedure call based model. They stop all processes before the change which leads to disruption of service till the change is completed. Moreover, their work does not address validity of on-line change in a distributed system that uses unrestricted message passing model.

2.2.5 Model-Based Development of Adaptive Software

Zhang *et al.* [39–41] proposed a model-based development of dynamically adaptive software. Their approach separates the model specifying the adaptive behavior from the model specifying the non-adaptive behavior. They use global invariants to specify properties that should be satisfied by adaptive programs regardless of adaptations. They enumerate different execution domains in which the program is required to execute, and build a state-based model in each domain. They enumerate possible adaptations of the program from one domain to another. Furthermore, they introduce A-LTL, an extension to linear temporal logic, to specify an adaptation from one program to another. They present three semantics of adaptation: one-point, guided and overlap. Similar to our goals, their work also addresses behavioral verification during adaptation. However, they do not consider general safety and liveness properties [42, 43] during adaptation. Their approach seems more suitable for quiescence adaptation and its application in mixed-mode adaptation is not straightforward.

2.2.6 Others

Several other works that addressed runtime adaptation include Podus [44–46], Durra [47], Polylith [48] and Dynamic ML [49, 50]. Similar to the limitations of the works discussed earlier, these approaches also suffer from one or more of the following limitations: (i) apply only to a single process change, (ii) apply only to distributed systems that communicate via RPC, but not in the case of asynchronous message passing model, and (iii) verify only structural changes; do not consider behavioral verification of system during adaptation. Other surveys of adaptation approaches can be found in [11, 38, 51, 52].

Chapter 3

Modeling Adaptation

In this chapter, we introduce a formal model for adaptation in asynchronous programs. We first present an informal overview of how adaptation occurs in a distributed system in Section 3.1. Then, we use the ideas discussed in Section 3.1 to formalize the model of adaptation in Section 3.2. In Section 3.3, we give definitions used in formal reasoning about the correctness of adaptation. Finally, in Section 3.4, we describe the concrete representation of programs and adaptations using guarded commands.

3.1 Adaptation Overview

We consider compositional adaptation as one that adds, removes, or replaces a component during adaptation. A component implements a part of the desired behavior of the system. A component (formally defined later in the chapter) consists of one or more *fractions*, where each fraction is associated with one process in the system. For the discussion in this chapter, we assume that adaptation replaces a component; addition and removal can be considered

as special cases of replacement.

We refer to the component that gets replaced as the *old component* and the component that replaces the old component as the *new component*. To replace an old component with a new component requires replacing each fraction of the old component with the corresponding fraction of the new component at all processes. An adaptation in a distributed system involves multiple steps that are executed at various processes. For example, consider a protocol that provides encrypted communication between a sender and a receiver. Such a protocol consists of two types of *fractions*, namely, *encryption fraction* at the sender that encrypts the packets before sending and *decryption fraction* at the receiver that decrypts the encrypted packets received from the sender. To replace such a protocol, each fraction of the protocol needs to be replaced. Thus, the adaptation in a distributed program involves multiple steps that are executed at various processes. We consider the replacement of a fraction at a single process as an atomic step of adaptation, and call it an *atomic adaptation*.

The old program, *i.e.*, the program before adaptation uses the old component and the new program, *i.e.*, the program after adaptation uses the new component. An adaptation replaces the old component with the new component, or equivalently we can say that the adaptation replaces the old program being executed by the system with the new program.

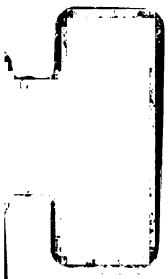
We assume that the old program and the new program are independently correct, *i.e.*, by themselves they can execute and produce acceptable behavior. The goal of verifying adaptation is to ensure that: (i) the adaptation ends in a state from where the system satisfies the behavior of the new program, and (ii) the (overlapping) behavior during adaptation is acceptable (as defined by specification during adaptation).

To verify the behavior during adaptation we need to classify the states of the program

during adaptation. The intermediate states that occur during adaptation are due to overlapping of the old program and the new program. The properties satisfied by these intermediate states may be different from the old program and the new program. Consequently, the behavior expected during adaptation needs to be specified separately from the old program and the new program. For example, in the case of adaptation of encrypted communication protocol discussed above, consider the system in which the adaptation has replaced the encryption fractions at the sender but has yet to replace the decryption fractions at the receiver. During adaptation the sender may continue to send packets that may be buffered at the receiver or the sender may be blocked from sending more packets until the receiver has replaced the decryption fractions. Clearly, there are different possible behaviors during adaptation, and the expected behavior during adaptation needs to be specified separately from the behavior of the old program and the new program. Towards this end, we define the notion of *intermediate program*.

Intermediate program. An intermediate program arises due to overlapping of behavior of the old program and the new program. The first atomic adaptation modifies the old program into the first intermediate program. Similarly, other atomic adaptations modify one intermediate program into the next intermediate program. The last atomic adaptation results in the new program. The specification during adaptation identifies the requirements for these intermediate programs.

We now present the formal model for adaptation and adaptive systems.



3.2 Abstract Model of Adaptation

We model a process as an automaton $\mathcal{A}$ represented as a tuple $\langle S, \Sigma, \delta, S_0 \rangle$, where

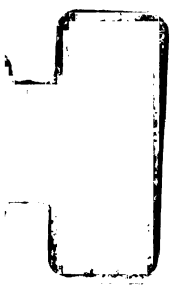
- $S(\mathcal{A})$ - a set of states
- $\Sigma(\mathcal{A})$ - a set of actions
- $\delta(\mathcal{A})$ - a state-transition relation, where $\delta(\mathcal{A}) \subseteq S(\mathcal{A}) \times \Sigma(\mathcal{A}) \times S(\mathcal{A})$
- $S_0(\mathcal{A})$ - a nonempty subset of $S(\mathcal{A})$ known as initial states

Each element (s, π, s') of $\delta(\mathcal{A})$ is known as a *transition*, where $s, s' \in S(\mathcal{A})$ and $\pi \in \Sigma(\mathcal{A})$. If $\mathcal{A}$ has a transition (s, π, s') it means that π is *enabled* in state s and executing action π in state s will lead to state s'. A transition of the form $(s, \_, s')$ is an abbreviation for a transition whose source is s and target is s', where the action that caused the transition is not of interest.

A program consists of a set of process automata. We assume the sets of actions of automata are disjoint. We consider *asynchronous* or *interleaved* execution for a program, where at any time only a single process can execute its action. This approach can be viewed as a reduction of concurrency to non-determinism, where a concurrent execution gives rise to many possible corresponding interleaving orders. We could have used more complex automata such as [53–55] to model concurrency, but we adopt an interleaving model as it results in a simpler theory for specification and verification of adaptation.

3.2.1 Adaptation as a set of automata

We model an adaptation Δ using a 5-tuple as follows:



- $\mathcal{I}$ - a set of automata
- P - an automaton of the old program, $P \in \mathcal{I}$
- Q - an automaton of the new program, $Q \in \mathcal{I}$
- Σ_a - a set of special type of actions known as *adaptive actions*, $\Sigma_a \cap (\bigcup_{\mathcal{A} \in \mathcal{I}} \Sigma(\mathcal{A})) = \phi$
- S_{map} - a state mapping is a partial function $(\bigcup_{\mathcal{A} \in \mathcal{I}} S(\mathcal{A})) \times \Sigma_a \rightarrow (\bigcup_{\mathcal{A}' \in \mathcal{I}} S(\mathcal{A}'))$ that satisfies the following two properties:

  i. $\forall s, s', \pi_a, \mathcal{A}_1, \mathcal{A}_2 : s \in S(\mathcal{A}_1), s' \in S(\mathcal{A}_2), \pi_a \in \Sigma_a, \mathcal{A}_1, \mathcal{A}_2 \in \mathcal{I}$:

  $((s, \pi_a), s') \in S_{map} \Rightarrow \mathcal{A}_1 \neq \mathcal{A}_2$, and

  ii. $\forall s_1, s_2, s_1', s_2', \pi_a, \mathcal{A}, \mathcal{A}_1, \mathcal{A}_2 : s_1, s_2 \in S(\mathcal{A}), s_1' \in S(\mathcal{A}_1), s_2' \in S(\mathcal{A}_2),$

  $\pi_a \in \Sigma_a, \mathcal{A}, \mathcal{A}_1, \mathcal{A}_2 \in \mathcal{I}$:

  $((s_1, \pi_a), s_1') \in S_{map} \wedge ((s_2, \pi_a), s_2') \in S_{map} \Rightarrow \mathcal{A}_1 = \mathcal{A}_2$

The old program, the new program, and all intermediate programs are modeled as automata. We also assume that the states of the automata in $\mathcal{I}$ are pair-wise disjoint. Given an adaptive action, the state mapping defines an automaton and the states of that automaton in which the adaptive action can execute, and the resulting automaton and the state of the resulting automaton in which the adaptive action terminates. The state mapping function satisfies two properties. The first property ensures that executing an adaptive action results in a change of automaton (whereas, executing an action results in a state of the same automaton). The second property states that if an adaptive action can be executed in different states of the automaton, then it will result in an unique automaton (the resulting states may be different, but they will be of the same automaton).

Note that the state mapping is a partial function, as it may not be possible to perform corresponding atomic adaptation in all states. Each element $((s, \pi_a), s')$ of S_{map} can be represented as a triplet (s, π_a, s'). Similar to the state-transition relation of an automaton, a state mapping S_{map} can be defined as a subset of $(\bigcup_{\mathcal{A} \in \mathcal{I}} S(\mathcal{A})) \times \Sigma_a \times (\bigcup_{\mathcal{A}' \in \mathcal{I}} S(\mathcal{A}'))$ with the restriction that if $(s, \pi_a, s') \in S_{map}$ and $(s, \pi_a, s'') \in S_{map}$ then $s' = s''$. Each element of S_{map} is known as *adaptive transition*. If S_{map} has an adaptive transition (s, π_a, s'), where $s \in S(\mathcal{A})$ and $s' \in S(\mathcal{A}')$, it means that π_a is *enabled* in $\mathcal{A}$ and executing π_a in state s of $\mathcal{A}$ will lead to state s' of $\mathcal{A}'$.

Note that the range of S_{map} is $S(\mathcal{A}')$ and not $S_0(\mathcal{A}')$. In other words, we do not require an adaptive action to terminate in an initial state of the resulting automaton.

Now, given the state mapping of adaptation Δ, we can define an *automata-transformation* (partial) function $\delta_a : \mathcal{I} \times \Sigma_a \rightarrow \mathcal{I}$. We have, $((\mathcal{A}, \pi_a), \mathcal{A}') \in \delta_a$ iff $\exists s, s' : s \in S(\mathcal{A}), s' \in S(\mathcal{A}') : (s, \pi_a, s') \in S_{map}$. Each element $((\mathcal{A}, \pi_a), \mathcal{A}')$ (equivalently, $(\mathcal{A}, \pi_a, \mathcal{A}')$) of δ_a is known as an *atomic adaptation*. Thus, each atomic adaptation is modeled as transforming one automaton to another automaton.

The automata-transformation function represents an adaptation lattice defined as follows:

Adaptation Lattice. *Adaptation lattice* (cf. Fig. 3.1) is a finite directed acyclic graph in which each node is labeled with an automaton and each edge is labeled with an atomic adaptation, such that,

1. There is a single *start node* P having no incoming edges. The start node is associated with the automaton representing the old program. The automata-transformation

function (correspondingly, S_{map}) satisfies the following condition:

$\forall \mathcal{A}, \pi_a :: (\mathcal{A}, \pi_a, P) \notin \delta_a$

2. There is a single *end node* Q having no outgoing edges. The end node is associated with the automaton representing the new program. The automata-transformation function (correspondingly, S_{map}) satisfies the following condition:

   $\forall \mathcal{A}, \pi_a :: (Q, \pi_a, \mathcal{A}) \notin \delta_a$

3. Each intermediate node R has at least one incoming edge and at least one outgoing edge. It is associated with the automaton representing the intermediate program. The automata-transformation function (correspondingly, S_{map}) satisfies the following condition:

   $\forall \mathcal{A} : \mathcal{A} \neq P : (\exists \mathcal{A}', \pi_a :: (\mathcal{A}', \pi_a, \mathcal{A}) \in \delta_a) \wedge$

   $\forall \mathcal{A} : \mathcal{A} \neq Q : (\exists \mathcal{A}', \pi_a :: (\mathcal{A}, \pi_a, \mathcal{A}') \in \delta_a)$

A path in the lattice from the start node to the end node is called *adaptation path.*

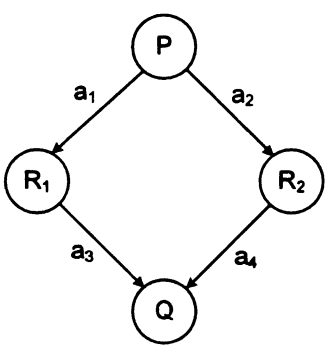


Figure 3.1: An example of an adaptation lattice.

3.2.2 Adaptation as an automaton

In the previous subsection we defined adaptation as a 5-tuple. An adaptation can also be viewed as an automaton defined as follows:

- $S(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{(\mathcal{A}, s) \mid s \in S(\mathcal{A})\}$
- $\Sigma(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{(\mathcal{A}, \pi) \mid \pi \in \Sigma(\mathcal{A})\} \cup \Sigma_a$
- $\delta(\Delta) = \bigcup_{\mathcal{A} \in \mathcal{I}} \{((\mathcal{A}, s), (\mathcal{A}, \pi), (\mathcal{A}, s')) \mid (s, \pi, s') \in \delta(\mathcal{A})\} \cup$
  $\{((\mathcal{A}, s), \pi_a, (\mathcal{A}', s') \mid (s, \pi_a, s') \in S_{map}\}$
- $S_0(\Delta) \subseteq (\mathcal{P}, S(\mathcal{P}))$

In definition of $S_0(\Delta)$ we use $S(\mathcal{P})$, and not $S_0(\mathcal{P})$, because adaptation should be able to start at any point in the execution of $\mathcal{P}$.

Modeling adaptation as an automaton allows us to verify some general properties of adaptation not concerning any overlapping behavior. On the other hand, modeling adaptation as a set of automata is important to identify individual intermediate automata during adaptation to verify properties due to overlapping behavior of the old program and the new program.

3.3 Adaptation Specification

In this section, we give some formal definitions used in specifying and verifying adaptive programs. We adapt these definitions from Arora and Kulkarni [56, 57].

Definition (State predicate). A state predicate X of $\mathcal{A}$ is any subset of $S(\mathcal{A})$. We say X is true in state s if $s \in X$.

Definition (Closure). A state predicate X of $\mathcal{A}$ is closed in $\mathcal{A}$ (respectively, $\delta(\mathcal{A}), \Sigma(\mathcal{A})$) iff the following condition holds:

$\forall s, s', \pi :: ((s, \pi, s') \in \delta(\mathcal{A})) \Rightarrow (s \in X \Rightarrow s' \in X)$

Definition (Computation). A computation of program $\mathcal{A}$ (respectively, adaptation Δ) is a sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfying the following conditions:

- For first state s_0 in σ, $s_0 \in S_0(\mathcal{A})$ (respectively, $S_0(\Delta)$)
- If σ is infinite then $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)
- If σ is finite and terminates in state s_l, then for all π, there does not exist a state s such that $(s_l, \pi, s) \in \delta(\mathcal{A})$ (respectively, $\delta(\Delta)$), and $\forall j : 0 < j \leq |\sigma| : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)

Definition (Specification). A specification of $\mathcal{A}$ is a set of computations. Given a specification, a computation in a specification is known as an *acceptable* computation. Following Alpern and Schneider [42], a specification can be decomposed into a safety specification and a liveness specification. As shown in [58], for a rich class of specifications, safety specification can be represented as a set of bad transitions that must not occur in program computations.

Definition (Satisfies). $\mathcal{A}$ satisfies a specification if each computation of $\mathcal{A}$ is in the specification. $\mathcal{A}$ satisfies a specification from X iff (i) X is closed in $\mathcal{A}$, and (ii) each computation of $\mathcal{A}$ is in the specification and starts from a state where X is true (i.e., $S_0(A) \subseteq X$).

Definition (Invariant). The state predicate X of $\mathcal{A}$ is an invariant iff $\mathcal{A}$ satisfies the specification from X. Note that, if X is an invariant of $\mathcal{A}$, then $X \supseteq S_0(\mathcal{A})$. Informally speaking, the invariant predicate includes the set of all states reached in the "acceptable" (correct) computations of $\mathcal{A}$. Note that the invariant predicate may include states that are not reachable in all computations of the program. However, computations from those states satisfy the specification and, hence, those states may be valuable in adding recovery transitions to provide fault tolerance [58].

Definition (Safety during adaptation). Similar to the specification of $\mathcal{A}$, safety specification during adaptation Δ is specified as a set of bad transitions that must not occur in computations of adaptation Δ.

Liveness during adaptation. We argue that the specification during adaptation should be a safety specification. This is due to the fact that one often wants the adaptation to be completed as quickly as possible. Hence, it is desirable not to delay adaptation to satisfy the liveness specification during adaptation. Rather, it is desirable to guarantee that, after adaptation, the program reaches states from where its (new) safety and liveness specifications are satisfied. Thus, the implicit liveness specification during adaptation is that the adaptation completes. In other words, the liveness specification during adaptation is that each intermediate program eventually executes its adaptive action. For these reasons, we have omitted the representation of liveness specification of the program.

3.3.1 Fault-tolerance

In this subsection we give formal definitions for specifying and verifying fault-tolerance properties of adaptive programs. These definitions are also adapted from Arora and Kulkarni [56, 57].

Definition (Fault class). Let Σ_f be a set of fault actions. A fault class $F(\mathcal{A})$ for program $\mathcal{A}$ is a subset of the set $S(\mathcal{A}) \times \Sigma_f \times S(\mathcal{A})$. We use $\mathcal{A}[]F$ to denote the transitions obtained by taking the union of the transitions in $\delta(\mathcal{A})$ and the transitions in $F(\mathcal{A})$. A fault class $F(\Delta)$ for adaptation Δ is:

$$\bigcup_{\mathcal{A} \in \mathcal{I}} \{((\mathcal{A}, s), \pi_f, (\mathcal{A}, s')) \mid (s, \pi_f, s') \in F(\mathcal{A})\}.$$

Definition (Fault-span). A state predicate T is a fault-span (F-span) of $\mathcal{A}$ from invariant S iff (i) $S \subseteq T$, and (ii) T is closed in $\mathcal{A}[]F$. A fault-span of a program includes the set of states that a program can reach in the presence of faults and it is closed under the execution of program and fault actions.

Definition (Computation in presence of faults). A computation of program $\mathcal{A}$ (respectively, adaptation Δ) in the presence of faults is a sequence of states $\sigma = \langle s_0, s_1, ... \rangle$ satisfying the following conditions:

- For first state s_0 in σ, $s_0 \in S_0(\mathcal{A})$ (respectively, $S_0(\Delta)$)
- If σ is infinite then $\forall j : j > 0 : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}) \cup F(\mathcal{A}))$ (respectively, $\delta(\Delta) \cup F(\Delta)$)
- If σ is finite and terminates in state s_l, then for all π, there does not exist a state s such that $(s_l, \pi, s) \in \delta(\mathcal{A})$ (respectively, $\delta(\Delta)$), and $\forall j : 0 < j \leq |\sigma| : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}) \cup F(\mathcal{A}))$ (respectively, $\delta(\Delta) \cup F(\Delta)$)

- If σ is infinite then $\exists n : n \geq 0 : (\forall j : j > n : (\exists \pi :: (s_{j-1}, \pi, s_j) \in \delta(\mathcal{A}))$ (respectively, $\delta(\Delta)$)

The first requirement captures that the computation begins in a initial state of the program (respectively adaptation). The second requirement captures that in each step, either a program (respectively, program or adaptive) transition or a fault transition is executed. The third requirement captures that faults do not have to execute, i.e., if the program reaches a state where only a fault transition can be executed then the fault transition need not be executed. Finally, the fourth requirement captures that the number of fault-occurrences in the computation is finite. This requirement is the same as that made in the previous work [59–62] to ensure that eventually recovery can occur.

Definition (Fault-tolerance (F-tolerant)). $\mathcal{A}$ is F-tolerant for specification *spec* from S iff the following two conditions hold: $(i)\mathcal{A}$ satisfies *spec* from S, and (ii) there exists T such that T is an F-span of $\mathcal{A}$ from S, and every computation of $\mathcal{A}[]F$ starting in a state where T is true satisfies *spec*.

Remark 1. Henceforth, whenever the invariant S, the program $\mathcal{A}$, and the specification *spec* are clear from the context, we will omit them; thus, "T is a F-span of $\mathcal{A}$ from S for *spec*" abbreviates to "T is a F-span".

Remark 2. Different types of tolerance specifications that normally occur in practice, namely, *masking*, *fail-safe*, and *non-masking* tolerance have been considered in [56–58]. In this dissertation, we assume masking fault-tolerance unless specified otherwise. The definitions can be easily extended to consider fail-safe and non-masking tolerance during adaptation.

3.4 Concrete Representation

In this section, we discuss the programming notation we use to describe the system. For brevity, we express programs using guarded commands [63, 64]. This gives a compact representation of the program defined as automata in Section 3.2 (in terms of state space and transitions). Translating the guarded command representation of the program to its automata representation is straightforward, as we discuss in this section.

Furthermore, the guarded command representation is closely related to a concrete implementation. Specifically, techniques for obtaining a guarded command representation from a program written in a general purpose language, such as C, are discussed in [65]. Also, techniques for transforming a program in guarded commands into a program in general purpose languages are discussed in [66–68].

3.4.1 Program

A program $\mathcal{P}$ is specified by a finite set of processes and channels. A process p is specified by a set of variables and a finite set of *actions*. The processes in a program communicate with one another by sending and receiving messages over unbounded channels that connect the processes. A channel from process p to process q is denoted by a channel variable $C_{p,q}$, which is an unbounded queue. Only process p can append an item of data to the rear of the queue $C_{p,q}$ and only process q can delete an item at the head of the queue $C_{p,q}$. Each variable has a predefined nonempty domain. A state of a process is obtained by assigning each variable a value from its respective domain. The state of the channel connecting p and q is given by the value of the queue $C_{p,q}$. The state of the program is given by the state of

all the processes and the channels. The state space of the program $\mathcal{P}$, $S(\mathcal{P})$, is the set of all possible states of $\mathcal{P}$. We use $s(x)$ to denote the value of variable x in state s, and $V(p)$ to denote the set of variables of process p. A state predicate of $\mathcal{P}$ is a boolean expression over process and channel variables.

Note that a state predicate may be characterized by the set of all states in which its boolean expression is true and, therefore, is a subset of the state space of the program.

Action. An action of p is uniquely identified by a name, and is of the form

$$\langle name \rangle \ : \ \langle guard \rangle \ \rightarrow \ \langle statement \rangle$$

A guard of each action is a state predicate of $\mathcal{P}$. The statement of each action is such that its execution updates zero or more process or channel variables. The sending of a message from p to q causes a message to be appended at the tail of the queue $C_{p,q}$. The receipt of a message from q by p is modeled by removing a message from the head of the queue $C_{p,q}$.

The set of actions of the program $\mathcal{P}$, $\Sigma(\mathcal{P})$ is given by the set of names of all the actions of all the processes of $\mathcal{P}$. Each action of p gives the set of transitions of the form (s, π, s') such that the guard of action π is true in state s and execution of statement of π in s results in state s'. Thus, the state-transition relation $\delta(\mathcal{P})$ is obtained from the set of actions of all the processes of $\mathcal{P}$. We say that an action of p is *enabled* in a state of p iff its guard evaluates to true in that state.

3.4.2 Component

A component is specified by a finite set of fractions that are involved in providing a common functionality. Intuitively, a component implements a part of the desired behavior of the system, such as some algorithm or protocol. A component fraction is specified by a set of variables and a finite set of actions that are associated with a single process. A component (respectively, fraction) is syntactically the same as a program (respectively, process), with the only difference that some variables of the component are designated as *input*, whose values are supplied by the program with which it is composed. The composition of the component and the program is the union of the variables and actions of the component and the program.

3.4.3 Adaptive action

An adaptive action is a special type of action, which is identified by a unique name and is of the form

$$\langle name \rangle \ : \ \langle guard \rangle \ \rightarrow \ TransformTo(p', \Phi).$$

If the adaptive action is an action of process p, then when the statement of the adaptive action is executed, p is replaced by p' and state-mapping Φ is used to initialize the variables of p'. Each adaptive action π_a gives a set of adaptive transitions of the form (s, π_a, s') such that the guard of π_a is true in state s of process p and execution of the statement of π_a results in state $s' = \Phi(s)$ of process p'. The state mapping function $S_{map}(\Delta)$ is obtained from the set of all adaptive actions.

From a modeling perspective, we consider that the adaptive action replaces the entire process, even if only a small part of it is actually changed. In an actual implementation, the adaptive action can be performed in various ways, such as by blocking execution of some method, or by loading/unloading some class. However, for verification we need to consider only the effect of the adaptive action. Additionally, considering each adaptive action as a generic form of process replacement gives the developer freedom to implement the adaptive action based on the platform and the language used.

3.4.4 State mapping

We define the following classes of state mapping Φ that occur during atomic adaptation:

- *Identity mapping.* In identity mapping, the names and the values of the variables remain the same. Formally, $V(p) \subseteq V(p')$ and for all s, $(\Phi(s))(y) = s(y)$.

- *Quasi mapping.* In quasi mapping, the name of the variable of the new process is different from that of the old process, though its value is the same as the value of some equivalent variable in the old process state. Formally, for a variable y of $V(p')$, there exists a variable x of $V(p)$ such that for all s, $(\Phi(s))(y) = s(x)$.

- *Initial mapping.* In initial mapping, the variables of the new process are initialized to some value as in the initial state of the new process. Formally, for a variable y of $V(p')$, for all s, $(\Phi(s))(y) = y_0$, where $y_0 \in S_0(y)$ and $S_0(y)$ is the set of values from domain of y that y can take in the initial states of process p'.

- *Functional mapping.* In functional mapping, the value of the variable of the new process is some function of the values of variables of the old process.

- *Arbitrary mapping.* An arbitrary mapping is a special type of functional mapping, where all variables of the new process are assigned some arbitrary value. Formally, for a variable y of $V(p')$, for all s, $(\Phi(s))(y) = y_d$, where $y_d \in D(y)$ and $D(y)$ denotes the domain of variable y.

- *Mixed mapping.* Most mappings that occur in practice are mixed mappings, in which variables of the new process $V(p')$ are divided into disjoint sets, and one of the above mappings is associated with each set.

Notation. We use "." to denote the *belongs to* relation. For example, if variable v belongs to process p, it is denoted by $p.v$, and action a of process p is denoted by $p.a$. A process p of program $\mathcal{P}$ is denoted by $\mathcal{P}.p$, and a fraction i of component C is denoted by $C.i$. For brevity, we avoid using belongs to relation if it is obvious from the context.

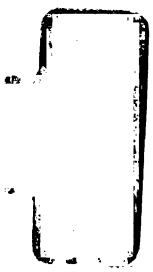
Chapter 4

Verifying Adaptation in Absence of Faults

In this chapter, we introduce the notion of *transitional-invariant* and *transitional-invariant lattice* to verify the correctness of adaptation. We first define transitional-invariant in Section 4.1. Next, in Section 4.2, we define transitional-invariant lattice and give a theorem to prove correctness of adaptation. In Section 4.3, we present a case study of adaptation in the message communication application to demonstrate the use of transitional-invariant lattice. Finally, we discuss some of the questions raised by this work in Section 4.4.

4.1 Transitional-Invariant

As discussed in Chapter 3, the program during adaptation consists of actions of the old program and actions of the new program. Therefore, we consider intermediate programs obtained after one or more atomic adaptations. Similar to the invariants that are used to iden-



tify "legal" program states and are closed under program execution, we define transitional-invariants.

Definition (Transitional-invariant). Let R be an intermediate program in the adaptation Δ. A *transitional-invariant* is a predicate that is closed in R.

Note that the actions of an intermediate program are the old program's actions that are not yet removed and the new program's actions that are already added. However, the adaptive actions do not necessarily preserve the transitional-invariant. Now, we define transitional-invariant lattice.

4.2 Transitional-Invariant Lattice

A *transitional-invariant lattice* is an adaptation lattice with each node having one predicate and that satisfies the following five conditions:

1. **Safety of old program.** The start node P is associated with an invariant S_P of the program before adaptation.

2. **Safety of new program.** The end node Q is associated with an invariant S_Q of the program after adaptation.

3. **Safety of intermediate program.** Each intermediate node R is associated with a predicate TS_R that is a transitional-invariant for any intermediate program at R (*i.e.*, an intermediate program obtained by performing adaptations from the entry node to R). Furthermore, any intermediate program at R satisfies the (safety) specification during adaptation from TS_R.

Cor

are s

duri

intui

say t

•

•

•

Th

and su

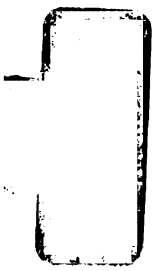
4. **Safety of adaptive action.** If a node labeled R_i has an outgoing edge labeled a to a node labeled R_j, then for all adaptive transitions (s, a, s') in S_{map} where TS_{R_i} is true in state s, TS_{R_j} is true in state s'. In other words, $\forall s, s' : (s, a, s') \in S_{map} : s \in TS_{R_i} \Rightarrow s' \in TS_{R_j}$. Furthermore, all the adaptive transitions (s, a, s') satisfy the safety specification during adaptation.

5. **Progress of adaptation.** If a node labeled R has outgoing edges labeled $a_1, a_2, ..., a_k$ to nodes labeled $R_1, R_2, ..., R_k$, respectively, then in all computations of adaptation there exists a transition (s, s') such that for some $i : 1 \leq i \leq k : (s, a_i, s') \in S_{map}$. Furthermore, $\forall s : s \in TS_R : (\forall a, s' : a \in \Sigma_a - \{a_1, ..., a_k\} : (s, a, s') \notin S_{map})$.

Correctness of adaptation. Intuitively, an adaptation is correct if the following conditions are satisfied: If the adaptation begins in a legitimate state of the old program, then safety during adaptation is met and the resulting state of the new program is legitimate. With this intuition, if adaptation begins in a state where invariant of the old program is true, then we say that adaptation is correct if:

- Adaptation terminates in a state where invariant of the new program is true
- During adaptation safety specification during adaptation is satisfied
- Eventually adaptation terminates

The following theorem states that finding a transitional-invariant lattice is necessary and sufficient for proving correctness of adaptation.



Theorem 4.1. *Given S_P as the invariant of the program before adaptation and S_Q as the invariant of the program after adaptation, the adaptation from P to Q is correct if and only if there is a transitional-invariant lattice for the adaptation with the start node associated with S_P and the end node associated with S_Q.*

Proof.

($\Rightarrow$) If the transitional-invariant lattice exists, then adaptation is correct.

If the stated conditions are satisfied, then the specification of the old program is satisfied when the adaptation starts. Also, the existence of the transitional-invariant lattice during adaptation ensures that for each intermediate program that occurs during adaptation, the specification during adaptation is satisfied. Moreover, from the definition of the transitional-invariant lattice, each adaptive action satisfies safety specification during adaptation. Also, in each intermediate program eventually some adaptive action will be executed, which ensures the liveness of adaptation. Furthermore, the last adaptive action terminates in the invariant of the new program, from where the system satisfies the behavior of the new program. Thus, the existence of the transitional-invariant lattice proves the correctness of adaptation.

($\Leftarrow$) If adaptation is correct, then transitional-invariant lattice exists (proof by construction).

Let adaptation consist of n adaptive actions $a_1, ..., a_n$. Consider all adaptive actions that can occur in a state of the old program. Since the adaptation is correct, each of these adaptive actions occur in a state of the old program where S_P holds, and execution of these adaptive actions satisfies the safety during adaptation.

Now, consider the intermediate program I_1 reached after execution of a_1. In all computations of the old program till the execution of a_1, the invariant S_P is satisfied since the old program is correct. Since the adaptation is correct, the intermediate program I_1 satisfies the specification during adaptation (otherwise, a_1 is not permitted in a state of the old program). Once a_1 is executed, we consider all the computations of the intermediate program I_1, and identify the transitional-invariant TS_{I_1} associated with it. Similarly, we consider all computations of the intermediate program reached after the execution of some adaptive action other than a_1 in a state of the old program, and find the transitional-invariant corresponding to that intermediate program. In this way, we construct the first level of intermediate programs and corresponding transitional-invariants starting from the old program.

Now, for each intermediate program at the first level we consider all possible adaptive actions that can occur in some state of its computations. We can then identify the transitional-invariants at the second level in the lattice by considering all the computations of the intermediate programs reached due to the execution of adaptive actions in the states of the corresponding intermediate programs at first level.

In this way, we can continue to find transitional-invariants at various levels in the lattice. Since the adaptation is correct, the atomic adaptation in each intermediate program at level $n-1$ will result in a state of the new program where S_Q holds.

Thus, the correctness of adaptation proves the existence of the transitional-invariant lattice. □

4.3 Case Study: Reliable Message Communication

In this section, we present an example that illustrates how the transitional-invariant lattice can be used to verify correctness of adaptation in the context of a simple message communication program. The communication program that we consider is an abstraction of the communication aspect of the applications such as video conferencing, audio streaming, or any distributed application where messages are transferred over wired or wireless channel. We first describe the fault-intolerant message communication program in Section 4.3.1. Then, we describe the FEC-based *proactive component* in Section 4.3.2. Next, in Section 4.3.3, we discuss adaptation of adding the proactive component to the fault-intolerant message communication program. In Section 4.3.3, we also identify the transitional-invariant lattice for the adaptation. In Chapter 5, we continue with the message communication program to discuss the adaptation of replacing the proactive component with the acknowledgment-based *reactive component.*

4.3.1 Fault-Intolerant Communication Program

Specification of the communication program. An infinite queue of messages at sender process s is to be sent to two receiver processes r_1 and r_2 via two unicast channels and copied into corresponding infinite queues at the receivers. Faults may cause loss of messages in the channel.

The message communication program is shown in Figure 4.1. Only `send` and `receive` actions of the program are shown, since only those actions are considered for adaptation.

Processes s, r_1, and r_2 maintain queues sQ, $r_1.rQ$, and $r_2.rQ$ respectively. sQ con-

program $\mathcal{P}_{intol}$

process s

var sQ : **queue of integer**
m : **integer**

begin

send : $\neg$isEmpty(sQ) $\rightarrow$ $m := \texttt{head}(sQ)$;
$C_{s,r_1}, C_{s,r_2} := C_{s,r_1} \circ m, C_{s,r_2} \circ m$

end

process $r_i[i = 1, 2]$

var rQ : **queue of integer**
m : **integer**

begin

receive : $\neg$isEmpty(C_{s,r_i}) $\rightarrow$ $m := \texttt{head}(C_{s,r_i})$;
$rQ := rQ \circ m$

end

Figure 4.1: Message communication program (fault-intolerant version).

tains messages that s needs to send to r_1 and r_2. The messages received by r_i from s are stored in $r_i.rQ$. Let mQ be the queue of all messages to be sent. (mQ is an auxiliary variable that is used only for the proof.) Initially, $sQ = mQ$. The function $\texttt{head}(sQ)$ returns the message at the front of sQ, and $\texttt{head}(sQ, k)$ returns k messages from the front of sQ. The notation $sQ \circ d$ denotes the concatenation of sQ and $\langle d \rangle$.

Invariant. The invariant of the communication program is $S_P = S1 \wedge S2$, where

$S1 = \forall i : (m_i \in r_1.rQ \vee m_i \in r_2.rQ) \Rightarrow m_i \in mQ$, and

$S2 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in C_{s,r_1} \veebar m_i \in r_1.rQ)$

$\wedge (m_i \in C_{s,r_2} \veebar m_i \in r_2.rQ)))$.

In the above invariant, $S1$ indicates that messages received by the receivers are sent by the sender. $S2$ indicates that a message m_i is not yet sent by the sender, or it is in the channel, or it is already received by the receiver, all exclusive.

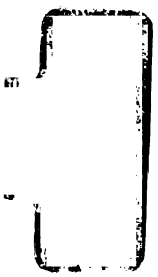
Notation. The symbol $\veebar$ denotes *exactly one* operator, *i.e.*, $x \veebar y \veebar z$ implies exactly one of x, y and z is true.

4.3.2 Proactive Component

The proactive component sends extra messages to the receiver, which the receiver can use to recover from the lost messages. It consists of two types of fractions: encoder and decoder. The encoder fraction is added at the sender process and the decoder fraction is added at the receiver process. The encoder takes $(n - k)$ data packets and encodes them to add k parity packets. It then sends the group of n (data and parity) packets. The decoder needs to receive at least $(n - k)$ packets of a group to decode all the data packets. This component provides tolerance to certain message loss faults (discussed in Chapter 5).

Figure 4.2 shows the abstract version of the proactive component. The encoder and the decoder fractions of the component are shown. The encoder fraction consists of two actions: `encode` and `fec_send`. The decoder consists of two actions: `decode` and `fec_receive`. These fractions are composed with the process that will use them. The composition of a fraction and a process is done by union, which is equivalent to combining the actions of the fraction and the process. We assume that appropriate renaming is performed so that there are no inconsistencies in the definitions of the variables of the fractions and the processes. The message communication program composed with the proactive component is shown in Figure 4.3.

Specification of program using the proactive component. The program using the proactive component satisfies the same specification as the communication program.



Component *fec*

Fraction *encoder*

inp sQ : **queue of integer**
r_1, r_2

var n, k, u, l, m : **integer** {initially, $u = l = m = 0$}
encQ : **array [integer**, $0..n-1$**] of integer** {initially, $encQ = \perp$}

begin

encode : $true \rightarrow encQ[u, 0..n-1] := \text{fec\_encode}(\texttt{head}(sQ, n-k))$;
$u := u + 1$

[] **fec_send** : $encQ[l, m] \neq \perp \rightarrow C_{s,r_1} := C_{s,r_1} \circ \{l, m, encQ[l, m]\}$;
$C_{s,r_2} := C_{s,r_2} \circ \{l, m, encQ[l, m]\}$;
$m := (m + 1)$ **mod** n;
if $m = 0$ **then**
$l := l + 1$
fi

end

Fraction $decoder_i$

inp *rQ* : **queue of integer**
s

var n, k, x, y, p, m : **integer** {initially, $p = 0$}
rbufQ : **array [integer**, $0..n-1$**] of integer** {initially, $rbufQ = \perp$}

begin

fec_receive : $\neg\text{isEmpty}(C_{s,r_i}) \rightarrow x, y, m := \texttt{head}(C_{s,r_i})$;
$rbufQ[x, y] := m$

[] **decode** : $\texttt{count}(rbufQ[p, 0..n-1] \neq \perp) >= (n-k) \rightarrow$
$rQ := rQ \circ \text{fec\_decode}(rbufQ[p, 0..n-1])$;
$p := p + 1$

end

Figure 4.2: Proactive component.

Invariant. The invariant of the program using the proactive component is $S_Q = S1 \wedge S_F$, where

$$S_F = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ$$
$$\veebar ((m_i \in r_1.rQ \veebar m_i \in \text{data}(encQ \cup C_{s,r_1} \cup r_1.rbufQ))$$
$$\wedge (m_i \in r_2.rQ \veebar m_i \in \text{data}(encQ \cup C_{s,r_2} \cup r_2.rbufQ)))).$$

Program $\mathcal{P}_{fec}$

process s

var : $\mathcal{P}_{intol}.s.\text{var} \cup fec.encoder.\text{var}$

begin

fec.encoder.encode

[] fec.encoder.fec_send

end

process $r_i[i = 1, 2]$

var : $\mathcal{P}_{intol}.r_i.\text{var} \cup fec.decoder_i.\text{var}$

begin

fec.decoder.fec_receive

[] fec.decoder.decode

end

Figure 4.3: Message communication program (with proactive component).

We use the notation $m_i \in \text{data(encQ} \cup C_{s,r_1} \cup r_1.rbufQ)$ to imply that message m_i can be generated from the data in $\{\text{encQ} \cup C_{s,r_1} \cup r_1.rbufQ\}$. In the above invariant, S_F indicates that the message is either at the sender, or already received by the receiver, or it can be generated from the data in the channel and the buffers at the sender and the receiver.

4.3.3 Adaptation: Addition of the Proactive Component

The adaptation of adding the proactive component converts the program shown in Figure 4.1 to the one shown in Figure 4.3. We first require an *adapt-ready* version of the program $\mathcal{P}_{intol}$ as shown in Figure 4.4. (An adapt-ready program is one that is composed with adaptive actions.) We now give the specification during adaptation of adding the proactive component.

Specification during adaptation. The specification during adaptation is that S_1 continues

program $\mathcal{P}_{a\text{-}intol}$

process s

var : $\mathcal{P}_{intol}.s.$var

begin

$\mathcal{P}_{intol}.s.$send

[] a_1 : *true* $\rightarrow$ *TransformTo*$(\mathcal{P}_{a\text{-}ip_1}.s, \Phi a_1)$;

end

process $r_i[i = 1, 2]$

var rQ : **queue of integer**

begin

$\mathcal{P}_{intol}.r_i.$receive

[] $a_{(i+1)}$: $a_1 \wedge$ isEmpty(C_{s,r_i}) $\rightarrow$ *transformTo*$(\mathcal{P}_{fec}.r_i, \Phi a_{(i+1)})$;

end

Figure 4.4: Message communication program (fault-intolerant version, adapt-ready).

to be true during adaptation.

We describe the adaptation by identifying the intermediate programs and the corresponding transitional-invariants during adaptation after each atomic adaptation.

program $\mathcal{P}_{a\text{-}ip_1}$

process s

var : $\mathcal{P}_{intol}.s.$var

begin

a_4 : $a_2 \wedge a_3$ $\rightarrow$ *TransformTo*$(\mathcal{P}_{fec}.s, \Phi a_4)$;

end

process $r_i[i = 1, 2]$: same as in Fig. 4.4

Figure 4.5: Intermediate program $\mathcal{P}_{a\text{-}ip_1}$.

The execution of adaptive action a_1 in $\mathcal{P}_{a\text{-}intol}$ results in intermediate program $\mathcal{P}_{a\text{-}ip_1}$ shown in Figure 4.5. $\mathcal{P}_{a\text{-}ip_1}$ does not send any packets, but the packets that are there in the channel can still be received by the receivers r_1 and r_2. In the execution of $\mathcal{P}_{a\text{-}ip_1}$ eventually all the packets in the channel are read and no new packets are added in the

channel from the sender to the receiver. Thus, the guards of the adaptive actions a_2 and a_3 eventually get enabled. The transitional-invariant of $\mathcal{P}_{a\text{-}ip_1}$ is: $TS_1 = S_1 \wedge S_2$, where S_1, S_2 are as defined earlier in Section 4.3.1.

program $\mathcal{P}_{a\text{-}ip_2}$
process s : same as in Fig. 4.5
process r_1 : same as in Fig. 4.3
process r_2 : same as in Fig. 4.4

Figure 4.6: Intermediate program $\mathcal{P}_{a\text{-}ip_2}$.

Since a_1 and a_2 occur independently, we consider both possible orderings among them. The execution of adaptive action a_2 in $\mathcal{P}_{a\text{-}ip_1}$ results in intermediate program $\mathcal{P}_{a\text{-}ip_2}$ shown in Figure 4.6. In $\mathcal{P}_{a\text{-}ip_2}$, receiver r_1 has replaced its fraction, whereas receiver r_2 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_2. Eventually, in the execution of $\mathcal{P}_{a\text{-}ip_2}$, the guard of adaptive action a_3 gets enabled and a_3 is executed resulting in intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

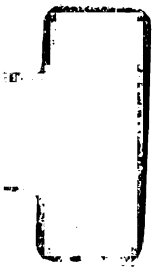
The transitional-invariant of $\mathcal{P}_{a\text{-}ip_2}$ is $TS_2 = S_1 \wedge S_3$, where

$$S_3 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in r_1.rQ) \wedge (m_i \in C_{s,r_2} \veebar m_i \in r_2.rQ))$$
$$\wedge \text{ isEmpty}(C_{s,r_1}) = true \wedge r_1.rbufQ = \perp \wedge r_1.p = 0).$$

program $\mathcal{P}_{a\text{-}ip_3}$
process s : same as in Fig. 4.5
process r_1 : same as in Fig. 4.4
process r_2 : same as in Fig. 4.3

Figure 4.7: Intermediate program $\mathcal{P}_{a\text{-}ip_3}$.

The execution of adaptive action a_3 in $\mathcal{P}_{a\text{-}ip_1}$ results in intermediate program $\mathcal{P}_{a\text{-}ip_3}$ shown in Figure 4.7. In $\mathcal{P}_{a\text{-}ip_3}$, receiver r_2 has replaced its fraction, whereas receiver r_1



has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_1. Eventually, in the execution of $\mathcal{P}_{a\text{-}ip_3}$, the guard of adaptive action a_2 gets enabled and a_2 is executed resulting in intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

The transitional-invariant of $\mathcal{P}_{a\text{-}ip_3}$ is $TS_3 = S_1 \wedge S_4$, where

$$S_4 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in r_2.rQ) \wedge (m_i \in C_{s,r_1} \veebar m_i \in r_1.rQ))$$
$$\wedge \text{isEmpty}(C_{s,r_2}) = true \wedge r_2.rbufQ = \perp \wedge r_2.p = 0).$$

program $\mathcal{P}_{a\text{-}ip_4}$
process s : same as in Fig. 4.5
process $r_i[i = 1, 2]$: same as in Fig. 4.3

Figure 4.8: Intermediate program $\mathcal{P}_{a\text{-}ip_4}$.

In intermediate program $\mathcal{P}_{a\text{-}ip_4}$ shown in Figure 4.8, only adaptive action a_4 is enabled, and execution of a_4 results in new program $\mathcal{P}_{fec}$. The transitional-invariant of $\mathcal{P}_{a\text{-}ip_4}$ is $TS_4 = S_1 \wedge S_5$, where

$$S_5 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar (m_i \in r_1.rQ \wedge m_i \in r_2.rQ))$$
$$\wedge \text{isEmpty}(C_{s,r_1}) = true \wedge r_1.rbufQ = \perp \wedge r_1.p = 0$$
$$\wedge \text{isEmpty}(C_{s,r_2}) = true \wedge r_2.rbufQ = \perp \wedge r_2.p = 0.$$

We now give the state mappings for the adaptive actions in the adaptation that are used in initializing the state of the new fraction at each process.

State mapping. The state mapping for each adaptive action is shown in Table 4.1. Each adaptive action initializes the state of the new process when it is executed based on this mapping.

Based on the description of adaptation in this section, we find the transitional-invariant lattice as shown in Figure 4.9 for the adaptation of adding the proactive component. Thus,

Mapping Function	Process Affected	New State
Φ_{a_1}	s	Identity mapping
Φ_{a_2}	r_1	$\{rQ, s\}$ - Identity mapping, $\{n, k, x, y, p, m, rbufQ\}$ - Initial mapping
Φ_{a_3}	r_2	$\{rQ, s\}$ - Identity mapping, $\{n, k, x, y, p, m, rbufQ\}$ - Initial mapping
Φ_{a_4}	s	$\{sQ, r_1, r_2\}$ - Identity mapping, $\{n, k, u, l, m, encQ\}$ - Initial mapping

Table 4.1: State mappings for the adaptation.

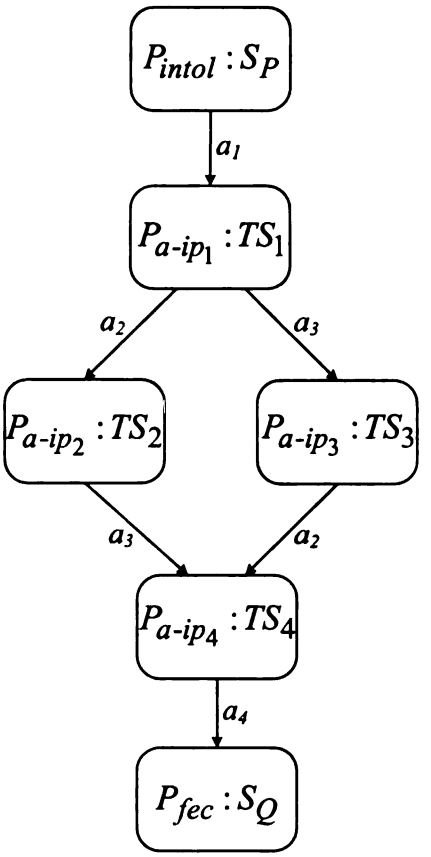


Figure 4.9: Adaptation lattice for addition of proactive component.

we have the following theorem.

Theorem 4.2. *The adaptation lattice of Figure 4.9 is the transitional-invariant lattice for the adaptation of adding the proactive component. Hence, the adaptation is correct.* □

4.4 Discussion

In this section, we discuss some of the questions raised by this work.

Why is the specification during adaptation a safety specification?

The specification of the application before adaptation can be arbitrary. However, during adaptation the specification should be a safety specification. It is not desirable to delay the adaptation to satisfy liveness during adaptation. Rather, we would expect the adaptation to complete as quickly as possible and the new program to satisfy the safety and liveness after adaptation. For example, consider a transaction processing system with liveness guarantees to commit or abort. In this case, either the adaptation should not start in the middle of the transaction, or if the adaptation can be started in the middle of the transaction then the liveness should be met once the adaptation is completed. Thus, the implicit liveness specification during adaptation is that adaptation completes.

How is transitional-invariant lattice constructed?

Techniques [64, 69] developed to calculate invariants of a program can also be used to find transitional-invariants. For a given adaptation model, we can perform reachability analysis for each intermediate program obtained after execution of the atomic adaptation. The reachability computation for an intermediate program helps in identifying the transitional-invariant for that intermediate program, and we can construct a transitional-invariant lattice for the given adaptation. Furthermore, the techniques for dynamically discovering likely invariants from the execution of the system such as [70–72] can be used to find transitional-invariants for the intermediate programs in the adaptation lattice.

Can transitional-invariant lattice approach be used to verify existing adaptation tech-

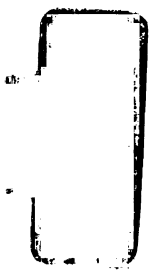
niques?

Yes, the transitional-invariant lattice approach can also be used to verify existing adaptation techniques. We give an outline of how existing adaptation techniques can be verified using transitional-invariant lattice. To verify existing adaptation approaches, we first need to identify all atomic adaptations and the corresponding adaptive actions. We then need to consider all possible orderings and concurrency among adaptive actions, and identify intermediate programs after each adaptive action. Next, we find transitional-invariants for each intermediate program and check if the transitional-invariants imply safety of adaptation.

Can adaptation lattice approach be used in the case of parallel adaptation?

Yes, the adaptation lattice approach can also be used to verify parallel adaptation. We discuss the use of adaptation lattice approach in the case of parallel adaptation using a simple example. Consider a system consisting of two processes, a sender process and a receiver process. Both processes communicate using an encryption protocol. The adaptation requires that eventually the current (or, old) encryption protocol be replaced by another (or, new) encryption protocol. The parallel adaptation adds the new encryption protocol alongside the current encryption protocol. In other words, the sender process has two types of fractions (for encryption) and the receiver process has two types of fractions (for decryption). All users using the sender process to send data to users at the receiver process initially use the old encryption protocol. Once the new encryption protocol is added, some users use the old protocol whereas some users use the new protocol. Eventually, after all users have stopped using the old protocol, it can be removed from the system.

The adaptation lattice in this case is as shown in Figure 4.10. Program P uses the



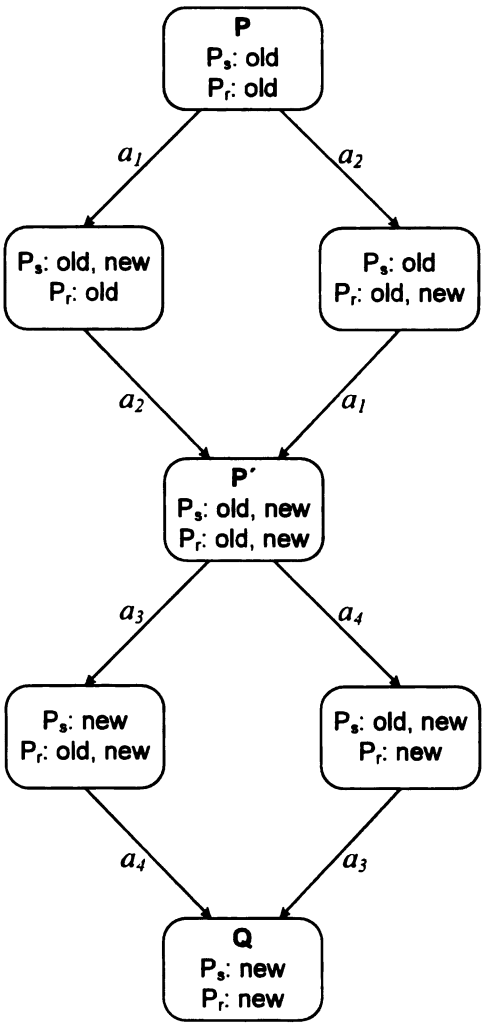


Figure 4.10: Adaptation lattice for parallel adaptation.

old protocol fractions at both the processes. Eventually, after adaptation is completed, program Q uses the new protocol fractions at both the processes. Program P' uses both types of fractions at both the processes. During adaptation, program P' can stay active for an unbounded time. The correctness requirements for P' are that the old (respectively, new) fraction at the sender process communicates only with the old (respectively, new) fraction at the receiver process. The correctness requirements for the intermediate programs between P and P' are that the old fraction at the sender process communicates only with the old

fraction at the receiver process, and the communication from the new fraction at the sender process to the receiver process is buffered. Similarly, the correctness requirements for the intermediate programs between P' and Q are that the new fraction at the sender process communicates only with the new fraction at the receiver process, and no communication occurs between the old fraction at the sender process and the old fraction at the receiver process.

Chapter 5

Verifying Adaptation in Presence of Faults

In this chapter, we introduce the notion of *transitional-faultspan* and *transitional-faultspan lattice* to verify the fault-tolerance properties during adaptation. We first define transitional-faultspan in Section 5.1. Then, in Section 5.2, we define transitional-faultspan lattice and give a theorem to prove the correctness of adaptation in the presence of faults. Finally, in Section 5.4, we present a case study of adaptation in the message communication application to demonstrate the use of transitional-faultspan lattice.

5.1 Transitional-Faultspan

Let F_P be the fault class of the old program and F_Q be the fault class of the new program. Let S_P be an invariant and T_P be a F_P-span of the old program. Similarly, let S_Q be an invariant and T_Q be a F_Q-span of the new program. The old program is F_P-tolerant, and

the new program is F_Q-tolerant. Let F be the fault class during adaptation.

In the context of adaptation, we define transitional-faultspans to identify the set of states that the program can reach while performing adaptation in the presence of faults.

Definition (Transitional-faultspan). Let R be an intermediate program in the adaptation Δ, and TS be a transitional-invariant of R. A *transitional-faultspan* (F-span) of R from TS is a predicate TT that satisfies following two conditions: $(i)\, TS \subseteq TT$, and $(ii)\, TT$ is closed in $R[]F$.

Now, we define transitional-faultspan lattice.

5.2 Transitional-Faultspan Lattice

A *transitional-faultspan (F-span) lattice* is an adaptation lattice where each node is associated with two predicates, a transitional-invariant and a transitional-faultspan, and the following conditions are satisfied:

0. **Correctness in absence of faults.** The adaptation lattice obtained by considering the transitional-invariants only forms a transitional-invariant lattice.

1. **Fault-tolerance of old program.** The entry node P is associated with a F_P-span T_P of the program before adaptation.

2. **Fault-tolerance of new program.** The exit node Q is associated with a F_Q-span T_Q of the program after adaptation.

3. **Fault-tolerance of intermediate program.** Each intermediate node R is associated with a predicate TT_R that is a transitional-faultspan (F-span) from TS_R for any

intermediate program at R (i.e., intermediate program obtained by performing adaptations from the entry node to R). Furthermore, any intermediate program at R is F-tolerant from TS_R.

4. **Safety of adaptive action.** If a node labeled R_i has an outgoing edge labeled a to a node labeled R_j, then for all adaptive transitions (s, a, s') in S_{map} where TT_{R_i} is true in state s, TT_{R_j} is true in state s'. In other words, $\forall s, s' : (s, a, s') \in S_{map} : s \in TT_{R_i} \Rightarrow s' \in TT_{R_j}$. Furthermore, all the adaptive transitions (s, a, s') satisfies the safety specification during adaptation.

5. **Progress of adaptation.** If a node labeled R has outgoing edges labeled $a_1, a_2, ..., a_k$ to nodes $R_1, R_2, ..., R_k$, respectively, then in all computations of adaptation there exists a transition (s, s') such that for some $i : 1 \leq i \leq k : (s, a_i, s') \in S_{map}$. Furthermore, $\forall s : s \in TT_R : (\forall a, s' : a \in \Sigma_a - \{a_1, ..., a_k\} : (s, a, s') \notin S_{map})$.

Correctness of adaptation in presence of faults. Intuitively, an adaptation is correct in presence of faults F if the following conditions are satisfied: If the adaptation begins in a legitimate state of the old program then during adaptation each intermediate program is F-tolerant and the resulting state of the new program is legitimate. With this intuition, if the adaptation begins in a state where the fault-span of the old program is true, then we say that the adaptation is correct if:

- Adaptation terminates in a state where fault-span of the new program is true
- During adaptation, each intermediate program is F-tolerant

- Eventually adaptation terminates

The following theorem states that finding a transitional-faultspan lattice is necessary and sufficient for proving correctness of adaptation.

Theorem 5.1. *Given S_P as the invariant of the program before adaptation, T_P as the faultspan used to show that the program before adaptation is tolerant to F_P, S_Q as the invariant of the program after adaptation, and T_Q as the faultspan used to show that the program after adaptation is tolerant to F_Q, the adaptation from P to Q is correct in presence of faults F if and only if there is a transitional-faultspan (F-span) lattice for the adaptation with start node associated with S_P and T_P, and end node associated with S_Q and T_Q.*

Proof.

The proof of this theorem is similar to Theorem 4.1 discussed in Chapter 4. □

5.3 Adaptation of Self-Stabilizing Programs

In this section, we consider the adaptation by Gouda *et al.* in [73], where the authors have focused on adapting from one self-stabilizing [59] program into another self-stabilizing program. We show that this is an instance of our approach where all the transitional-faultspan predicates are *true*.

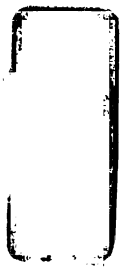
A program is self-stabilizing if starting from an arbitrary state it eventually recovers to a legitimate state. Thus, in transforming from one stabilizing program to another, we can let all the fault-spans (*i.e.*, fault-span of the old program, fault-span of the new program and transitional-faultspans of the intermediate programs) be true. With this approach, if the

old program starts in any state, eventually the new program execution begins although the state of the new program may be arbitrary. Since the new program is self-stabilizing, it will eventually recover to legitimate states.

Note that in [73] the corresponding transitional-invariants may not exist. Specifically, even if the old program begins in legitimate states, the new program may initially be in illegitimate states before recovery takes place. Moreover, the approach in [73] allows arbitrary behavior during adaptation and, hence, the specification during adaptation may not be met.

5.4 Case Study: Reliable Message Communication (Continued)

In this section, we continue with the example of Chapter 4. We consider the adaptation that replaces the proactive component with the reactive component. We first discuss the adapt-ready version of the proactive component and the faults tolerated by the proactive component in Section 5.4.1. Next, in Section 5.4.2, we describe the acknowledgment-based reactive component. We then discuss the adaptation of replacing the proactive component by the reactive component in Section 5.4.3. Finally, in Section 5.4.3, we identify the transitional-faultspan lattice to verify the correctness of this adaptation in the presence of faults.



5.4.1 Proactive Component

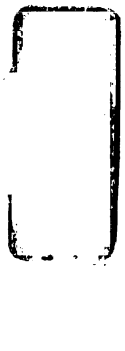
We discussed the proactive component in Chapter 4. The adapt-ready version of the proactive component is shown in Figure 5.1.

Program $\mathcal{P}_{aa\text{-}fec}$
process s
var : $\mathcal{P}_{fec}.s.\text{var}$
begin
fec.encoder.encode
[] fec.encoder.fec_send
[] aa_1 : *true* $\rightarrow$ *transformTo*$(\mathcal{P}_{aa\text{-}ip_1}.s, \Phi_{aa_1})$
end
process $r_i[i = 1, 2]$
var : $\mathcal{P}_{fec}.r_i.\text{var}$
begin
fec.decoder.fec_receive
[] fec.decoder.decode
[] $aa_{(i+2)}$: $aa_2 \wedge \text{isEmpty}(C_{s,r_i})$ $\rightarrow$ *transformTo*$(\mathcal{P}_{ack}.r_i, \Phi_{aa_{(i+2)}})$
end

Figure 5.1: Message communication program (with proactive component, adapt-ready).

The specification of the program using proactive component is discussed in Chapter 4. Additionally, it tolerates message loss faults of class $F1$ (cf. Figure 5.2). Faults of class $F1$ causes a loss of up to k messages in a group. In writing the fault transitions, we use the following auxiliary variables: m^g to denote a message m from group g, and $lostCount_i^g$ to denote the number of messages lost in group g in the channel from s to r_i. Initially, $\forall g :: lostCount_i^g = 0$. We now give the fault-span of the program using the proactive component.

Fault-span. The $F1$-span of the program using the proactive component is $T_Q = S_Q$. The fault-span is same as the invariant since the proactive component provides masking



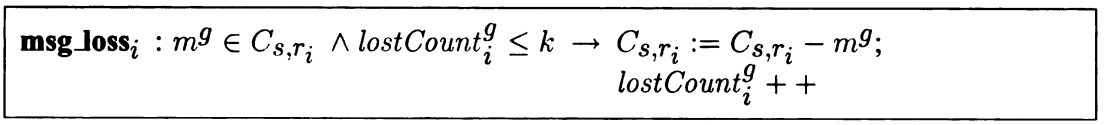


Figure 5.2: Fault class $F1$.

fault-tolerance.

5.4.2 Reactive Component

The reactive component deals with the message loss by retransmitting the lost packets. It uses acknowledgments to confirm the receipt of messages sent by the sender, and negative acknowledgments to confirm the loss of messages sent by the sender. It consists of $aSnd$ fraction at the sender and $aRcv$ fraction at the receiver. The $aSnd$ fraction adds a group and a packet number in each packet. It maintains a window of size w and sends all packets in that window to the receiver. It waits for the acknowledgment of receipt of a group before moving the window one group forward. If it receives a negative acknowledgment for any packet, it sends that packet again to the receiver. When the $aRcv$ fraction at the receiver receives a packet out of order, it waits for a few more packets before sending a negative acknowledgment to the sender. When all packets in a group are received, it sends an acknowledgment for that group to the sender.

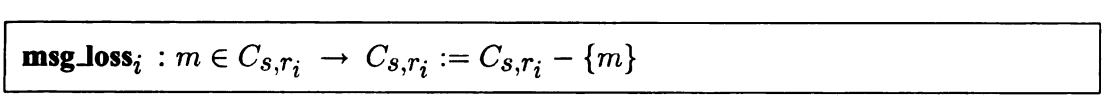


Figure 5.3: Fault class $F2$.

The reactive component provides tolerance to message loss faults $F2$ shown in Figure 5.3. Faults of class $F2$ causes loss of messages from the channel. For simplicity, we assume that acknowledgment messages are not lost; however, the component can be easily extended to deal with faults that lose acknowledgments by using *timeout* guards.

Component *ack*

Fraction *aSnd*

inp sQ : **queue** of **integer**
$r1, r2$

var n, w, g_i, p_i : **integer** {initially, $p_i = g_i = 0$}
g_a, g_{na}, p_{na} : **integer** {initially, $p_i = g_i = 0$}
$sQcopy_i$: **queue** of **integer** {initially, Empty}
snt_i : **array** $[0..w-1, 0..n-1]$ of **integer** {initially $\perp$}

param i : $i = 1, 2$

begin

copy$_i$: isEmpty$(sQcopy_i) \rightarrow sQcopy_i := sQ$

[] **send**$_i$: $\neg$isEmpty$(sQcopy_i) \wedge snt_i[g_i, p_i] = \perp \rightarrow$
$snt_i[g_i, p_i] := \{g_i, p_i, \texttt{head}(sQcopy_i)\}$;
$C_{s,r_i} := C_{s,r_i} \circ snt_i[g_i, p_i]$;
$p_i := (p_i + 1) \bmod n$;
if $p_i = 0$ **then**
$g_i := (g_i + 1) \bmod w$
fi

[] **resend**$_i$: $\texttt{type}(C_{r_i,s}) = \text{nack} \rightarrow g_{na}, p_{na} := \texttt{head}(C_{r_i,s})$;
if $snt_i[g_{na}, p_{na}] \neq \perp$ **then**
$C_{s,r_i} := C_{s,r_i} \circ snt_i[g_{na}, p_{na}]$
fi

[] **ack_rcv**$_i$: $\texttt{type}(C_{r_i,s}) = \text{ack} \rightarrow g_a, snt_i[g_a, 0..n-1] := \texttt{head}(C_{r_i,s}), \perp$

end

Figure 5.4: Acknowledgment component: sender fraction.

Figures 5.4 and 5.5 shows the abstract version of the reactive component. The *aSnd* fraction consists of four types of actions: `copy`, `send`, `resend`, `ack_rcv`. The *aRcv* fraction consists of three types of actions: `receive`, `deliver`, and `send_nack`. These fractions are composed with processes that will use them. The message communication program composed with the reactive component is shown in Figure 5.6. We now give the specification of the program using the reactive component.

Specification of program using the reactive component. Program using the reactive component satisfies the same specification as the communication program (cf. Chapter 4).

Component *ack*

Fraction $aRcv_i$

inp rQ
s

var n, w, g, p : **integer** {initially $k = ng = 0$}
k, ng, m : **integer** {initially $k = ng = 0$}
$rbufQ$: **array** $[0..w-1, 0..n-1]$ of **integer** {initially $\perp$}
ug : **array** $[0..w-1]$ of **boolean** {initially *false*}

param j : $0 \leq i \leq w-1$

begin

receive : $\neg\text{isEmpty}(C_{s,r_i}) \rightarrow g, p, m := \text{head}(C_{s,r_i});$
$rbufQ[g,p], ug[g] := m, true$

[] **deliver**$_j$: $ug[j] = true \rightarrow$ **if** $\text{count}(rbufQ[j, 0..n-1] \neq \perp) = n$ **then**
$rQ := rQ \circ rbufQ[j, 0..n-1];$
$rbufQ[j, 0..n-1], C_{r_i,s} := \perp, C_{r_i,s} \circ \text{ack}(j);$
$ug[j], ng := \textit{false}, (j+1) \bmod w$
fi

[] **send_nack** : $\text{count}(ug[0..w-1] = true) > 2 \rightarrow$ **for** $k = 0$ **to** $n-1$
if $rbufQ[ng, k] = \perp$ **then**
$C_{r_i,s} := C_{r_i,s} \circ \text{nack}(ng, k)$
fi

end

Figure 5.5: Acknowledgment component: receiver fraction.

Additionally, it tolerates message loss faults $F2$.

Invariant. The invariant of the program using the reactive component is $S_R = S_1 \wedge S_A$, where

$$S_A = \forall i : m_i \in mQ \Rightarrow$$

$$((m_i \in sQcopy_1 \veebar m_i \in r_1.rQ$$

$$\veebar (m_i \notin (sQcopy_1 \cup r_1.rQ) \Rightarrow (m_i \in snt_1 \wedge (m_i \in C_{s,r_1} \veebar m_i \in r_1.rbufQ))))$$

$$\wedge (m_i \in sQcopy_2 \veebar m_i \in r_2.rQ$$

$$\veebar (m_i \notin (sQcopy_2 \cup r_2.rQ) \Rightarrow (m_i \in snt_2 \wedge (m_i \in C_{s,r_2} \veebar m_i \in r_2.rbufQ))))).$$

In the above invariant, S_A indicates that for a message m, exactly one of the following

Program $\mathcal{P}_{ack}$

process s

var : $\mathcal{P}_{aa-fec}.s.\text{var} \cup ack.aSnd.\text{var}$

param $i : i = 1, 2$

begin

ack.aSnd.copy$_i$

[] **ack.aSnd.send**$_i$

[] **ack.aSnd.send_again**$_i$

[] **ack.aSnd.ack_rcv**$_i$

end

process $r_i[i = 1, 2]$

begin

var : $\mathcal{P}_{aa-fec}.r_i.\text{var} \cup ack.aRcv_i.\text{var}$

param $k : 0 \leq k \leq w - 1$

begin

ack.aRcv.receive

[] **ack.aRcv.deliver**$_k$

[] **ack.aRcv.send_nack**

end

Figure 5.6: Message communication program (with reactive component).

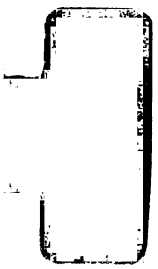
is true:

- m is at the sender, and is not yet sent
- m is received by the receiver
- m is buffered by the sender, and m is either in the channel or is buffered at the receiver.

Fault-span. The $F2$-span of the program using the reactive component is $T_R = S_1 \wedge T_A$, where

$T_A = \forall i : m_i \in mQ \Rightarrow$

$((m_i \in sQcopy_1 \veebar m_i \in r_1.rQ \veebar (m_i \notin (sQcopy_1 \cup r_1.rQ) \Rightarrow m_i \in snt_1))$



$\wedge\ (m_i \in sQcopy_2 \veebar m_i \in r_2.rQ \veebar (m_i \notin (sQcopy_2 \cup r_2.rQ) \Rightarrow m_i \in snt_2)))$.

In the above fault-span, T_A indicates that for a message m, exactly one of the following is true:

- m is at the sender, and is not yet sent
- m is received by the receiver
- if m is sent by the sender and not yet received by the receiver, then m is buffered by the sender.

5.4.3 Adaptation: Replacement of Proactive Component with Reactive Component

The adaptation of replacing the proactive component with the reactive component converts the program shown in Figure 5.1 to the one shown in Figure 5.6. We now give the specification during adaptation for the replacement of the proactive component with the reactive component.

Specification during adaptation. The specification during adaptation is that S_1 continues to be true during adaptation in presence of faults F_1.

We now describe the adaptation by discovering the intermediate programs and the corresponding transitional-invariants and transitional-faultspans during adaptation. We identify the intermediate programs after each atomic adaptation.

The execution of adaptive action aa_1 in $\mathcal{P}_{aa\text{-}fec}$ results in intermediate program $\mathcal{P}_{aa\text{-}ip_1}$ shown in Figure 5.7. $\mathcal{P}_{aa\text{-}ip_1}$ does not encode any new packets, but will send

Program $\mathcal{P}_{aa\text{-}ip_1}$

process s

var : $\mathcal{P}_{aa-fec}.s.\text{var}$

begin

fec.encoder.fec_send

[] $aa_2 : aa_1 \wedge l = u \rightarrow$ *transformTo*$(\mathcal{P}_{aa\text{-}ip_2}.s, \Phi_{aa_2})$;

end

process $r_i[i = 1, 2]$: same as in Fig. 5.1

Figure 5.7: Intermediate program $\mathcal{P}_{aa\text{-}ip_1}$.

any remaining encoded packets. In the execution of $\mathcal{P}_{aa\text{-}ip_1}$, eventually all the encoded packets are sent to the receivers. Thus, the guard of adaptive action aa_2 becomes true. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_1}$ is: $TS_5 = S_Q \wedge S_6$, where S_Q is defined earlier in Chapter 4, and

$S_6 = (\forall j : j \geq u : encQ[j, 0..n-1] = \perp) \wedge (l \leq u)$.

In the above transitional-invariant, S_6 indicates that no new packets will be encoded by the sender. The transitional-faultspan TT_5 of $\mathcal{P}_{aa\text{-}ip_1}$ is same as TS_5.

Program $\mathcal{P}_{aa\text{-}ip_2}$

process s

var : $\mathcal{P}_{aa-fec}.s.\text{var}$

begin

$aa_5 : aa_3 \wedge aa_4 \rightarrow$ *transformTo*$(\mathcal{P}_{ack}.s, \Phi_{aa_5})$;

end

process $r_i[i = 1, 2]$: same as in Fig. 5.1

Figure 5.8: Intermediate program $\mathcal{P}_{aa\text{-}ip_2}$.

The execution of aa_2 results in intermediate program $\mathcal{P}_{aa\text{-}ip_2}$ shown in Figure 5.8. $\mathcal{P}_{aa\text{-}ip_2}$ does not send any packets, but the packets that are there in the channel can still be received by the receivers r_1 and r_2. In the execution of $\mathcal{P}_{aa\text{-}ip_2}$ eventually all the

packets in the channel are read and no new packets are added in the channel from sender to receiver. Thus, the guards of the adaptive actions aa_3 and aa_4 eventually becomes true. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_2}$ is: $TS_6 = S_1 \wedge S_7 \wedge S_8$, where S_1 is defined earlier in Chapter 4, and

$S_7 = (\forall j : j \geq u : encQ[j, 0..n-1] = \perp) \wedge (l = u)$, and

$S_8 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar ((m_i \in r_1.rQ \veebar m_i \in \text{data}(C_{s,r_1} \cup r_1.rbufQ))$

$\wedge (m_i \in r_2.rQ \veebar m_i \in \text{data}(C_{s,r_2} \cup r_2.rbufQ)))).$

In the above transitional-invariant, S_7 indicates that there are no encoded packets left at the sender for sending, and S_8 indicates that all packets that are sent are either received by the receivers or are in the corresponding channels. The transitional-faultspan TT_6 of $\mathcal{P}_{aa\text{-}ip_2}$ is same as TS_6.

Program $\mathcal{P}_{aa\text{-}ip_3}$
process s : same as in Fig. 5.8
process r_1 : same as in Fig. 5.6
process r_2 : same as in Fig. 5.1

Figure 5.9: Intermediate program $\mathcal{P}_{aa\text{-}ip_3}$.

Since aa_3 and aa_4 occur independently, we consider both possible orderings between them. The execution of adaptive action aa_3 in $\mathcal{P}_{aa\text{-}ip_2}$ results in intermediate program $\mathcal{P}_{aa\text{-}ip_3}$ shown in Figure 5.9. In $\mathcal{P}_{aa\text{-}ip_3}$, receiver r_1 has replaced its fraction, whereas receiver r_2 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_2. Eventually, in the execution of $\mathcal{P}_{aa\text{-}ip_3}$ the guard of adaptive action aa_4 gets enabled and aa_4 is executed resulting in intermediate program $\mathcal{P}_{aa\text{-}ip_5}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_3}$ is $TS_7 = S_1 \wedge S_7 \wedge S_9 \wedge S_{10}$, where

$S_9 = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ$

$\veebar\ (m_i \in r_1.rQ\ \wedge\ (m_i \in r_2.rQ\ \veebar\ m_i \in \text{data}(C_{s,r_2} \cup r_2.rbufQ))))$, and

$S_{10} = \text{isEmpty}(C_{s,r_1}) = true \wedge r_1.rbufQ = \perp$.

The transitional-faultspan TT_7 of $\mathcal{P}_{aa\text{-}ip_3}$ is same as TS_7.

Program $\mathcal{P}_{aa\text{-}ip_4}$
process s : same as in Fig. 5.8
process r_1 : same as in Fig. 5.1
process r_2 : same as in Fig. 5.6

Figure 5.10: Intermediate program $\mathcal{P}_{aa\text{-}ip_4}$.

The execution of adaptive action aa_4 in $\mathcal{P}_{aa\text{-}ip_2}$ results in intermediate program $\mathcal{P}_{aa\text{-}ip_4}$ shown in Figure 5.10. In $\mathcal{P}_{aa\text{-}ip_4}$, receiver r_2 has replaced its fraction, whereas receiver r_1 has not yet replaced its fraction and can receive any remaining packets in the channel from s to r_1. Eventually, in the execution of $\mathcal{P}_{aa\text{-}ip_4}$ the guard of adaptive action aa_3 gets enabled and aa_3 is executed resulting in intermediate program $\mathcal{P}_{aa\text{-}ip_5}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_4}$ is $TS_8 = S_1 \wedge S_7 \wedge S_{11} \wedge S_{12}$, where

$S_{11} = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ$

$\veebar\ ((m_i \in r_1.rQ\ \veebar\ m_i \in \text{data}(C_{s,r_1} \cup r_1.rbufQ))\ \wedge\ m_i \in r_2.rQ))$, and

$S_{12} = \text{isEmpty}(C_{s,r_2}) = true \wedge r_2.rbufQ = \perp$.

The transitional-faultspan TT_8 of $\mathcal{P}_{aa\text{-}ip_4}$ is same as TS_8.

Program $\mathcal{P}_{aa\text{-}ip_5}$
process s : same as in Fig. 5.8
process $r_i[i = 1, 2]$: same as in Fig. 5.6

Figure 5.11: Intermediate program $\mathcal{P}_{aa\text{-}ip_5}$.

In intermediate program $\mathcal{P}_{aa\text{-}ip_5}$ shown in Figure 5.11, only adaptive action aa_5 is

enabled, and execution of aa_5 results in the new program $\mathcal{P}_{ack}$. The transitional-invariant of $\mathcal{P}_{aa\text{-}ip_5}$ is $TS_9 = S_1 \wedge S_{10} \wedge S_{12} \wedge S_{13}$, where

$$S_{13} = \forall i : m_i \in mQ \Rightarrow (m_i \in sQ \veebar (m_i \in r_1.rQ \wedge m_i \in r_2.rQ)).$$

The transitional-faultspan TT_9 of $\mathcal{P}_{aa\text{-}ip_5}$ is same as TS_9.

We now give the state mappings for the adaptive actions in the adaptation that are used in initializing the state of the new fraction at each process.

State mapping. The state mapping for each adaptive action is shown in Table 5.1. Each adaptive action initializes the state of the new process when it is executed based on this mapping.

Mapping Function	**Process Affected**	**New State**
Φ_{aa_1}	s	Identity mapping
Φ_{aa_2}	s	Identity mapping
Φ_{aa_3}	r_1	$\{rQ, s\}$ - Identity mapping, $V(r_1) - \{rQ, s\}$ - Initial mapping
Φ_{aa_4}	r_2	$\{rQ, s\}$ - Identity mapping, $V(r_2) - \{rQ, s\}$ - Initial mapping
Φ_{aa_5}	s	$\{sQ, r_1, r_2\}$ - Identity mapping, $V(s) - \{sQ, r_1, r_2\}$ - Initial mapping

Table 5.1: State mappings for the adaptation.

Based on the description of the adaptation in this section, we find the transitional-faultspan (F-span) lattice as shown in Figure 5.12 for the adaptation of replacing the proactive component with the reactive component. Thus, we have the following theorem.

Theorem 5.2. *The adaptation lattice of Figure 5.12 is a transitional-faultspan (F1-span) lattice for the adaptation of replacing the proactive component with the reactive component. Hence, the adaptation is correct in presence of faults.* □

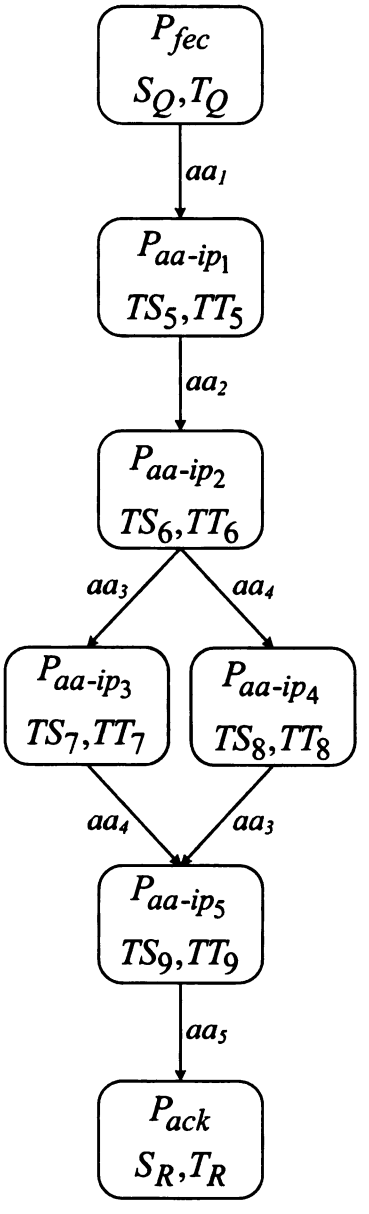


Figure 5.12: Adaptation lattice for replacement of proactive component with reactive component.

Chapter 6

Case Study: Mixed Mode Adaptation

In this chapter, we discuss the case study of mixed-mode adaptation. We study mixed-mode adaptation in the context of protocol change as discussed in dynamically adaptable middleware [1, 74, 75]. Specifically, we consider two leader election protocols and adaptation that changes one leader election protocol to another at run-time. We replace a leader election protocol that elects a leader based on process identification to a leader election protocol that elects a leader based on process value, where the value of a process can be defined using process's battery life, its average distance to other processes, etc.

In the rest of this chapter, we first discuss the two leader election protocols in Section 6.1; we discuss the system model, protocol specifications and descriptions in this section. Next, in Section 6.2, we discuss the adaptation of leader election protocols; we discuss the overlap communication scenarios, state mappings and verification of the adaptation in this section. In Section 6.3, we present the performance results of mixed-mode adaptation. Finally, in Section 6.4, we discuss the limitations of mixed-mode adaptation.

6.1 Leader Election Protocols

Leader election is a fundamental problem in distributed computing. The basic description of leader election problem is stated as: *eventually elect a unique leader.* For example, in the case of group communication protocols, the leader election is employed to elect a new coordinator whenever a group coordinator fails. Numerous leader election protocols have been proposed in the literature for a variety of applications. We discuss two versions of the leader election protocol that is based on the termination detection protocol by Dijkstra and Scholten [76]. The protocols discussed in this section are an abstraction of the protocol discussed in [77]. We assume that there is another module at all (or selected) processes that monitors the status of the leader process. A monitor process starts an instance of the leader election protocol whenever it detects a failure of the leader.

6.1.1 System Model and Assumptions

The system consists of n processes. All processes have unique identifiers which we denote by id. Each process maintains a variable ldr, which denotes the value of the leader that the protocol elected, and a variable ϵ, which denotes if the process is in election or not . We make the following assumptions about the system:

- *Bi-directional channel.* The channels between processes are bidirectional, *i.e.*, if the system has a channel $C_{p,q}$ from p to q, then it also has a channel $C_{q,p}$.
- *Static processes.* We assume that processes are static and the network is connected. If the processes are mobile and the network suffers from partitioning, then the protocol can be extended as discussed in [77].

- *Process failure and reliable communication.* A process or a link can fail, however, for simplicity, we assume that while the election is going on there is no process or link failure. We assume reliable communication, *i.e.*, if process p sends a message m to process q, then eventually q receives the message m.

6.1.2 Specification of Leader Election Protocols

We consider the following problem specification of the leader election protocols that we use in this case study:

Safety: $\Box(i.ldr \neq j.ldr \Rightarrow i.\epsilon \vee j.\epsilon)$

Liveness: $\Box\Diamond\forall i, j : i.ldr = j.ldr$

The safety property asserts that no two stable processes (*i.e.*, processes not in the election) can have different leaders. The liveness property asserts that eventually a unique leader is elected.

We now discuss two leader election protocols, one that elects the process with the maximum id as the leader, and one that elects the process having the maximum value as the leader.

6.1.3 Leader Election based on Process ID

The leader election protocol, *ldrId*, that elects the process with maximum id as a leader is shown in Figure 6.1. In addition to the safety and liveness properties discussed earlier in Section 6.1.2, the protocol also satisfies the following liveness property:

Liveness of *ldrId*: $\Box\Diamond i.ldr = max\{k \mid d_{i,k} < \infty\}$,

Component *ldrId*

Fraction i

inp N : **list**

var $startElection, \epsilon, ack$: **bool** {initially $false$}

p, num, max, ldr : **integer** {initially $p = num = max = 0, ldr = \perp$}

$src\langle Num, Id\rangle$: $\langle$**integer,** $1..n\rangle$ {initially $\langle 0, i\rangle$}

W, chd : **list** {initially $W, chd = \phi$}

begin

startElection : $\neg\epsilon \wedge startElection \rightarrow$

$src, num := \langle num, i\rangle, num + 1;$

ELECT.$src := src;$

for $j = 1$ **to** $n \wedge j \in N$

$C_{i,j}$.add(ELECT);

$W, ack, \epsilon, p, max, chd := N, true, true, i, i, \phi;$

$startElection := false$

[] **joinElection** : ELECT($ldrId$) $\in C_{j,i} \rightarrow$

$C_{j,i}$.remove(ELECT);

if $\neg\epsilon \vee (\epsilon \wedge$ ELECT.$src > src)$ **then**

$src :=$ ELECT.$src;$

$num := src.Num + 1;$

for $k = 1$ **to** $n \wedge (k \neq j \wedge k \in N)$

$C_{i,k}$.add(ELECT);

$W, ack, \epsilon, p, max, chd := N - \{j\}, true, true, j, i, \phi;$

else if $\epsilon \wedge src =$ ELECT.src **then**

ACK.$\{src, chd\} := src, false;$

$C_{i,j}$.add(ACK)

fi

[] **ackToParent** : $\epsilon \wedge W = \phi \wedge src.Id \neq i \wedge ack \rightarrow$

$ack := false;$

ACK.$\{src, chd, id\} := src, true, max;$

$C_{i,p}$.add(ACK)

Figure 6.1: Leader election algorithm based on node Id.

which asserts that the elected leader is the process having maximum id among all connected processes.

The protocol uses three types of messages as shown in Table 6.1. It also shows the fields associated with each message type. The variables used by the leader election protocol are

```
[] ackReceive  : ε ∧ ACK(ldrId) ∈ C_{j,i} →
                     C_{j,i}.remove(ACK);
                     if ack ∧ src = ACK.src then
                       W := W − {j};
                       if ACK.chd then
                         chd = chd + {j};
                         max := MAX(ACK.id, max);
                       fi
                     fi
[] electLeader : ε ∧ W = φ ∧ src.Id = i ∧ ack →
                     ack, ε, ldr := false, false, max;
                     LD.{src, id} := src, max;
                     for j = 1 to n ∧ j ∈ chd
                       C_{i,j}.add(LD)
[] setLeader   : ε ∧ LD(ldrId) ∈ C_{j,i} ∧ ¬ack →
                     C_{j,i}.remove(LD)
                     if src = LD.src then
                       ldr, ε := LD.ld, false;
                       for k = 1 to n ∧ k ∈ chd
                         C_{i,k}.add(LD)
                     fi
end
```

Figure 6.1: Leader election algorithm based on node Id (Continued).

shown in Table 6.2.

Message	Meaning	Message Fields
ELECT	for building a spanning tree	*type* : protocol type *src* : computation index of the election
ACK	to acknowledge the receipt of ELECT message	*type* : protocol type *src* : computation index of the election *chd* : denotes if the sender is a child *id* : maximum id as seen by the sender
LD	to announce a leader	*type* : protocol type *src* : computation index of the election *id* : id of leader elected in the election

Table 6.1: Message types used in the protocol.

Variable	Type	Meaning
ϵ	*bool*	indicates whether process is currently in election or not
ldr	$int(1..n)$	*id* of the leader process
num	*int*	index of the last computation that this process started
$src\langle Num, Id\rangle$	$\langle int, int(1..n)\rangle$	computation index of the last computation in which this process participated
p	$int(1..n)$	parent process in last computation
ack	*bool*	indicates if ACK message is sent to parent or not
W	*list*	list of neighbors from which ACK is being awaited
C	*list*	list of my children in current computation
max	$int(1..n)$	maximum *id* among my children in current computation

Table 6.2: Variables maintained by each process in the protocol.

6.1.3.1 Description of the protocol

When the process i detects the failure of the leader, it sets the value of the variable *startElection* to true to start the instance of the *ldrId* protocol to elect a new leader. The fraction at i begins the election by starting a diffusing computation by sending an ELECT message to all of its neighbors. The ELECT message has a field that contains the *computation-index* of the diffusing computation. When a process receives the ELECT message, it sets the neighbor from which it first received the message as its parent. It then propagates the ELECT message to all of its neighbors. In this way, during the first phase a spanning tree is build.

When a fraction i receives an ELECT message from a process that is not its parent, it immediately responds by sending an ACK message. The ACK message has a field that identifies if the message is from a child or not, and a field that denotes the maximum id as seen by the process. The fraction i sends an ACK message to its parent only when it

has received ACK messages from all of its neighbors. When a fraction i has received ACK messages from all of its neighbors, it first finds the process having maximum id among all its children. It then sends the ACK message to its parent.

When the source process that started the computation (election) receives ACK messages from all of its neighbors, it can compute the leader by finding the process having maximum id among all its children. It then starts a diffusing computation to forward the leader information to all the processes.

As one or more processes can concurrently detect failure of a leader, it is possible that more than one process can start elections independently, thereby, leading to concurrent diffusing computations. To ensure correctness of the protocol, it is required that each process participate in only one diffusing computation. This is done by associating a *computation-index* to each computation. The computation-index is a pair $\langle num, id \rangle$, where id represents the identifier of the process, and num is an integer. When a process participating in a diffusing computation, receives another diffusing computation with higher computation-index, it stops participating in the current computation in favor of the diffusing computation with higher computation-index. Two computation-index are compared as follows:

$\langle num_1, id_1 \rangle > \langle num_2, id_2 \rangle \Leftrightarrow ((num_1 > num_2) \vee ((num_1 = num_2) \wedge (id_1 > id_2)))$.

6.1.4 Leader Election based on Process Value

The leader election protocol, *ldrVal* that computes the leader based on the value of a process is shown in Figure 6.2. This protocol satisfies the following liveness property in addition to the safety and liveness properties discussed in Section 6.1.2:

Component *ldrVal*

Fraction i

inp $value$: **int**

N : **list**

var $startElection, \epsilon, ack$: **bool** {initially *false*}

p, num, ldr : **int** {initially $p = num = 0, ldr = \perp$}

$src\langle Num, Id\rangle, max\langle Val, Id\rangle$: $\langle$**int,** $1..n\rangle$ {initially $\langle 0, i\rangle$}

N, W, chd : **list** {initially $W, chd = \phi$}

begin

startElection : $\neg\epsilon \wedge startElection \rightarrow$

$src, num := \langle num, i\rangle, num + 1;$

ELECT.$src := i;$

for $k = 1$ **to** $n \wedge k \in N$

$C_{i,k}$.add(ELECT);

$W, ack, \epsilon, p, max, chd := N, true, true, i, \langle value, i\rangle, \phi;$

$startElection := false$

[] **joinElection** : ELECT($ldrVal$) $\in C_{j,i} \rightarrow$

$C_{j.i}$.remove(ELECT);

if $\neg\epsilon \vee (\epsilon \wedge$ ELECT.$src > src)$ **then**

$src :=$ ELECT.$src;$

$num := src.Num + 1;$

for $k = 1$ **to** $n \wedge (k \neq j \wedge k \in N)$

$C_{i,k}$.add(ELECT);

$W, ack, \epsilon, p, chd := N - \{j\}, true, true, j, \phi;$

$max := \langle value, i\rangle;$

else if $\epsilon \wedge src =$ ELECT.src **then**

ACK.$\{src, chd\} := src, false;$

$C_{i,j}$.add(ACK)

fi

[] **ackToParent** : $\epsilon \wedge W = \phi \wedge src.Id \neq i \wedge ack \rightarrow$

$ack := false;$

ACK.$\{src, chd, val, id\} := src, true, max.\{Val, Id\};$

$C_{i,p}$.add(ACK)

Figure 6.2: Leader election algorithm based on node value.

Liveness of $ldrVal$: $\Box\Diamond(i.ldr = j \Rightarrow j.value = max\{k.value \mid d_{i,k} < \infty\}$,

which asserts that the elected process is the process having maximum value among all connected processes.

```
[] ackReceive  : ϵ ∧ ACK(ldrVal) ∈ C_{j,i} ∧ ack →
                        C_{j,i}.remove(ACK);
                        if src = ACK.src then
                          W := W − {j};
                          if ACK.chd then
                            chd = chd + {j};
                            max := MAX(⟨ACK.val, ACK.id⟩, max)
                          fi
                        fi
[] electLeader : ϵ ∧ W = φ ∧ src.Id = i ∧ ack →
                        ack, ϵ, ldr := false, false, max.Id;
                        LD.{src, id} := src, max.Id;
                        for k = 1 to n ∧ k ∈ chd
                          C_{i,k}.add(LD)
[] setLeader   : ϵ ∧ LD(ldrVal) ∈ C_{j,i} ∧ ¬ack →
                        C_{j,i}.remove(LD);
                        if src = LD.src then
                          ldr, ϵ := LD.ld, false;
                          for k = 1 to n ∧ k ∈ chd
                            C_{i,k}.add(LD)
                        fi
end
```

Figure 6.2: Leader election algorithm based on node value (Continued).

The value of a process is calculated based on the resources available at the process, such as, battery power, CPU load, distance to other processes, and degree of the process. The leader election protocol is independent of how the value of a process is calculated. We assume that there is a separate component that monitors the resources at a process and computes the value of the process; and the leader election protocol fraction at the process has access to this value.

6.1.4.1 Description of the protocol

The *ldrVal* protocol is similar to the *ldrId* protocol described earlier where the leader is computed based on the process whose id is maximum. The *ldrVal* protocol is different from *ldrId* protocol in the following ways:

- The *ldrVal* protocol has an additional input variable, $value$ that indicates the value of the process.
- The max variable in *ldrVal* is represented as a pair $\langle Val, Id \rangle$ and is of type $\langle int, int(1..n) \rangle$; $max.Id$ denotes the id of the process and $max.Val$ represents the value of the process.
- The ACK message of *ldrVal* has a field, ACK.val that carries the information about the process having maximum value among all of its children. This field is of type: $\langle int, int(1..n) \rangle$. The equivalent field in ACK message of *ldrId* is ACK.id, which is of type int. In the case where two processes have the same value, the tie is broken in favor of the process having higher id. We have the following property:

  $max_1 > max_2 \Leftrightarrow ((max_1.Val > max_2.Val) \vee ((max_1.Val = max_2.Val) \wedge max_1.Id > max_2.Id)))$.

6.2 Adaptation

We now consider adaptation that dynamically replaces the leader election protocol *ldrId* to the leader election protocol *ldrVal*. When the adaptation is initiated, an instance of *ldrId* can be underway, or it is also possible that no instance of *ldrId* is underway. If the

processes are not engaged in election, then replacing one leader election protocol to another leader election protocol can be done easily; by replacing the corresponding fractions and initializing the new fractions using initial, identity or quasi state mapping.

However, it is impossible for a process to locally determine whether other processes are in election or not. One way to deal with this is to use a centralized control during adaptation and/or enforce synchrony by *quiescing* the processes (blocking the processes from starting new election). This type of approach causes service interruption where the leader election service is not available till the adaptation is completed. Also, there is communication overhead because of extra messages required for synchronization.

We consider mixed-mode adaptation that lets the old and the new protocol overlap during adaptation. As a result, a process where the new fraction is installed, can start the election right away without waiting for other processes to finish the adaptation. Thus, the time for which the application is blocked or the service interruption time is reduced. Moreover, since all processes can perform replacement of fractions independently (in parallel), there is a little or no need for extra messages to achieve synchrony. Thus, the communication overhead during adaptation is reduced.

The program using *ldrId* protocol is shown in Figure 6.3, and the program using *ldrVal* protocol is shown in Figure 6.4. We first give the specification during adaptation for the adaptation that replaces *ldrId* with *ldrVal*.

Specification during adaptation. The specification during adaptation is that the safety property of leader election protocols as discussed in Section 6.1.2 continues to be true during adaptation.

Adapting *ldrId* protocol to *ldrVal* protocol requires replacing each fraction of *ldrId*

Program $\mathcal{P}_{ldrId}$

process $i[i = 1, .., n]$

begin

$ldrId.i$.**startElection**

[] $ldrId.i$.**joinElection**

[] $ldrId.i$.**ackToParent**

[] $ldrId.i$.**ackReceive**

[] $ldrId.i$.**electLeader**

[] $ldrId.i$.**setLeader**

end

Figure 6.3: Program using $ldrId$.

Program $\mathcal{P}_{ldrVal}$

process $i[i = 1, .., n]$

begin

$ldrVal.i$.**startElection**

[] $ldrVal.i$.**joinElection**

[] $ldrVal.i$.**ackToParent**

[] $ldrVal.i$.**ackReceive**

[] $ldrVal.i$.**electLeader**

[] $ldrVal.i$.**setLeader**

end

Figure 6.4: Program using $ldrVal$.

with the corresponding fraction of $ldrVal$. The replacement of fraction at each process is an atomic adaptation and is modeled by an adaptive action. The intermediate programs that occur during adaptation are as shown in Figure 6.5.

Program $\mathcal{P}_{ip(ldrId-ldrVal)}$

process $i[i = 1, .., n; i \neq j]$

$\mathcal{P}_{ldrId}.i \cup aa_i : true \rightarrow TransformTo(\mathcal{P}_{ldrVal}.i, \Phi_{aa_i})$

process $j[j = 1, .., n; i \neq j]$

$\mathcal{P}_{ldrVal}.j$

Figure 6.5: Intermediate program during adaptation from $\mathcal{P}_{ldrId}$ to $\mathcal{P}_{ldrVal}$.

In this example, the fractions of the two protocols at all processes are similar. Also, as

discussed later in this section, the overlap communication scenarios for each intermediate program are similar. We take advantage of this symmetry by modeling the adaptation as an adaptive program (cf. Section 3.2.2 (adaptation as an automaton) of Chapter 3).

To model each adaptive action, we introduce two new boolean variables $idActive$ and $valActive$. We do *restriction composition* [58] of $ldrId$ program with $idActive$, and of $ldrVal$ with $valActive$. A restriction of program P by Z is a program whose actions are of the form $Z \wedge g \rightarrow st$, for each action $g \rightarrow st$ of P. Thus, actions of $ldrId$ will be enabled only if $idActive$ is $true$, and actions of $ldrVal$ will be enabled only if $valActive$ is true. Initially, when the old program is running, $idActive$ is $true$ and $valActive$ is $false$ at each process. The adaptive action at each process atomically sets $idActive$ to $false$ and $valActive$ to $true$. The adaptation modeled as an adaptive program is shown in Figure 6.6.

We allow these adaptive actions to execute independently of each other. In other words, at each process the fraction of $ldrId$ protocol can be replaced by the fraction of $ldrVal$ protocol independent of other processes. During this replacement, the state of the new fraction needs to be initialized appropriately to ensure correctness of adaptation.

Furthermore, independent replacement of fractions at various processes will lead to a situation where during adaptation some of the processes have fractions of $ldrId$ active and some of the processes have fractions of $ldrVal$ active. This will cause overlap of communication between the two protocols. The communication between two protocols need to be handled appropriately to ensure correctness of adaptation.

The correctness of adaptation requires that the safety property, which asserts that two processes not in the election cannot have different leaders, also needs to be satisfied during

Program $\mathcal{P}_{ldrId-ldrVal}$

process $i[i = 1, .., n]$

var $idActive$: **bool** {initially $true$}
$valActive$: **bool** {initially $false$}
$ldrId.i.\{\textbf{var}\} \cup ldrVal.i.\{\textbf{var}\}$

begin

$idActive \wedge ldrId.i.\textbf{startElection}$
[] $idActive \wedge ldrId.i.\textbf{joinElection}$
[] $idActive \wedge ldrId.i.\textbf{ackToParent}$
[] $idActive \wedge ldrId.i.\textbf{ackReceive}$
[] $idActive \wedge ldrId.i.\textbf{electLeader}$
[] $idActive \wedge ldrId.i.\textbf{setLeader}$
[] $valActive \wedge ldrVal.i.\textbf{startElection}$
[] $valActive \wedge ldrVal.i.\textbf{joinElection}$
[] $valActive \wedge ldrVal.i.\textbf{ackToParent}$
[] $valActive \wedge ldrVal.i.\textbf{ackReceive}$
[] $valActive \wedge ldrVal.i.\textbf{electLeader}$
[] $valActive \wedge ldrVal.i.\textbf{setLeader}$
[] aa_i : $idActive \rightarrow idActive, valActive := false, true;$
$s' := \Phi_{aa_i}(s)$
[] **discardElect** : $valActive \wedge \texttt{ELECT}(ldrId) \in C_{j,i} \rightarrow C_{j,i}.\texttt{remove(ELECT)}$
[] **discardAck** : $valActive \wedge \texttt{ACK}(ldrId) \in C_{j,i} \rightarrow C_{j,i}.\texttt{remove(ACK)}$
[] **acceptLd** : $valActive \wedge \texttt{LD}(ldrId) \in C_{j,i} \rightarrow true \vee ldrVal.i.\textbf{setLeader}$

end

Figure 6.6: Adaptive program to adapt from $\mathcal{P}_{ldrId}$ to $\mathcal{P}_{ldrVal}$.

adaptation. Furthermore, when all processes have finished replacing their fractions, the program should be in a reachable (invariant) state of the $ldrVal$. This is required to ensure that once the adaptation is complete, the new program satisfies its specification.

We now describe the state mapping that we use in the adaptation, and also discuss various overlapping communication scenarios that arise during the adaptation and how we deal with them.

6.2.1 State Mapping and Overlap Communication

During adaptation, a process replaces the fraction of *ldrId* with the fraction of *ldrVal*. Since the protocol fractions are replaced independently at each process, we need to explicitly deal with the communication between the two protocols. A process with the old fraction may receive a message from the process with the new fraction and vice-versa. In Table 6.3, we show different overlap communication scenarios that can occur during adaptation and how they are dealt. Any message of type `ELECT` from the new fraction to the old fraction is buffered. Any message of type `ELECT` or `ACK` from the old fraction to the new fraction is discarded. The new fraction accepts a message of type `LD` from the old fraction. The scenarios where the new fraction sends a message of type `ACK` or `LD` to the old fraction do not arise because the new fraction always discards any message of type `ELECT` from the old fraction. We introduce the following actions: `discardElect`, `discardAck`, and `acceptLd` as shown in Figure 6.6 to deal with overlap communication scenarios during adaptation.

Message Type / *Overlap scenario*	ELECT	ACK	LD
Old (*ldrId*) to New (*ldrVal*)	discard	discard	accept
New (*ldrVal*) to Old (*ldrId*)	buffer	NA	NA

Table 6.3: Overlap communication between protocols.

To ensure correctness while dealing with overlap communication, the new fractions are initialized using the state mapping defined in the Table 6.4. When the replacement is done, the old fraction is in any one of the four states as described in the Table 6.4. As described in the first case, if the old fraction is not involved in the election, then identity mapping is

done, *i.e.*, each variable of the new fraction is initialized to the corresponding variable of the old fraction. In the second case, the old fraction is involved in the election, and is waiting to receive ACK from its children. In this case, the old fraction can locally determine that the source process has not yet elected the leader. The new fraction in this case is initialized to its initial state, *i.e.*, it is not taking part in election any more. The num and src variables of the new fraction are assigned the same value as that of the old fraction. In the third case, the fraction is waiting to receive the leader value (*i.e.*, LD message) from the parent. In this case, an identity mapping is done. In the last case, the process is the source of the election, *i.e.*, it started the election. In this case, we replace the fraction and have the new fraction start the election again. We initialize the state of the new fraction to its initial state, and num variable of the new fraction is initialized to the same value as that of the old fraction. For brevity, in Figure 6.6 we show the state mapping action as $s' := \Phi_{aa_i}(s)$. This is equivalent to a set of assignment statements that assign values to each variable of the new fraction using the state mapping Φ_{aa_i} of Table 6.4.

(old) state s of process i	**Description of (old) state s**	**(new) state s' of process i**
$\neg ldrId.\epsilon$	not in election	Identity mapping
$ldrId.\epsilon \wedge ldrId.W \neq \phi \wedge ldrId.src \neq i$	in election and waiting for ACK from children and is not source of election	$\{num, src\}$ - Identity mapping; Initial mapping
$ldrId.\epsilon \wedge ldrId.W = \phi$	in election and waiting for LD	Identity mapping
$ldrId.\epsilon \wedge ldrId.W \neq \phi \wedge ldrId.src = i$	in election and waiting for ACK from children and is source of election	$\{num, \epsilon\}$ - Identity mapping; $\{ldrVal.startElection := true$; - Functional mapping; Initial mapping

Table 6.4: State mapping (Φ_{aa_i}) for each atomic adaptation aa_i.

6.2.2 Verifying Adaptation

To verify adaptation, we need to verify all intermediate programs that occur during adaptation. The intermediate programs that occur during the adaptation from $ldrId$ to $ldrVal$ are as shown in Figure 6.5. Since the protocol fractions and the overlap communication scenarios are symmetrical, we modeled the adaptation as the adaptive program shown in Figure 6.6. Thus, instead of verifying all intermediate programs, we verify the adaptive program of Figure 6.6.

Safety of adaptive program. The safety property for the adaptive program $\mathcal{P}_{ldrId-ldrVal}$ is similar to the safety property of the leader election protocols, and given as follows:

$$\forall p_1, p_2 : p_1, p_2 \in \{ldrId, ldrVal\} : p_1.i.ldr \neq p_2.j.ldr \Rightarrow p_1.i.\epsilon \vee p_2.j.\epsilon.$$

Invariant. We establish the following invariant for the adaptive program that implies the safety is not violated during adaptation:

$I \equiv P_E \wedge P'_E \wedge P_L \wedge P'_L$, where

$$P_E \equiv i.idActive \wedge j.valActive \wedge j.\epsilon \wedge j.ack \wedge j.W \neq \phi \Rightarrow \texttt{ELECT}(ldrVal) \in C_{j,i}$$

$$P'_E \equiv j.valActive \wedge \neg j.\epsilon \wedge (\exists i : i \in j.N : i.idActive \wedge j.src = i.src \wedge i.\epsilon \wedge i.ack \wedge C_{i,j} = \phi)$$
$$\Rightarrow \exists k : k.p = k \wedge k.\epsilon \wedge k.ack \wedge k.W \neq \phi$$

$$P_L \equiv i.idActive \wedge j.valActive \wedge j.\epsilon \wedge j.p = i \wedge \neg j.ack \wedge j.W = \phi$$
$$\Rightarrow ((i.\epsilon \wedge i.src = j.src \wedge \neg i.ack \wedge i.W = \phi) \vee$$
$$(\neg i.\epsilon \wedge \texttt{LD}(ldrId) \in C_{i,j}) \vee$$
$$(i.\epsilon \wedge i.ack \wedge i.W \neq \phi \wedge i.src = j.src) \vee$$
$$(i.\epsilon \wedge i.ack \wedge i.W \neq \phi \wedge i.src \neq j.src \Rightarrow j \in i.W))$$

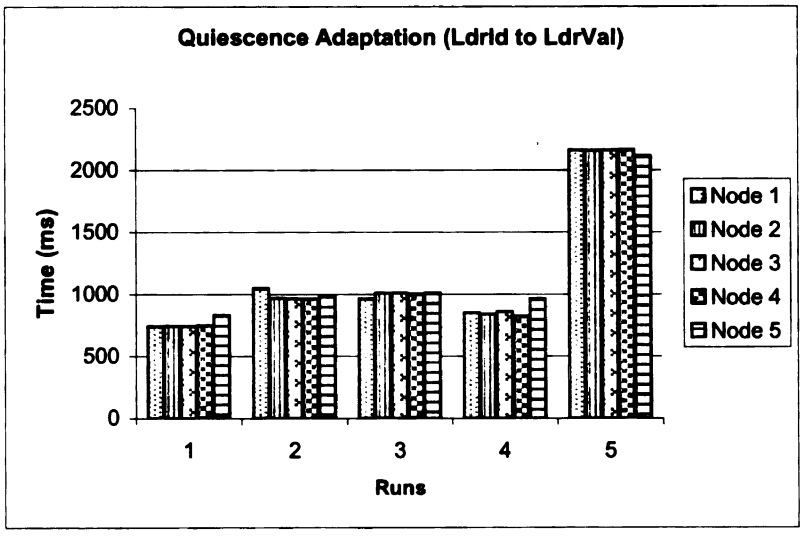
$$P'_L \equiv i.idActive \wedge j.valActive \wedge i.\epsilon \wedge \neg j.\epsilon \wedge i.p = j \wedge \neg i.ack \wedge i.W = \phi \wedge i.src = j.src$$
$$\Rightarrow \texttt{LD}(ldrId) \in C_{j,i} \vee \exists k : k.p = k \wedge k.\epsilon \wedge k.ack \wedge k.W \neq \phi$$

In the above invariant, P_E asserts that if process j starts an election after adapting to *ldrVal* fraction, then any ELECT message that j sends to process i, which is still using *ldrId* fraction, remains in the channel $C_{j,i}$ until i gets adaptated. P'_E asserts that if j was in election and waiting for ACK(s) from its neighbor(s) when it replaced its fraction then the source of that election is still in the election. P_L asserts that if process j was in election and waiting for LD when it replaced its fraction and the parent of j is still using the old fraction then one of the following is true: (i) the parent of j is also waiting for LD, (ii) the parent of j is not in election and the channel $C_{i,j}$ has a message LD(*ldrId*), (iii) the parent of j is still waiting for ACK(s) from its neighbor(s), or (iv) if the parent of j has started a new election then it will not receive ACK from j till it replaces its fraction. P'_L asserts that if a process i using the old fraction is waiting for a message LD from its parent j, and j has completed its election and is now using the new fraction, then one of the following is true: (i) the channel $C_{j,i}$ has a message LD(*ldrId*) which was sent by process j before it got adapted, or (ii) the source of that election is still in the election.

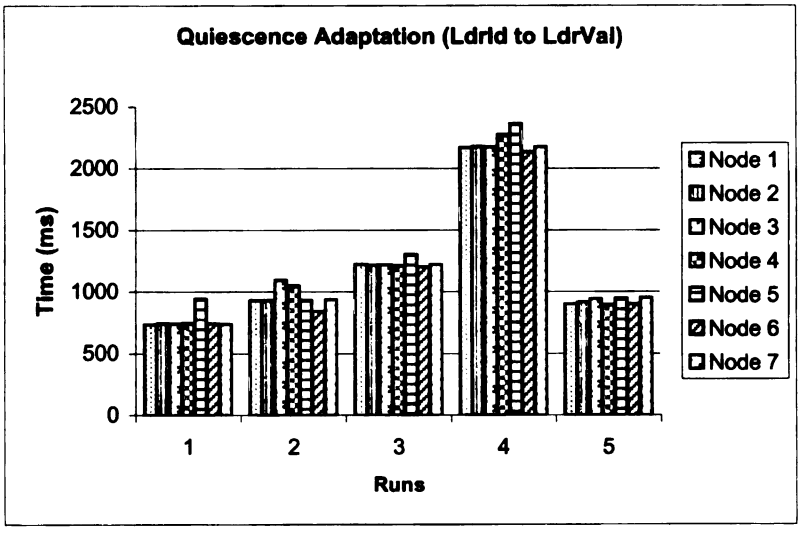
6.3 Performance of Mixed-Mode Adaptation

In this section, we compare the performance of mixed-mode and quiescence adaptation. We consider two different configurations for the following discussion: (2) a connected network (straight line) of 5 processes and 4 edges, and (2) a connected network of 7 processes and

11 edges. We have considered other network topologies and obtained similar results. In this discussion, we consider adaptation from *ldrId* to *ldrVal*. We have also implemented adaptation from *ldrVal* to *ldrId* and obtained similar performance results.



(a) Configuration 1



(b) Configuration 2

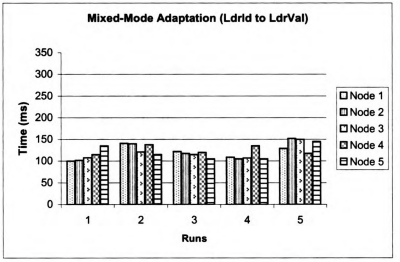
Figure 6.7: Quiescence adaptation.

Figure 6.7 shows the time required for quiescence adaptation to adapt from Id-based leader election protocol to value-based leader election protocol. It shows two configura-

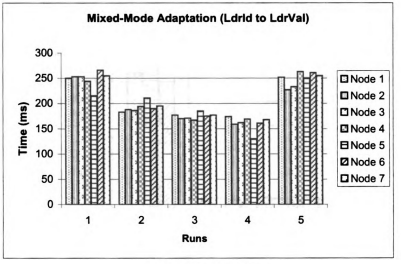
tions and 5 runs for each configuration. In each run it shows the time taken by each process for adaptation. In run number 5 of configuration 1 and run number 4 of configuration 2, the time taken for adaptation is almost twice the average time taken by the process over other runs. This is because when the adaptation started the instance of the leader election protocol is already running. The adaptation waits for the election to complete before replacing the protocol.

Figure 6.8 shows the time required for the mixed-mode adaptation to adapt from Id-based leader election protocol to value-based leader election protocol. It shows two configurations and 5 runs for each configuration; and in each run the time taken by each process to finish the adaptation. The time taken in each run is almost the same for a given configuration regardless of whether the instance of leader election protocol is underway or not when adaptation occurs.

Figure 6.9 shows the comparison between the average time taken by quiescence and mixed-mode adaptation to adapt from Id-based leader election protocol to value-based leader election protocol. The average time taken by quiescence adaptation is almost 8 times that of mixed-mode adaptation in the case of configuration 1, and 6 times that of mixed-mode adaptation in the case of configuration 2. The result is as expected, because the quiescence adaptation sends more messages for synchronization during adaptation. Furthermore, in the case of quiescence adaptation there is more processing time required at each process during adaptation as channels and other resources used by the existing protocol at that process need to be released before the new protocol can be installed. On the contrary, in the case of mixed-mode adaptation the new protocol is able to deal with overlap communication and hence explicit release of channels and other resources is not necessary.



(a) Configuration 1

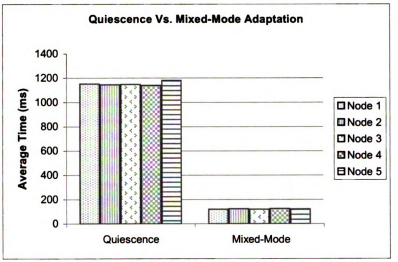


(b) Configuration 2

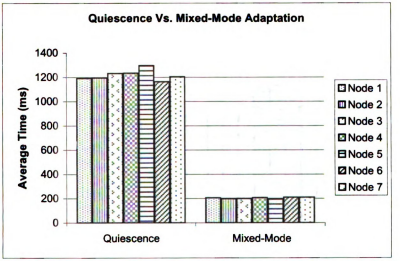
Figure 6.8: Mixed-mode adaptation.

Figure 6.10 shows the time required for electing a leader by Id-based and value-based leader election protocols. It is clear that both the protocols take almost the same amount of time to do an election.

Furthermore, from Figures 6.9 and 6.10 it can be observed that time taken for quiescence adaptation is more than twice the time it takes for leader election. Also, the time



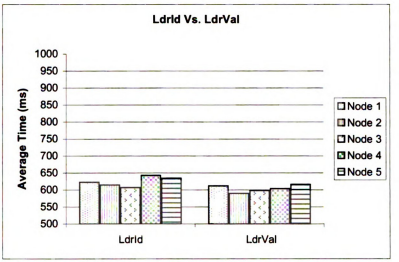
(a) Configuration 1



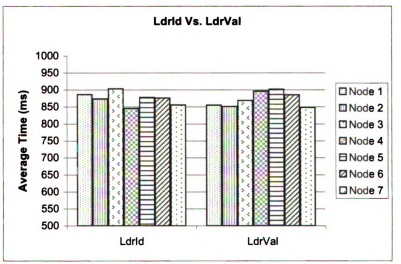
(b) Configuration 2

Figure 6.9: Quiescence vs. mixed-mode adaptation.

for mixed-mode adaptation is less than one-fourth of the time it takes for leader election. Clearly, if a user at some process requests an election at the same instant when the system decided to adapt (using quiescence adaptation), then user would notice a long delay which could be up to twice the time it normally experiences for an election. However, if mixed-mode adaptation is chosen, the adaptation would be almost transparent to the user.



(a) Configuration 1



(b) Configuration 2

Figure 6.10: Time for leader election protocols.

6.4 Limitations of Mixed-Mode Adaptation

One of the limitations with mixed-mode adaptation is that it requires the adaptation developer to have a deeper knowledge of the components involved in adaptation. Additionally, such adaptation may require support from components themselves. Nonetheless, in our experience, we find that components involved in mixed-mode adaptation exhibit var-

ious levels of mixed-mode behaviors. Consequently, based on the details available about the components involved in adaptation, an adaptation developer can provide an appropriate mixed-mode behavior during adaptation. However, in cases where components are not conducive in the development of mixed-mode adaptation, adaptation based on system structure such as quiescence adaptation may be more appropriate.

Chapter 7

Tradeoffs in Adaptation

In this chapter we identify various tradeoffs that arise in developing adaptation. We identify tradeoffs in verification complexity, completion time, and communication overhead during adaptation. Concurrent executions are generally considered faster than sequential executions. Specifically, with respect to adaptation, we expect that concurrent execution of atomic adaptations (if possible after considering any dependencies) would be faster than sequential execution. However, verification complexity increases exponentially with increase in concurrency, and also message communication overhead increases with increase in concurrency. In the rest of this chapter, we first discuss tradeoff between concurrency and verification complexity in Section 7.1. In Section 7.2, we discuss tradeoffs between concurrency and communication overhead. Finally, we discuss a simple casestudy to demonstrate tradeoffs in adaptation in the publish-subscribe application.

7.1 Concurrency v/s Verification Complexity

As discussed in Chapter 3, to verify a given adaptation, we consider all possible orderings of concurrent atomic adaptations. As a result, in the lattice, we have multiple paths from start node to end node to encompass all possible orderings among concurrent atomic adaptations. This increases the number of intermediate programs that need to be verified.

Putting concurrent atomic adaptations in various possible orderings is a potential cause of the explosion in the size of the lattice. For example, if an adaptation consists of n atomic adaptations that can be executed concurrently, then there are $n!$ different orderings and $2^n - 2$ different intermediate programs. Thus, $2^n - 2$ transitional-invariants need to be identified corresponding to each intermediate program. The lattice in this case is as shown in Figure 7.1(a) for $n = 3$. To identify all these transitional-invariants and verify the corresponding intermediate programs is a difficult process.

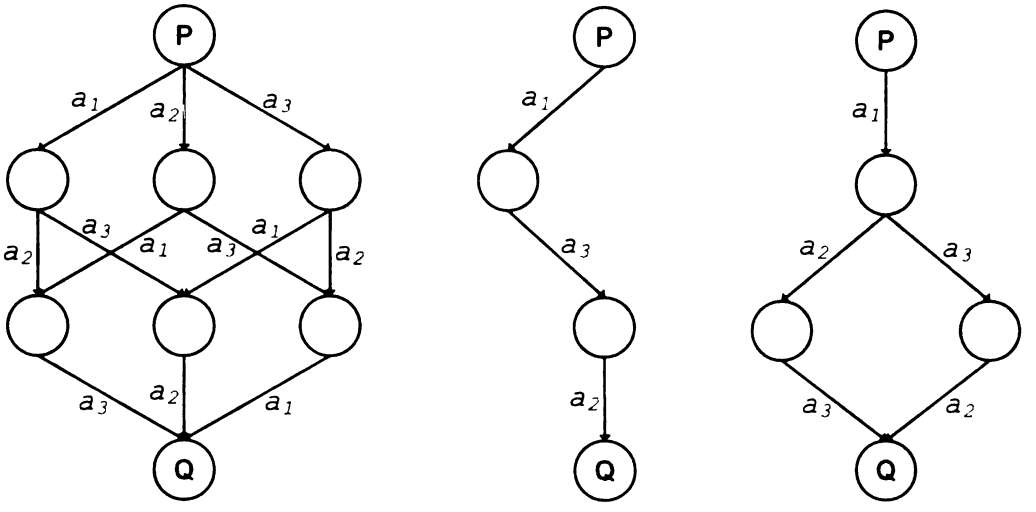


Figure 7.1: Executing three atomic adaptations.

Clearly, the specification during adaptation is satisfied, if the adaptation follows any

path in the lattice. So instead of choosing to verify all adaptation paths, if we verify only one path (*e.g.*, a_1, a_3, a_2), then the lattice would be as shown in Figure 7.1(b). For specification during adaptation to be satisfied the adaptation must follow this path, *i.e.*, a_1 should occur before a_3, and a_3 should occur before a_2. In this case there is no concurrency among adaptive actions during adaptation, and we are able to reduce the cost of verification from $O(2^n)$ to $O(n)$. Specifically, for n concurrent atomic adaptations, the number of transitional-invariants that need to be identified is reduced to $n - 1$.

Alternatively, we could have chosen the lattice as shown in Figure 7.1(c). In this case the cost of verification is more compared to the lattice in Figure 7.1(b), but less compared to the lattice in Figure 7.1(a). Also, the concurrency in the lattice in Figure 7.1(c) is more compared to the lattice in Figure 7.1(b), but less when compared to the lattice in Figure 7.1(a).

Thus, based on the tradeoff between concurrency of adaptation and complexity of verifying that adaptation, we can choose a subgraph (sublattice) of a given lattice that has all the properties of the lattice defined in Chapters 4 and 5. If we verify only a sublattice, we also need to constrain the adaptation so that it follows only path of the sublattice.

7.2 Concurrency v/s Message Complexity

Many systems are limited by communication overhead and message delays. Specifically, for wireless and mobile systems, energy-communication tradeoff may require system to reduce communication overhead whenever possible. For designing adaptation in such systems, communication overhead should also be taken into account. In this section, we show

how concurrency during adaptation affects the communication overhead.

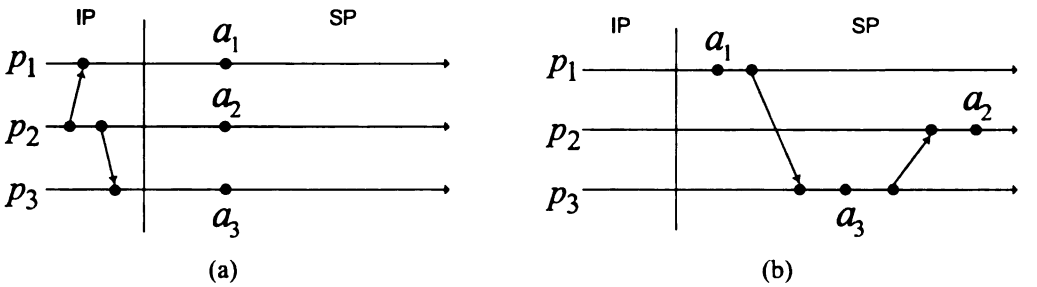


Figure 7.2: Space-time diagram of adaptation.

Consider the case where all atomic adaptations are executed concurrently. This is described by the lattice in Figure 7.1(a), and the space-time diagram for this adaptation is shown in Figure 7.2(a). We show only the minimum number of adaptation-specific messages in the space-time diagram. There may be other application-specific messages that we do not consider as they are not related to adaptation. We divide adaptation into two phases: (i) *initialization phase*, and (ii) *synchronization phase*. In the initialization phase (denoted by `IP` in the figures) the initiator process that decides on the adaptation informs other processes of this decision. In the synchronization phase (denoted by `SP` in the figures) processes exchange messages to co-ordinate the execution of adaptive actions.

In Figure 7.2(a), process p_2 is the initiator that informs other processes to start performing any steps required for adaptation. In this case, there are at least two messages required to initiate adaptation. Now, if the adaptation were to occur according to the lattice of Figure 7.1(b), then we can make process p_1 as the initiator, and the space-time diagram would be as shown in Figure 7.2(b). In this case, we got rid of the initialization messages that were required for the adaptation described by Figure 7.2(a). In both the cases, the minimum number of adaptation-specific messages required during adaptation is two. Thus, we

did not increase any communication overhead by reducing concurrency. We now consider another scenario where the number of messages can actually be reduced if concurrency is reduced during adaptation.

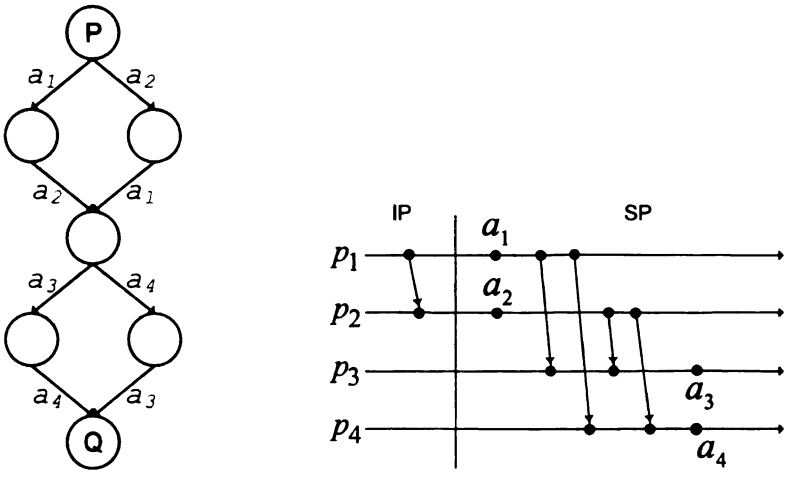
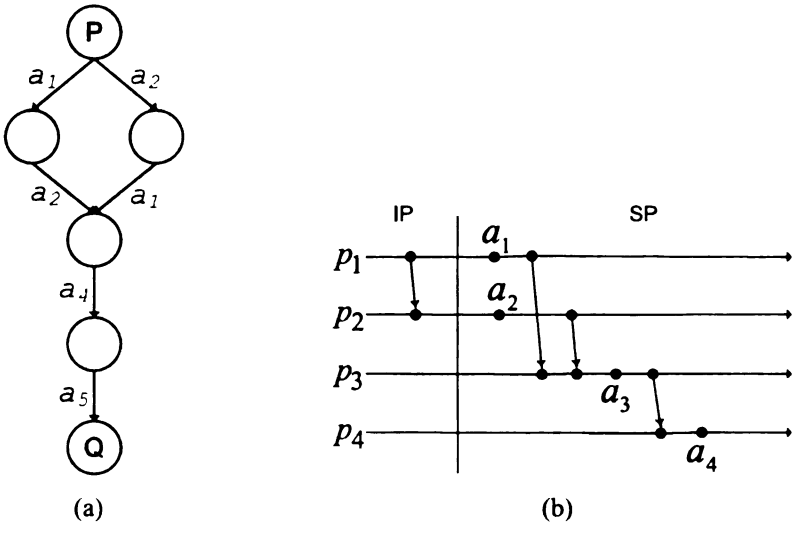


Figure 7.3: Adaptation with concurrency.

Consider the lattice of Figure 7.3(a) that describes the adaptation consisting of four adaptive actions a_1, a_2, a_3, and a_4 occurring at processes p_1, p_2, p_3, and p_4 respectively. Adaptive actions a_1 and a_2 are independent of each other and can occur concurrently. Similarly, a_3 and a_4 can occur concurrently. The corresponding space-time diagram is shown in Figure 7.3(b). The minimum number of adaptation-specific messages required during adaptation is five. Now, if were to reduce concurrency in this case, we can have the adaptation that is described by the lattice of Figure 7.4(a), and corresponding space-time diagram as shown in Figure 7.4(b). In this case, the minimum number of adaptation-specific messages required is reduced to four.

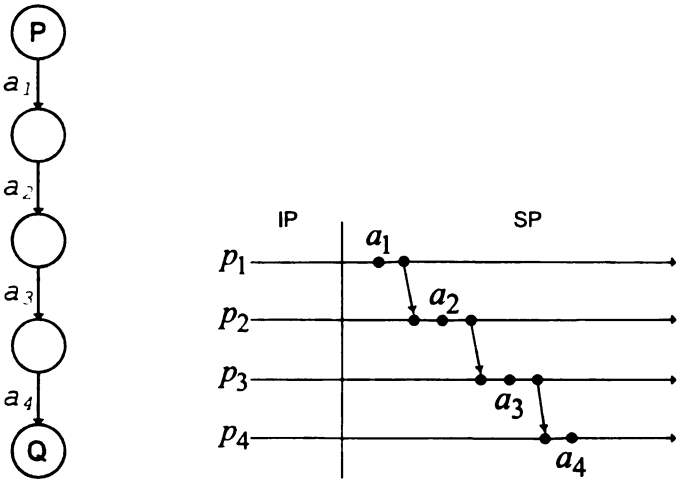
Further, if we have no concurrency during adaptation as described by the lattice of Figure 7.5(a), then the space-time diagram would be as shown in Figure 7.5(b). In this case,



(a) (b)

Figure 7.4: Adaptation with (reduced) concurrency.

a minimum of only three adaptation-specific messages are required during adaptation.



(a) (b)

Figure 7.5: Adaptation with no concurrency.

Thus, by reducing concurrency during adaptation, it is possible to reduce the number of messages required during adaptation. However, from the space-time diagrams of Figure 7.2-7.5, it is clear that time required to complete adaptation would probably be less when there is more concurrency during adaptation. Thus, while designing adaptation, one should consider various factors such as concurrency during adaptation, message delays and

communication overhead, and verification complexity.

7.3 Case Study: Publish-Subscribe Application

In this section, we illustrate the tradeoffs due to concurrent adaptive actions during adaptation using a simple publish-subscribe application. We consider the publish-subscribe application with two publishers (senders) and two subscribers (receivers). Both the receivers subscribe to receive data from both the senders. For reliable communication between publishers and subscribers we consider two protocols, namely, the proactive protocol based on forward error correction and the reactive protocol based on acknowledgments. These protocols are discussed earlier in Chapters 4 and 5.

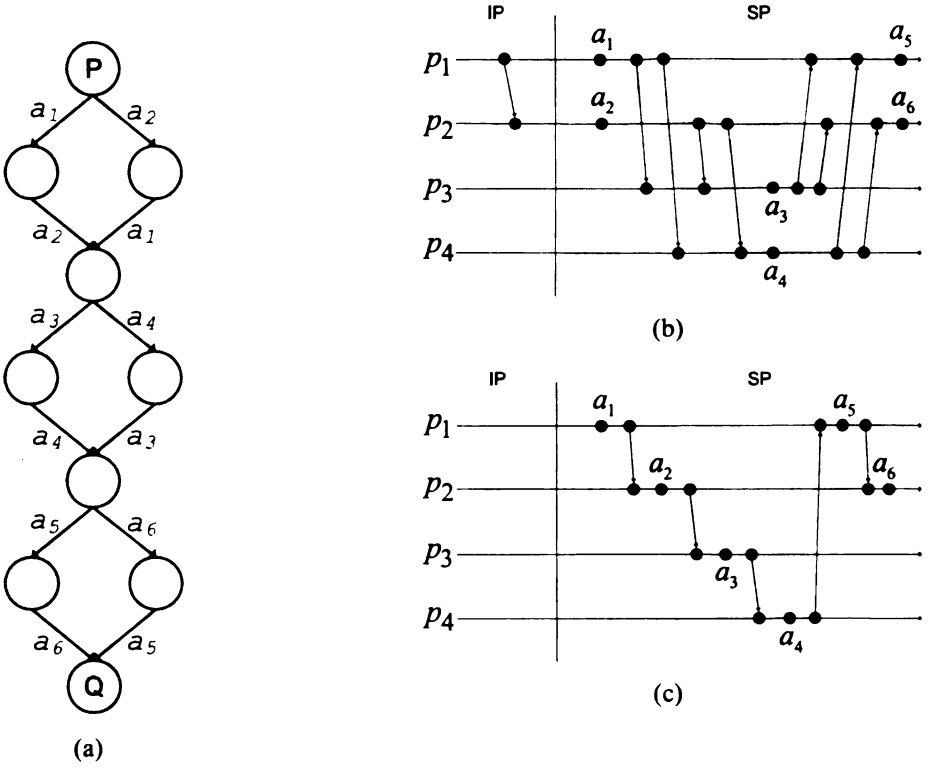


Figure 7.6: Adaptation in publish-subscribe application.

The adaptation in publish-subscribe application replaces the proactive protocol with the reactive protocol. The adaptation is done by first blocking the two senders. The blocking of two senders can be done concurrently, *i.e.*, independent of each other. We note that the local guards of the adaptive actions that block the senders need to be true before they can be executed. As a result, though the two adaptive actions are independent of each other, they may not necessarily execute at the same instant during adaptation. Once the protocol fractions at both the senders are blocked, the protocol fractions at both the receivers can be replaced concurrently. Finally, once the receivers have replaced to the new protocol fractions, the protocol fractions at the senders can be replaced. These replacement of fractions at the senders can also occur concurrently. The adaptation lattice in this case is shown in Figure 7.6(a). The corresponding space-time diagram is shown in Figure 7.6(b). The verification complexity and communication overhead can be reduced for this adaptation as discussed in Section 7.1. Specifically, if all the adaptive actions are serialized then the space-time diagram for the adaptation is as shown in Figure 7.6(c). Clearly, the communication overhead is reduced from a minimum of 9 messages to a minimum of 5 messages. Also, the number of intermediate programs that need to considered for verification of adaptation is reduced from 8 to 5.

Chapter 8

Testing Adaptation

In order to specify and verify the behavior of the system during dynamic adaptation, we presented an approach based on *adaptation lattice* in Chapters 3, 4, and 5. Due to complexity of the adaptive systems, the verification is often done on an abstract model of the system. In this chapter, we present an approach for testing adaptation to gain assurance about the implementation of adaptation.

Predicate detection is a common approach used in testing and debugging of distributed systems, as many problems in distributed systems can be formulated as an instance of *Global Predicate Evaluation (GPE)* [78]. Typical properties of distributed systems such as deadlock detection, mutual exclusion, termination and many more properties can be tested using predicate detection techniques. Numerous approaches [78–83] have recognized a variety of predicate classes and presented algorithms for predicate detection. In this paper, we discuss predicate detection approach for testing adaptive systems.

Due to overlapping behavior of the old program and the new program during adaptation, the existing algorithms for predicate detection cannot be applied directly. Specifically, these

algorithms do not deal with the system whose code is being changed. In many cases, the algorithms can be modified so that if any error is detected during adaptation then it can be mapped to a particular step of adaptation that caused the error.

With this motivation, we extend the existing algorithms to test the system during adaptation. In particular, we classify the predicates to be detected during adaptation into two types: (i) *adaptation-stable* predicates, and (ii) *adaptation-transient* predicates. We call a predicate adaptation-stable if the predicate holds throughout during adaptation, and call it adaptation-transient if it holds only in some interval during adaptation. We show how existing algorithms can be modified to detect both these types of predicates during adaptation. Furthermore, we show how we can reduce the cost of testing by testing only atomic adaptations.

The rest of the chapter is organized as follows. In Section 8.1, we review preliminary concepts of distributed computation, causal precedence, consistent global state and consistent cut. Then, we introduce *adaptation vector* in Section 8.3. We also give a brief overview of vector clocks used to track causality among events in Section 8.3. Subsequently, in Section 8.4, we discuss algorithms to test adaptation. To reduce the cost of testing, in Section 8.5, we identify a subset of states during adaptation to do limited testing. Finally, we give a summary of this chapter in Section 8.6.

8.1 Preliminaries

As discussed in Chapter 3, a program $\mathcal{P}$ consists of a set of n processes $\{p_1, p_2, ..., p_n\}$ communicating via asynchronous messages on interprocess channels. We do not assume

the channels to be FIFO (unless the application requires so). Furthermore, in this chapter, we do not specify the channel states explicitly; a channel state can be constructed from considering the local states of the processes. The execution of a process consists of a sequence of *events*. An event is the execution of a process *action*. An event is one of the three types: local (or internal) event, send event or receive event. A action corresponding to send event has a statement of the form: `send` (m) `to` p_i. A receive event has a corresponding action that has the following form: `rcv` (m) `from` $p_i \rightarrow stmts$.

We now present formal definitions of distributed computation, causal precedence, consistent global state and consistent cut. In this section, we consider partial order semantics for a program.

Definition (Distributed computation). A *distributed computation* r of a program $\mathcal{P}$ describes a single execution of $\mathcal{P}$ by a collection of traces $r[i]$ for each process p_i. Each $r[i]$ is a finite alternating sequence of *states* and *events*. For example, the trace of process p_i is $s_i^0 e_i^1 s_i^1 e_i^2 ...$, where s_i^k denotes the local state of p_i immediately after event e_i^k, and s_i^0 denotes the initial state before any actions are executed. A distributed computation is commonly depicted using a *space-time diagram* as shown in Figure 8.1.

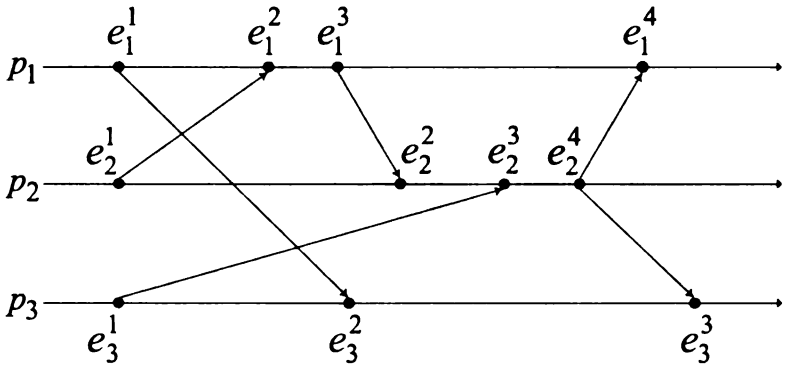


Figure 8.1: Space-time diagram of a distributed computation.

Formally, a distributed computation is a partially ordered set defined by the pair $(E, \rightarrow)$, where E is the set containing all events and $\rightarrow$ is the happened-before relation [84] that defines the causal precedence relation on events (or states). The happened-before relation on states is defined as follows:

Definition (Causal precedence). The state s causally precedes the state t (denoted as $s \rightarrow t$) if and only if one of the following holds:

- if s and t are states on the same process and s occurred before t
- if action following s is the send of a message and the action before t is the corresponding receive
- there exists a state x such that $s \rightarrow x$ and $x \rightarrow t$.

We use the notation $\rightarrow$ to denote the causal-precedence relation for both states and events. If for two events e_1 and e_2, neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$, then e_1 and e_2 are said to be concurrent and denoted by $e_1 \parallel e_2$.

Not all events are relevant for testing of dynamic adaptation, and hence to simplify the testing, we only consider a subset of events of distributed computation. Let $R \subseteq E$ be the set of relevant events. The poset $(R, \rightarrow)$ describes an abstraction of the distributed computation.

Definition (Global state and cut). We defined a global state as a state of all the processes and channels in the system. We, however, ignore the channel states and represent global state as a n-tuple of local states $(s_1^{c1}, ..., s_n^{cn})$. A channel state can be constructed from the set of all messages that have been sent but not received yet.

A *cut* C associated with a global state g is a set of events, one event per process, such that event $e \in C$ if and only if the process state immediately after event e is a part of g. A global history H associated with the cut C (or corresponding global state g) of $\mathcal{P}$ is defined as the subset $h_1^{c_1} \cup ... \cup h_n^{c_n}$, where $h_i^{c_i}$ is the local history of process p_i containing first c_i (*i.e.*, $e_i^1 e_i^2 ... e_i^{c_i}$) events.

Definition (Consistent cut and consistent global state). A cut C is *consistent* if for all events e in the corresponding global history H, we have $(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in H$. A global state g is consistent if the cut corresponding to it is consistent. Intuitively, a consistent global cut corresponds to a view of the run which could be obtained given the existence of a global clock.

Given a space-time diagram, it may not always be possible to say how the global execution actually occurred. For example, it is not clear in Figure 8.1 if a_1^2 or a_3^2 occurred first. In other words, there are many total orders of a distributed computation r. Thus, several possible global executions correspond to a given distributed computation r. We call each total order of r as an *observation*. In other words, a sequence of consistent global states $g_0 g_1 g_2 g_3 ...$ is an observation, where g_0 denotes the initial global state $(s_1^0, ..., s_n^0)$, and each global state g_i is obtained from previous state g_{i-1} by some process executing a single event. For two such global states g_{i-1} and g_i, we say that g_{i-1} *leads-to* g_i. The set of all consistent global states of a computation along with the leads-to relation defines a *lattice of global states* or *computation lattice*. A path in the computation lattice corresponds to an observation, and each observation has a corresponding path in the computation lattice. Thus, the computation lattice represents the set of all possible observations of the computation.

Remark. In the following sections, unless mentioned otherwise, we will use global state to mean consistent global state.

8.2 Testing

Various global properties of the system need to be tested during adaptation. Examples of the properties that can be tested during adaptation include deadlock detection, token loss detection, and in general monitoring. We classify the properties to be checked during adaptation into two categories: (i) *adaptation-stable* properties, and (ii) *adaptation-transient* properties. If a property is to be satisfied for all states during adaptation, then it is known as adaptation-stable property, and if a property is to be satisfied for some interval during adaptation, then it is known as adaptation-transient property. A predicate used to specify adaptation-stable property is known as *adaptation-stable predicate*, and a predicate used to specify adaptation-transient property is known as *adaptation-transient predicate*. From definition, it is easy to observe that adaptation-stable predicates are specified in terms of variables that are not affected (added or removed) due to atomic adaptations. On the contrary, adaptation-transient predicates are specified in terms of variables that are affected (added or removed) at some point during adaptation.

Given a test predicate, a predicate evaluation strategy, in general, would construct and test every global state of the system during adaptation. General predicate testing is normally considered impractical for reasonably big systems, as the number of global states can be exponential in the number of processes.

While testing for arbitrary predicates may be very expensive, efficient testing is feasible

if predicates have certain characteristics. Specifically, in [78–83], authors have identified efficient algorithms for a variety of predicate classes such as conjunctive predicates, disjunctive predicates, observer-independent predicates, stable or unstable predicates. However, in context of testing adaptation, these algorithms either cannot be employed directly or are inefficient if used directly as code of the system is changing during adaptation. In particular, these algorithms need to be extended to classify the global states constructed during adaptation into specific intermediate programs. Towards this end, we introduce *adaptation vector* to distinguish intermediate program states.

8.3 Adaptation Vector

Adaptation vector is used to identify the intermediate program states. An adaptation vector A is a vector of n elements, where n is the number of atomic adaptations. Each element of the adaptation vector, denoted by $A[i]$, is boolean valued (1 represents *true*, 0 represents *false*); $A[i] = 1$ denotes execution of atomic adaptation a_i in past and $A[i] = 0$ denotes that atomic adaptation a_i has not yet executed. For simplicity of discussion, we assume that the adaptation consists of n atomic adaptations $a_1, a_2, ..., a_n$ and atomic adaptation a_i occurs at process p_i. Our approach can be easily extended if multiple atomic adaptations occur at a process.

Each node of the adaptation lattice is assigned an adaptation vector. A value of $[0, ..., 0]$ is assigned to the start node, which denotes the old program where no atomic adaptations have occurred. A value of $[1, ..., 1]$ is assigned to the end node, which denotes the new program reached after all atomic adaptations have occurred. If $[a_1, a_2, ..., a_n]$ is an adap-

tation vector, then the value of $[1, 1, 0, ..., 0]$ denotes an intermediate program reached after execution of atomic adaptations a_1 and a_2. Given an intermediate program and its corresponding adaptation vector, we can determine what atomic adaptations occurred in the past to reach that intermediate program. However, we cannot determine the order (or causal precedence relation) among those atomic adaptations.

8.3.1 Implementation of Adaptation Vectors

Each process keeps its own local adaptation vector AV, which is updated as the process learns about new atomic adaptation. The adaptation vector is maintained as follows:

- When a process p_i executes its own atomic adaptation a_i, it updates the adaptation vector AV_i by $AV_i[i] = 1$.
- When a process p_i sends a message m to process p_j, it attaches the current value of AV_i to m. This value is denoted by $m.AV$.
- When a process p_i receives a message m from process p_j, it updates its adaptation vector value as $AV_i = AV_i \vee m.AV$, where the OR operation over vectors is defined on a component-by-component basis.

8.3.2 Implementation of Vector Clocks

A vector clock system [84, 85] is a mechanism that assigns vector timestamps to each event such that comparing the timestamps of two events indicates the causal relation between two events. Each vector timestamp is of size n, where n is the number of processes. Each process p_i has a vector $V_i[1..n]$ of integers, which is maintained as follows:

- $V_i[1..n]$ is initialized to $[0, 0, ..., 0]$.

- If e_i is a relevant event, then p_i increments its vector clock entry as $V_i[i] := V_i[i] + 1$. It also associates the vector timestamp V_i with the event e_i which is denoted as $e_i.V$.

- When a process p_i sends a message m, it attaches the current value of V_i to m. This value is denoted by $m.V$

- When p_i receives a message m, it updates its vector clock value as $V_i = \max(V_i, m.V)$, where the maximum operator over vectors is defined on a component-by-component basis.

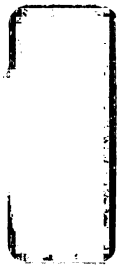
If $e.V$ and $f.V$ are two timestamps associated with distinct events e and f respectively, then the fundamental property associated with vector clocks is described as follows:

$\forall(e, f) \in R \times R : ((e \rightarrow f) \Leftrightarrow e.V < f.V)$, where

$e.V < f.V \equiv (\forall k : (e.V[k] \leq f.V[k]) \wedge \exists k : (e.V[k] < f.V[k]))$.

8.4 Detecting Global Predicates During Adaptation

Given a global state of the system and the value of adaptation vector in that state, we can identify the predicates from the adaptation lattice that should be true in that state. Adaptation-stable predicates need to be checked for all global states constructed during adaptation. The intermediate program in which the adaptation-stable predicate was detected can be easily identified from the value of adaptation vector associated with the state in which the predicate was detected. On the other hand, adaptation-transient predicates need to be checked only in states of the corresponding intermediate program.



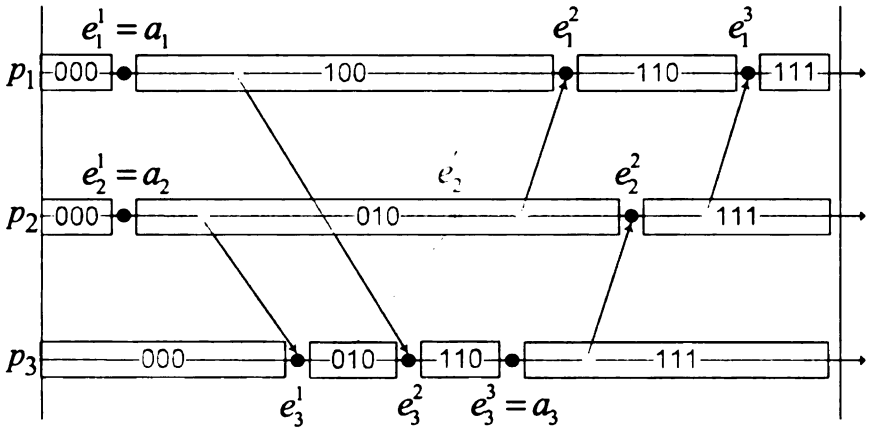


Figure 8.2: Identifying intermediate program states.

Figure 8.2 shows how each process is divided into different sections based on the value of its adaptation vector AV. Whenever a local state s_i^k of process p_i is to be collected, the state s_i^k is assigned an adaptation timestamp whose value (denoted as $s.AV$) is equal to the current value of AV_i.

We construct a global state as a n-tuple of local states $(s_1, s_2, ..., s_n)$. This global state is a state of the intermediate program whose adaptation vector is equal to $s_1.AV \vee s_2.AV \vee ... \vee s_n.AV$, where

$$s_i.AV \vee s_j.AV = [s_i.AV[0] \vee s_j.AV[0], ..., s_i.AV[n] \vee s_j.AV[n]]$$

We now discuss the algorithm for detecting *weak conjunctive* [79] adaptation predicates. A predicate is called *weak* if it is true for some observation of the distributed computation, and is similar to *possibly* predicate in [78]. *Conjunctive predicates* are of the form $C_1 \wedge ... \wedge C_n$, where each C_i is a local predicate. This class of predicates allow each process to independently evaluate its local predicate. A weak conjunctive predicate is true if and only if there exists an observation in which all conjuncts are true in some global state. This type of predicate typically describes some bad or undesirable property; in other words,

predicates that should never become true in the system. The algorithms discussed in this section are extensions of algorithms in [79].

8.4.1 Detecting Adaptation-Stable Predicates

In [79] it has been shown that to detect a weak conjunctive predicate it is necessary and sufficient to find a set of concurrent states in which local predicates are true. Let $wcp = C_1 \wedge ... \wedge C_n$ denote the weak conjunctive predicate to be detected. We discuss the centralized algorithm in which one process serves as a checker process and all other processes involved in wcp are referred to as non-checker processes.

The algorithms for non-checker and checker processes are shown in Figures 8.3 and 8.4 respectively. Each non-checker process maintains its own vector clock and adaptation vector. The values of vector clock and adaptation vector are updated as discussed earlier in this section. Whenever the local predicate of a process becomes true for the first time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its vector clock and adaptation vector, and sends it to the checker process.

The checker process maintains a separate queue for each process involved in the wcp. Whenever a debug message is received from the process it is enqueued in the queue corresponding to that process. It is assumed (for sake of efficiency) that the checker process gets its message from any process in FIFO order. If the underlying computation is not FIFO, the checker process can ensure FIFO property by using sequence numbers in messages. The algorithm compares the vector clock values at the head of the queues to determine if a consistent global state can be constructed.

process p_i (non-checker)

var V : **array** {initially, $\forall j : i \neq j : V[j] = 0; V[i] = 1$}
/* vector clock */

AV : **array** {initially, $\forall j :: AV[j] = 0$}
/* adaptation vector */

$firstflag$: **boolean** {initially, $firstflag = true$}
/* first time the local predicate is true after any send event */

$local\_pred$: **boolean expression**
/* the local predicate to be tested */

□ **replace** adaptive action $\{a_i\}$ **with**
$a_i; AV[i] = 1$

□ **replace** send statement $\{\texttt{send}\ (m)\ \texttt{to}\ p_j\}$ **with**
$\texttt{send}\ (m, V, AV)\ \texttt{to}\ p_j;$
$V[i], firstflag := V[i] + 1, true$

□ **replace** receive action $\{\texttt{rcv}\ (m)\ \texttt{from}\ p_j \rightarrow stmts\}$ **with**
$\texttt{rcv}\ (m, m.V, m.AV)\ \texttt{from}\ p_j \rightarrow stmts;$
$\forall j : V[j] = \mathbf{max}(V[j], m.V[j]);$
$\forall j : AV[j] = AV[j] \vee m.AV[j]$

□ **add** local predicate detection action
if $(local\_pred = true) \wedge firstflag$ **then**
$firstflag := false;$
$\texttt{send}\ (dbg, V, AV)\ \texttt{to}\ p$ (checker process)
fi

Figure 8.3: Algorithm for adaptation-stable predicate (non-checker process p_i).

The algorithm is initiated whenever any debug message is received from a non-checker process. If the corresponding queue is non-empty, then the message is inserted in the queue; otherwise it is checked if the message lead to the case where the conjunctive predicate became true. The algorithm maintains $changed$ and $newchanged$ variables to ensure that only those heads of the queue are compared which have not been compared earlier. When the while loop terminates if all the queues are non-empty, then the intermediate program can be ascertained in which wcp was first detected.

```
process p (checker)
  var q_1, ..., q_n                : queue of (V, AV)
      changed, newchanged          : set of {1, 2, ..., n}
  □ rcv (elem) from P_k →
  /* elem.V and elem.AV denotes the value of vector clock and adaptation vector */
      insert(q_k, elem);
      if (head(q_k) = elem) then
        changed := {k};
        while (changed ≠ φ) do
          newchanged := {};
          for i in changed ∧ j in {1, 2, ..., n}
            if (¬empty(q_i) ∧ ¬empty(q_j)) then
              if head(q_i).V < head(q_j).V then
                newchanged := newchanged ∪ {i};
              fi
              if head(q_j).V < head(q_i).V then
                newchanged := newchanged ∪ {j};
              fi
            fi
            changed := newchanged;
            for i in changed
              deletehead(q_i);
        od /* end while */
        if ∀i : ¬empty(q_i) then
          found := true;
          int_program_AV := [⋁_{i=1}^{n} head(q_i).AV[1], ..., ⋁_{i=1}^{n} head(q_i).AV[n]]
           /* intermediate program in which the predicate was detected */
        fi
      fi
```

Figure 8.4: Algorithm for adaptation-stable predicate (checker process).

8.4.2 Detecting Adaptation-Transient Predicates

Consider a weak conjunctive predicate wcp that is to be checked for some intermediate program I during adaptation. Let I be represented by the adaptation vector I_{AV}. The goal is to detect wcp during the interval of adaptation that corresponds to the intermediate program I.

When detecting adaptation-transient predicates, the non-checker processes have to check for local predicates only for some interval of adaptation as defined by the intermediate program in which the adaptation-transient predicate is to be checked. Specifically, each non-checker process identifies a set of local states that are potential states of the intermediate program I. Let the current value of adaptation vector in state s of process p_i be AV_i, which is denoted by $s.AV$. We first present the following lemma in the context of identifying potential states of an intermediate program locally at each process.

Lemma 7.1. For state s of process p_i, $s.AV[i] = 0 \Rightarrow \forall j : j \neq i : AV_j[i] = 0$

Proof. From definition of adaptation vector in Section 8.3, adaptive action a_i occurs at process p_i. The lemma states that if process p_i has not performed its atomic adaptation, then adaptation vectors at all other processes denotes the same. Also, if adaptation vectors at all other processes denote that process p_i has not performed its atomic adaptation, then it does not imply that p_i has not performed its atomic adaptation. The proof is apparent from the implementation of adaptation vectors. □

We now discuss the following two cases at each process p_i to identify the potential states of intermediate program I:

Case 1 : $I_{AV}[i] = 0$. This is the case where intermediate program I occurs before the atomic adaptation at process p_i is executed. s is a potential state of the intermediate program I, if $s.AV \vee I_{AV} = I_{AV}$. For example, if $AV_I = [1, 1, 0, ..., 0]$ and $s.AV = [1, 0, 0, ..., 0]$ then s is a potential state of intermediate program I, whereas $s.AV = [1, 0, 1, ..., 0]$ is not a potential state of intermediate program I.

Case 2 : $I_{AV}[i] = 1$. This is the case where intermediate program I is reached after the atomic adaptation at process p_i is executed. From Lemma 7.1, it is clear that in this case

checking the condition $s.AV \vee I_{AV} = I_{AV}$ is not sufficient. For example, if $s.AV = [0, 0, ..., 0]$ then the condition is true, but atomic adaptation at process p_i is not executed (because $s.AV[i] = 0$). Therefore, we also need to check that $s.AV[i] = 1$, *i.e.*, atomic adaptation at process i has executed. In other words, we also need to check for the following condition: $s.AV[i] = I_{AV}[i]$.

Based on the above discussion, we modify the algorithm of Figure 8.3 for non-checker process so it can detect adaptation-transient predicates. The modified non-checker process algorithm is shown in Figure 8.5. In the algorithm, $plist$ maintains a list of - predicate, the corresponding intermediate program in which it needs to be checked, and the checker process that is checking that predicate.

We maintain one checker process algorithm for each adaptation-transient predicate that needs to be checked. The checker process algorithm is modified so that whenever it finds a consistent state (*i.e.*, all queues are non-empty) it checks the adaptation vectors to ensure if that consistent state belongs to intermediate program I. The modified checker process algorithm to detect adaptation transient predicate is shown in Figure 8.6.

8.5 Testing Only Atomic Adaptations

In Section 8.4, we showed how we can test predicates that have certain characteristics during adaptation. However, if the test run is large then it increases the cost of testing. It is, therefore, desirable if we can reduce the cost of testing by doing a partial testing on a small subset of states rather than testing the entire run of adaptation. In this case, the states chosen should be such that testing predicates in these states would still give reasonable

process p_i (non-checker)

var V : **array** {initially, $\forall j : i \neq j : V[j] = 0; V[i] = 1$}
/* vector clock */

AV : **array** {initially, $\forall j :: AV[j] = 0$}
/* adaptation vector */

$firstflag$: **boolean** {initially, $firstflag = true$}
/* first time the local predicate is true after any send event */

$plist\ \langle IP_{AV}, local\_pred, cp \rangle$
: **array** ⟨ **array, boolean expression, process id** ⟩
/* local predicates to be tested */

□ **replace** adaptive action $\{a_i\}$ **with**
$a_i; AV[i] = 1$

□ **replace** send statement $\{\texttt{send}\ (m)\ \texttt{to}\ p_j\}$ **with**
$\texttt{send}\ (m, V, AV)\ \texttt{to}\ p_j;$
$V[i], firstflag := V[i] + 1, true$

□ **replace** receive action $\{\texttt{rcv}\ (m)\ \texttt{from}\ p_j \rightarrow stmts\}$ **with**
$\texttt{rcv}\ (m, m.V, m.AV)\ \texttt{from}\ p_j \rightarrow stmts;$
$\forall j : V[j] = \mathbf{max}(V[j], m.V[j]);$
$\forall j : AV[j] = AV[j] \vee m.AV[j]$

□ **add** local predicate detection action
for p **in** $plist$
if $p.IP_{AV}[i] = AV[i] \wedge (p.IP_{AV} \vee AV) = p.IP_{AV}$
$\wedge p.local\_pred = true \wedge firstflag$ **then**
$firstflag := false;$
$\texttt{send}\ (dbg, V, AV)\ \texttt{to}\ p.cp$ (checker process)
fi

Figure 8.5: Algorithm for adaptation-transient predicates (non-checker process p_i).

assurance about correctness of adaptation.

With this motivation, we choose the states before and after each atomic adaptation for testing. For adaptation to be correct, each atomic adaptation should occur in some "safe state" of the system. Informally, each atomic adaptation should occur when the system is ready to deal with this change. Thus, instead of checking all the states of intermediate programs, we can check all the possible states in which the atomic adaptation could have

```
process p (checker)
  var q_1, ..., q_n              : queue of (V, AV)
      changed, newchanged        : set of {1, 2, ..., n}
      IP_AV                      : array
                                 /* adaptation vector of being checked */
  □ rcv (elem) from P_k →
  /* elem.V and elem.AV denotes the value of vector clock and adaptation vector */
      insert(q_k, elem);
      if (head(q_k) = elem) then
        changed := {k};
        while (changed ≠ φ) do
          newchanged := {};
          for i in changed ∧ j in {1, 2, ..., n}
            if (¬empty(q_i) ∧ ¬empty(q_j)) then
              if head(q_i).V < head(q_j).V then
                newchanged := newchanged ∪ {i};
              fi
              if head(q_j).V < head(q_i).V then
                newchanged := newchanged ∪ {j};
              fi
            fi
            changed := newchanged;
            for i in changed
              deletehead(q_i);
        od; /* end while */
        if ∀i : ¬empty(q_i) then
          int_program_AV := [⋁_{i=1}^{n} head(q_i).AV[1], ..., ⋁_{i=1}^{n} head(q_i).AV[n]]
          if IP_AV = int_program_AV then
            found := true
          fi
        fi
      fi
```

Figure 8.6: Algorithm for adaptation-transient predicate (checker process).

occurred (or could have lead to). Moreover, since the set of possible states before and after atomic adaptations is expected to be much smaller than the set of all states it may also be possible to check for arbitrary predicates in those states.

For example, consider the adaptation lattice of Figure 4.2. In this case, when atomic adaptation a_3 is executed the predicate associated with node R_1 should be true, and when a_3 has completed execution the predicate associated with node Q should be true.

We now discuss how we calculate, for each atomic adaptation, the set of states in which the atomic adaptation could have occurred. The approach to find the set of states after each atomic adaptation is similar. We first define some notions that we borrow from Venkatesan and Dathan [82]. A *spectrum* $spec_i$ in process p_i is a sequence of consecutive states of p_i. The first (respectively, last) event of $spec_i$ is denoted by $first_i(spec_i)$ (respectively, $last_i(spec_i)$. A state s_i^j in $spec_i$ is the first state in $spec_i$ if $\forall s : s \in spec_i : s_i^j \rightarrow s$. Similarly, a state s_i^j in $spec_i$ is the last state in $spec_i$ if $\forall s : s \in spec_i : s \rightarrow s_i^j$. The *first consistent cut state* of s_i^j at process p_k, denoted by $first_k(s_i^j)$, is the earliest state s_k^l in p_k such that s_k^l and s_i^j can be in a consistent state. Similarly, the *last consistent cut state* of s_i^j at process p_k, denoted by $last_k(s_i^j)$, is the latest state s_k^l in p_k such that s_k^l and s_i^j can be in a consistent state. For example, consider Figure 8.1 of Section 8.1. Let s_i^j be the state of p_i immediately before the execution of event e_i^j, and $s_i^{j'}$ be the one immediately after the execution of that event. In Figure 8.1, $first_1(s_2^4) = s_1^{3'}$, and $last_1(s_2^4) = s_1^4$.

Now, let $e_i^k = a_i$ be the atomic adaptation at process p_i. Let s_i^k be the state of the process p_i just before the execution of atomic adaptation a_i. To identify the set of global states in which a_i could have occurred, we need to identify the local states at all processes that are consistent with state s_i^k. We first identify the spectrum of states in each process that are consistent with s_i^k. Let $spec_j(a_i)$ denote the spectrum of states in process p_j, possibly empty, that are consistent with state s_i^k. We have,

$$spec_j(a_i) = [first_j(s_i^k), last_j(s_i^k)]$$

Note that $spec_i(a_i) = s_i^k$. We use the property of the vector clock to find $first_j(s_i^k) = s_j^l$ and $last_j(s_i^k) = s_j^m$. If $s_i^k.V[j] = 0$, *i.e.*, no state of p_j precedes s_i^k, then $l = 0$; otherwise,

$s_j^l.V[j] = s_i^k.V[j] \wedge s_j^{l-1}.V[j] < s_i^k.V[j]$, and

$(s_j^m.V[i] \leq s_i^k.V[i]) \wedge ((s_j^m = final\_state_j) \vee (s_j^{m+1}.V[i] > s_i^k.V[i]))$

We use $final\_state_j$ to denote the state in which process p_j terminates.

In this manner, we can find the spectra $spec_1(a_i), ..., spec_n(a_i)$ at processes p_1, ..., p_n respectively. Since the trace is typically small, it may be possible to construct all possible consistent states from these spectra. This allows us to check even arbitrary predicates in all possible states in which atomic adaptation could have occurred. Furthermore, given all these spectra we can use the algorithms from Venkatesan and Dathan [82] to test for different types of predicates using intersection of these spectra. Moreover, if the predicates are of certain types, we can also use algorithms discussed in Section 8.5 on the collected trace for each atomic adaptation.

8.6 Chapter Summary

In this chapter, we discussed testing of adaptation in distributed systems using predicate detection. We identified two classes of predicates, namely adaptation-stable and adaptation-transient predicates, that occur during adaptation. We introduced adaptation vector to identify intermediate program states in testing dynamic adaptation.

Furthermore, to reduce the cost of testing, we identified a subset of states to do limited testing of adaptation. Specifically, instead of checking for all intermediate program states,

we presented an approach to test the system before and after each atomic adaptation. We discussed an approach to calculate spectra at each process consistent with the state before (or after) each atomic adaptation. We can use the existing algorithms for predicate detection on this smaller trace.

Chapter 9

Component Family: Design of Adaptive Components

In this chapter, we discuss *component family* design to support runtime adaptation. We first discuss, in Section 9.1, the issues that arise in developing compositional adaptation. Then, in Section 9.2, we present the abstract design of component family. In Section 9.3, we discuss the concrete design of component family. Next, in Sections 9.4 and 9.5, we illustrate the use of component family using the case studies. We discuss some questions related to component family design in Section 9.6. In Section 9.7, we discuss the related work and finally, in Section 9.8, we summarize the advantages of component family.

9.1 Introduction

In compositional adaptation, which is the primary focus of this dissertation, some algorithmic or structural parts of the system are added, removed or replaced at runtime. Com-

positional adaptation enables an application to adopt new algorithms and strategies for addressing concerns that were not known at the time the original application is developed.

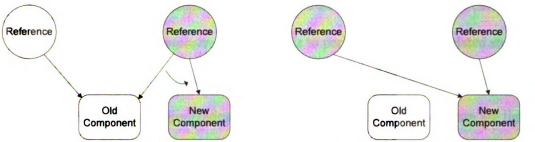
Typically, in the case of compositional adaptation, *separations of concerns* principle [64, 86] is employed to separate the adaptation concern. The separation of concerns is an ubiquitous software engineering principle which states that for a given problem different kinds of concerns should be identified and separated to cope with complexity and achieve robustness, adaptability, maintainability and reusability. In other words, it states that software should be decomposed in such a way that different aspects of the problem are solved in well-separated modules or parts of the software. Consequently, adaptive applications are designed to separate the functionality that gets adapted from rest of the application. In object-oriented programming, separated concerns are modeled as objects and classes, while in structural programming these concerns are modeled as functions or procedures. We argue that these two programming paradigms are not adequate to model concerns that spread across multiple processes. For example, in the case of adaptive distributed systems, functionality that gets adapted may involve changes to multiple processes across the system. To deal with this we consider component-based paradigm, where the basic unit of concerns are components [28, 87].

A component is defined as follows : "a component is an executable unit of code that provides physical black-box encapsulation of related services. Its services can only be accessed through a consistent, published interface that includes an interaction standard. A component must be capable of being connected to other components (through a communications interface) to form a larger group" [87]. Although objects or procedures form the underlying fabric of software solutions, it are components that provide the effective

granularity to model adaptive systems. In other words, object-oriented programming or structural programming may be used to realize component-based systems. Thus, compositional adaptation can be considered as addition, removal or replacement of components.

Existing approaches to building adaptive software have some notion of a component, which is similar to the one defined above. In many of the approaches, the term component refers to a replaceable class or object [88]. However, in our approach, a component can consist of several objects and these objects may be spread across different processes of a system. Regardless of how a component is defined, adaptive software needs some mechanisms for examining and understanding the state of its components (introspection) and also some mechanisms to modify the components (intercession). There are three main issues that occur while developing introspection and intercession mechanisms for compositional adaptation:

- **Reference update**. As shown in Figure 9.1, when a component is replaced (respectively, added or removed) the references pointing to the component need to be updated. Specifically, when an old component is exchanged for a new component it is necessary to update all the references to the old component such that they refer to the new component. This is important to ensure that the system is not left in an inconsistent state after component replacement. A common solution to this problem is achieved through indirection, which allows decoupling of an application. The standard practice of *programming to an interface* [89] is useful in achieving this indirection. This allows an application to access a component while ensuring that access methods are not coupled to the implementation of the component.



(a) Incorrect/incomplete update (b) Correct/complete update

Figure 9.1: Reference update during adaptation.

- **State transfer**. When a component is replaced, the state of the old component needs to be transferred to the new component. This is necessary for performance as well as correctness reasons. It is not optimal to let the new component start from initial state, as the new component may have to do the computations again which the old component already performed. Moreover, in some cases it may be incorrect to have the new component start from an arbitrary initial state. If two components are to be interchanged at runtime, then mechanisms to extract the state from the old component and inject the equivalent state into the new component need to be considered.

- **Synchronization**. When a component is added, removed or replaced, it is important to control the access to the component during the change. For example, once the state of the old component is extracted for transfer to the new component, the application should not modify the old component as it would lead to unexpected results. It is necessary to have proper intra-process and inter-process synchronization during adaptation, as lack of synchronization can lead to undesirable and incorrect behavior.

We now consider the three main limitations concerning most of the existing approaches for building adaptive software:

1. **Tight coupling**. Most adaptive softwares have an entity that is normally categorized as a *framework* or a *composer*, whose goal is to perform adaptation related steps. In other words, it primarily deals with the above three issues, namely, reference update, state transfer and synchronization. Most approaches to adaptive software are tightly coupled in terms of components that are adapted and framework that performs adaptation. Specifically, the frameworks are either not reusable with other components and applications, or the role of the framework is not clearly defined. Many times the introspection and intercession related steps are tightly coupled with the component functionality. These kind of couplings leads to the following four problems: (i) makes the verification of adaptation difficult, (ii) restricts independent development of components, (ii) restricts reuse of component and framework, and (iv) modifying introspection and intercession mechanisms are difficult.

2. **Large number of adaptations**. There are several components available for a given functional requirement. For example, to provide reliable communication one can use a proactive component based on forward error correction or a reactive component based on acknowledgments. The main motivation for adaptation is to provide an appropriate component for a given functionality. Now, consider the scenario where there are n different components that provide similar interface and functionality, and the choice of the component depends on application requirements and environment conditions. In this case, there are $n(n-1)$ possible adaptations among these components. Considering the fact that each adaptation needs to address introspection and intercession issues and also leads to the four problems related to tight coupling,

identifying all these adaptations is a difficult and unfeasible task. Moreover, if the developer of the component wants to expose the details of that component to only a subset of developers of other components then adaptation between that component and the remaining components may not be possible.

3. **Unanticipated components.** While developing adaptations among existing components, it is difficult to anticipate new components those would be developed later. Furthermore, while developing a new component one does not know about all the existing components that can potentially be replaced by the new component. Moreover, for the case where one does not know the details of some components, it is impossible to provide assurance for adaptations to and from those components. As there is a growing need to build systems that are autonomic in nature, it is desirable to build applications that can support the use of unanticipated components with little or no human intervention while still being able to guarantee assurance of adaptation.

To overcome above limitations, we introduce the notion of *component family* – a *systematic* and *extensible* repository of components that provide similar functionality. There are two main aspects to component family: (i) overall design of the library of components, and (ii) design of individual components in the library. The overall design of component family is based on the ideas of program families as proposed by Parnas [90] for the study of reusable software. We first list the design goals for component family and then give a formal definition of component family:

1. **Adaptation ready.** The components should be designed with adaptation in mind. In other words, components should provide additional methods required during adapta-

tion. For example, the components discussed in [3, 91] provide *checkstate* function to check state of the component for safety, *block* to temporarily block the component operations, etc.

2. **Separation of adaptation logic**. The adaptation logic should be separate from the core functionality of components, thereby, simplifying specification and verification of adaptation. The application developer is primarily concerned with using component functionality and should not have to deal with details of adaptation among components.

3. **Replaceable adaptation logic**. Adaptation from one component to another can be done in multiple ways. The design should support easy replacement of adaptation logic. For example, as discussed in Chapter 6, replacing one component by another can be done either using quiescence adaptation or mixed-mode adaptation. Clearly, the adaptation logic should be modeled separately and should be easy to replace.

4. **Language and platform neutral**. The design should be independent of any particular language or platform so that component family can be implemented in different languages on a variety of platforms.

5. **Application independent**. The design should be applicable to components from different domains.

6. **Extensible**. The design should be extensible so that new components that are developed later can be easily integrated into the family.

Considering the above goals, we first discuss the abstract design of a component family and then describe the concrete representation of the family.

9.2 Abstract Component Family

Definition. A component family is a strongly connected directed graph, say (V, E), where (i) each vertex in V denotes a component, (ii) all components have similar interfaces and syntactically one component in the family can be replaced by another, and (iii) Each arc $(v_1, v_2) \in E$ denotes that there exists an adaptation from v_1 to v_2.

We illustrate a component family in Figure 9.2. Consider the subgraph consisting of vertices A, B, C, and D, which represents the four components in the family. Direct adaptations are defined for each arc in this graph. The arc from A to D denotes that there exists an adaptation from A to D. In other words, arc (A, D) implies that there is a corresponding adaptation lattice (as discussed in Chapter 3) that defines the adaptation from A to D. We say that arc (A, D) is a verified arc if verified adaptation from A to D exists. Otherwise stated, there exists a transitional-invariant or transitional-faultspan lattice corresponding to the adaptation from A to D.

Furthermore, as shown in the Figure 9.2, there exists a path from component A to component B, which implies that there exists a sequence of adaptations through which A can be replaced by B. Such a path exists from A to D, from D to C, and from C to B.

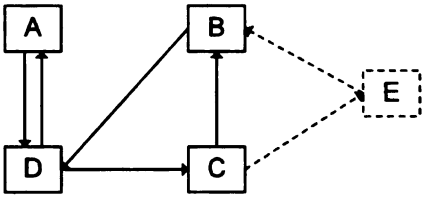


Figure 9.2: An example of a component family

9.2.1 Abstract Component Structure

As discussed earlier in this section, in order to keep a graph of component family strongly connected, we need at least two adaptations associated with each component, such that one adaptation is to the component and one adaptation is from the component. A component has different adaptation related actions corresponding to different adaptations that it is involved in. However, the part of the component that performs the actual functionality remains the same irrespective of the adaptations.

Reckoning the adaptation requirements, each component in a component family is designed to consist of two parts: (*i*) a *functional* part, and (*ii*) an *adapt-active* part. The adapt-active part is involved in state-transfer and synchronization related actions that are needed only during adaptation. In other words, functions of the adapt-active part are invoked only during adaptation.

Each component in a family consists of exactly one functional part. A component may have zero or more adapt-active parts. The adapt-active part of a component corresponds to a particular adaptation the component is involved in. A component may not have an adapt-active part (in other words, have an empty adapt-active part), if the component does not perform any introspection or intercession related actions during adaptation. Also, a component may have an adapt-active part that is shared with multiple adaptations that it is

associated with. From an implementation perspective, depending on the adaptation that the component is involved in, the appropriate adapt-active part corresponding to that adaptation should be loaded before adaptation. This can be triggered internally (by some monitoring module) or externally (by an user).

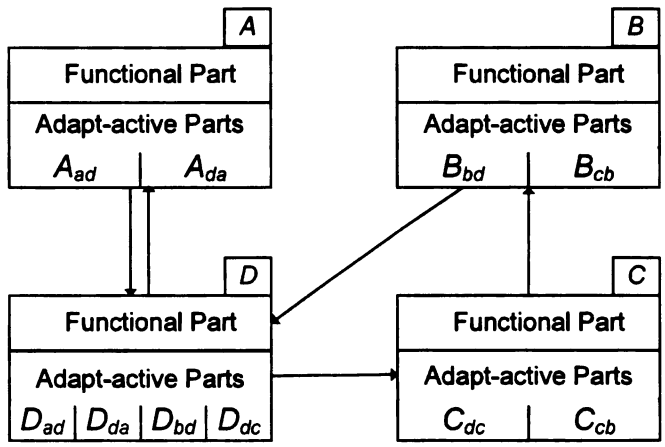


Figure 9.3: Structure of a component in a family

For example, consider the components A, B, C, and D in a component family as shown in Figure 9.3. The component B has two adapt-active parts, B_{bd} and B_{cb}. The part B_{bd} is used during adaptation from B to D, and the part B_{cb} is used during adaptation from C to B.

The abstract design of a component family, as discussed in this section, is useful when developing high-level abstract models of the components and adaptation. However, there are several concerns that need to be addressed from implementation perspective while developing the component families. In the next section, we discuss the concrete representation of component family.

9.3 Concrete Component Family

The various architectural parts of a component family and high-level relationships among them are shown in Figure 9.4.

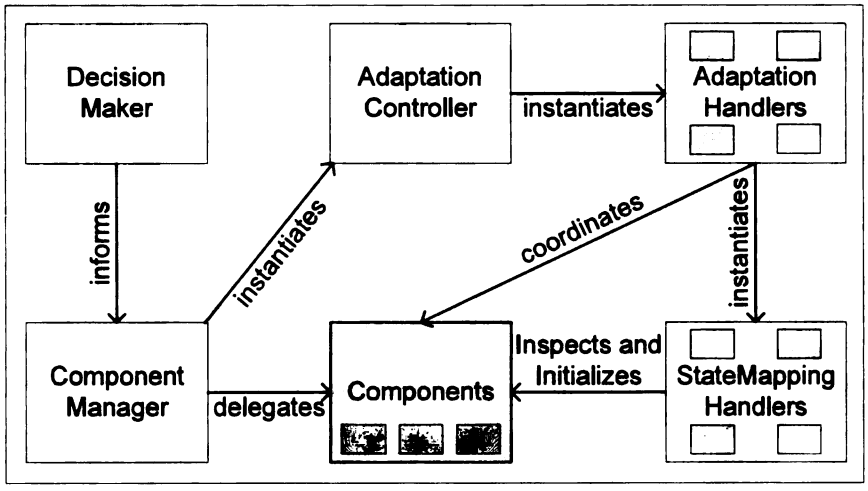


Figure 9.4: Architecture of a component family

- **Components**. At the core of the component family are the set of components that provide the actual functionality required by the application/user. In the absence of component family, the application developer uses the components directly.

- **Component Manager**. To provide adaptation transparency to the application using the components, we decouple the application from the components by introducing an entity between the application and the components. We call this entity *ComponentManager*. *ComponentManager* accepts the requests from the application and delegates that request to the currently active component. It also controls the exchange of information between *DecisionMaker* and *AdaptationController*.

- **Decision Maker**. *DecisionMaker* decides what is the appropriate component to be used based on the environment conditions. The entity external to the component family may also decide on when and what to adapt. For example, a user requirement may define the adaptation. In this case, *DecisionMaker* that is part of the family may be either disabled or made to co-ordinate with the external decision maker to reach a decision. The details of *DecisionMaker* is beyond the scope of this work. The reader is referred to [92–95] for more information on decision makers.

- **Adaptation Controller**. Once *ComponentManager* has learnt of the decision from *DecisionMaker* and decided to adapt it passes that information to *AdaptationController*. *AdaptationController* selects an appropriate *AdaptationHandler* to execute the adaptation.

- **Adaptation Handlers**. Each *AdaptationHandler* implements the logic for adaptation. It executes tasks related to reference updates, synchronization, and state transfer. Specifically, it deals with any dependency and synchronization related issues while adding, removing, or replacing the components. It also instantiates an appropriate *StateMappingHandler* to perform any required state transfer during adaptation.

- **StateMapping Handlers**. *StateMappingHandlers* are responsible to map the state of the old component to the new component during component replacement. There is a separate *StateMappingHandler* for each adaptation. Various mappings are discussed in Chapter 3 and *StateMappingHandlers* implement those mappings.

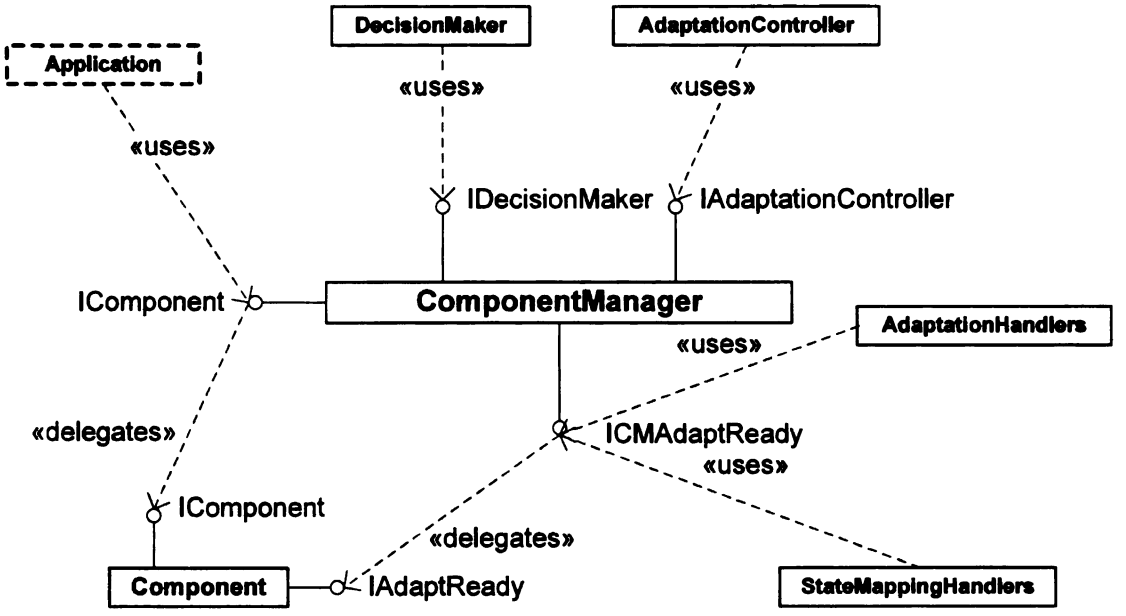


Figure 9.5: Interfaces of a component family.

9.3.1 Component Family Interfaces

Figure 9.5 shows how different parts of a component family are related to *ComponentManager*. It shows that *ComponentManager* implements four interfaces, namely, *IComponent*, *ICMAdaptReady*, *IDecisionMaker* and *IAdaptationController*. All the functional components in the family provide two interfaces *IComponent* and *IAdaptReady*. *ICMAdaptReady* interface extends *IAdaptReady* interface. *IComponent* interface specifies the core functionality provided by the component. *IAdaptReady* interface specifies adaptation-related functionality provided by the component. It maps to the adapt-active parts discussed earlier in the abstract design of component family. The application is external to the component family and it uses *IComponent* interface provided by *ComponentManager*.

ComponentManager instantiates the component that will provide the functionality requested by the application. At any time, *ComponentManager* maintains exactly one active instance of some component. It uses delegation pattern to delegate any service request from

the application to the active component.

ComponentManager also provides *IDecisionMaker* interface for use by the decision maker. The choice of how the decision maker is implemented affects the design of *IDecisionMaker*. The following are two main issues that need to be considered while designing *IDecisionMaker*: (i) Whether the decision maker will be a part of the family or external to the family, and (ii) In the case of a distributed system, which process will have the decision maker. In the case where the decision maker is part of the family, *ComponentManager* creates and initializes an instance of *DecisionMaker*. *DecisionMaker* uses *IDecisionMaker* interface to collect information from *ComponentManager* that is required to make a decision about adaptation and to inform *ComponentManager* of the decision.

ComponentManager also provides *IAdaptationController* interface that is used by *AdaptationController*. *ComponentManager* creates and initializes an instance of *AdaptationController* and informs *AdaptationController* of the decision made by *DecisionMaker*. *AdaptationController* uses *IAdaptationController* to get details about the current active component and any other information that is required by it.

Once *AdaptationController* has learnt of the decision, it selects an appropriate *AdaptationHandler* that will perform the adaptation. As shown in Figure 9.6, each *AdaptationHandler* extends *AbstractAdaptationHandler*, which provides *IAdaptationHandler* interface that is used by *AdaptationController*. Each *AdaptationHandler* also instantiates an appropriate *StateMappingHandler*, which performs any required state transfer to map the state of old component to the new component.

AdaptationHandlers and *StateMappingHandlers* use *ICMAdaptReady* interface of *ComponentManager* to communicate with the components to coordinate the adaptation

and state transfer. *ComponentManager* delegates the requests received from *Adaptation-Handlers* and *StateMappingHandlers* to the active component.

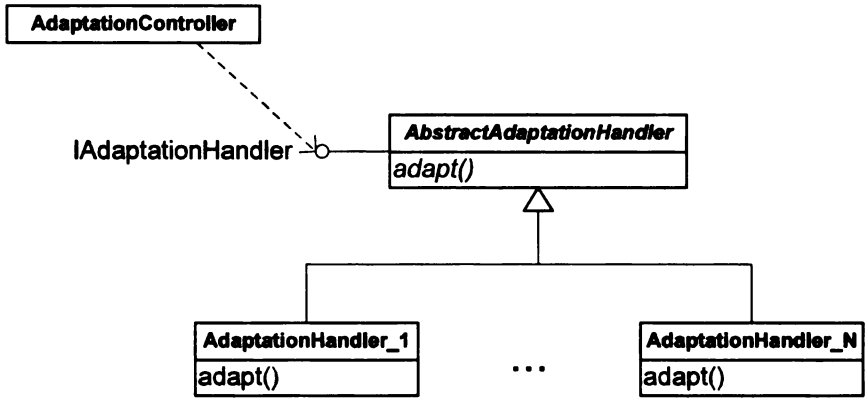


Figure 9.6: Adaptation handlers.

We now give a brief overview of delegation pattern used by *ComponentManager*.

9.3.1.1 Delegation Pattern

Delegation pattern is a technique where a component outwardly expresses some behavior, but delegates the actual responsibility of implementing that behavior to another component. Delegation is a way of extending and reusing a class by writing another class with additional functionality that uses instances of original class to provide the original functionality. Chain of responsibility and observer pattern [89] use delegation pattern. The sample code in Java programming language that illustrates delegation pattern is shown in Figure 9.7.

In this example, *ComponentManager* class has instances of two components *ComponentX* and *ComponentY*. *ComponentManager*, *ComponentX*, and *ComponentY* all implement *IComponent* interface. *ComponentManager* maintains an active instance of one component (which can be changed). *ComponentManager* can delegate the call to *foo* function to either *ComponentX* or *componentY*.

```
// Common Interface
public interface IComponent {
    public void foo();
}

// Component X
public class ComponentX implements IComponent {
    public void foo() {
        // Component X implementing foo
    }
}

// Component Y
public class ComponentY implements IComponent {
    public void foo() {
        // Component Y implementing foo
    }
}

// Component Manager
public class ComponentManager implements IComponent {
    IComponent instance = new ComponentX();

    // For Delegation to X
    public void toX {
        instance = new ComponentX();
    }

    // For Delegation to Y
    public void toY {
        instance = new ComponentY();
    }

    // foo is delegated
    public void foo () {
        instance.foo();
    }
}
```

Figure 9.7: An example of delegation pattern.

9.3.2 Component Family Instantiation

In the previous subsection, we discussed the different interfaces of a component family. In this section, we discuss instantiation of a component family. We consider two cases depending on the type of components represented by the component family: (i) local components, and (ii) distributed components.

We use the term local components to refer to components that are installed at a single process, and distributed components to denote components installed across multiple processes. Examples of adaptations involving local components include: (i) replacing local monitoring component that monitors some environmental conditions and takes appropriate actions, (ii) replace logging component that stores a log of events at a process. In this case, the application of component family is relatively easy. The issues related to inter-process synchronization do not occur in this case, as a result, *AdaptationHandlers* are also less complex. We focus our discussion on distributed components as it offers more challenges and the results in the case of distributed components can be trivially applied in the case of local components.

We first identify the challenges during adaptation involving distributed components that any implementation of a component family needs to address and then discuss our implementation.

- **Atomicity**. When a distributed component is replaced in an application, we need to ensure *atomicity*. In other words, all local fractions of the distributed component should be replaced across all or selected processes of the application or none should be replaced.

- **Minimal blocking** Another challenge is that during adaptation involving a distributed component, the application should not be blocked. While it is desirable that the adaptation of a distributed component be entirely non-blocking, it is not always possible to do so due to dependency among component fractions. We, therefore, require that blocking introduced during the adaptation be minimal.

- **Synchronization.** To avoid interference between concurrent versions of distributed components, some global synchronization is often required. For example, if a process that has added a component fraction interacts with another that has not then the results can be unpredictable.

- **Transparency and noninterference.** The execution of an application that is using a distributed component should not be affected while the adaptation is underway.

The code that addresses these challenges is implemented as *AdaptationHandlers* in the component family. The design of component family separates out the implementation of *AdaptationHandlers*, thereby, allowing different ways to deal with these challenges. Several implementations of adaptation logic, written by different people can be part of the same family and an appropriate implementation can be chosen for execution based on environment conditions and application requirements. In the next section, we discuss one particular implementation.

Remark. For sake of brevity, we will use the term component to mean distributed component in the rest of this chapter.

9.3.3 Component Family Implementation

In this subsection, we discuss an implementation of *AdaptationHandler*, which specifies adaptation logic to replace one component by another. *ComponentManager* at one of the process acts as *initiator*. *DecisionMaker* at the initiator process may be the only active *DecisionMaker* in the system, or if there are *DecisionMakers* at other processes then the *DecisionMaker* at the initiator process may coordinate with *DecisionMakers* at other processes to decide on adaptation. Regardless, once *ComponentManager* at the initiator process learns of the decision and it decides to adapt, it informs *AdaptationController* about it. The *AdaptationController* will instantiate appropriate *AdaptationHandler* that will execute the adaptation. *AdaptationHandlers* at different processes coordinate to execute the adaptation. We use a variation of *distributed reset protocol* [96, 97] to achieve inter-process synchronization among component fractions. We first discuss the distributed reset protocol in brief and then give details of how it is used to achieve synchronization during adaptation.

9.3.3.1 Distributed Reset Protocol

The reset subsystem in [96, 97] can be embedded in an arbitrary distributed system to allow processes to reset the system to a given global state. In the model described in [96, 97], each process consists of an application module and a reset module. The application module at any process may begin the reset operation. The function of a reset module is to (1) reset the state of the application to a state that is reachable from the given global state, and (2) inform the application module when the reset operation is complete.

Each reset operation satisfies the following two properties: (1) Every reset operation is

non-premature, *i.e.*, if the reset operation completes, then all processes have been reset and the program state is reachable from the given global state, and (2) Every reset operation *eventually completes*, *i.e.*, if an application module at a process initiates a reset operation, eventually the reset module at that process informs the application module that the reset operation is complete. The reset solutions in [96, 97] allow the program computation to proceed concurrently with the reset, to any extent that does not interfere with the correctness of the reset.

To simplify the reset operation, the algorithms in [96, 97] maintain a rooted spanning tree of all non-failed processes. It uses this spanning tree to perform a *diffusing computation* [76] in which each process resets its state. The diffusing computation begins at the root of the spanning tree. The root of the tree resets the state of its local application module and initiates a reset wave that propagates along the tree towards the leaves; whenever the reset wave reaches a process, the process resets the state of its local application module and propagates the reset wave to its children. After the reset wave reaches a leaf, it is reflected as a completion wave towards the root. A process propagates the completion wave to its parent when it receives the completion wave from all its children. The reset is complete when the root receives the completion wave from all its children.

We now discuss how the distributed reset protocol is used to replace a component across a distributed application.

9.3.3.2 Using Distributed Reset Protocol for Adaptation

AdaptationHandlers at all the processes communicate using a variation of the distributed reset protocol. The reset protocol consists of two waves: a *reset-initialization wave*, and

a *replacement wave*. The replacement wave consists of two sub-waves, namely, a *reset-transition wave* and a *reset-completion wave*. We first present the outline describing the reset process and then explain each of the reset waves in detail. The *AdaptationHandler* at the initiator process initiates the reset by sending the reset-initialization wave. In the reset-initialization wave, all processes change to the *transit state* and initialize the component fraction of the new component. Thus, in the transit state, a process has initialized the new component fraction, although it is still using the old component fraction. After all processes have set themselves into the transit state, the reset-initialization wave completes successfully. In case any process does not set itself into transit state during the reset-initialization wave, the reset-initialization wave completes unsuccessfully. The option of unsuccessful completion is provided to deal with the case where the processes need to obtain their component fraction remotely and they fail to do so, or sufficient resources are not available at a process to start the new component. If the reset-initialization wave completes unsuccessfully, component replacement is abandoned.

Upon successful completion of the reset-initialization wave, the *AdaptationHandler* at the initiator starts the replacement wave. The replacement wave begins with the reset-transition wave from the initiator (root) towards leaves. The *AdaptationHandler* at each process receiving the reset-transition wave removes the old component fraction and adds the new component fraction. It uses services provided by *IAdaptReady* interface of the component fraction to ensure safety (correctness) of such replacement. *IAdaptReady* interface of each component fraction provides *checkState* function to determine the state of the component fraction. After a leaf process has completed the replacement of its component fraction, it sends the reset-completion wave to its parent. Further, if a non-leaf

process has completed the replacement of its component fraction and it has received the reset-completion wave from all of its children, it propagates the reset-completion wave to its parent. The reset-completion wave eventually reaches the initiator. We allow another reset to start once the first reset is completed (successfully or unsuccessfully). We now explain the reset waves in detail.

Reset-initialization wave. The *AdaptationHandler* at the initiator initializes the reset by sending the reset-initialization wave to all of its neighbors. The *AdaptationHandler* uses the *adaptation initialization protocol* to communicate with other reset modules. The adaptation initialization protocol communicates information such as the name of the component, the location of the server where components are available, etc. The *AdaptationHandler* at each process that receives the reset-initialization wave performs the following tasks:

1. It sets its parent to the first process from which it received the reset-initialization wave.

2. It propagates the reset-initialization wave to all of its neighbors except its parent.

3. If a process receives the reset-initialization wave again it informs the sender of the identity of its parent. This information is used to form a tree.

4. If the process that receives the reset-initialization wave is a leaf (*i.e.*, no neighbor process has set its parent to this process), it initializes the new component and sets itself into the transit state, where it is still using the old component while waiting to use the new component. If the process fails to initialize the new component, it sets itself into *error state*. The process then communicates its state (transit or error) to its parent.

5. When a process has received transit state message from all of its children, it sets itself into transit state by initializing the new component. If it receives the error state information from any of its children or if it fails to initialize the new component fraction, it sends the error state message to its parent. Eventually, the root process receives the transit state or the error state information from its children. If it receives the error state information from any of its children, it can restart the reset-initialization wave or abandon the component replacement based on the threshold value set for the number of reset-initialization waves that can be initiated. If the component replacement is abandoned, other processes would be informed about it so that they can return to normal state. If the root process receives transit state information from all of its children, it initializes the new component and sets itself into transit state.

Reset-transition wave. When all processes are in transit state, *i.e.*, at the successful completion of the reset-initialization wave, the *AdaptationHandler* at the initiator starts the reset-transition wave. The *AdaptationHandler* at each process that receives the reset-transition wave performs the following tasks:

1. It propagates this wave to all of its children.

2. It invokes the *checkState* function of the component, which returns one of the three values: *safetoremove*, *safetoblock* or *unsafetoremove*.

   (a) If the function returns *safetoremove*, the *AdaptationHandler* requests the *ComponentManager* to remove the old component fraction and activate the new component. It then sets itself into the normal state.

(b) If the function returns *safetoblock*, the *AdaptationHandler* requests the *ComponentManager* to block the component fraction. It periodically calls *checkState* until it returns *safetoremove*. After a component fraction receives information about other component fractions being removed or other application processes being blocked, eventually, *checkState* function at the blocking process will return *safetoremove*. When that occurs, we follow case 2a.

(c) If the function returns *unsafetoremove*, it is periodically invoked till it returns *safetoblock* or *safetoremove*, in which case we follow the case 2b or 2a respectively.

During the transition phase, *AdaptationHandler* at each process will normally invoke a sequence (zero or more) of operations provided by *ICMAdaptReady* interface. These operations are invoked before or after each invocation of *checkState*. The details of what operations are invoked are dependent on the adaptation logic and are specific to the application and the component at hand. Typically, these operations are related to state transfer and to ensure that eventually *checkState* function returns *safetoremove*.

Reset-completion wave. The transition wave is reflected towards the initiator (root) as the reset-completion wave. The leaf process sends the reset-completion wave to its parent after it removes the old component fraction and starts using the new component fraction. Any non-leaf process, which completes the component fraction replacement and receives the reset-completion wave from all of its children, sends the reset-completion wave to its parent. Once the initiator has replaced its component fraction and received the reset-completion wave from all of its children, the component replacement is complete.

9.3.3.3 Customizing Reset Protocol

In the above discussion, we laid out a general overview of how the reset protocol is used in adaptation. However, depending on the application context, each implementation of *AdaptationHandler* customizes the reset protocol. For example, depending on the components and the application, the adaptation logic to replace the component may not require all of the two reset waves. In the case of mixed-mode adaptation discussed in Chapter 6 we need only one reset wave. Moreover, the operations invoked at each process during the reset waves will normally vary depending on the components at hand. Specifically, as discussed above, during transition wave each process will invoke a sequence of operations of *ICMAdaptReady* interface based on the adaptation logic.

9.3.3.4 Fault-Tolerant Reset Protocol

By treating the reset protocol discussed earlier as an intolerant application and using the fault-tolerance components from [96, 97], we can make the reset protocol fault-tolerant. If we were to add the fault-tolerance component from [96], the resulting protocol will ensure that stabilizing fault-tolerance [59] is provided to faults including process/channel failures/repairs and transients. Thus, even if these faults occur, eventually the application will recover to a state from where subsequent component replacements will be correct. If we were to add the fault-tolerance component from [97], in addition to the stabilizing fault-tolerance to these faults, the resulting protocol will provide masking fault-tolerance to process/channel failures/repairs. Thus, if only process/channel failures/repairs occur then the component replacement will be always correct. Moreover, if more general faults such as

transients occur then the protocol will recover to a state from where subsequent component replacements will be correct.

9.4 Case Study: Leader Election Component Family

In this section, we discuss the case study of leader election component family, which we denote by *LEFamily*. *LEFamily* consists of components that implement leader election protocol. In Chapter 6, we discussed the abstract model of two leader election protocols, namely *ldrId* and *ldrVal*, and mixed-mode adaptation from *ldrId* to *ldrVal*. In this section, we discuss the details of different interfaces and implementation of different parts of *LEFamily*.

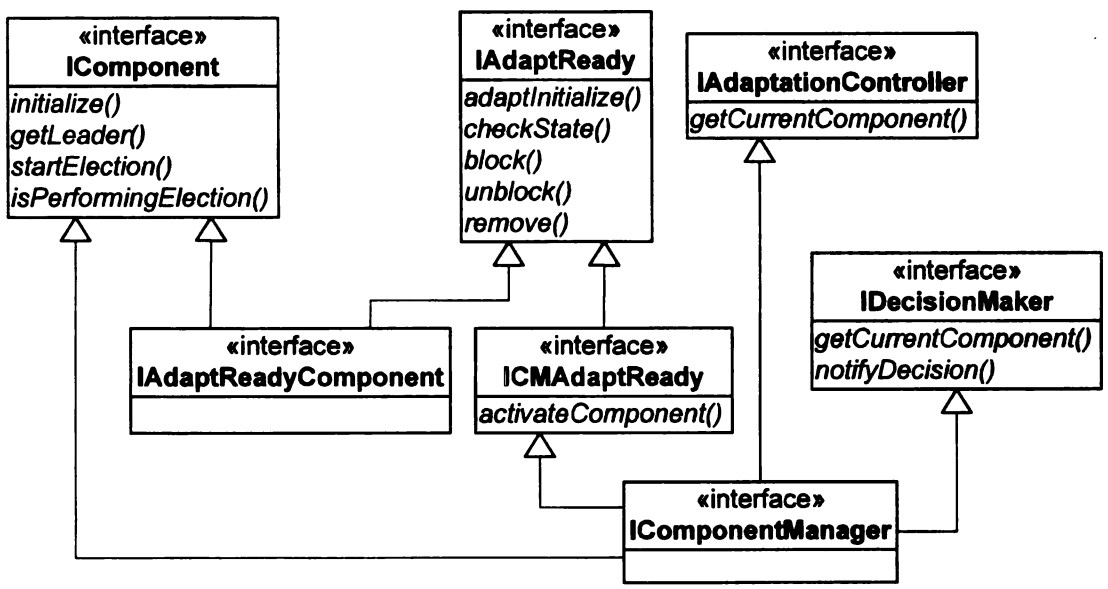


Figure 9.8: Interfaces of the leader election component family.

9.4.1 Interfaces of the Family

The interfaces of *LEFamily* are shown in Figure 9.8. We now discuss the functions provided by each interface:

- ***IComponent.*** This interface provides functions supported by leader election components.
  - `initialize` function is used by the application to initialize the leader election component before using it. The initialization parameters that a component needs from an application include host address and neighbor list.
  - `getLeader` function returns the address of the leader. In our implementation, `getLeader` function blocks if leader election is underway. `getLeader` can also have a non-blocking implementation that will return the address of the last known leader process.
  - `startElection` function is used by the application to start the election.
  - `isPerformingElection` function is used by the application to check if the protocol is performing election.
- ***IAdaptReady.*** This interface provides functions that correspond to adapt-active parts of the component. They include functions that are used by *AdaptationHandlers* to perform adaptation.
  - `adaptInitialize` function is used to initialize the component during adaptation. It uses *StateMappingObject* to read the state of the previous component.

- `checkState` function returns one of the three values: *safetoremove*, *safetoblock* or *unsafetoremove*.
- `block` function blocks the component from starting a new election.
- `unblock` removes the block so that the component can start a new election.
- `remove` function releases any resources and transfers the current state of the component to *StateMappingObject*.

- ***IAdaptReadyComponent***. This interface extends *IComponent* and *IAdaptReady* interface. Both the leader election components in *LEFamily* implement this interface.

- ***ICMAdaptReady***. This interface extends *IAdaptReady* and provides additional functions that are used by *AdaptationHandlers* to perform adaptation. It includes `activateComponent` function that activates the new component (so that calls to *ComponentManager* are now delegated to the new component).

- ***IAdaptationController***. This interface provides `getCurrentComponent` that is used by *AdaptationController* to get the current active component.

- ***IDecisionMaker***. This interface provides `getCurrentComponent` that is used by *DecisionMaker* to get the current active component and `notifyDecision` to inform *ComponentManager* of the decision. The *ComponentManager* provides this interface only at the initiator node (a designated node that will initiate adaptation) as *DecisionMaker* is only available at the initiator node.

- ***IComponentManager***. This interface extends *IComponent*, *ICMAdaptReady*, *IAdaptationController* and *IDecisionMaker* interfaces. *IDecisionMaker* interface is

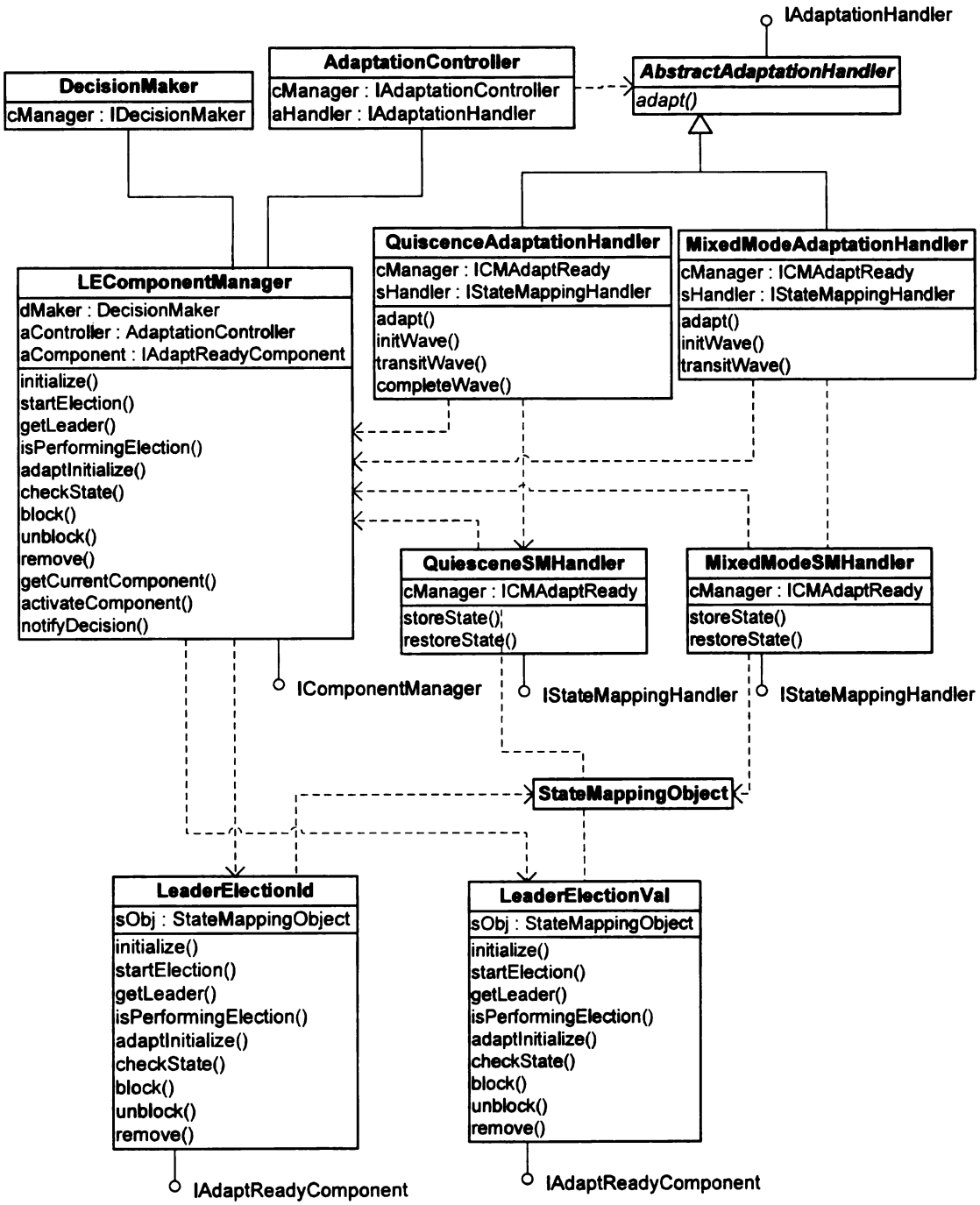


Figure 9.9: Classes of the leader election component family.

extended by *IComponentManager* only at the initiator node.

9.4.2 Classes of the Family

The class diagram of *LEFamily* is shown in Figure 9.9. It consists of the following classes:

- ***LEComponentManager.*** This class implements *IComponentManager* interface. At the initiator node, it creates and maintains an instance of *DecisionMaker.* It informs *DecisionMaker* once it starts using a new component. It also maintains an instance of *AdaptationController.* At the initiator node, it informs *AdaptationController* of the new component that need to be installed. At the non-initiator nodes it creates an instance of *AdaptationController*, which listens on the adaptation port. The `initialize` function of *LEComponentManager* class is used to create an instance of *LeaderElectionId* or *LeaderElectionVal.* It delegates the calls received on *IComponent* and *IAdaptReady* to the active instance of the component.

- ***LeaderElectionId.*** This class implements leader election protocol based on process id. The abstract version of the protocol is discussed in Chapter 6.

- ***LeaderElectionVal.*** This class implements leader election protocol based on process value. The abstract version of the protocol is discussed in Chapter 6.

- ***DecisionMaker.*** This class is installed only at the initiator node and it notifies *LEComponentManager* of the decision. In our current implementation, *DecisionMaker* gets the decision (on when and what to adapt) directly from the user and passes that decision to *LEComponentManager.*

- ***AdaptationController.*** This class uses strategy pattern [89] to instantiate the class that implements the adaptation logic. It has two implementations:

- *Initiator.* At the initiator node it gets the input from *ComponentManager.* Upon receiving the input, it initializes the appropriate *AdaptationHandler* (*QuiescenceAdaptationHandler* or *MixedModeAdaptationHandler*). In our current implementation, we let the user choose the type of adaptation. Another possibility is to let *AdaptationController* decide itself the type of adaptation based on some rules.

- *Non-Initiator.* At the non-initiator node, *AdaptationController* listens on the adaptation port for requests from the other nodes. Based on initialization parameters received during reset-initialization wave, it instantiates an appropriate *AdaptationHandler.*

The *AdaptationController* can be extended to check for the availability of the new component, to retrieve the new component from the server, and to verify if the new component satisfies the contract of the family. In our current implementation we assume that the new component is available and is verified.

- ***AbstractAdaptationHandler.*** In our current implementation, this class provides abstract `adapt` function, whose behavior is defined in the derived classes. This class can be extended to provide behaviors common to all *AdaptationHandlers.*

- ***QuiescenceAdaptationHandler.*** This class is derived from *AbstractAdaptationHandler* and it implements the quiescence adaptation between *LeaderElectionId* and *LeaderElectionVal.* At the initiator node, it acts as the root of the reset protocol, and at non-initiator nodes it acts as a participant in the reset protocol.

- ***MixedModeAdaptationHandler***. This is similar to *QuiescenceAdaptationHandler* except that it implements mixed-mode adaptation.

- ***QuiescenceSMHandler***. This class is used by *QuiescenceAdaptationHandler* to do state mapping from *LeaderElectionId* to *LeaderElectionVal* and vice-versa during adaptation. It gets a reference to *LEComponentManager* from *QuiescenceAdaptationHandler*, and also maintains *StateMappingObject*. It stores the state of the old component in *StateMappingObject* and uses it to initialize the state of the new component.

- ***MixedModeSMHandler***. This class is similar to *QuiescenceSMHandler*, however, it implements the mapping required by mixed-mode adaptation.

- ***StateMappingObject***. This class stores the state of the components during adaptation. The reference to this object is passed to the old component, which stores its current state in this object. The *StateMappingHandlers* use appropriate state mapping on this object to map the state of the old component to the new component, and then passes the reference to this object to the new component during initialization of the new component.

9.4.3 Performance Results

When comparing performance of adaptive systems with non-adaptive systems, it is reasonable to assume that adaptive systems will have some performance overhead. When the system is not adapting, the adaptive system is functionally equivalent to corresponding

non-adaptive system. Therefore, there should be little or no overhead from the mechanisms that make non-adaptive systems adaptive.

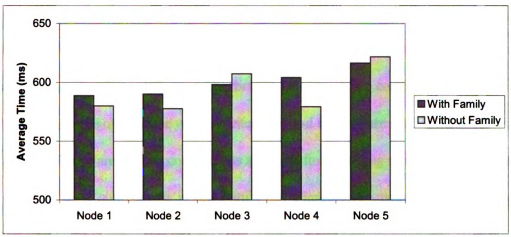


Figure 9.10: Component family performance.

We now present experimental results to show that the component family design does not impose any significant overhead on the use of components. The chart in Figure 9.10 shows the average time of leader election (using *ldrId* component) at each node for the two cases: (i) application using the leader election protocol directly, and (ii) application using the component family. The time of leader election is calculated from the instance when the application makes a request for leader election. In the case of component family, the request by application for leader election is delegated to the active component (in this case *ldrId*). We observe that the time for leader election in both cases is almost similar. At three nodes (node 1, node 2 and node 4) the time for leader election is less when component is used directly, whereas at the other two nodes (node 3 and node 5) it is slightly more. We conclude that overhead induced by component family is negligible. In particular, any overhead due to delegation is within the variance of the time for leader election.

9.5 Case study: Reliable Communication Component Family

In this section, we discuss the component family consisting of components that provide reliable communication. We describe abstract version of the component family in this section. The details of concrete version (interfaces and classes) are similar the case study of Section 9.4. There are various components available for providing reliable communication. We consider three components in this family (cf. Figure 9.11), namely, (i) forward error correction (`fec`) component, (ii) acknowledgment (`ack`) component, and (iii) forward error correction with acknowledgment (`fecAck`) component. Components `fec` and `ack` are similar to the proactive component discussed in Chapter 4 and the reactive component discussed in Chapter 5, respectively. For simplicity, we discuss only one sender and one receiver; however, the case study can be easily extended for multiple sender and multiple receivers.

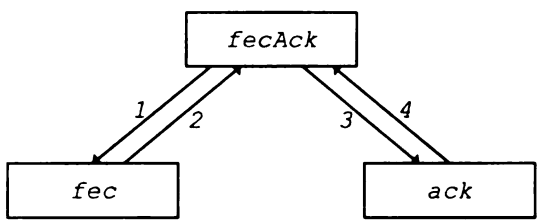


Figure 9.11: Reliable communication component family.

9.5.1 Components of the Family

We now discuss abstract models of the components and the adaptations involved in the component family. In Chapters 4 and 5, we gave details of these components. The com-

ponents discussed in this chapter have unbounded buffer, whereas the reactive component discussed in Chapter 5 had bounded buffer. Also, while the reactive component in Chapter 5 used both acknowledgments and negative acknowledgments, `ack` component discussed in this chapter uses only negative acknowledgments.

Forward Error Correction (`fec`) Component. `fec` component is used to deal with message loss by sending extra packets. The receiver can recover the lost packets without requesting retransmission of lost packets. `fec` component consists of two types of fractions: encoder and decoder. The encoder fraction is added at the sender process and the decoder fraction is added at the receiver process. The encoder takes $(n - k)$ data packets and encodes them to add k parity packets. It then sends the group of n (data and parity) packets. The decoder needs to receive at least $(n - k)$ packets of a group to decode all the data packets. Thus, if less than k packets in a group are lost then the receiver can recover the lost packets by using the extra packets from that group.

The abstract version of `fec` component is shown in Figure 9.12, which consists of encoder fraction `s_fec` at sender process s and decoder fraction `r_fec` at receiver process r. The encoder fraction reads the input from sQ, encodes the input and sends it to r. The decoder fraction receives the input from s, decodes it and delivers it to the output listener, rQ.

Acknowledgment (`ack`) Component. `ack` component deals with message loss by retransmitting the lost packets. It uses negative acknowledgments to confirm the message loss. It consists of the encoder fraction at the sender and the decoder fraction at the receiver. The encoder fraction adds a group and a packet number in each packet. If it receives a negative acknowledgment for any packet, it sends that packet again to the decoder fraction.

Component *fec*

Fraction *s_fec*

inp sQ : **queue of integer**
r
var n, k, u, l, m : **integer** {initially, $u = l = m = 0$}
encQ : **array [integer,** $0..n-1$**] of integer** {initially, $encQ = \perp$}
begin
encode : $\neg$isEmpty(sQ) $\rightarrow$ $encQ[u, 0..n-1] :=$ fec_encode(head($sQ, n-k$));
$u := u + 1$
[] **fec_send** : $encQ[l, m] \neq \perp \rightarrow$ $C_{s,r} := C_{s,r} \circ \{l, m, encQ[l, m]\}$;
$m := (m + 1)$ **mod** n;
if $m = 0$ **then**
$l := l + 1$
fi
end

Fraction *r_fec*

inp rQ : **queue of integer**
s
var n, k, x, y, p, m : **integer** {initially, $p = 0$}
rbufQ : **array [integer,** $0..n-1$**] of integer** {initially, $rbufQ = \perp$}
begin
fec_receive : $\neg$isEmpty($C_{s,r}$) $\rightarrow$ $x, y, m :=$ head($C_{s,r}$);
$rbufQ[x, y] := m$
[] **decode** : count($rbufQ[p, 0..n-1] \neq \perp$) $>= (n-k) \rightarrow$
$rQ := rQ \circ$ fec_decode($rbufQ[p, 0..n-1]$);
$p := p + 1$
end

Figure 9.12: Forward error correction component.

When the decoder fraction detects a packet loss it sends a negative acknowledgment to the encoder fraction. The decoder fraction detects packet loss when it starts receiving packets from a new group before all the packets of the current group are received. When all packets in a group are received, it delivers them to the output listener.

The abstract version of `ack` component is shown in the Figure 9.13, which consists of encoder fraction `s_ack` at sender process s and decoder fraction `r_ack` at receiver process

Component *ack*

Fraction *s_ack*

inp sQ : **queue of integer**
r

var $n, g, p, g_{na}, p_{na}, m$: **integer** {initially, $p = g = 0$}
snt : **array [integer**, $0..n-1$] **of integer** {initially $snt = \perp$}

begin

ack_send : $\neg$isEmpty(sQ) $\rightarrow$ $snt[g,p] := \{g, p, \mathtt{head}(sQ)\}$;
$C_{s,r} := C_{s,r} \circ snt[g,p]$;
$p := (p+1) \bmod n$;
if $p = 0$ **then**
$g := g + 1$
fi

[] **resend** : $\mathtt{type}(C_{r,s}) = \mathtt{nack} \rightarrow g_{na}, p_{na} := \mathtt{head}(C_{r,s})$;
$C_{s,r} := C_{s,r} \circ snt[g_{na}, p_{na}]$

end

Fraction *r_ack*

inp rQ
s

var n, k, x, y, p, m : **integer** {initially, $p = 0$}
$rbufQ$: **array [integer**, $0..n-1$] **of integer** {initially, $rbufQ = \perp$}

begin

ack_receive : $\neg$isEmpty($C_{s,r}$) $\rightarrow$ $x, y, m := \mathtt{head}(C_{s,r})$;
$rbufQ[x,y] := m$

[] **deliver** : $\mathtt{count}(rbufQ[p, 0..n-1] \neq \perp) = n \rightarrow$
$rQ := rQ \circ rbufQ[p, 0..n-1]$;
$p := p + 1$

[] **send_nack** : $\mathtt{count}(rbufQ[p+1, 0..n-1] \neq \perp) > 0 \rightarrow$
for $k = 0$ **to** $n-1$
if $rbufQ[p,k] = \perp$ **then**
$C_{r,s} := C_{r,s} \circ \mathtt{nack}(p,k)$
fi

end

Figure 9.13: Acknowledgment component.

r. Similar to `fec` component, the encoder fraction has sQ and r as the input parameters.

The decoder fraction has rQ and s as the input parameters.

Component *fecAck*

Fraction *s_fecAck*

inp sQ : **queue of integer**
r

var n, k, u, l, m : **integer** {initially, $u = l = m = 0$}
encQ : **array [integer,** $0..n-1$**] of integer** {initially, $encQ = \perp$}
ackMode : **boolean** {initially, *true*}

begin

encode : $\neg$isEmpty(sQ) $\rightarrow$ $encQ[u, 0..n-1] :=$
fec_encode(head($sQ, n-k$));
$u := u + 1$

[] **fecAck_send** : $encQ[l, m] \neq \perp \rightarrow$ $C_{s,r} := C_{s,r} \circ \{l, m, encQ[l, m]\}$;
if $ackMode = true$ **then**
$snt[l, m] = \{l, m, encQ[l, m]\}$;
fi
$m := (m + 1)$ **mod** n;
if $m = 0$ **then**
$l := l + 1$
fi

[] **resend** : $ackMode = true \wedge$ type($C_{r,s}$) = nack $\rightarrow$
$gna, pna :=$ head($C_{r,s}$);
$C_{s,r} := C_{s,r} \circ snt[gna, pna]$

end

Figure 9.14: Forward error correction with acknowledgment: sender fraction.

Forward Error Correction with Acknowledgment (`fecAck`) Component. `fecAck` component uses both forward error correction and acknowledgments. If the rate of message loss is high and more than k packets are lost in a group, then negative acknowledgments can be used for retransmission of the lost packets. If the rate of message loss is low and less than k packets are lost in a group, then the receiver can recover the lost packets without requesting any retransmission.

The abstract version of `fecAck` component is shown in Figures 9.14 and 9.15, which consists of `s_fecAck` fraction at the sender process and `r_fecAck` fraction at the receiver

Component *fecAck*

Fraction *r_fecAck*

inp *rQ* : **queue of integer**
s

var n, k, x, y, p, m : **integer** {initially, $p = 0$}
rbufQ : **array [integer**, $0..n-1$] **of integer** {initially, $rbufQ = \perp$}
ackMode : **boolean** {initially, *true*}

begin

fec_receive : $\neg$isEmpty($C_{s,r}$) $\rightarrow$ $x, y, m := \texttt{head}(C_{s,r})$;
$rbufQ[x, y] := m$

[] **decode** : $\texttt{count}(rbufQ[p, 0..n-1] \neq \perp) >= (n-k) \rightarrow$
$rQ := rQ \circ \text{fec\_decode}(rbufQ[p, 0..n-1])$;
$p := p + 1$

[] **send_nack** : $ackMode = true \wedge \texttt{count}(rbufQ[p+1, 0..n-1] \neq \perp) > 0 \rightarrow$
for $k = 0$ **to** $n-1$
if $rbufQ[p, k] = \perp$ **then**
$C_{r,s} := C_{r,s} \circ \texttt{nack}(p, k)$
fi

end

Figure 9.15: Forward error correction with acknowledgment: receiver fraction.

process.

9.5.2 Adaptations of the Family

We now consider the adaptations that exist in this family as shown in Figure 9.11. There are four adaptations that exist in this family, namely, (1) `fecAck` to `fec`, (2) `fec` to `fecAck`, (3) `fecAck` to `ack`, and (4) `ack` to `fecAck`. The arcs in the graph are labeled accordingly. (Note that the maximum number of possible adaptations in a family of three components is six, and the minimum number of required adaptations to keep the graph strongly connected is three).

The adapt-active parts of each fractions that are involved in adaptations are shown in

Adapt-active parts for adaptation 1 : `fecAck` to `fec`

Fraction : `s_fecAck`

a_{11s} : true $\rightarrow$ *ack_mode* := false

a_{12s} : $a_{11s} \wedge a_{11r} \rightarrow$ `sremove s_fecAck` $[n, k, u, l, m, eQ]$

Fraction : `r_fecAck`

a_{11r} : true $\rightarrow$ *ack_mode* := false

a_{12r} : $a_{11s} \wedge a_{11r} \rightarrow$ `sremove r_fecAck` $[n, k, p, m, rbufQ]$

Fraction : `s_fec`

a_{12s} : a_{12s}(`s_fecAck`) $\rightarrow$ `sadd s_fec` $[n, k, u, l, m, eQ]$

Fraction : `r_fec`

a_{12r} : a_{12r}(`r_fecAck`) $\rightarrow$ `sadd r_fec` $[n, k, p, m, rbufQ]$

Adapt-active parts for adaptation 2 : `fec` to `fecAck`

Fraction : `s_fec`

a_{21s} : true $\rightarrow$ `sremove s_fec` $[n, k, u, l, m, eQ]$

Fraction : `r_fec`

a_{21r} : true $\rightarrow$ `sremove r_fec` $[n, k, p, m, rbufQ]$

Fraction : `s_fecAck`

a_{21s} : a_{21s}(`s_fec`) $\rightarrow$ `sadd s_fecAck` $[n, k, u, l, m, eQ]$; *ackMode* := *true*

Fraction : `r_fecAck`

a_{21r} : a_{21r}(`r_fec`) $\rightarrow$ `sadd r_fecAck` $[n, k, p, m, rbufQ]$; *ackMode* := *false*

a_{22r} : a_{21s} $\rightarrow$ *ackMode* := *true*

Figure 9.16: Adaptations 1 and 2 in reliable communication component family.

Figures 9.16 and 9.17. The adapt-active parts of the fractions contain atomic adaptations that are associated with them. The name of each atomic adaptation has three subscripts (*e.g.*, a_{13s}). The first subscript denotes the adaptation (cf. label of the arc in Figure 9.11). The second subscript denotes the order in the sequence of atomic adaptations. If two atomic adaptations has the same second subscript, it means that they can be executed in any order. The third subscript denotes the process that the atomic adaptation is associated with. For example, atomic adaptation a_{13s} denotes that in adaptation 1 (`fecAck` to `fec`), it is third in the sequence of atomic adaptations and it occurs at process s.

Adaptation 1: `fecAck` *to* `fec` *having enhanced-primitive relationship.* Consider `fec` and `fecAck` components as shown in Figures 9.12, 9.14 and 9.15, respectively. `fecAck`

Adapt-active parts for adaptation 3 : `fecAck` to `ack`

Fraction : `s_fecAck`

a_{31s} : true $\rightarrow$ `block encode`

a_{33s} : a_{32r} $\rightarrow$ `remove s_fecAck`

Fraction : `r_fecAck`

a_{32r} : $a_{31s} \wedge encQ[l,m] = \perp \wedge \neg$isEmpty$(C_{s,r})$ $\rightarrow$ `remove r_fecAck`

Fraction : `s_ack`

a_{34s} : $a_{33s} \wedge a_{33r}$ $\rightarrow$ `add s_ack`

Fraction : `r_ack`

a_{33r} : a_{32r} $\rightarrow$ `add r_ack`

Adapt-active parts for adaptation 4 : `ack` to `fecAck`

Fraction : `s_ack`

a_{41s} : true $\rightarrow$ `block send`

a_{43s} : a_{42r} $\rightarrow$ `remove s_ack`

Fraction : `r_ack`

a_{42r} : $a_{41s} \wedge \wedge \neg$isEmpty$(C_{s,r})$ $\rightarrow$ `remove r_ack`

Fraction : `s_fecAck`

a_{44s} : $a_{43s} \wedge a_{43r}$ $\rightarrow$ `add s_fecAck`

Fraction : `r_fecAck`

a_{43r} : a_{42r} $\rightarrow$ `add r_fecAck`

Figure 9.17: Adaptations 3 and 4 in reliable communication component family.

component is syntactically compatible with `fec` and it also provides all the services that `fec` provides. In this case, `fecAck` is an enhanced version of `fec`, which is a primitive version. During the adaptation that replaces `fecAck` to `fec`, the fractions can be changed arbitrarily, provided (i) `fecAck` component is running in a primitive mode before the adaptation begins, and (ii) state of `fecAck` component is transferred to `fec` component. As shown in Figure 9.16, first *ack_mode* is set to false (a_{11s} and a_{11r}); as a result the `fecAck` component is now running in primitive mode, *i.e.*, in a mode compatible with `fec`. Now, the fractions can be changed arbitrarily (a_{12s} and a_{12r}). There are two atomic adaptations named a_{12s}, one associated with fraction `s_fecAck` and another associated with fraction `s_fec`. This implies that `sremove` and `sadd` can be considered as an

atomic action that affects both fractions `s_fecAck` and `s_fec`. Similarly, removal of `r_fec` and addition of `r_fecAck` is done in an atomic manner. Note that `sremove` and `sadd` carry an extra argument, which represents the state information that is transferred from the existing component to the new component. In this case, the state of `fecAck` component is transferred to `fec` component. The adaptation lattice corresponding to this adaptation is shown in Figure 9.18(a).

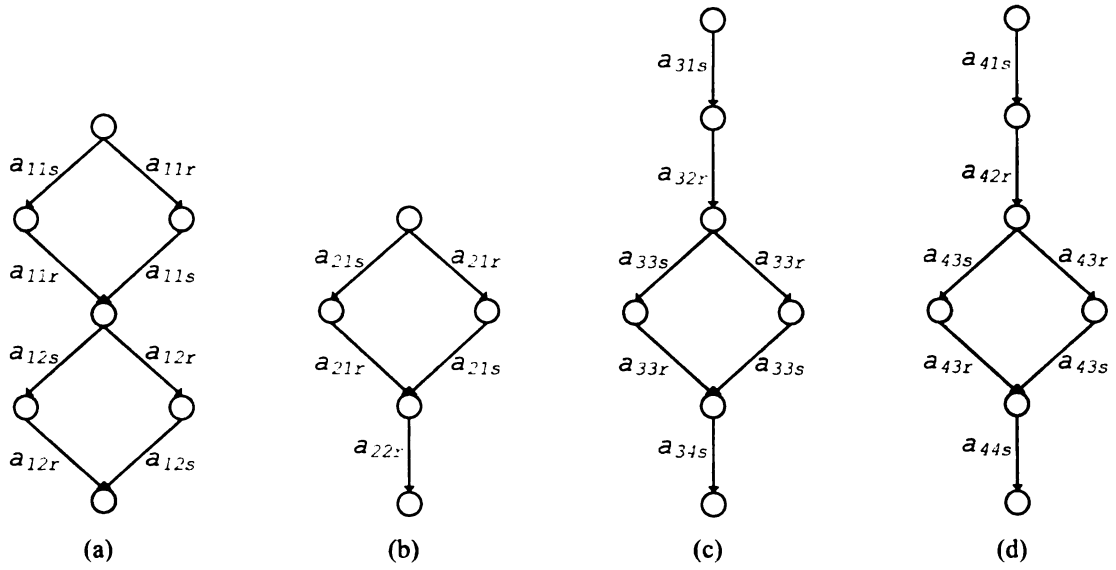


Figure 9.18: Adaptation lattices.

Adaptation 2: *`fec` to `fecAck` having primitive-enhanced relationship.* As discussed in the previous adaptation, in this case, the fractions can be changed arbitrarily, as the components share a primitive-enhanced relationship. We need to ensure that (i) state of `fec` component is transferred to `fecAck` component, and (ii) `fecAck` runs in a primitive mode till the adaptation is complete. Initially, *ack_mode* is set to false, so `fecAck` runs in a mode that is compatible with `fec`. After the adaptation is complete, *ack_mode* is set to true. The adaptation from `fec` to `fecAck` is as shown in Figure 9.16 and the adaptation

lattice corresponding to this adaptation is shown in Figure 9.18(b).

Adaptation 3: *fecAck to ack*. The adaptation that replaces *fecAck* to *ack* is as shown in Figure 9.17. Unlike adaptations 1 and 2, components *fecAck* and *ack* do not share an enhanced-primitive relationship, because *fecAck* component is not designed to run in *ack*-only mode. Specifically, fecAck_send action of *fecAck* is not compatible with ack_send action of *ack*. Hence, *fecAck* component does not provide all the functionality that *ack* component does. Therefore, the fractions cannot be changed arbitrarily. First, encoder fraction *s_fecAck* at sender s needs to be blocked from encoding more packets (a_{31_s}). Once all the packets that are already encoded are sent by s, and received by receiver r, then decoder fraction *r_fecAck* can be removed (a_{32_r}). After *r_fecAck* fraction is removed, the removal of *s_fecAck* fraction (a_{33_s}) and the addition of *r_ack* fraction (a_{33_r}) can be done in any order. Finally, after *s_fecAck* fraction at s is removed and *r_ack* fraction at r is added, encoder fraction *s_ack* is added at s (a_{34_s}). The adaptation lattice corresponding to this adaptation is shown in Figure 9.18(c). (Note that *fecAck* and *ack* could be modified so that they satisfy the enhanced-primitive relationship. We have chosen not to do so to illustrate the case where the components are not related by an enhanced-primitive relationship.)

Adaptation 4: *ack to fecAck*. The adaptation that replaces *ack* to *fecAck* is similar to adaptation 3 discussed above. Since the components do not share an enhanced-primitive relationship, the fractions need to be changed in a specific order as shown in Figure 9.17. The adaptation lattice corresponding to this adaptation is shown in Figure 9.18(d).

9.6 Discussion

In this section, we address some questions related to the component family design.

Can there be multiple adaptation paths between two components? If yes, then which adaptation path should be chosen?

Yes. There can be multiple paths between two components. In this case, additional factors could be taken into account while deciding the appropriate path for adaptation. To this end, for each arc, we could associate several factors, *e.g.*, the time or resources required for adaptation, types of faults that could be tolerated in the adaptation. Based on the factors associated with each arc, we can compute the characteristics for different paths. These characteristics can then be used to determine the suitable path.

How do we develop components that satisfy the enhanced-primitive relationship?

One way to design such components is to use inheritance. Inheritance provides syntactic compatibility between components. To ensure semantic compatibility, we need to extend the inheritance relationship, such that, a derived component provides all services that a parent component provides.

How does the component family design help when adaptation involves components providing different functionalities, say security and reliability?

In this case, there will be two separate component families, namely, a family of components that provide reliability and a family of components that provide security. The application will have to perform separate adaptations for reliability and security components.

There may exist a scenario, although undesirable as it violates the principle of separa-

tion of concerns, where some component, say C, provides reliability as well as security. In this scenario, C will be present in both the families. Now, if the application that is using C to provide security and reliability decides to use only a security component, then the application can perform adaptation to replace C with a security component. Also, existence of such components could be used in adaptations where one needs to trade off between two desirable properties such as reliability and security. Specifically, if the application were to replace a reliability component by a security component (may be because of environment changes that require it to have security but where reliability cannot be provided due to other constraints such as energy management) then the application could first replace the reliability component with C and then replace C by the security component.

How can we perform the adaptation where some component is removed although not replaced by other component?

If such a scenario is desired for a particular component family then that family should have a *default component* which is equivalent to having no component at all. For example, in the context of our case study in Section 9.5, the default component would be one that provides no recovery for lost messages. Thus, removal of a component is equivalent to replacing that component by the default component. This approach is similar to that in [3].

How does component family assist in independent development of components?

Consider a scenario, where components, say A and B, are developed for a component family $\mathcal{F}$. These components can be developed independently and we can still perform adaptation from A to B and vice-versa, even if direct adaptations were not to exist between A and B. One way to achieve this would be if adaptations from A and B to some compo-

nent C and vice-versa were identified. In this case, adaptations between A and B could be done via C. Specifically, this approach is easy to comprehend when a component family has a primitive component and different enhanced components are developed corresponding to that primitive component. The adaptation between any two enhanced components can be done via the corresponding primitive component. Examples of such adaptation can be found in [91].

How can component family design be used to develop parallel adaptation?

To implement parallel adaptation, *ComponentManager* can be modified so that it maintains active instances of more than one component. It can delegate the request received from the application to the appropriate component. Also, the switching algorithms discussed in [98] for parallel adaptation of distributed agreement protocols can be implemented as *AdaptationHandlers*.

9.7 Related Work

In this chapter, we introduced the notion of component family to build a systematic and extensible library of adaptive components. The idea is based on the original notion of program families proposed by Parnas [86, 90]. A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for which it is profitable or useful to consider as a group. However, the idea of program families does not address issues related to adaptation. In this work, we try to study different components with adaptation in mind. Moreover, we focus on components with similar functionalities and interfaces. In other words, our focus is more narrow compared to the general goals for program families

where components with (slightly) different functionalities and interfaces are also studied together (for different reasons).

In feature-oriented programming [99, 100] and software product lines [101], goals of product lines (creating family of related programs) are addressed. Product families provides an architecture that is based on commonality and similarity. In product families, various product variants can be derived from the basic product family, thereby, allowing reuse of products in the family. The main focus of product lines (product families) is from the perspective of static development and reuse. In contrast, the component family design addresses dynamic reuse of components. The current design of component family focuses on components having similar functionalities and interfaces. We believe that the product family and feature-oriented programming work can be leveraged when we consider extensions to the component family to build a library of adaptive components that have different interfaces.

9.8 Summary

In this chapter, we presented a systematic and extensible component family design to build a library of adaptive components. In summary, we discuss the following advantages:

1. Simplifying adaptation between components and enabling independent development of new components. Consider the case where an application is using a component from a component family $\mathcal{F}$ consisting of n components. In this case, to provide adaptation between any two components of a family $\mathcal{F}$, we need a minimum of only n adaptations (in this case, the graph consists of a directed cycle). When a new component is developed that

will be a part of $\mathcal{F}$, it suffices to have only two more adaptations while still keeping the graph strongly connected. For example, if a new component E is added to the component family shown in Figure 9.2, only two adaptations, say from C to E and from E to B, are enough to keep the graph strongly connected. Also, since every component implements *IAdaptReady* interface, it becomes easy to develop adaptation between these components.

2. Simplifying verification of adaptation. To verify that the adaptation between components is correct (*e.g.*, by using the approach in Chapters 4 and 5), it is required that after adaptation the component continues to correctly perform its functionality, and specification during adaptation is satisfied. The separation of adaptation logic from component functionality simplifies the task of specifying and verifying adaptation. Moreover, the adaptation lattice specification has a direct mapping to the implementation (*AdaptationHandlers*). Furthermore, if the number of such adaptations is low, then less verification needs to be performed.

3. Reusability of components and adaptations. The design of component family not only enhances the reuse of components but also promotes the reuse of adaptations between components. For example, consider two components X and Y and that the adaptation from X to Y exists. Now, consider a component Z and the adaptation from Z to Y. In this case, by providing the adaptation from Z to X, the adaptation from Z to Y can be done in two steps while reusing the adaptation that already exists from X to Y. We note that if the direct adaptation from Z to Y were to exist, it would not necessarily be fast or simple. In fact, there are cases, as discussed in next point, where a two (or more) step adaptation is simpler than a direct adaptation between components.

4. Simplifying adaptation in case of an *enhanced-primitive* relationship among com-

ponents. The state-transfer and synchronization during adaptation is in general difficult between arbitrary components. However, if one component is an enhancement of another component, then the state-transfer and synchronization can be simplified in adaptation between those two components (cf. Section 9.5 for an example). To take advantage of this, we define the enhanced-primitive relationship between components. We say that a component A is an *enhanced* component of component B (respectively, component B is a *primitive* component of A) iff A is syntactically and semantically compatible with B, *i.e.*, it extends the interface of B, and it provides all services that B provides.

Now, consider a scenario where A is an enhanced component of B and that B is being replaced by A. In this scenario, fractions of B can be replaced in an arbitrary order by fractions of component A, as each fraction of A can provide the required service to the remaining fractions of B. Moreover, the fractions of A can communicate with the remaining fractions of B using a protocol that the latter understands. Thus, in this case, the synchronization requirement among component fractions is relaxed, and also transferring state to/from primitive component is easy.

In a case where two enhanced components, say C and D, are derived from the same primitive component, say Z, the adaptation from C to D can be done in two steps; by first replacing C by Z and then replacing Z by D. Since the adaptation for enhanced-primitive relationship is relatively easy, the direct adaptation from C to D may not necessarily be fast or easy. We note that the idea of an enhanced-primitive relationship can be extended to have a multi-level hierarchy of components, where components at a higher level are enhanced version of components at lower level.

5. Easy replacement of adaptation logic. An adaptation from one component to another

component can be performed in several ways. For example, we discussed quiescence and mixed-mode adaptations in this dissertation. Moreover, we used reset-based approach to achieve synchronization among processes in distributed system during adaptation. However, other approaches or variations of reset protocol can be used for synchronization. The environment conditions or resource constraints may determine which adaptation logic will provide optimum performance. The component family design has *AdaptationHandlers* that implement adaptation logic and *AdaptationController* can choose among different *AdaptationHandlers* that will execute the adaptation. This clear separation of adaptation logic makes it easy to replace between different adaptation logics.

Chapter 10

Conclusion and Future Work

Software systems undergo adaptation for reasons that include changes in environment conditions, fixing of some bugs, or changes in requirements. In the case of distributed systems, adaptation causes changes to multiple processes. It is required that these changes to multiple processes be properly synchronized in order for the adaptation to provide acceptable behavior. Furthermore, in the case of adaptation in distributed systems, the behavior of the old program and the behavior of the new program may overlap causing mixed-mode behavior during adaptation. To gain assurance in adaptation it is important to formally specify and verify the behavior of the system during adaptation.

10.1 Contributions

In this dissertation, we made the following contributions:

- We presented an approach based on *adaptation lattice* to model the adaptation in distributed systems [102]. The adaptation lattice approach identifies various overlap

scenarios that can occur during adaptation. It classifies system during adaptation into intermediate programs and specifies the specification for the intermediate programs. Specifically, various properties that need to be checked during adaptation can be specified using adaptation lattice.

- We presented an approach based on *transitional-invariant lattice* to verify the correctness of adaptation in the absence of faults, and *transitional-faultspan lattice* to verify the correctness of adaptation in the presence of faults [102]. In the transitional-invariant lattice approach, we identify the invariants associated with the intermediate programs to show the safety during adaptation. In the transitional-faultspan lattice approach, we identify the invariants and the faultspans associated with the intermediate programs to verify the fault-tolerance properties during adaptation. The approach applies to different types of fault-tolerances, namely, failsafe, masking and non-masking. The transitional-invariant and transitional-faultspan lattices also satisfy the liveness during adaptation which states that eventually adaptation is completed.

- We discussed the case study of mixed-mode adaptation using leader election protocols [103]. We showed the performance benefits that can be obtained by mixed-mode adaptation compared to quiescence adaptation. Specifically, mixed-mode adaptation reduces the service interruption time and the communication overhead during adaptation. We also showed how the lattice-based approach can be used to specify and verify mixed-mode behavior during adaptation.

- We identified the tradeoffs that occur in adaptation [104]. The tradeoff shows that

increasing concurrency among adaptive actions leads to increase in verification complexity. Moreover, reducing concurrency leads to decrease in communication overhead due to adaptation. These tradeoffs should assist the adaptation developer in choosing concurrency during adaptation based on complexity of adaptation, adaptation completion time, and resource availability during adaptation.

- Because of complexity, specification and verification of adaptation in distributed systems is often done on abstract model. To gain assurance in implementation, it is important that verification be supplemented by testing. In this context, we presented an approach for testing of adaptation using predicate detection techniques [105]. We introduced *adaptation vector* to classify intermediate program states that occur during adaptation. We also introduce two classes of predicates that occur during adaptation, namely, *adaptation-stable* and *adaptation-transient* predicates. We showed how existing techniques for predicate detection can be extended for testing adaptation.

- We described *component family* design to support adaptation and build an adaptation-verified library of components. The components in the component family provides adaptation-related services in addition to the core functionality to support adaptation. The design of component family also separates the adaptation logic from the component functionality, thereby, simplifying the specification, verification and implementation of the adaptation logic. The design also simplifies independent development of new components and adaptation among components. In this context, we have presented some of the results of *component family* approach in [106]. The component family design integrates the framework for adaptation while ensuring separation of

adaptation logic. The framework supports both mixed-mode and quiescence adaptation. A preliminary design of this framework has been published in [3], and in a recent case study in sensor networks we have used this framework [107].

10.2 Future Research

This dissertation has created the possibility of several new research directions. Some of these are outlined below:

- *Automatic generation of lattices.* In this dissertation, we used transitional-invariant and transitional-faultspan lattice to verify adaptation. However, identifying the lattice is a complex task. We would like to explore methods to generate these lattices in a semi-automatic fashion. Specifically, considering all possible atomic adaptations, we would like to explore approaches that can generate models of different intermediate programs. Also, we would like to explore approaches to identify transitional-invariants and transitional-faultspans automatically.

- *Component family having components with varied interfaces.* In the component family design discussed in Chapter 9, we considered components having same interfaces. We would like to explore ways to extend the design to deal with components having same functionality but (slightly) different interfaces.

- *Applying component family in new domains.* In this dissertation, we presented component family design and illustrated it in context of algorithms in group communication application such as leader election protocols and reliable communication pro-

tocols. We would like to apply this design in other domains, such as web services, middleware components, and other application-level components.

- *Runtime verification.* The component family design allows new components to be included and used in the family at runtime. Currently, we expect that verification of adaptation involving the new component is done offline. We would like to explore approaches to do runtime checking of new components and adaptations involving those components. In this context, we would like to leverage the design of component family as it separates the adaptation logic from component functionality.

APPENDICES

Appendix A

Model-Checking of Adaptive Leader Election Program

In this appendix, we describe the model-checking of adaptive leader election program using SPIN. The leader election programs are discussed in Chapter 6. The code of the adaptive leader election program in Promela language is shown in Section A.1. In Section A.2, we show the results of model checking safety and liveness properties of the adaptive leader election program.

A.1 Adaptive Leader Election Program

The adaptive leader election program in promela is shown below. We consider five processes. All processes are initially using *ldrId* protocol. The program should cover following scenarios for model-checking: (i) adaptation occurs after election, (ii) adaptation occurs before election, (iii) adaptation occurs while election is underway, and (iv) elec-

tion occurs while adaptation is underway. To ensure that above scenarios are covered, we choose two process (namely, process 3 and process 4) to start the election. As discussed in Chapter 6, each process performs its adaptive action independent of others. All processes perform adaptation that replaces the *ldrId* protocol with *ldrVal*.

```
#define N 5 /* number of nodes */
#define L 10    /* buffer size of channel */

/* types of messages */
mtype = { LDRID, LDRVAL, ELECT, ACK, LD };

/* structure to store neighbors for a node */
/* nL[i] (0 <= i < N) = true if i is a neighbor */
typedef ArrayNeighbors {
   bool nL[N] = false
};

/* neighbor list for each node */
hidden ArrayNeighbors Nbhrs[N];

/* computation index (election index) */
typedef  compIndex {
   byte num;
   byte id
};

/* defining channel structure */
typedef Arraychannels {
   chan ch[N] = [L] of {mtype, mtype, compIndex, byte, byte, bool
      }
};

Arraychannels A[N];

hidden compIndex rcvtmp;
hidden byte junkByteID;
bool startNodes[N] = false;    /* nodes that will start election
   */
byte value[N] = 0;        /* value at each node */
bool ADAPT[N] = false;      /* boolean variable indicating if node
    is adapting */

/* for joining in a election - LDRID */
inline joinElectionId(j)  {
```

```
protocolType == LDRID && Nbhrs[j-1].nL[myId-1] && A[j-1].ch[
   myId-1]?[LDRID(ELECT,rcvsrc,tmpbyte,junkByteID,tmpbool)] ->
   atomic {
      A[j-1].ch[myId-1]?LDRID(ELECT,rcvsrc,tmpbyte,junkByteID,
         tmpbool) ->

      /* if E then check if waiting for leader (LD might be in
          queue?) */
      /* need to remove LD from the queue */
      if
      :: E && waitCount == 0 && A[p-1].ch[myId-1]?[LDRID(LD,
         rcvtmp,ldr,junkByteID,tmpbool)] ->
         A[p-1].ch[myId-1]?LDRID(LD,rcvtmp,ldr,junkByteID,
            tmpbool);
      :: else -> skip
      fi;

      if
      :: (!E || (E && (rcvsrc.num > src.num || (rcvsrc.num ==
         src.num && rcvsrc.id > src.id)))) ->

         src.num = rcvsrc.num; src.id = rcvsrc.id;
         mynum = src.num + 1;
         waitCount = 0;
         loopvar = 1;
         do
         :: loopvar <= N ->
            if
            :: (loopvar != j && Nbhrs[myId-1].nL[loopvar-1])
               ->
               A[myId-1].ch[loopvar-1]!LDRID(ELECT,src,tmpbyte
                  ,junkByteID,tmpbool);
               wait[loopvar-1] = true;
               waitCount++;
            :: else ->
               wait[loopvar-1] = false;
            fi;
            loopvar++;
         :: loopvar > N ->
            break
         od;
         if
         :: E -> printf("MSC: Stopping old election. Joining
            new election\n");
         :: else -> printf("MSC: Joining new election\n");
         fi;
         ack = true; E = true; p = j; max = myId;
```

```
            :: E && (rcvsrc.num == src.num && rcvsrc.id == src.id)
               ->
               A[myId-1].ch[j-1]!LDRID(ACK, src, tmpbyte, junkByteID,
                  false);
            :: else -> skip
            fi;
         }
}

inline ackReceiveId(j) {
   protocolType == LDRID && wait[j-1] && A[j-1].ch[myId-1]?[LDRID
      (ACK, rcvsrc, maxchildId, junkByteID, isChild)] ->
      atomic {
         A[j-1].ch[myId-1]?LDRID(ACK, rcvsrc, maxchildId, junkByteID
            , isChild) ->
         if
         :: E ->
            if
            :: ack && src.num == rcvsrc.num && src.id == rcvsrc.
               id ->
               wait[j-1] = false;
               waitCount--;
               if
               :: isChild ->
                  chd[j-1] = true;
                  if
                  :: max < maxchildId -> max = maxchildId;
                  :: else -> skip
                  fi;
               :: else -> skip
               fi;
            :: else -> skip
            fi;
         :: else -> skip
         fi;
      }
}

/* for joining in a election - LDRVAL */
inline joinElectionVal(j) {
   protocolType == LDRVAL && Nbhrs[j-1].nL[myId-1] && A[j-1].ch[
      myId-1]?[LDRVAL(ELECT, rcvsrc, tmpVal, tmpbool)] ->
      atomic {
         A[j-1].ch[myId-1]?LDRVAL(ELECT, rcvsrc, tmpVal, tmpbool) ->

         /* if E then check if waiting for leader (LD might be in
            queue?) */
```

```
        /* need to remove LD from the queue */
        if
        :: E && waitCount == 0 && A[p-1].ch[myId-1]?[LDRVAL(LD,
           rcvtmp,tmpVal,tmpbool)] ->
           A[p-1].ch[myId-1]?LDRVAL(LD,rcvtmp,tmpVal,tmpbool);
        :: else -> skip
        fi;

        if
        :: (!E || (E && (rcvsrc.num > src.num || (rcvsrc.num ==
           src.num && rcvsrc.id > src.id)))) ->

           src.num = rcvsrc.num; src.id = rcvsrc.id;
           mynum = src.num + 1;
           waitCount = 0;
           loopvar = 1;
           do
           :: loopvar <= N ->
              if
              :: (loopvar != j && Nbhrs[myId-1].nL[loopvar-1])
                 ->
                 A[myId-1].ch[loopvar-1]!LDRVAL(ELECT,src,tmpVal
                    ,tmpbool);
                 wait[loopvar-1] = true;
                 waitCount++;
              :: else ->
                 wait[loopvar-1] = false;
              fi;
              loopvar++;
           :: loopvar > N ->
              break
           od;
           if
           :: E -> printf("MSC: Stopping old election. Joining
              new election\n");
           :: else -> printf("MSC: Joining new election\n");
           fi;
           ack = true; E = true; p = j;
           maxVal.id = myId; maxVal.num = value[myId-1];
        :: E && (rcvsrc.num == src.num && rcvsrc.id == src.id)
           ->
           A[myId-1].ch[j-1]!LDRVAL(ACK,src,tmpVal,false);
        :: else -> skip
        fi;
    }
}
```

```
inline ackReceiveVal(j) {
   protocolType == LDRVAL && wait[j-1] && A[j-1].ch[myId-1]?[
      LDRVAL(ACK, rcvsrc , tmpVal , isChild)] ->
      atomic {
         A[j-1].ch[myId-1]?LDRVAL(ACK, rcvsrc , tmpVal , isChild) ->
         if
         :: E ->
            if
            :: ack && src.num == rcvsrc.num && src.id == rcvsrc.
               id ->
               wait[j-1] = false;
               waitCount--;
               if
               :: isChild ->
                  chd[j-1] = true;
                  if
                  :: ((maxVal.num < tmpVal.num) || (maxVal.num ==
                      tmpVal.num && maxVal.id < tmpVal.id)) ->
                     maxVal.id = tmpVal.id; maxVal.num = tmpVal.
                        num;
                  :: else -> skip
                  fi;
               :: else -> skip
               fi;
            :: else -> skip
            fi;
         :: else -> skip
         fi;
      }
}

inline OverlappedMessage(j) {
   protocolType == LDRVAL && A[j-1].ch[myId-1]?[LDRID(rcvdMsgType
      , rcvsrc , tmpbyte , junkByteID , tmpbool)] ->
      atomic {
         A[j-1].ch[myId-1]?LDRID(rcvdMsgType , rcvsrc , tmpbyte ,
            junkByteID , tmpbool);
         printf("MSC: Message from LDRID\n");
         if
         :: rcvdMsgType == ELECT ->
            skip;
         :: rcvdMsgType == ACK ->
            skip;
         :: rcvdMsgType == LD ->
            if
            :: src.num == rcvsrc.num && src.id == rcvsrc.id ->
               E = false;
```

```
                ldr = tmpbyte;
                loopvar = 1;
                do
                :: loopvar <= N ->
                   if
                   :: chd[loopvar-1] -> A[myId-1].ch[loopvar-1]!
                      LDRVAL(LD, src, ldr, junkByteID, tmpbool);
                      chd[loopvar-1]=false;
                   :: else -> skip
                   fi;
                   loopvar++
                :: loopvar > N ->
                   break
                od;
             :: else -> skip
             fi;
          fi;
       }
}

/* defining the process */
proctype node (bool st; byte mynumber) {
   bool startElection = st, E = false, ack = false, isChild =
      false;
   byte myId = mynumber, maxchildId;
   byte p = 0, mynum = 0, max = 0, ldr = 0;
   compIndex src, rcvsrc, maxVal, tmpVal;
   bool wait[N] = false, chd[N] = false, tmpbool;
   byte waitCount = 0, loopvar, tmpbyte;
   mtype protocolType = LDRID;   /* initially LDRID is running */
   mtype rcvdMsgType;

end: do

   /* LDRID : start election */
   :: protocolType == LDRID && (!E && startElection) ->
      atomic {
         printf("MSC: Starting new election\n");
         src.num = mynum;
         src.id = myId;
         mynum = mynum + 1;
         waitCount = 0;
         loopvar = 1;
         do
         :: loopvar <= N ->
            if
            :: Nbhrs[myId-1].nL[loopvar-1] ->
```

```
            A[myId-1].ch[loopvar-1]!LDRID(ELECT, src, tmpbyte,
                junkByteID, tmpbool);
            wait[loopvar-1] = true;
            waitCount++;
         :: else ->
            wait[loopvar-1] = false;
         fi;
         loopvar++;
      :: loopvar > N ->
         break
      od;
      ack = true; E = true; p = myId; max = myId;
      startElection = false;
   }

:: joinElectionId(1);
:: joinElectionId(2);
:: joinElectionId(3);
:: joinElectionId(4);
:: joinElectionId(5);

:: ackReceiveId(1);
:: ackReceiveId(2);
:: ackReceiveId(3);
:: ackReceiveId(4);
:: ackReceiveId(5);

/* LDRID : ack to parent */
:: protocolType == LDRID && E && waitCount == 0 && myId != src
   .id && ack ->
   atomic {
      ack = false;
      A[myId-1].ch[p-1]!LDRID(ACK, src, max, junkByteID, true);
   }

/* LDRID : elect leader */
:: protocolType == LDRID && E && waitCount == 0 && myId == src
   .id && ack ->
   atomic {
      ack = false;
      E = false;
      ldr = max;
      loopvar = 1;
      do
      :: loopvar <= N ->
         if
```

```
            :: chd[loopvar-1] -> A[myId-1].ch[loopvar-1]!LDRID(LD
               ,src,ldr,junkByteID,tmpbool);
               chd[loopvar-1] = false;
            :: else -> skip
            fi;
            loopvar++
         :: loopvar > N ->
            break
      od;
   }

/* LDRID : receive and forward leader */
:: protocolType == LDRID && E && !ack && myId != src.id && A[p
   -1].ch[myId-1]?[LDRID(LD,rcvsrc,ldr,junkByteID,tmpbool)] ->
   A[p-1].ch[myId-1]?LDRID(LD,rcvsrc,ldr,junkByteID,tmpbool);
   atomic {
      if
      :: src.num == rcvsrc.num && src.id == rcvsrc.id ->
         E = false;
         loopvar = 1;
         do
         :: loopvar <= N ->
            if
            :: chd[loopvar-1] -> A[myId-1].ch[loopvar-1]!LDRID
               (LD,src,ldr,junkByteID,tmpbool);
               chd[loopvar-1]=false;
            :: else -> skip
            fi;
            loopvar++
         :: loopvar > N ->
            break
         od;
      :: else -> skip
      fi;
   }

/* adaptive action */
:: ADAPT[myId-1] && protocolType == LDRID ->
   atomic {
      printf("MSC: Adapting protocol to LDRVAL at node: %d\n",
          myId);
      protocolType = LDRVAL;
      /* perform state transfer actions */
      if
      :: E && waitCount != 0 && src.id != myId ->
         E = false; ack = false;
         waitCount = 0; p = 0; ldr = 0;
```

```
            loopvar = 1;
            do
            :: loopvar <= N ->
               wait[loopvar-1] = false;
               chd[loopvar-1] = false;
               loopvar++;
            :: loopvar > N -> break
            od;
         ::  E && waitCount != 0 && src.id == myId ->
            E = false; ack = false;
            waitCount = 0; p = 0; ldr = 0;
            loopvar = 1;
            do
            :: loopvar <= N ->
               wait[loopvar-1] = false;
               chd[loopvar-1] = false;
               loopvar++;
            :: loopvar > N -> break
            od;
            startElection = true;
         :: else -> skip
         fi;
      }

:: OverlappedMessage (1);
:: OverlappedMessage (2);
:: OverlappedMessage (3);
:: OverlappedMessage (4);
:: OverlappedMessage (5);

/* LDRVAL : start election */
:: protocolType == LDRVAL && (!E && startElection) ->
   atomic {
      printf("MSC: Starting new election\n");
      src.num = mynum;
      src.id = myId;
      mynum = mynum + 1;
      waitCount = 0;
      loopvar = 1;
      do
      :: loopvar <= N ->
         if
         :: Nbhrs[myId-1].nL[loopvar-1] ->
            A[myId-1].ch[loopvar-1]!LDRVAL(ELECT, src, tmpVal,
               tmpbool);
            wait[loopvar-1] = true;
            waitCount++;
```

```
            :: else ->
               wait[loopvar-1] = false;
            fi;
            loopvar++;
         :: loopvar > N ->
            break
      od;
      ack = true; E = true; p = myId;
      maxVal.id = myId; maxVal.num = value[myId-1];
      startElection = false;
   }

:: joinElectionVal(1);
:: joinElectionVal(2);
:: joinElectionVal(3);
:: joinElectionVal(4);
:: joinElectionVal(5);

:: ackReceiveVal(1);
:: ackReceiveVal(2);
:: ackReceiveVal(3);
:: ackReceiveVal(4);
:: ackReceiveVal(5);

/* LDRVAL : ack to parent */
:: protocolType == LDRVAL && E && waitCount == 0 && myId !=
   src.id && ack ->
   atomic {
      ack = false;
      A[myId-1].ch[p-1]!LDRVAL(ACK, src, maxVal, true);
   }

/* LDRVAL : elect leader */
:: protocolType == LDRVAL && E && waitCount == 0 && myId ==
   src.id && ack ->
   atomic {
      ack = false;
      E = false;
      ldr = maxVal.id;
      loopvar = 1;
      do
      :: loopvar <= N ->
         if
         :: chd[loopvar-1] -> A[myId-1].ch[loopvar-1]!LDRVAL(
            LD, src, maxVal, tmpbool);
            chd[loopvar-1] = false;
         :: else -> skip
```

```
            fi;
            loopvar++
        :: loopvar > N ->
            break
        od;
    }

/* LDRVAL : receive and forward leader */
:: protocolType == LDRVAL && E && !ack && myId != src.id && A[
    p-1].ch[myId-1]?[LDRVAL(LD,rcvsrc,tmpVal,tmpbool)] ->
    A[p-1].ch[myId-1]?LDRVAL(LD,rcvsrc,tmpVal,tmpbool);
    atomic {
        if
        :: src.num == rcvsrc.num && src.id == rcvsrc.id ->
            E = false;
            ldr = tmpVal.id;
            loopvar = 1;
            do
            :: loopvar <= N ->
                if
                :: chd[loopvar-1] -> A[myId-1].ch[loopvar-1]!
                    LDRVAL(LD,src,tmpVal,tmpbool);
                    chd[loopvar-1]=false;
                :: else -> skip
                fi;
                loopvar++
            :: loopvar > N ->
                break
            od;
        :: else -> skip
        fi;
    }
od
}

/* initial process */
init {
    /* defining the neighbors */
    Nbhrs[0].nL[1] = true; Nbhrs[1].nL[0] = true;
    Nbhrs[0].nL[2] = true; Nbhrs[2].nL[0] = true;
    Nbhrs[0].nL[4] = true; Nbhrs[4].nL[0] = true;
    Nbhrs[1].nL[2] = true; Nbhrs[2].nL[1] = true;
    Nbhrs[1].nL[4] = true; Nbhrs[4].nL[1] = true;
    Nbhrs[2].nL[3] = true; Nbhrs[3].nL[2] = true;

    /* processes that will start the election */
    startNodes[3] = true;
```

```
    startNodes[4] = true;

    value[3] = 1;

    byte proc;
    atomic {
        proc = 1;
        do
        :: proc <= N ->
            if
            :: startNodes[proc-1] -> run node(true, proc);
            :: else -> run node(false, proc)
            fi;
            proc++;
        :: proc > N ->
            break
        od;

        do
        :: !ADAPT[0] -> ADAPT[0] = true;
        :: !ADAPT[1] -> ADAPT[1] = true;
        :: !ADAPT[2] -> ADAPT[2] = true;
        :: !ADAPT[3] -> ADAPT[3] = true;
        :: !ADAPT[4] -> ADAPT[4] = true;
        :: ADAPT[0] && ADAPT[1] && ADAPT[2] && ADAPT[3] && ADAPT[4]
            -> break;
        od
    }
}
```

A.2 Model-Checking Results

In this section, we discuss results of safety and liveness properties that we verify for the adaptive leader election program of Section A.1 using SPIN.

A.2.1 End States

In this subsection we show the results of verifying the program for valid end states. The valid end states for the adaptive leader election program discussed in section A.1 are those

in which every process that was instantiated has reached the end of its code. Specifically, those are ones in which all process have completed the election (started by process 1) and the adaptation (from *ldrId* to *ldrVal*). As shown in the result, there are no invalid end states.

```
Bit statespace search for:
   never claim          - (not selected)
   assertion violations - (disabled by -A flag)
   cycle checks         - (disabled by -DSAFETY)
   invalid end states   +

State-vector 2332 byte, depth reached 1251, errors: 0
3.36632e+07 states, stored
1.14232e+08 states, matched
1.47896e+08 transitions (= stored+matched)
5.90994e+08 atomic steps

hash factor: 1.99354 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
78637.270    equivalent memory usage for states (stored*(State-
   vector + overhead))
16.777    memory used for hash array (-w26)
0.360     memory used for DFS stack (-m10000)
0.623     other (proc and chan stacks)
0.017     memory lost to fragmentation
17.777    total actual memory usage
```

A.2.2 Safety Property of Leader Election

In this subsection, we consider the safety property which states that if any two processes are not in election then they have the same leader. For sake of reducing redundancy, we show the result for the case in which we verify the property for process 1 and process 2. Similar results are obtained when we consider any two processes. The never claim describing the

property is shown first, which is followed by the verification result.

Never claim

```
#define p    (node[1]:E || node[2]:E || node[1]:ldr == node[2]:ldr
    )

    /*
     * Formula As Typed: []p
     * The Never Claim Below Corresponds
     * To The Negated Formula !([]p)
     * (formalizing violations of the original)
     */

never {    /* !([]p) */
T0_init:
    if
    :: (! ((p))) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

Verification result

```
Bit statespace search for:
    never claim          +
    assertion violations + (if within scope of claim)
    acceptance   cycles  + (fairness disabled)
    invalid end states   - (disabled by never claim)

State-vector 2336 byte, depth reached 1383, errors: 0
3.36991e+07 states, stored
1.09351e+08 states, matched
1.4305e+08 transitions (= stored+matched)
5.64732e+08 atomic steps

hash factor: 1.99142 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
79260.243    equivalent memory usage for states (stored*(State-
    vector + overhead))
8.389     memory used for hash array (-w26)
0.040     memory used for bit stack
0.320     memory used for DFS stack (-m10000)
```

```
0.584     other (proc and chan stacks)
0.056     memory lost to fragmentation
9.389     total actual memory usage
```

A.2.3 Liveness Property of Leader Election

In this subsection, we consider the liveness property which states that eventually all processes have the same leader. The never claim describing the property is shown first, which is followed by the verification result.

Never claim

```
#define p   (node[1]:ldr == node[2]:ldr && node[2]:ldr == node
    [3]:ldr && node[3]:ldr == node[4]:ldr && node[4]:ldr == node
    [5]:ldr && node[5]:ldr == node[1]:ldr)
    /*
     * Formula As Typed: []<>p
     * The Never Claim Below Corresponds
     * To The Negated Formula !([]<>p)
     * (formalizing violations of the original)
     */

never {     /* !([]<>p) */
T0_init:
    if
    :: (! ((p))) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4:
    if
    :: (! ((p))) -> goto accept_S4
    fi;
}
```

Verification result

```
Bit statespace search for:
    never claim          +
    assertion violations + (if within scope of claim)
    acceptance   cycles  + (fairness disabled)
    invalid end states   - (disabled by never claim)
```

```
State-vector 2336 byte, depth reached 1391, errors: 0
3.3996e+07 states, stored
1.10277e+08 states, matched
1.44273e+08 transitions (= stored+matched)
5.7067e+08 atomic steps

hash factor: 1.97402 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
79958.646    equivalent memory usage for states (stored*(State-
   vector + overhead))
8.389     memory used for hash array (-w26)
0.040     memory used for bit stack
0.320     memory used for DFS stack (-m10000)
0.585     other (proc and chan stacks)
0.055     memory lost to fragmentation
9.389     total actual memory usage
```

A.2.4 Safety Property During Adaptation

In this subsection, we consider a safety property during adaptation. The property states that if a process using *ldrVal* protocol sends a message of type ELECT to a process using *ldrId* protocol, then that message is buffered. The never claim describing the property is shown first, which is followed by the verification result.

Never claim

```
#define p    (!(node[1]:protocolType == LDRID && node[2]:
   protocolType == LDRVAL && node[2]:E && node[2]:ack && node[2]:
   waitCount !=0) || (A[1].ch[0]??[LDRVAL(ELECT)]))

   /*
    * Formula As Typed: []p
    * The Never Claim Below Corresponds
    * To The Negated Formula !([]p)
    * (formalizing violations of the original)
    */

never {     /* !([]p) */
```

```
T0_init:
   if
   :: (! ((p))) -> goto accept_all
   :: (1) -> goto T0_init
   fi;
accept_all:
   skip
}
```

Verification result

```
Bit statespace search for:
   never claim          +
   assertion violations + (if within scope of claim)
   acceptance   cycles  + (fairness disabled)
   invalid end states   - (disabled by never claim)

State-vector 2336 byte, depth reached 1383, errors: 0
3.36991e+07 states, stored
1.09351e+08 states, matched
1.4305e+08 transitions (= stored+matched)
5.64732e+08 atomic steps

hash factor: 1.99142 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
79260.243    equivalent memory usage for states (stored*(State-
   vector + overhead))
8.389     memory used for hash array (-w26)
0.040     memory used for bit stack
0.320     memory used for DFS stack (-m10000)
0.684     other (proc and chan stacks)
0.176     memory lost to fragmentation
9.609     total actual memory usage
```

BIBLIOGRAPHY

Bibliography

[1] W. K. Chen, M. Hiltunen, and R. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, pp. 635–643, April 2001.

[2] P. McKinley and U. Padmanabhan, "Design of composable proxy filters for mobile computing," in *Proceedings of the Workshop on Wireless Networks and Mobile Computing*, 2001.

[3] S. S. Kulkarni, K. N. Biyani, and U. Arumugam, "Composing distributed fault-tolerance components," in *Proceedings of International Workshop on Principles of Dependable Systems - PoDSy, at DSN*, pp. W127–136, June 2003.

[4] J. Hallstrom, W. Leal, and A. Arora, "Scalable evolution of highly available systems," *Transactions of the Institute for Electronics, Information and Communication Engineers*, vol. E86-D, no. 10, pp. 2154–2164, 2003.

[5] B. Redmond and V. Cahill, "Supporting unanticipated dynamic adaptation of application behavior," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pp. 205–230, 2002.

[6] S. M. Sadjadi, *Transparent Shaping of Existing Software to Support Pervasive and Autonomic Computing*. PhD thesis, Michigan State University, 2004.

[7] J.-C. Fabre and T. Perennou, "FRIENDS: A flexible architecture for implementing fault tolerant and secure distributed applications," in *Proceedings of the European Dependable Computing Conference*, pp. 3–20, 1996.

[8] R. Keller and U. Hölzle, "Binary component adaptation," *Lecture Notes in Computer Science*, vol. 1445, 1998.

[9] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions of Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[10] N. Amano and T. Watanabe, "A software model for flexible and safe adaptation of mobile code programs," in *Proceedings of the International Workshop on Principles of Software Evolution*, pp. 57–61, ACM Press, 2002.

[11] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng, "Composing adaptive software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[12] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, 2003.

[13] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications," in *Proceedings of the International Workshop on Self-Managed Systems (WOSS)*, 2004.

[14] D. L. Metayer, "Describing software architecture styles using graph grammers," *IEEE Transaction on Software Engineering*, vol. 24, no. 7, pp. 521–533, 1998.

[15] G. Taentzer, M. Goedicke, and T. Meyer, "Dynamic change management by distributed graph transformation: Towards configurable distributed systems," in *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation*, vol. 1764 of *LNCS*, Springer, 1998.

[16] M. Wermelinger, A. Lopes, and J. L. Fiadeiro, "A graph based architectural (re)configuration language," in *Proceedings of the 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundatations of Software Engineering (ESEC/FSE 2001)*, vol. 26 of *Software Engineering Notes*, pp. 21–32, 2001.

[17] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of Conference on Fundamental Approaches to Software Engineering*, vol. 1382 of *Lecture Notes in Computer Science (LNCS)*, pp. 21–35, 1998.

[18] C. E. Cuesta, P. de la Fuente, and M. Barrio-Solarzano, "Dynamic coordination architecture through use of reflection," in *Proceedings of ACM Symposium on Applied Computing*, pp. 134–140, ACM Press, 2001.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th International Conference on Software Engineering*, pp. 177–186, 1998.

[20] J. Kramer, J. Magee, and M. Sloman, "Configuring distributed systems," in *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop*, pp. 1–5, ACM Press, 1992.

[21] S. McCamant and M. D. Ernst, "Predicting problems caused by component upgrades," in *ESEC/FSE: Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Helsinki, Finland), pp. 287–296, September 2003.

[22] L. Mariani and M. Pezzè, "A technique for verifying component-based software," in *International Workshop on Test and Analysis of Component Based Systems*, (Barcelona, Spain), pp. 17–30, March 27–28, 2004.

[23] S. Chaki, N. Sharygina, and N. Sinha, "Verification of evolving software," in *Proceedings of the 3rd International Workshop on Specification and Verification of Component-based Systems*, pp. 55–61, 2004.

[24] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Software - Practice and Experience*, vol. 23, no. 9, pp. 949–964, 1993.

[25] B. P. Lientz and E. B. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer-Application Software in 487 Data Processing Organizations*. Addison-Wesley, August 1980.

[26] N. Chapin, J. Hale, K. Khan, J. Ramil, and W. G. Than, "Types of software evolution and software maintenance," *Journal of Software Maintenance and Evolution*, no. 13, pp. 3–30, 2001.

[27] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and H. C. Cheng, "A taxonomy of compositional adaptation," Tech. Rep. MSU-CSE-04-17, Michigan State University, May 2004.

[28] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2nd ed., 2002.

[29] J. Kramer and J. Magee, "Analysing dynamic change in software architectures: A case study," in *Proceedings of IEEE International Conference on Configurable Distributed Systems*, 1998.

[30] C. Canal, E. Pimentel, and J. M. Troya, "Specification and refinement of dynamic software architectures," in *Proceedings of the Working IFIP Conference on Software Architecture*, pp. 107–126, 1999.

[31] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, and J. E. Robbins, "A component- and message-based architectural style for gui software," in *Proceedings of the 17th International Conference on Software Engineering*, pp. 295–304, ACM Press, 1995.

[32] I. Lee, *DYMOS: A Dynamic MOdification System*. PhD thesis, University of Wisconsin, 1983.

[33] R. P. Cook, "*mod - a language for distributed programming," *IEEE Transactions on Software Engineering*, vol. 6, no. 6, pp. 563–571, 1980.

[34] J. Magee and J. Kramer, "Dynamic configuration for distributed systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 424–436, 1985.

[35] J. Magee, J. Kramer, and M. Sloman, "Constructing distributed systems in conic," *IEEE Transactions on Software Engineering*, vol. 15, no. 6, 1989.

[36] B. Liskov, "Distributed programming in argus," *Communications of the ACM*, pp. 300–312, March 1988.

[37] T. Bloom, *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, Massachusetts Institute of Technology, 1983.

[38] D. Gupta, *On-line Software Version Change.* PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, 1994.

[39] J. Zhang and B. Cheng, "Using temporal logic to specify adaptive program semantics," *Journal of Systems and Software (JSS)*, vol. 79, no. 10, pp. 1361–1369, 2006.

[40] J. Zhang and B. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of International Conference on Software Engineering*, May 2006.

[41] J. Zhang, Z. Yang, B. Cheng, and P. McKinley, "Adding safeness to dynamic adaptation techniques," in *Proceedings of ICSE 2004 Workshop on Architecting Dependable Systems*, (Edinburgh, Scotland, UK), May 2004.

[42] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, October 1985.

[43] B. Alpern and F. B. Schneider, "Proving boolean combinations of deterministic properties," in *Proceedings of the Second Symposium on Logic in Computer Science*, pp. 131–137, 1987.

[44] M. E. Segal and O. Frieder, "Dynamic program updating: a software maintenance technique for minimizing software downtime," *Software Maintenance : Research and Practice*, vol. 1, pp. 59–79, September 1989.

[45] M. E. Segal and O. Frieder, "Dynamically updating distributed software: Supporting change in uncertain and mistrustful environments," in *Proceedings of International Conference on Software Maintenance*, October 1989.

[46] M. E. Segal, "Extending dynamic program updating systems to support distributed systems that communicate via remote evaluation," in *Proceedings of the International Workshop on Configurable Distributed Systems*, pp. 188–199, 1991.

[47] M. R. Barbacci, D. L. Doubleday, and C. B. Weinstock, "Application level programming," in *Proceedings of International Conference on Distributed Computing Systems*, pp. 458–465, 1990.

[48] C. Hofmeister, *Dynamic Reconfiguration.* PhD thesis, Computer Science Department, University of Maryland, 1993.

[49] S. Gilmore, D. Kirli, and C. Walton, "Dynamic ml without dynamic types," Tech. Rep. ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, University of Edinburgh, December 1997.

[50] C. Walton, D. Kirli, and S. Gilmore, "An abstract machine for module replacement," tech. rep., Laboratory for the Foundations of Computer Science, University of Edinburgh, June 1998.

[51] M. Hicks, *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[52] M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, pp. 53–65, March 1993.

[53] D. Peled and W. Penczek, "Using asynchronous buchi automata for efficient automatic verification of concurrent systems," in *PSTV*, pp. 315–330, 1995.

[54] J. Shutt and R. Rubinstein, "Self-modifying finite automata: An introduction," in *Information Processing Letters*, vol. 56, pp. 185–190, 1995.

[55] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[56] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, 1993.

[57] A. Arora and S. S. Kulkarni, "Component based design of multitolerant systems," *IEEE transactions on Software Engineering*, vol. 24, pp. 63–78, January 1998.

[58] S. Kulkarni, *Component-based Design of Fault Tolerance*. PhD thesis, Ohio State University, 1999.

[59] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, 1974.

[60] A. Arora and S. S. Kulkarni, "Designing masking fault-tolerance via nonmasking fault-tolerance," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 435–450, 1998.

[61] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

[62] G. Varghese, *Self-stabilization by local checking and correction*. PhD thesis, MIT/LCS/TR-583, 1993.

[63] M. Gouda, *Elements of Network Protocol Desgin*. John Wiley & Sons, 1998.

[64] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[65] G. Holzmann, "Logic verification of ansi-c code with spin," *Proceedings of the The Sixth SPIN Workshop*, 2000.

[66] M. G. Gouda and T. McGuire, "Correctness preserving transformations for network protocol compilers," *Prepared for the Workshop on New Visions for Software Design and Productivity: Research and Applications*, 2001.

[67] M. Nesterenko and A. Arora, "Stabilization-preserving atomicity refinement," *Journal of Parallel and Distributed Computing*, vol. 62(5), pp. 766–791, 2002.

[68] M. Demirbas and A. Arora, "Convergence refinement," *Proceedings of the International Conference on Distributed Computing Systems*, 2002.

[69] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[70] M. D. Ernst, *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, (Seattle, Washington), 2000.

[71] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the International Conference on Software Engineering*, pp. 213–224, 1999.

[72] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the International Conference on Software Engineering*, pp. 449–458, 2000.

[73] M. G. Gouda and T. Herman, "Adaptive programming," *IEEE Transactions on Software Engineering*, vol. 17, pp. 911–921, 1991.

[74] R. V. Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, "Building adaptive systems using ensemble," *Software - Practice & Experience*, vol. 28, pp. 963–979, July 1998.

[75] N. Sridhar, S. M. Pike, and B. W. Weide, "Dynamic module replacement in distributed protocols," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, May 2003.

[76] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computation," in *Information Processing Letters*, vol. 11, pp. 1–4, August 1980.

[77] S. Vasudevan, J. Kurose, and D. Towsley, "Design and analysis of a leader election algorithm for mobile ad hoc networks," in *Proceedings of the 12th IEEE International Conference on Network Protocols*, pp. 350–360, IEEE Computer Society, October 2004.

[78] O. Babaoglu and K. Marzullo, "Consistent global states of distributed systems: Fundamental concepts and mechanisms," in *Distributed Systems* (S. Mullender, ed.), pp. 55–96, Addison-Wesley, 1993.

[79] V. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 299–307, March 1994.

[80] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 63–75, Feb 1985.

[81] R. Cooper and K. Marzullo, "Consistent detection of global predicates," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, no. 12, pp. 167–174, 1991.

[82] S. Venkatesan and B. Dathan, "Testing and debugging distributed programs using global predicates," *IEEE Transactions of Software Engineering*, vol. 21, no. 2, pp. 163–177, 1995.

[83] V. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1323–1333, 1996.

[84] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[85] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pp. 215–226, 1989.

[86] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, 1972.

[87] P. Allen and S. Frost, *Component-Based Development for Enterprise Systems*. Cambridge University Press, 1998.

[88] K. Czarnecki and U. Eisenecker, *Generative Programming*. Addison-Wesley, May 2000.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[90] D. M. Hoffman and D. M. Weiss, eds., *Software Fundamentals - Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

[91] K. Biyani, "Dynamic composition of distributed components," Master's thesis, Michigan State University, 2003.

[92] E. P. Kasten, *An Integrated Approach to Autonomous Computation for Data Streaming Applications*. PhD thesis, Michigan State University, 2007.

[93] "IBM systems journal, special issue on autonomic computing," 2003.

[94] T. Jebara and A. Pentland, "Statistical imitative learning from perceptual data," in *Proceedings of the Second International Conference on Development and Learning*, pp. 191–196, June 2002.

[95] R. Laddaga, M. L. Swinson, and P. Robertson, "Seeing clearly and moving forward," *IEEE Intelligent Systems*, vol. 15, pp. 46–50, 2000.

[96] A. Arora and M. G. Gouda, "Distributed reset," *IEEE Transactions on Computers*, vol. 43, no. 9, pp. 1026–1038, 1994.

[97] S. S. Kulkarni and A. Arora, "Multitolerance in distributed reset," *Chicago Journal of Theoretical Computer Science*, 1998.

[98] P. T. Wojciechowski and O. Rütti, *Formal Methods for Open Object-Based Distributed Systems*, vol. 3535 of *Lecture Notes in Computer Science*, ch. On Correctness of Dynamic Protocol Update, pp. 275–289. Springer Berlin / Heidelberg, 2005.

[99] D. Batory, "Multi-level models in model-driven development, product lines, and metaprogramming," *IBM Systems Journal*, vol. 45, no. 3, 2006.

[100] J. Liu and D. Batory, "Automatic remodularization and optimized synthesis of product families," in *Generative Programming and Component Engineering (GPCE)*, October 2004.

[101] P. Clements and L. Northrop, *Software Product Lines*. Addison Wesley, 2001.

[102] S. Kulkarni and K. Biyani, "Correctness of component-based adaptation," in *Proceedings of International Symposium on Component-based Software Engineering - CBSE, at ICSE*, vol. 3054 of *Lecture Notes in Computer Science*, pp. 48–58, May 2004.

[103] K. Biyani and S. Kulkarni, "Mixed-mode adaptation in distributed systems: A case study," in *Proceedings of International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS, at ICSE*, May 2007.

[104] K. Biyani and S. Kulkarni, "Concurrency tradeoffs in dynamic adaptation," in *Proceedings of International Workshop on Assurance in Distributed Systems and Networks - ADSN, at ICDCS*, July 2006.

[105] K. Biyani and S. Kulkarni, "Testing dynamic adaptation in distributed systems," in *Proceedings of International Workshop on Automation of Software Test - AST, at ICSE*, May 2007.

[106] K. Biyani and S. Kulkarni, "Building component families to support adaptation," in *Proceedings of International Workshop on Design and Evolution of Autonomic Application Software - DEAS, at ICSE*, May 2005.

[107] M. Arumugam, S. Kulkarni, and K. Biyani, "Adaptation in sensor–actuator networks: A case study," in *Proceedings of the Third International Conference on Networked Sensing Systems - INSS*, June 2006.