Contents

1 Introduction **9**

1.1 A Door Closing Protocol 9

1.2 Self-Stabilization using Domain Restriction 12

1.3 Self-Stabilization in Computer Networks 15

1.4 Criticisms of Self-Stabilization 18

1.4.1 Distinction between Program Code and State 18

1.4.2 Faults that Stop versus Faults that Continue 19

1.4.3 Permitting Initial Errors 19

1.4.4 Periodic Message Sending in the Self-stabilization Model 19

1.5 Brief History of Self-Stabilization 20

1.5.1 Refinements of Dijkstra's model 21

1.5.2 Existing solutions for Specific Tasks 21

1.5.3 Existing General Technique 22

1.6 Local Checking and Correction: A Preview 22

1.6.1 Example of Checking and Correcting on a Single Link Subsystem 22

1.6.2 Extending the Idea to a General Network 25

1.6.3 Examples of Local Checking and Correction 26

1.6.4 Why Local Checking is Useful 28

1.7 Thesis Organization 29

1.7.1 Basic Definitions and Examples: Chapters 2 - 4 30
1.7.2 Local Checking and Correction: Chapters 5 - 7 31
1.7.3 Local Checking and Global Correction: Chapters 8 - 9 33
1.8 Reading the Thesis 34

2 The I/O Automaton Model **35**
2.1 The I/O Automaton Model 36
2.1.1 Why use the I/O Automaton Model? 36
2.1.2 Four Important Features of the I/O Automaton Model 37
2.1.3 Specifying Correctness in the I/O Automaton Model 39
2.2 Formal Summary of the I/O Automaton Model 42
2.2.1 Composition and Hiding 45
2.2.2 Useful Notation 47
2.2.3 Modelling Asynchronous Protocols 47

3 Stabilization: Definitions and Properties **49**
3.1 Definitions of Stabilization based on Executions 49
3.2 Definitions of Stabilization based on External Behavior 52
3.3 Discussion on the Stabilization Definitions 55
3.4 Proof Technique 56
3.4.1 Proving that an Automaton Solves a Problem 57
3.4.2 Proving that an Automaton Stabilizes to another Automaton . . 59
3.5 Modularity Theorem 61
3.5.1 Discussion of the Modularity Theorem 65
3.6 Summary 66

4 Local Checking and Correction in a Shared Memory Model 69

4.1 Modelling Shared Memory Protocols 70

4.2 A Reset Protocol on a Tree 70

4.3 Tree Correction for Shared Memory Systems 76

4.3.1 Weakening the Fairness Requirement 83

4.4 Rediscovering Dijkstra's Protocols 84

4.5 Summary 89

5 Local Checking and Correction for Network Protocols 90

5.1 Modelling Network Protocols 91

5.1.1 Modelling Network Topology 92

5.1.2 Modelling Network Links 93

5.1.3 Modelling Network Nodes 95

5.1.4 Network Automata 97

5.2 Implementing Our Model in Real Networks 98

5.3 Locality 99

5.3.1 Link Subsystems and Local Predicates 100

5.3.2 Local Checkability 101

5.3.3 Local Correctability 102

5.4 Local Correction Theorem 104

5.4.1 Overview 104

5.4.2 Precise Statement of the Result 105

5.4.3 Overview of the Transformation Code 106

5.4.4 Constructing Augmented Automata: Formal Description 109

5.5 Intuitive Proof of Local Correction Theorem 114

5.5.1 Intuition Behind Counter Based Matching 114

5.5.2 Intuition Behind Local Snapshots 116

5.5.3 Intuition Behind Local Resets . . . 117
5.5.4 Intuition Behind Local Correction Theorem . . . 118
5.6 Formal Proof of Local Correction Theorem . . . 119
5.6.1 Overview of Formal Proof . . . 119
5.6.2 Phases . . . 121
5.6.3 Clean Phases . . . 123
5.6.4 Quiet Links: Establishing Link Predicates . . . 128
5.6.5 Projecting Behaviors of $\mathcal{N}|Q$ . . . 131
5.6.6 Tying up the Proof . . . 133
5.7 Implementing Local Checking in Real Networks . . . 133
5.8 Summary . . . 134

6 Stabilizing Mutual Exclusion and Tree Correction 136
6.1 Overview of Token Passing Protocol . . . 137
6.2 Specification of Token Passing Protocol . . . 138
6.3 Adding Local Checking and Correction . . . 139
6.4 Removing Unexpected Packet Transitions . . . 145
6.5 Tree Correction Theorem . . . 148
6.6 Summary . . . 149

7 Stabilizing Network Reset 152
7.1 Synchronization . . . 153
7.1.1 Data Link Synchronization . . . 153
7.1.2 Network Synchronization . . . 155
7.2 Existing Solutions . . . 158
7.3 Specifying the Desired Behaviors of a Reset Protocol . . . 159
7.3.1 Interface to Reset Protocol . . . 159

7.3.2 Difficulties in Specifying the Reset Problem 161
7.3.3 Formal Specification of Reset Problem 162
7.3.4 Alternative Specifications of Reset Problem 168
7.4 Overview of the solution 169
7.4.1 Problems with a Simple Reset Protocol 170
7.4.2 The basic idea behind the Reset protocol 172
7.4.3 Overview of the Code 173
7.4.4 Example 178
7.5 Reset Automaton 180
7.6 Reset Protocol is Locally Checkable and Correctable 182
7.6.1 Overview of Predicates 182
7.6.2 Proving that the Local Predicates of the Reset Protocol are Closed 185
7.6.3 Reset Protocol is Locally Correctable 187
7.7 The behaviors of a reset protocol after it stabilizes 190
7.7.1 Why the Termination Lemma Works 191
7.7.2 Why behaviors of the reset automaton are timely, consistent, and causal 193
7.8 Local Correctability and Dynamic Network Protocols 201
7.9 Summary 203

8 Global Correction Theorem **205**

8.1 Statement of Global Correction Theorem 206
8.2 Proof of the Global Correction Theorem 208
8.2.1 Construction of $\mathcal{N}^+$ 208
8.2.2 All reset triggers disappear in Linear Time 209
8.2.3 All Local Predicates Hold and All Signals Stop in Linear Time . 219
8.2.4 Proof of Global Correction Theorem is Modular 220

8.3 A Locally Checkable Spanning Tree protocol . . . 220
8.3.1 Previous Work on Spanning Tree Protocols . . . 220
8.3.2 Code for Locally Checkable Spanning Tree Protocol . . . 222
8.3.3 Local Predicates for $\mathcal{T}$ . . . 224
8.3.4 Fast Computation of Spanning Tree after all Link Predicates Hold 227
8.4 One Way Predicates and Local Checking . . . 230
8.5 Simple Stabilizing Spanning Tree Protocol . . . 232
8.6 Stabilizing topology update protocol . . . 234
8.7 Summary . . . 236

9 Compiling Synchronous Protocols 238

9.1 Non-interactive Protocols . . . 239
9.2 Synchronous Protocols . . . 242
9.3 Results . . . 243
9.4 Distributed Program Checking . . . 245
9.5 Rollback Compiler . . . 246
9.6 Simplified Resynchronizer Compiler . . . 247
9.6.1 Resynchronizer Code . . . 249
9.6.2 Proof Sketch for the Simplified *Resynchronizer* Construction. . . . 251
9.7 Extensions . . . 254
9.7.1 Better Synchronous Checkers for Deterministic Protocols . . . 254
9.7.2 Randomized Protocols . . . 256
9.8 Summary and open problems . . . 257

10 Conclusions and Open Questions 258

10.1 Contributions . . . 259
10.1.1 General Techniques . . . 259

10.1.2 New or Improved Stabilizing Solutions for Specific Problems . . 263
10.1.3 Modularity Theorem 264
10.1.4 Modelling 265
10.1.5 Understanding Existing Work 267
10.2 Open Questions and Further Problems 269
10.2.1 Modelling 270
10.2.2 Increased Understanding of Local Checking and Correction . . . 271
10.2.3 New Algorithms 273
10.2.4 New Directions 274
10.3 Summing Up 276

A Notation **278**

B Proofs for Chapter 5 **280**
B.1 Any Execution of $\mathcal{N}^+$ is infinite 280
B.2 Basic Properties of links 281
B.3 Time Bounds for Correction Phases 281
B.4 Proof that Clean Edges remain Clean 284
B.5 How Links Become Quiet: Detailed Proofs 286

C The AAG reset protocol **294**
C.1 Why three phases are used in the AAG protocol 294
C.2 Overview of the changes required for stabilization 296
C.3 Mating Relation is not Transitive 297

D Proofs for Reset Protocol in Chapter 7 **301**
D.1 Proving that the Local Predicates of the Reset Protocol are Closed . . . 301
D.2 Proving that the Reset Protocol Behaviors are Timely, Causal, and Consistent 309

D.2.1 Useful Claims and Lemmas for Reset Protocol 309
D.2.2 Every Behavior of $\mathcal{R}|L$ is timely 314
D.2.3 Every Behavior of $\mathcal{R}|L$ satisfies the consistency property 318
D.2.4 Every Behavior of $\mathcal{R}|L$ is causal 324
D.2.5 The Main Theorem 324

E Dijkstra's Token Protocol as an Example of Counter Flushing 325

Chapter 1

Introduction

In physics we often talk about systems that *stabilize* to a good state after initial perturbations. For example, a spring eventually stabilizes after being compressed. More generally, systems can stabilize to good behavior after an initial perturbation, where a behavior is a description of how the state changes with time. For example, a missile with a tracking system will continue to move towards its target after it is momentarily thrown off course by bad weather. In these examples drawn from physics and control theory, the states are *continuous* variables and the state transitions are described by differential equations.

By contrast, in this thesis we will concentrate on computer systems, and especially systems of computers that are interconnected by networks. In such systems, states are described by *discrete* variables and state transitions are described by transition rules, often in the form of programs. We will focus on the ability of such computer systems to stabilize to "correct behavior" after *arbitrary* initial perturbation. This property was called *self-stabilization* by Dijkstra [Dij74]. The "self" emphasizes the ability of the system to stabilize by *itself* without manual intervention.

1.1 A Door Closing Protocol

A story illustrates the basic idea. Imagine that you live in a house in Alaska in the middle of winter. You establish the following protocol (set of rules) for people who enter and leave your house. Anybody who leaves or enters the house must shut the

door after them. If the door is initially shut, and nobody makes a mistake, then the door will eventually return to the closed position. Suppose, however, that the door is initially open or that somebody forgets to shut the door after they leave. Then the door will stay open until somebody passes through again. This can be a problem if heating bills are expensive and if several hours can go by before another person goes through the door. It is often a good idea to make the door closing protocol self-stabilizing. This can be done by adding a spring (or automatic door closer) that constantly restores the door to the closed position.

We can model this situation as a state transition system. To keep things simple, let us assume that the door is only used to leave the house, and another door is used to enter. The state of the system consists of two Boolean variables, *in_threshold* and *door_open*. Variable *in_threshold* is *true* if and only if a person is in the threshold of the door waiting to go out. Variable *door_open* is *true* if and only if the door is open. We use state transitions called ENTER_THRESHOLD, OPEN_DOOR and LEAVE to model the action of a person entering the threshold, opening the door, and leaving respectively. We will model errors by allowing the initial values of the two Boolean variables to be arbitrary.

The code for these routines is described in Figure 1.1. The actions are described in terms of "preconditions" and "effects". Preconditions are the enabling conditions that must be true before an action can be taken. For instance, we don't allow the OPEN_DOOR action to be taken unless there is a person in the threshold. Effects are the results of an action. For instance, the LEAVE action shuts the door. This style of description is used throughout the thesis.

The code also specifies that certain actions take place in time t if they are continuously enabled. By this we mean that if the preconditions of the action remain true for time t, then the action must occur within time t. We will use such timing conditions throughout the thesis.

Next, we say that the door closing system is correct if whenever the door is open, there is somebody in the threshold. We write this more formally using the predicate $OK_Door \equiv$ (If $door_open = true$ then $in_threshold = true$). For the door closing system to be self-stabilizing we want OK_Door to eventually hold regardless of what state the system starts in. But it is easy to see that if initially $door_open = true$ and $in_threshold = false$ then the predicate OK_Door may never hold.

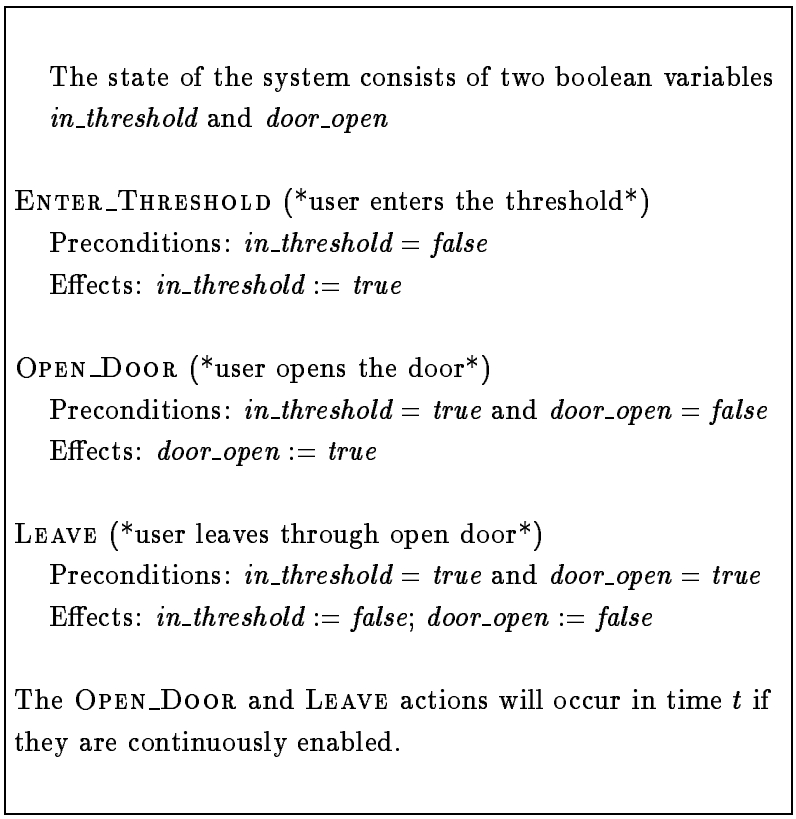


Figure 1.1: Door Closing System

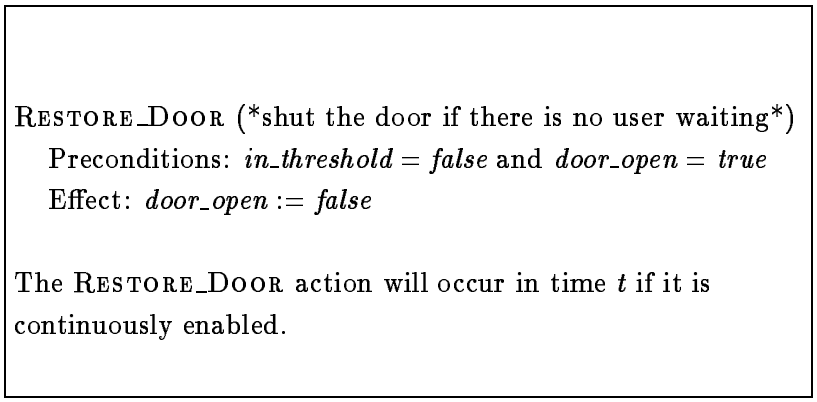


Figure 1.2: Extra action that models automatic door closing in a Door Closing System.

We can model the addition of an automatic door closer using the action RESTORE_DOOR shown in Figure 1.2. This door closer works by detecting whether there is anybody in the threshold.

1.2 Self-Stabilization using Domain Restriction

Consider Figures 1.1 and 1.2 again. Some readers may object that we caused the problem by allowing *door_open* to be an independent variable. Isn't it possible to hardwire relationships between variables to avoid illegal states? In fact, such a technique is actually used in a revolving door!

A revolving door (see Figure 1.3) can be modelled as having three states instead of four. A person enters the revolving door, gets into the middle of the door, and finally leaves. It is physically impossible to leave the door open and yet there is a way to exit through the door.

This technique, which we will call *domain restriction*, is a simple but powerful method for removing illegal states in computer systems that contain a single shared memory. Consider two processes A and B that have access to a common memory as shown in the first part of Figure 1.4. Suppose both processes should not run concurrently because they can interfere with each other. Thus we would like to provide *mutual exclusion* for the two processes. To achieve this we can pass a token between

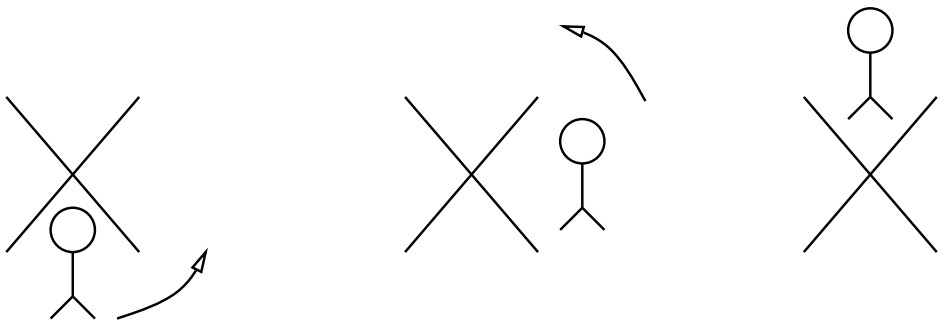


Figure 1.3: Exiting through a revolving door.

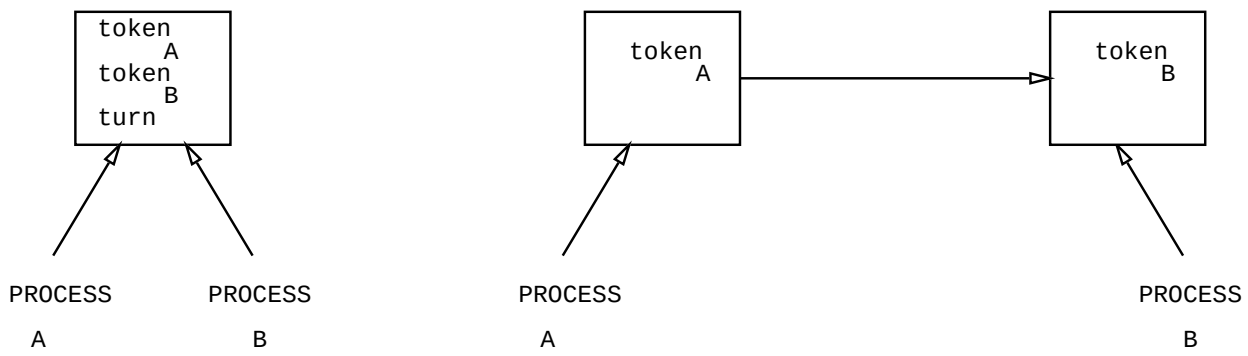
A and B. A process can "run" only when it has the token.

One way to implement this is to use two boolean variables $token_A$ and $token_B$. The token variable for a process is set to *true* whenever the process has the token. To pass the token, Process A sets $token_A$ to *false* and sets $token_B$ to *true*. In a self-stabilizing setting, however, this is not a good implementation. For instance, the system will deadlock if $token_A = token_B = false$ in the initial state. The problem is that we have some extra and useless states. The natural solution is to restrict the domain to a single bit called *turn*, such that $turn = 1$ when A has the token and $turn = 0$ when B has the token. By using domain restriction, [1] we ensure that *any possible state is also a legal state.*

In this thesis, we will sometimes use domain restriction to avoid illegal states *within a single node* of a computer network. Domain restriction can be implemented in many ways. The most natural way is by restricting the number of bits allocated to a set of variables so that every possible value assigned to the bits corresponds to a legal assignment of values to each of the variables in the set. Another possibility is to modify the code that reads variables so that only values within the specified domain are read. Almost all the automata described in this thesis are *finite state* machines. Domain restriction can be performed (for finite state machines) by enumerating the legal states and then adding suitable checks to the code.

Unfortunately, domain restriction cannot solve all problems. Consider the same

[1] In this example, we are really *changing* the domain. However, we prefer the term domain restriction.



1. WITH SHARED MEMORY

2. WITHOUT SHARED MEMORY

Figure 1.4: Token passing among two processes

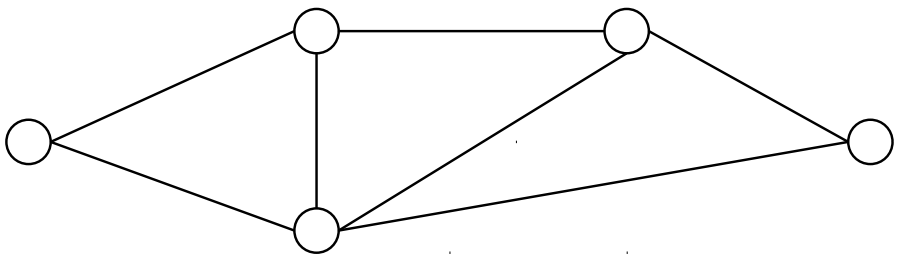


Figure 1.5: A typical mesh Network

two processes A and B that wish to achieve mutual exclusion. This time, however, (see Figure 1.4, Part 2) A and B are at two different nodes of a computer network. The only way they can communicate is by sending *token* messages to each other. Thus we cannot use a single *turn* variable that can be read by both processes. In fact, A must have at least two states: a state in which A has the token, and a state in which A does not have the token. B must also have two such states. Thus we need at least four combined states, of which two are illegal.

Thus domain restriction at each node *cannot prevent illegal combinations across nodes*. We need other techniques to detect and correct illegal states of a network. It should be no surprise that the title of Dijkstra's pioneering paper on self-stabilization [Dij74] was "Self-Stabilization in spite of Distributed Control."

1.3 Self-Stabilization in Computer Networks

In this thesis, we will explore self-stabilization properties for computer networks. A computer network consists of nodes that are interconnected by communication channels. The network topology (see Figure 1.5) is described by a graph. The vertices of the graph represent the nodes and the edges represent the channels. Nodes communicate with their neighbors by sending messages along channels. Many real networks such as the ARPANET, DECNET and SNA can be modelled in this way.

A network protocol consists of a program for each network node. Each program consists of code and inputs as well as local state. The global state of the network consists of the local state of each node as well as the messages on network links. We define a *catastrophic* fault as a fault that arbitrarily corrupts the global network state, but not the program code or the inputs from outside the network.

Self-stabilization formalizes the following intuitive goal for networks: *despite a history of catastrophic failures, once catastrophic failures stop, the system should stabilize to correct behavior without manual intervention.* Thus self-stabilization is an abstraction of a strong fault-tolerance property for networks. It is an important property of real networks because:

- **Catastrophic faults occur:** Most network protocols are resilient to common failures such as nodes and link crashes. However, many protocols cannot deal with *memory corruption.* But memory corruption does happen from time to time. For example, alpha particles are a common cause of memory corruption. It is also hard to prevent a malfunctioning device from sending out an incorrect message that carries erroneous state information. The malfunctioning node can then crash leaving an incorrect message on a channel.

- **Manual intervention has a high cost:** In a large decentralized network, restoring the network manually after a failure requires considerable coordination. As in the case of the AT&T network, the consequent network shutdown has a large dollar cost. Thus even if catastrophic faults occur rarely, (say once a year) there is considerable incentive to make network protocols self-stabilizing. In fact, a reasonable guideline is what we call Lauck's Principle [Lau90]. This principle states that *the network should stabilize preferably before the user notices and at least before the user logs a service call.* This may seem facetious. However, service calls are so expensive that this guideline is sometimes used to set timers for self-stabilizing protocols!

These issues are illustrated by the crash of the original ARPANET protocol ([Ros81] [Per83]). The designers used a sequence number to distinguish newer topology updates from older ones. Because the set of sequence numbers was finite, they used a circularly ordered number space. Hence, it was possible to have three sequence numbers a, b, c such that $a > b > c > a$. The protocol was carefully designed never to enter a state that contained the three sequence numbers a, b, and c. Unfortunately, a malfunctioning node injected three such updates into the network and crashed. After this the network cycled continuously between the three updates. It took days of detective work [Ros81] before the problem was diagnosed. With hindsight, the problem could have been avoided by making the protocol self-stabilizing.

Self-stabilization is also attractive because a self-stabilizing program does not require initialization. The concept of an initial state makes perfect sense for a single sequential program. However, for a distributed program an initial state seems to be an artificial concept. How was the distributed program placed in such an initial state? Did this require another distributed program? Self-stabilization avoids these questions by eliminating the need for distributed initialization.

Probably the most exciting reason for self-stabilization is that can provide a *uniform approach towards fault-tolerance*, thus leading to simplification as well as strengthening of existing fault-tolerant protocols. This is because self-stabilization can subsume such common fault models as link and node failures. However, in order to so the self-stabilization recovery mechanisms must be fast enough to provide adequate response time for such common failures.

There appears to be a hierarchy of faults ranging from very rare faults like memory corruption (that occur at most once every few days) to fairly common faults like link and node crashes (that may occur in the order of minutes) to very common faults like bit errors (that may occur every second). Thus it is adequate to recover from memory errors in the order of minutes, from link failures in the order of seconds, and from bit errors in the order of milliseconds. If the self-stabilization mechanism recovers too slowly (say in the order of minutes, as in [Per83]), then it is necessary to have separate, faster mechanisms to deal with common failures ([Per83]) like link and node failures.

On the other hand, if the self-stabilizing mechanisms recover in the order of seconds, then there is no need for separate mechanisms to deal with link and node failures. A good example of an existing self-stabilizing protocol that meets this criteria is the IEEE 802.1 bridge spanning tree protocol described in [Per85]. Clearly reducing the number of separate mechanisms leads to simpler protocols. It should be noted that it is unlikely that self-stabilization will be efficient enough to subsume the need for *all* other fault-recovery mechanisms; for example, most self-stabilizing protocols will probably need some retransmission scheme to deal with messages lost due to bit errors. However, the more mechanisms that can be subsumed, the simpler the resulting protocol.

In this thesis we will investigate methods for designing stabilizing protocols that have fast recovery times. Such protocols are not just faster and more fault-tolerant but also (by the arguments in the last paragraph) may be simpler than existing protocols. The example in Section 1.6.1 should clarify this point.

1.4 Criticisms of Self-Stabilization

Despite the claims of the previous section, there are several peculiar features of the self-stabilization model that are often criticized.

- The model allows network state to be corrupted but not program code. Isn't this distinction artificial? After all programs and state variables are stored as bits in memory.
- The model only deals with catastrophic faults that *stop*. There are other (e.g., Byzantine) models that deal with continuous faults. Aren't models that allow continuous faults preferable?
- A self-stabilizing program P is only supposed to *eventually* produce correct behavior. In the interim period, P is allowed to make mistakes. How can we make use of a program that can sometimes make mistakes?
- Most self-stabilizing network protocols require periodic message traffic. Using some theoretical measures of message complexity, the message complexity of a self-stabilizing protocol is unbounded. Thus a theoretician may question whether such protocols are worth the "cost".

We deal with each criticism in turn:

1.4.1 Distinction between Program Code and State

Program code can be protected against arbitrary corruption of memory by redundancy since code is rarely modified. Some static input (such as node IDs) can also be protected in this way and can be considered to be part of the program. Some changing input (such as the list of neighboring nodes in a network) can be protected by requiring that such input be the output of another self-stabilizing protocol. On the other hand, the state of a program is constantly being updated and it is not clear how one can prevent illegal operations on the memory by using checksums. It is even harder to prevent a malfunctioning node from sending out incorrect messages.

1.4.2 Faults that Stop versus Faults that Continue

In the Byzantine [LSP82] fault model, some fraction of faulty nodes can *continuously exhibit arbitrary faulty behavior*. By allowing continuous faults, the Byzantine fault model appears to be stronger than the self-stabilization model. However, network protocols with Byzantine robustness [Per88] are expensive because they require large amounts of redundancy in storage, processing, and message traffic. On the other hand, it is possible to make many protocols self-stabilizing with a small cost in extra message traffic and node processing.

In Byzantine models, only a fraction of nodes are allowed to exhibit arbitrary behavior. In the self-stabilization model, *all* nodes are permitted to start with arbitrary initial states. Thus, neither model subsumes the other. In theory, there is no reason why a protocol cannot be robust against *both* Byzantine failures and arbitrary initial states.

Assuming that faults stop in the self-stabilization model is only a modelling artifice. In practice, *we only need faults to stop for a period long enough for the protocol to stabilize*. Thus the self-stabilization model is especially appropriate for handling transient errors.

1.4.3 Permitting Initial Errors

A distributed database program cannot tolerate errors that may, for instance, wrongly credit an account with a million dollars! However, for most of the network protocols considered in this thesis, *errors are not as critical*. An example is a network protocol that computes routes between nodes. The nice thing about a routing protocol is that even if the network is completely fouled up, the worst thing that can happen is that network traffic stops for a while. Most of the stabilizing protocols described in this thesis are used for routing, scheduling, and resource allocation tasks. For such tasks, initial errors only result in a temporary loss of service.

1.4.4 Periodic Message Sending in the Self-stabilization Model

It is easy to show that any non-trivial self-stabilizing network protocol must send messages periodically. Periodic sending of messages may seem extremely ugly. However,

in real networks, each node periodically sends control messages to its neighbors to detect whether the neighbor is alive. For many of the protocols described in this thesis, the periodic message sending required for stabilization can be piggybacked on such "keep-alive" message traffic without appreciable loss of efficiency.

In a real implementation, periodic message sending is controlled by timers in order to keep the overhead bounded. We will not model these timers explicitly. *A timer can be implemented in a self-stabilizing fashion as long as the hardware clock in every node that's up continues to function.* For instance, we can implement a timer using a counter that is incremented every time the hardware clock ticks. When the counter reaches its maximum value, the sending of a message is enabled, and the counter is reset to 0. Assuming that the hardware clock continues to tick is not at all restrictive. For most computers, if the hardware clock stops, the node has effectively crashed!

1.5 Brief History of Self-Stabilization

Self-stabilization was introduced by Dijkstra in a seminal paper [Dij74]. In Dijkstra's model, a network protocol is modelled using a graph. The nodes of the graph contain finite state machines. The protocol is asynchronous, and the asynchrony is modelled by an adversarial scheduler called a "demon". At each stage, the demon is allowed to choose an arbitrary node in the graph to make a move. In a single move, a node is allowed to read the state of its neighbors, compute, and then possibly change its state. In this setting, Dijkstra described three self-stabilizing mutual exclusion protocols.

After Dijkstra's initial paper, work on self-stabilization languished for many years. However, in this period, at least three researchers recognized the importance of the concept, and championed its cause. Gouda and his co-workers at the University of Texas produced a number of papers (e.g., [BGW87],[GM90], [AG90]) in this area. Lamport's PODC address ([Lam84]) was probably responsible for awakening the interest of the theory community in self-stabilization. Independently, Tony Lauck, [Lau90], who is responsible for the architecture of DECNET, recognized the applicability of self-stabilization to real networks. At his insistence, self-stabilization was added as a requirement ([Per83]) for many DECNET protocols.

After Lamport's PODC address, a number of papers began to appear in this area. The contributions of these papers fall into three categories: refinements of Dijkstra's

model, solutions to specific tasks, and one general technique.

1.5.1 Refinements of Dijkstra's model

In Dijkstra's model, a node is allowed to read the state of all its neighbors and change its own state, all in one move. This level of atomicity is hard to achieve in a real network. [BGW87] suggest a model in which at each step, the demon can allow an *arbitrary subset* of nodes to make a move. Later [DIM90] introduced a model in which a node communicates with its neighbors by reading or writing to certain shared registers. Also, in their model, reading and writing of the shared register (and local computation) are separate atomic steps that can be arbitrarily interleaved. Other papers [Per83, AB89, GM90, KP90] model communication between nodes by the explicit sending of messages.

In Dijkstra's model, one node in the graph is assumed to be a "leader" in order to break symmetry. Dijkstra observed that some form of symmetry breaking is required for self-stabilizing mutual exclusion. Later models introduced other forms of symmetry breaking. In [Per83, AKY90], each node is assumed to have a distinct ID. [IJ90] introduced the use of randomization. Finally, most papers in this area assume the model is completely asynchronous. No assumptions are made about how long it takes for actions to be performed. By contrast, [Per83, Per85] assume upper bounds on message delivery and node processing times.

1.5.2 Existing solutions for Specific Tasks

Dijkstra's paper concentrated on the task of mutual exclusion on rings. Subsequent papers (e.g., [BP89, DIM90, IJ90]) continued to work on self-stabilizing mutual exclusion, but in different models. Solutions to other tasks have also appeared.

[Per83, SG89] describe self-stabilizing routing protocols to compute shortest path routes between every pair of nodes in a network. [Per85, AG90, AKY90] describe self-stabilizing protocols to compute a spanning tree in a network. [AB89, GM90] show how to establish reliable communication between a pair of nodes over a physical channel. A reset protocol is a protocol that can be used to "reset" a network to a prespecified initial state; a snapshot protocol can be used to find a consistent global

state of a network. [AG90, KP90] describe self-stabilizing snapshot and reset protocols. [Spi88a] describes a self-stabilizing virtual circuit protocol.

1.5.3 Existing General Technique

While many fundamental problems have been tackled, there is a lack of general methods. We know of only one general technique other than the work described in this thesis. Katz and Perry [KP90] show how to stabilize a large class of distributed algorithms by *centralized checking and correction at a leader*. The main technical difficulty in this approach is finding a self-stabilizing method to do checking and correction. For this purpose, [KP90] invented a self-stabilizing snapshot protocol. However, the need for centralized checking makes the performance of this approach rather poor as we see in the next section.

1.6 Local Checking and Correction: A Preview

The major theme of this thesis is the design of new and efficient general methods for making protocols self-stabilizing. All our methods are based on what we call *local checking*. Unlike [KP90], our methods are efficient because checking is local and decentralized. In this section, we give a preview of our ideas.

1.6.1 Example of Checking and Correcting on a Single Link Subsystem

To make the notion of local checking and correcting more concrete we quickly describe an example of a protocol that works between two nodes, a sender node and a receiver node (Figure 1.6) that are connected by two unidirectional links. The sender sends messages to the receiver who buffers the messages in a finite sized queue. Any message that arrives when the queue is full is dropped. A simple credit-based scheme can be used to prevent messages being dropped during normal operation.

The sender (Figure 1.6) keeps a credit register which stores the current credits available to the sender. Initially, assume that the receiver queue is empty and that the

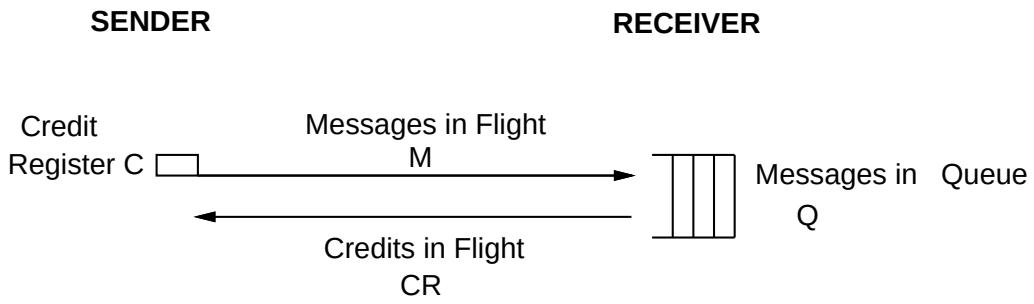


Figure 1.6: Credit Based Flow Control between a Sender and a Receiver

sender's credit register is equal to the size of the receiver queue (say Max). The sender only sends a message when the credit register is non-zero; after sending a message, the sender decrements its credit register by one. When the receiver removes a message from its queue, the receiver sends a CREDIT message back to the sender. When the sender receives such a CREDIT message, the sender increments its credit register by one.

It is easy to see that after proper initialization and assuming that no errors occur, no messages will be dropped. Under such conditions, the following condition (which we will later call a Local Predicate) holds at every instant. If at any instant we denote (see Figure 1.6) the value of the credit register by C, the number of messages in flight by M, the number of messages in the queue as Q and the number of credits in flight by CR, then it must be true that: $C + M + Q + CR = Max$. Intuitively, this can be seen by analogy to two banks that only transfer money between each other; assuming no errors, the total amount of "money" (credits plus messages) in and between the two banks must be conserved. This local predicate ensures that there is always room in the queue for the messages in flight since $M + Q \leq Max$.

Unfortunately, this simple credit based scheme runs into trouble if the system is either improperly initialized or there are errors on the link. Link errors can result in lost or even (less likely) added credits. Credit loss can result in slowing down the sender and possibly even deadlock; credit addition can lead to continuous dropping of messages.

We can make this protocol self-stabilizing by superimposing a periodic checking/correcting process (see Figure 1.7) on the original protocol. This process is triggered by a timer at the sender every few seconds. To initiate a checking phase, the

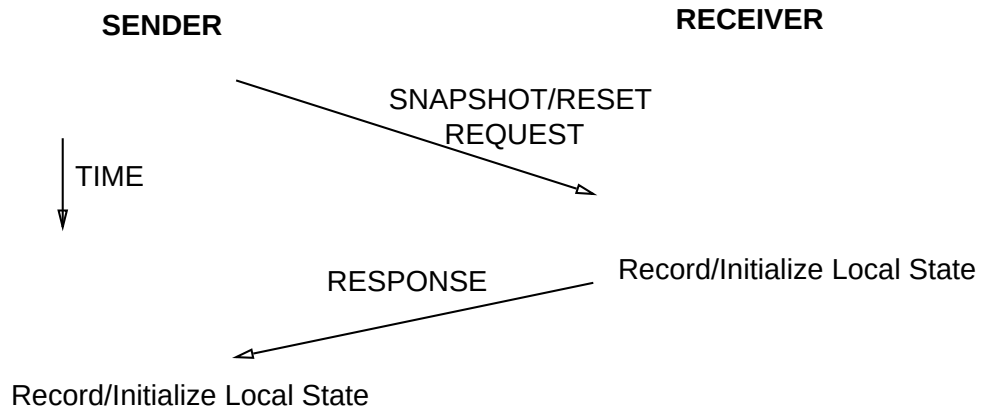


Figure 1.7: A Single Phase of Checking/Correction using Snapshots/Resets

sender (see Figure 1.7) sends a Snapshot ([CL85]) request message to the receiver. While checking, the sender also stops sending any messages on the link. When the receiver receives such a message, the receiver sends back a response containing the number of messages in its queue (Q) at the instant it sent the response. When the sender gets the response, the sender checks whether $Q + C = \mathit{Max}$, where C is the value of the credit register at the instant the response is received. If this condition is false, the sender infers that the local predicate is violated and initiates a reset phase.

To initiate a reset phase, the sender (see Figure 1.7) sends a Reset request message to the receiver. As in checking, the sender also stops sending any messages on the link until it gets a response.[2] When the receiver receives such a request, the receiver empties its queue and sends back a response. When the sender gets the response, the sender reinitializes its credit register to Max. In other words, the receiver reinitializes its local state on sending the response and the sender reinitializes its local state on receiving the response. Its not hard to see that if no errors occur during the reset phase, the local predicate will hold at the end of the reset phase.

Several optimizations can be added to this basic scheme. For example, it is possible to avoid having the sender stop sending messages during a checking phase by keeping track of some extra state variables. For this particular example, it is also possible to avoid a separate reset phase; instead when the sender receives a snapshot response, the sender can locally correct the credit register to account for any discrepancy.

[2]Chapter 5 contains details of how this protocol deals with lost request and responses.

We can now illustrate the point made earlier that self-stabilization can subsume the need for other fault-tolerance mechanisms. If the checking/correction procedure is activated fairly often (doing it once every second on a high speed link requires negligible overhead), then *there is no need for separate mechanisms when either the sender or receiver nodes or the two links crash and recover.* For a crash and recovery we do nothing special. Clearly the local predicate can be violated by such actions. However, after the next checking and correction phase the flow control scheme will begin working correctly. In the interim, messages may be lost but this is comparable to the time most protocols take to reinitialize after a link recovers; during this period the protocol is not providing service to the user. Thus the final scheme is both simple and fault-tolerant.

This fault-tolerant credit based scheme was proposed by us (for use on high speed links) at Digital Equipment Corporation ([CSV89]). Recently, we proposed a variant of this scheme for hop-by-hop flow control on ATM[3] at the ATM Forum meeting in Aug 1993. Local checking and correction is practical!

1.6.2 Extending the Idea to a General Network

Briefly, the rest of this thesis can be described as an extension (of the simple link checking and correction scheme described in the last subsection) to general network protocols.

So consider a network as shown in Figure 1.5. Recall that in the method of [KP90] there is a leader that periodically checks the network. If the leader discovers that the network is in an illegal state, the leader corrects the network by resetting it into a good state. Intuitively, *centralized* checking and correction is slow. It also has high message complexity.

Instead, we divide the network into a number of overlapping *link subsystems* as shown in Figure 1.8. A link subsystem consists of a pair of neighboring nodes and the channels between them. We wish to replace the *global, centralized* checking of [KP90] with *local, decentralized* checking. The intent, of course, is to allow each link subsystem to be checked in parallel. This results in faster stabilization.

[3] ATM stands for Asynchronous Transfer Mode. In ATM, messages are fixed sized "cells". There can also be multiple "circuits" per link each of which must be independently flow-controlled and checked.

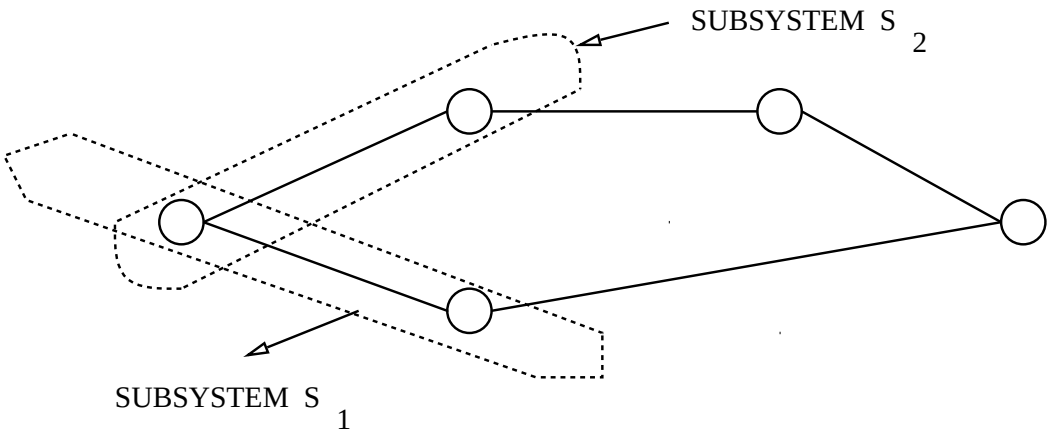


Figure 1.8: An Example of Two Overlapping Link Subsystems in a Network

We describe *sufficient* conditions under which these methods can be applied. Intuitively, a network protocol is *locally checkable* if whenever the protocol is in a bad state, some link subsystem is also in a bad state. Thus if the protocol is in a bad state, some link subsystem will be able to detect this fact locally. As in [KP90], we can correct a locally checkable protocol by doing (what we call) *global correction* of the network. However, in some cases we can do even better if the protocol is also *locally correctable.*

Intuitively, a network protocol is locally correctable if the network can be corrected to a good state by each link subsystem independently correcting itself to a good state. Clearly, this is non-trivial because link subsystems overlap (see Figure 1.8) at nodes. In the figure, the correction of link subsystem S_1 may cause subsystem S_2 to become incorrect.

1.6.3 Examples of Local Checking and Correction

We will go through three simple examples to make these notions clearer. The first example is not locally checkable, the second is locally checkable but does not appear to be locally correctable, and the third is both locally checkable and locally correctable.

For the first example, consider a token passing protocol in a line graph as shown in Figure 1.9. The line is oriented such that A is at the leftmost end and X is at

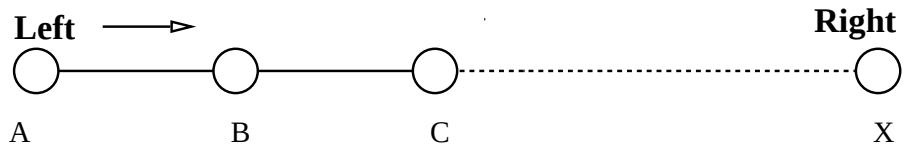


Figure 1.9: Token Passing in an Oriented Line Graph is not Locally Checkable

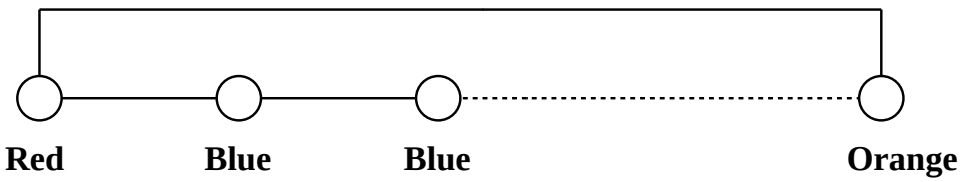


Figure 1.10: Coloring a Cycle is Locally Checkable but not Locally Correctable

the rightmost end. In normal operation, a single token is passed from left to right (i.e.. from A to X), and then back from right to left (i.e., from X back to A). Thus each node in the line will receive the token periodically. This protocol is not locally checkable if the graph has at least 3 nodes. Consider a typical link subsystem, for example the subsystem between neighbors B and C in Figure 1.9. Clearly, in normal operation it is possible for there to be no token at either B, C, or the channels between them. Thus having no tokens in a link subsystem is a legal state of a link subsystem. But this means that if there is no token in the entire network, no subsystem can detect this fact locally. Remember that subsystem checking is not coordinated.

For the second example, consider a protocol that colors the nodes of a cycle as shown in Figure 1.10. We require that the color of each node be either red, blue or green. We also require that the color of each node be different from that of its neighbors. Assume that in one atomic step a node can read the state of its neighbors and change its own state. However, steps of nodes can be arbitrarily interleaved.

Then the protocol is locally checkable. Suppose that node A has the same color as a neighbor B. Then, this can be detected within the link subsystem containing A and B.

However, it is not clear how to make this protocol locally correctable. Suppose

that all nodes are initially red. Then by the symmetry of the initial state, it appears that local corrections (or any corrections) are insufficient to correct the system to a good state. We could break symmetry with randomization. However, in this thesis we will only consider *deterministic* local correction procedures.

Consider the same problem of coloring the nodes of a graph, except that the graph is an *oriented* line graph as shown in Figure 1.9. Then the protocol is locally checkable *and* locally correctable. Suppose that that two nodes in a link subsystem (say B and C) have the same color. Then to correct the link subsystem, the color of the right node (i.e., C) is changed to any legal color different from the color of left node (i.e., B). Assume that correction actions occur in bounded time after they are enabled. Then within bounded time, node B will have a color different from that of A and will never change its color from this point on. Then within bounded time after node B's color stabilizes, node C will have a color different from that of B and will never change its color from this point on. By induction, we can show that all nodes are colored correctly in bounded time.

1.6.4 Why Local Checking is Useful

It is perhaps surprising that a number of useful network protocols are both locally checkable and locally correctable. In subsequent chapters we will describe locally checkable and correctable protocols for mutual exclusion, network resets, and end-to-end communication across an unreliable network. It may seem from the simple examples that our method is confined to acyclic graphs; this is not true: both the end-to-end and reset protocols work on arbitrary topologies. It also appears (see Chapter 7) that other existing protocols that work in dynamic networks (in which the topology can change due to link failures and recovery) are locally correctable. Protocols that are both locally checkable and correctable can be stabilized very quickly.

Protocols that are locally checkable but work on a tree topology can be stabilized in time proportional to the height of the tree. Thus we can remove the need for local correctability if the underlying topology is a tree. Another way to remove the need for local correctability (without restricting the topology to a tree) is to pay a price in stabilization time. Protocols that are locally checkable but not locally correctable can be made self-stabilizing by doing *global* correction using the network reset protocol developed in this thesis. The price for using global correction is that

the stabilization time now becomes proportional to the number of nodes. We describe stabilizing spanning tree and topology update protocols that use local checking and global correction.

We will also describe two compilers that can compile any deterministic synchronous protocol π into a self-stabilizing asynchronous version of π. The first compiler is stabilized by using local checking and correction, while the second compiler is stabilized using local checking and global correction. The significance of the compilers is that there are some network tasks (for example, computing a minimal spanning tree) for which a synchronous protocol exists but for which no locally checkable solution is known. Hence the compilers extend the range of our general techniques.

Thus while local checking cannot be used to solve every problem, there are a large number of useful protocols that can be efficiently stabilized using this notion. There are several benefits to this approach:

- The resulting protocols are efficient and stabilize quickly.
- The approach allows us to understand how to *design* self-stabilizing protocols in a systematic fashion. In fact, we will show that some existing self-stabilizing protocols can easily be understood in this framework.
- The approach allows us to *prove* self-stabilization properties of protocols in a modular way. This is because we limit ourselves to proving properties of link subsystems instead of arguing about global states.
- As a side benefit, local checking provides a useful debugging tool. Recall that each link subsystem periodically checks whether the subsystem is in a good state. Thus any violations can be logged for further examination. In a trial implementation of our reset procedure on the Autonet [MAM+90], local checking discovered bugs in the protocol code. In the same vein, local checking can provide a record of catastrophic, transient faults that are otherwise hard to detect.

1.7 Thesis Organization

The thesis is organized into three major parts as illustrated in Figure 1.11. The first part consists of three chapters on basic definitions and examples. The second part

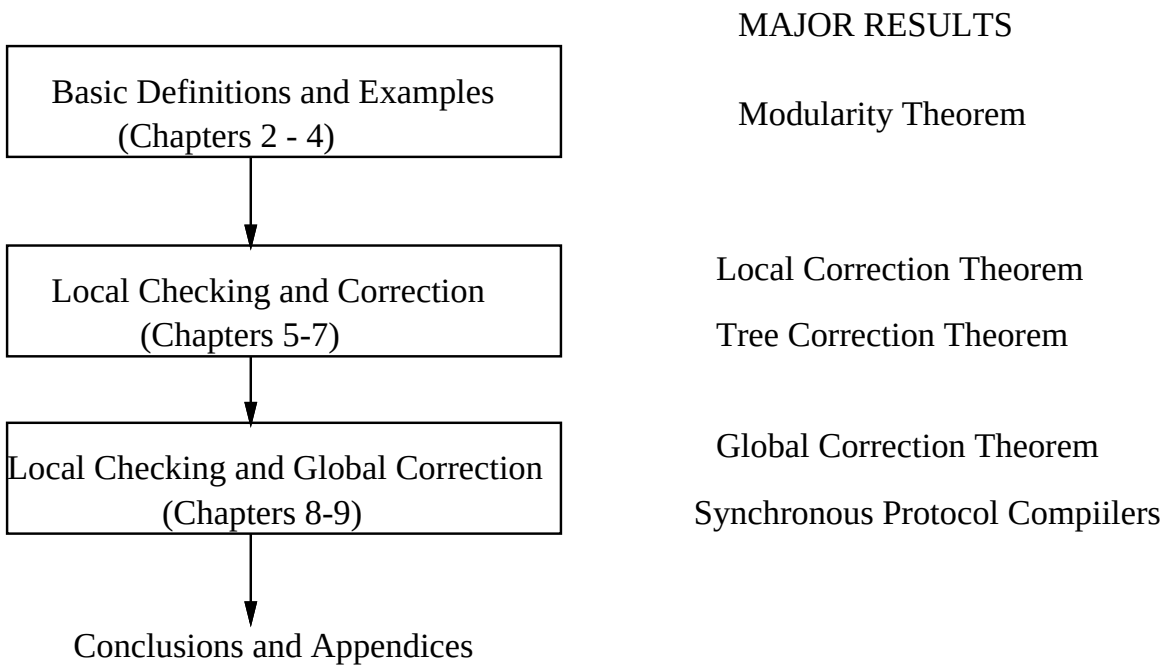


Figure 1.11: Thesis Organization

contains three chapters on local checking and *local* correction. The third part consists of two chapters on local checking and *global* correction. The final chapter presents our conclusions and contains a list of open questions. There are also several appendices. The first appendix is a list of frequently used notation. Next, there are appendices containing some details of proofs that were omitted in the main text for clarity. Finally, there is an index of commonly used terms and definitions. Figure 1.11 also summarizes the major results of the thesis.

We now describe each of the major parts in more detail below.

1.7.1 Basic Definitions and Examples: Chapters 2 - 4

Chapter 2 describes our model of computation, a variant of the timed Input/Output automata model ([MMT91], [LT89]. The model is basically a state machine model except that transitions are labelled with *action* names. By separating actions into *internal* and *external* actions, it is possible to define the correctness of an automaton in terms of its external *behavior*, where a behavior is a sequence of external actions.

Chapter 3 contains our definitions of stabilization. Roughly, we say that an automaton A stabilizes to some target set P of behaviors if every behavior of A has a suffix that is in P. The intuition is that the behaviors of A eventually begin to "look like" the behaviors in P. The actual definitions are slightly more complex in order to define what it means to stabilize in bounded time. Our behavior definitions are in contrast to previous definitions (e.g., [KP90] which are in terms of the states and executions of a system. We do have a definition of stabilization that corresponds to the standard definition; we use the behavior definition for *specification* and the standard definition for *proofs*.

Chapter 3 also contains our first important result. This is a *Modularity Theorem* that allows us to prove facts about the stabilization of a large system by proving facts about the stabilization of the system components. The theorem formalizes a "building block" approach to designing stabilizing protocols that we use throughout the thesis. Chapter 3 also describes a technique for proving stabilization properties. Chapter 3 is joint work with Nancy Lynch.

Chapter 4 contains a quick example of local checking and correction in the simplified shared memory model introduced by Dijkstra in [Dij74]. Because Dijkstra's model is so simple, it allows us to strip away extraneous detail and focus on the main ideas behind local checking and correction. However, readers who wish to concentrate on results for more realistic network models should skip Chapter 4 and go directly to Chapter 5. From Chapter 5 onwards, we use a network model suitable for modelling real networks. Chapter 4 is based on work done by the author. Independently, Anish Arora and Mohamed Gouda from the University of Texas at Austin have obtained similar results. Joint publication is planned.

1.7.2 Local Checking and Correction: Chapters 5 - 7

Chapter 5 begins by introducing our network model. Our network model is basically the standard asynchronous message passing model except for one important twist: *each link is restricted to store at most one packet at a time.* We argue that bounded storage link models are essential in a stabilizing context. We also argue our network model can be easily implemented in real networks. The network model of Chapter 5 is used for the reset of the thesis.

The concept of local checking and correction was introduced in [APV91b], and is joint work with Baruch Awerbuch and Boaz Patt. While these concepts are used throughout the thesis, [APV91b] did not present a formal description of the method. Instead [APV91b] described the method informally and showed how it could be used to stabilize two important protocols, one for end-to-end message delivery and one for network reset.

Chapter 5 gives a formal basis for the method of local checking and correction in message passing systems. The chapter contains formal definitions of local checkability and local correctability in a network model. These definitions are used to state the main result of the chapter, the *Local Correction Theorem*. This theorem shows that any locally checkable and correctable protocol can be transformed into an equivalent stabilizing protocol. The stabilization time of the resulting system is proportional to the height of a certain partial order that is used in the definition of local correctability. Chapter 5 is joint work with Nancy Lynch.

Chapter 6 applies the method of local checking to a simple mutual exclusion protocol. Chapter 6 also contains an important result, the *Tree Correction Theorem*. This theorem states that any locally checkable protocol on a tree can be efficiently stabilized in time proportional to the height of the tree. In other words, if the underlying topology is a tree we can dispense with the need for local correctability. The proof of this theorem is only sketched because we prove a corresponding tree correction theorem for shared memory systems in Chapter 4.

Chapter 7 links the second and third parts of the thesis by describing a stabilizing network reset protocol. Intuitively, a network reset protocol is a protocol that can be used by some other protocol P in order to restore P to a good state. Protocol P is given interfaces to make reset requests; the network reset protocol responds by providing reset *signals* at each network node. If each node (that implements P) locally initializes its state at the instant it receives a signal, then P will be restored to a good state.

In order to use such a network reset protocol as a tool for building other stabilizing protocols (as we do in the third part of the thesis) the network reset must itself be stabilizing. Chapter 7 applies the method of local checking and correction to create a stabilizing network reset protocol as described in [APV91b]. Chapter 7 is joint work with Baruch Awerbuch and Boaz Patt.

Chapter 7 also explores an interesting heuristic connection between locally correctable protocols and protocols that work in dynamic networks where links can fail and recover. The heuristic states that locally checkable protocols for dynamic networks can sometimes be made locally correctable. The basic idea is to use the link failure and recovery actions of the original protocol to locally correct link subsystems. This heuristic is the key to the proof of local correctability for our network reset protocol.

1.7.3 Local Checking and Global Correction: Chapters 8 - 9

The last part of the thesis contains two applications of *global* correction.

In Chapter 8 we prove another major result, the *Global Correction* theorem. This theorem states that any locally checkable protocol can be stabilized in time proportional to the number of network nodes. The Global Correction theorem shows that we can dispense with the need for local correctability and the need for the underlying topology to be a tree as long as we are willing to pay a higher price in stabilization time. The height of the underlying partial order in the local correction method and the height of the tree in in the tree correction method are typically smaller than the number of network nodes. Thus it pays to use local correction or tree correction wherever possible.

We present stabilizing protocols for computing a spanning tree and solving the topology update problem as examples of Global Correction. The spanning tree and topology update protocols are based on joint work with Baruch Awerbuch and Boaz Patt that is also described in [APV91a]. The protocols in Chapter 7 and 8 are both efficient and practical and can be applied to real networks.

Chapter 9 develops two compilers that can convert any synchronous protocol π into a self-stabilizing asynchronous version of π. The main compiler, the *Resynchronizer*, works by first applying the synchronizer protocol of [Awe85] to create an asynchronous version of π. Next we use global correction to make the resulting protocol stabilizing. This can be done by using a stabilizing reset protocol to periodically restart an asynchronous version of protocol π. The proof of the current version of the *Resynchronizer* protocol is incomplete. However, we have a proof of a much more complicated version of the *Resynchronizer* protocol that was originally reported in [AV91]. The construction in this thesis is much simpler than our original construction in [AV91] but its

proof is incomplete. Thus chapter 9 is best regarded as a set of useful ideas that need polishing. Chapter 9 is joint work with Baruch Awerbuch

1.8 Reading the Thesis

Most chapters and long sections contain a roadmap at the start that explains the organization of the chapter or section. Similarly most chapters end with a summary of the important ideas in the chapter. Since it is easy to forget a piece of notation or a definition, the reader may also wish to consult the notation appendix and the index.

Whenever the proof of a theorem or lemma is too long, we give an intuitive explanation of why the theorem or lemma works in the main text, and provide more details later or in the appendix. On a first reading, the reader is advised to skip the details.

We believe that self-stabilization is useful and practical and hope that systems readers can also read this thesis. Chapter 2 is written to help readers unfamiliar with formal methods to get comfortable with the formalism we use. A systems reader wishing to get a quick summary of the results can read the introduction, summary and main theorems in each chapter. Once the reader gets comfortable with our method of describing protocols it should also be easy to read the actual code of the protocols. The complicated (and important) pieces of code are heavily commented and are preceded by informal descriptions.

Chapter 2

The I/O Automaton Model

A formal description of an algorithm is a precise and unambiguous description of the algorithm. Formal descriptions of sequential algorithms have proved to be useful. Distributed algorithms are more complicated than sequential algorithms because they have to deal with *parallelism, asynchrony*, and *fault-tolerance*. Thus we will often give a formal presentation of the protocols in this thesis.

By describing algorithms formally, we hope to describe them precisely so as to avoid ambiguity and to permit careful proofs of correctness. However, it is often hard to do so and yet convey the important ideas. We will try to combat this by providing intuitive explanations along with formal descriptions.

To describe algorithms precisely, we use an underlying mathematical model. The idea is that after we model a real-life distributed algorithm, we can study the algorithm purely in terms of its mathematical properties. Despite this, we will often return to what these mathematical symbols represent.

In this chapter we will describe the computation model used to describe the distributed algorithms in this thesis. The first section of this chapter is an intuitive introduction to the I/O automaton model. This section is written for readers unfamiliar with the I/O automaton model. The second section of the chapter contains a formal description of the variant of the I/O automaton model that we use in the rest of the thesis. Readers already familiar with the I/O automaton model may wish to only read the formal description in Section 2.2.

2.1 The I/O Automaton Model

Our model of computation is a variant of the *timed Input/Output automaton* model [MMT], which in turn is based on the Input/Output automaton model of Lynch and Tuttle [LT89]. We will omit the word "timed" in what follows. For instance, we will refer to a timed I/O automaton simply as an automaton.

2.1.1 Why use the I/O Automaton Model?

We wish to model systems of processes that compute but also *communicate* with other processes. The processes do not have a common clock and so communication is *asynchronous*. A sequential algorithm computes some function of its input and then halts. By contrast, our processes can continuously receive input from and *react to* their environment.

Consider the example of a token passing ring. Such rings are the basis of a number of local area networks that interconnect computers in offices and on college campuses. A token passing ring consists of a number of processes, say 0 to $n-1$, connected together in a ring. To prevent more than one process from transmitting at the same time, a token packet is passed from process to process. A process can transmit only when it has a token. A process passes its token to its clockwise neighbor a bounded time after it finishes transmitting. It is easy to see that the system works correctly if there is exactly one token in the system initially.

The simplest model of the token passing system is a big state machine. Each node is a state machine that sends and receives packets and so are the channels between nodes. The state of a channel is the sequence of packets stored in the channel. Unfortunately, such monolithic models often do not have a property we call *compositionality*. A model is said to be *compositional* if we can infer the behavior of the system from the behavior of its components. This allows modular specification and modular proofs.

The I/O automaton model is essentially a state machine model. However, it has a few extra features that make it a compositional model.

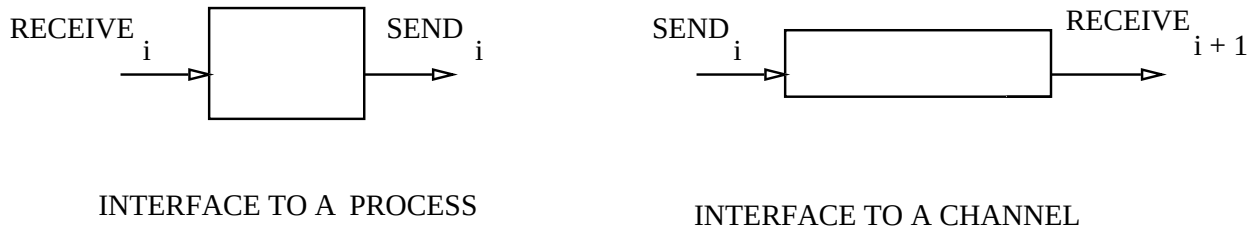


Figure 2.1: The Interface to Process i and the Channel between Process i and Process $i+1$ in a token passing system

2.1.2 Four Important Features of the I/O Automaton Model

In the I/O automaton model, both systems and processes are modelled by an I/O automaton. An I/O automaton is an automaton (i.e., a state machine) with the following additional features.

First, all state transitions of the automaton are labelled with names that are known as *actions*. Further, *all actions are classified into three categories: input, output, and internal.* Intuitively, *input actions* are actions caused by the external world or environment and to which the automaton must respond. *Output actions* are actions caused by the automaton and to which the environment must respond. Finally, *internal actions* are transitions that are neither input or output actions: such actions only change the internal state of the automaton.

As an example, Figure 2.1 shows a single process in the token passing system, say Process i. Of course, Process i must have interfaces to send and receive packets. We can model these interfaces using an input action $\text{RECEIVE}_i(p)$ to receive a packet p and an output action $\text{SEND}_i(p)$ to send a packet p. In the figure, we have not shown any internal actions. Figure 2.1 also shows the external interface to a channel between Process i and Process $i+1$. Notice that $\text{SEND}_i(p)$ is an input action and $\text{RECEIVE}_{i+1}(p)$ is an output action for this channel. Intuitively, this corresponds to the fact that when Process i sends a packet as an output action, that packet must simultaneously be stored in the channel using a channel input action.

The classification of actions allows a simple scheme for "plugging" together automata so that they can can communicate. The formal name for this scheme is *composition.* Composition is based on an idea we have already alluded to. When automata

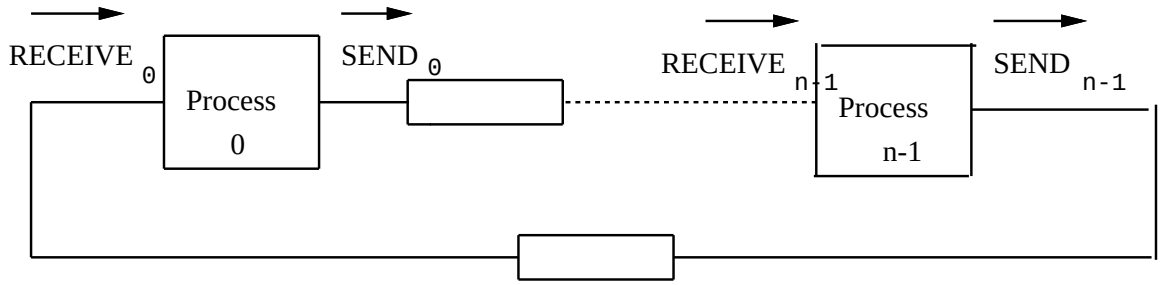


Figure 2.2: The token passing system formed by composition of processes and channels

are composed, the output actions of the automata are identified with input actions of other automata that share the same name. As an example, suppose we compose Process i with the channel between Process i and Process $i+1$. Then the output action $\text{SEND}_i(p)$ of Process i gets identified with the input action $\text{SEND}_i(p)$ of the channel. When we run the new composed automaton, whenever Process i performs a $\text{SEND}_i(p)$ action, the channel will simultaneously perform a $\text{SEND}_i(p)$ action. Thus the second feature of the I/O automaton model is that *automata communicate by simultaneous performance of shared actions.*

Continuing our example, we can compose the automata for Process $i, 0 \leq i \leq n-1$ and the channels between them to form a new automaton that represents the token passing system. This is shown in Figure 2.2. We assume that all arithmetic on process indices is mod n.

Suppose an automaton A wishes to perform an output action π that is also an input action of another automaton B. Some models allow automaton B to block its inputs in certain states. If B can block input actions, then A must somehow "handshake" with B before A can perform action π. This in turn implies that action π is jointly controlled by A and B. In the I/O automaton model, things are much simpler because of a third feature of the model. *Input actions are enabled in every state of an automaton.* Thus there is no need for handshaking, and an action is controlled solely by the originator of the action.

The assumption that inputs cannot be blocked is extremely natural for the message passing systems studied in this thesis. In real message passing systems, a process must be prepared to receive packets in any state. Of course, a process may choose to drop a packet when it receives it. On top of this basic model, processes may choose to implement a flow control scheme to prevent senders from sending to receivers when

the receivers are unable to process packets. However, such "flow control" is not part of the basic I/O automaton model.

Output and internal actions of an automaton A are under the control of A; hence they are also called *locally controlled actions*. Typically, we need to specify certain "liveness" guarantees on the performance of locally controlled actions. For example, in the token passing system we need to specify that a node holds the token for a bounded amount of time. We specify such guarantees using a fourth feature of timed automata. *The locally controlled actions of an automaton are partioned into a number of equivalence classes. Each class c in this partition has a time t_c which represents an upper bound on the performance of an action in class c.*

Intuitively, each class represents the set of actions under the control of one system component. The automaton will guarantee "fair turns" to the enabled actions in each class. Suppose some action of class c is enabled at time t. One can think of the automaton as having a scheduler that checks every class at time periods of at least t_c. If some action of class c is enabled when the scheduler checks the class, then some action in the class is performed. More precisely, we require that either some action of class c will be performed by time $t + t_c$, or no action of class c is enabled in some state that occurs before time $t + t_c$. Returning to our example, we specify that each $\text{SEND}_i(p)$ action at Process i is in a separate class with associated class time, say t_n. We do so because each Process i is a distinct system component that controls the sending of its own messages.

When we compose automata, the state set of the resulting automaton is the cross product of the state sets of the the component automata. More interestingly, the timing partition of the new automaton is the union of the timing partitions of the component automata. Thus composition preserves the timing guarantees of the constituent automata.

2.1.3 Specifying Correctness in the I/O Automaton Model

How do we specify the correctness of an automaton? We take the token passing system as an example. Our first attempt may be to specify correctness in terms of a set of legal states: a token passing system is correct if in any state there is exactly one token. But this allows implementations in which the token always remains at Process 1. Thus

we also need to specify that every process receives a token within a bounded amount of time. To do this, we need to describe *executions* of the system and to model how time passes.

To model how time passes, we use the concept of a timed state and a timed action. Intuitively, a timed state is a pair (s, t) where s is a state of the automaton and t is a time; t is read as the time associated with state s. Similarly, a timed action is a pair (a, t) where a is an action of the automaton.

When an automaton "runs", it generates a string representing an *execution* of the system the automaton models. This string is simply an alternating sequence of timed states and timed actions, that begins with a timed start state. An execution must respect the state transition rules of the automaton and the timing guarantees specified by each class of the automaton. Section 2.2 contains a more precise description.

Using this definition of an execution, we can say that a token passing system S is correct if for every execution α of S:

- There is exactly one token in any state of α.
- Within a bounded time after any state of α, Process i receives a token.

Specifying correctness using a set of legal executions is a reasonable solution that we will use sometimes. However, the correctness specification refers to the state of the implementing system. Ideally, we should treat the implementing system as a "black box" and describe its correctness in terms of its externally visible behavior.

Suppose we wished our token passing system to be used by other applications. Then we need to specify additional actions to act as an interface to such client applications. To do so we add two additional actions to each Process i. The first is an output action DELIVER_TOKEN$_i$ and the second is an input action RETURN_TOKEN$_i$. This is shown in Figure 2.3. Intuitively, DELIVER_TOKEN$_i$ is used by Process i to deliver a token to the external client; RETURN_TOKEN$_i$ is used by the external client to return the token to Process i. Suppose we now compose all process and channel automata. Next, we reclassify all actions of the composition as internal actions except the DELIVER_TOKEN and RETURN_TOKEN actions. Such a reclassification can be done formally using a "hide" operator.

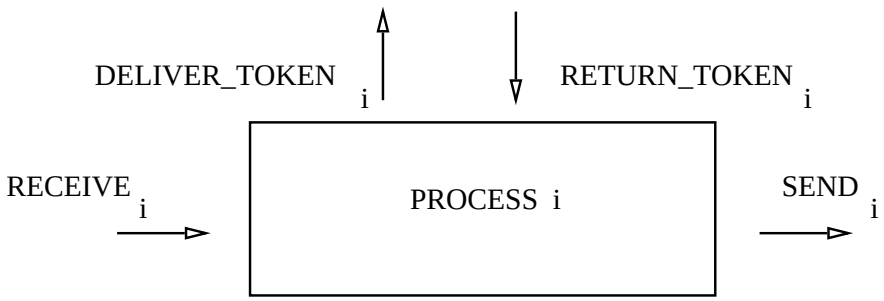


Figure 2.3: Process i with additional interfaces to an external client

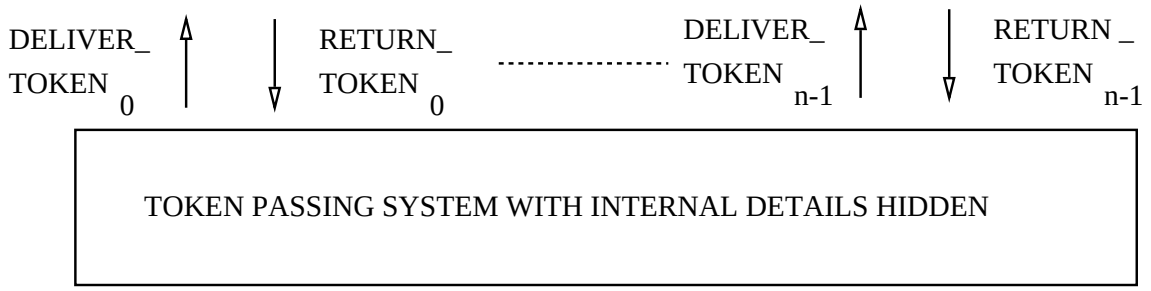


Figure 2.4: A modular token passing system and its external interface to clients

The resulting automaton is shown in Figure 2.4. It essentially reflects the interface between the token passing system and its clients. A natural question is: can we specify correctness of the system solely in terms of the external interface to the clients?

To answer this question, we first define an *external behavior* (or *behavior* for short) of an automaton. A behavior corresponding to an execution α is the subsequence of α consisting only of timed input and output actions, together with a *start time*. The start time of the behavior is equal to the time associated with the first state of α. A sequence β is said to be a behavior of automaton A if β is the behavior corresponding to some execution of A. Clearly any behavior of the automaton in Figure 2.4 will consist only of DELIVER_TOKEN and RETURN_TOKEN actions.

Using the notion of a behavior, we can define the "external" correctness of the token passing system as follows. A token passing system S is said to be correct if for any behavior β of S the following two properties hold in the sequence β:

- There must be a RETURN_TOKEN$_i$ between any DELIVER_TOKEN$_i$ and a later DELIVER_TOKEN$_j$ for any i, j.

- Suppose that in β, a RETURN_TOKEN$_i$ occurs in bounded time after every DELIVER_TOKEN$_i$ for all i. Then for any j and any suffix γ of β, a DELIVER_TOKEN$_j$ will occur in bounded time after the start of γ.

The first condition is a "safety" property. It guarantees that Process j will not receive the token until all processes that have received the token before j have returned the token. The second property is a "liveness property". It ensures that Process j will get the token periodically. However, we can only guarantee this property if all external clients return the token in bounded time after a token is delivered to them. Notice a modelling trick that is being used here. While an I/O automaton A must allow all possible inputs, we can specify that A exhibit correct behavior only on certain "well-formed" inputs.

2.2 Formal Summary of the I/O Automaton Model

We summarize our discussion so far. In this thesis, we will use the following model which is a special case of the model in [MMT91]. However, our terminology is slightly different from that of [MMT91].

An *automaton* A consists of five components:

- a finite set of actions $actions(A)$ that is partitioned into three sets called the set of *input, output, and internal* actions. The union of the set of input actions and the set of output actions is called the set of *external* actions. The union of the set of output and internal actions is called the set of *locally controlled* actions.
- A finite set of states called $states(A)$.
- A nonempty set $start(A) \subseteq states(A)$ of start states.
- A transition relation $R(A) \subseteq states(A) \times actions(A) \times states(A)$ with the property that for every state s and input action a there is a transition $(s, a, \tilde{s}) \in R(A)$.
- An equivalence relation $part(A)$ partitioning the set of locally controlled actions into equivalence classes, such that for each class c in $part(A)$ we have a positive real upper bound t_c. (Intuitively, t_c denotes an upper bound on the time to perform some action in class c.)

An action a is said to be *enabled* in state s of automaton A if there exist some $\tilde{s} \in states(A)$ such that $(s, a, \tilde{s}) \in R(A)$. An action a is *disabled* in state s of automaton A if it is not enabled in that state. Since one action may occur multiple times in a sequence, we often use the word *event* to denote a particular occurrence of an action in a sequence.

To model the passage of time we use a time sequence. A *time sequence* $t_0, t_1, t_2, \ldots$ is a non-decreasing sequence of non-negative real numbers; also the numbers grow without bound if the sequence is infinite. A *timed element* is a tuple (x, t) where t is a non-negative real and x is an element drawn from an arbitrary domain. A *timed state* for automaton A is a timed element (s, t) where s is a state of A. A *timed action* for automaton A is a timed element (a, t) where a is an action of A.

Let $X = (x_0, t_0), (x_1, t_1), \ldots$ be a sequence of timed elements. We will also use $x_j.time$ (which is read as the time associated with element x_j) to denote t_j.

We say that element x_j occurs within time t of element x_i if $j > i$ and $x_j.time \leq x_i.time + t$. We will use $X.start$ (which is read as the start time of X) to denote t_0.

Definition 2.2.1 *An* execution α *of automaton A is an alternating sequence of timed states and actions of A of the form $(s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), (s_2, t_2), \ldots$ such that the following conditions hold:*

1. *$s_0 \in start$ and $(s_i, a_{i+1}, s_{i+1}) \in R$ for all $i \geq 0$.*
2. *The sequence can either be finite or infinite, but if finite it must end with a timed state.*
3. *The sequence $t_0, t_1, t_2, \ldots$ is a time sequence.*
4. *If any action in any class c is enabled in any state s_i of α then within time $s_i.time + t_c$ either some action in c occurs or some state s_j occurs in which every action in c is disabled.*

Notice that the time assigned to any event a_i in α (i.e., $a_i.time$) is equal to the time assigned to the next state (i.e., $s_i.time$). Notice also that we have ruled out the possibility of so-called "Zeno executions" in which the execution is infinite but time stays within some bound.

Definition 2.2.2 *Consider an execution α of A. Let γ be the subsequence of α consisting of timed external actions, and let t_0 be the start time of α. The* behavior β *corresponding to α is the sequence $\beta = t_0, \gamma$.*

Notice that the start time of a behavior is the start time of the corresponding execution. The behaviors of automaton A are the behaviors corresponding to the executions of A.

Notice that a behavior is not the same "type" of sequence as an execution since a behavior consists of a start time followed by a sequence of timed actions. Formally, a *behavior sequence* β is a sequence $t_0, (a_1, t_1), (a_2, t_2), \ldots$ such that each a_i is drawn from some set of actions and such that $t_0, t_1, t_2, \ldots$ is a time sequence. Note that any behavior of an automaton is a behavior sequence. We will use $\beta.start$ to denote the first element in β; $\beta.start$ can be read as the start time of behavior sequence β. As before, we will use $a_j.time$ to denote t_j.

Consider an execution $\alpha = (s_0, t_0), (a_1, t_1), (s_1, t_1), (a_2, t_2), \ldots$. The untimed execution corresponding to α is the sequence $s_0, a_1, s_1, a_2, s_2, \ldots$. For brevity, we will frequently describe execution α by the corresponding untimed execution $s_0, a_1, s_2, \ldots$. By our notation, the time associated with any state s_i (or action a_j) in α is $s_i.time$ (or $a_j.time$).

Similarly consider a behavior $\beta = t_0, (b_1, t_1), (b_2, t_2), \ldots$ The untimed behavior corresponding to β is the sequence $b_1, b_2, \ldots$. We will often describe behavior β using the corresponding untimed behavior $b_1, b_2, \ldots$. Once again by our notation, the start time of β is $\beta.start$ and the time associated with any action b_j in β is $b_j.time$.

Notice that the time associated with the first state of an execution, $s_0.time$, is allowed to be an *arbitrary non-negative* real number. As we see below, this assumption allows a clean statement of an important lemma about stabilizing automata. In the timed automaton model [MMT91], each class also has an associated lower bound. In our model, the lower bound associated with each class is implicitly assumed to be zero. These two assumptions (or lack of assumptions) restrict us to modelling systems in which the value of time is not used by the protocol. In the protocols we describe later, we will use time only to model liveness guarantees and to measure time complexity.

The following lemma is useful later.

Lemma 2.2.3 *Consider any execution α of an automaton A. Any suffix of α that starts with a timed start state is also an execution of A.*

Proof: In essence, the lemma follows because we have no lower bounds on the time between actions. ∎

2.2.1 Composition and Hiding

To describe a collection of automata we will use a finite index set, say I. For example, an index set could be the set of vertices in a graph, or the set of edges in a graph. Thus we often speak of a collection $\{A_i, i \in I\}$ of automata, where I is an index set. Often we are interested in composing such a collection of automata.

Before automata can be composed they must obey certain restrictions. Clearly, the automata cannot share internal or output actions without violating the principle that each action is controlled by exactly one automaton. A collection of automata $\{A_i, i \in I\}$ is said to be *compatible* if the collection is finite, the output actions of all automata are disjoint, and the internal actions of any automaton are disjoint from the actions of any other automaton. Notice that this allows several automata to have a common input action. This can be used, for instance, to model a broadcast from one automaton to several other automata.

Let I be a finite index set. The composition of a compatible collection $\{A_i, i \in I\}$ is denoted by $A = \Pi_{i \in I} A_i$. A is the automaton formed as follows:

- An action π is an input action of A if π is the input action of some A_i in the collection and *is not* the output action of some other A_j in the collection. The set of output actions of A is the union of the output actions of the collection. The set of internal actions of A is the union of the internal actions of the collection.
- The set of states of A is the cross-product of the state sets of the automata in the collection. The set of initial states of A is the cross-product of the initial state sets of the automata in the collection.
- Let $s|i$ denote the projection of some state s of A onto automaton A_i. Then the transitions of A are the triples $(s, a, \tilde{s})$ such that for any $i \in I$:

- If a is an action of A_i then $(s|i, a, \tilde{s}|i)$ is a transition of A_i.
- If a is not an action of A_i then $s|i = \tilde{s}|i$.

- The partition of A is the union of the partitions in the collection. Thus if an action a belongs to some class c of any A_i, then a belongs to class c in A. The upper bound corresponding to class c in A is the upper bound corresponding to class c in A_i.

Note: Suppose we simply took the set of input actions of A to be the union of the set of input actions of the components. Then any action that is an input action of say A_i and an output action of say A_j will be classified both as an input and output action of A. It is to avoid this problem that the input actions of A are defined the way they are. Notice also that any action of the component automata is also an action of the composition.

We return to our claim that the I/O automaton model is *compositional*. We would like to show that the behavior of a composition can be inferred from the behavior of its components. We translate the following two lemmas from [MMT].

We use $\beta|A_i$ to represent the projection of a behavior β of A on to some constituent automaton A_i. The projection is the subsequence of β containing actions of A_i. We also assume that $\beta|A_i$ inherits the times of β in the natural way. Thus the time associated with any action in $\beta|A_i$ is the time associated with the corresponding action in β, and the start time of $\beta|A_i$ is the start time of β.

The first lemma shows how we can "cut" a behavior of the composition into behaviors of each of the pieces. The second lemma shows how we can "paste" a sequence of behaviors of the pieces into a behavior of the composition.

Lemma 2.2.4 Cut Lemma *Let $\{A_i, i \in I\}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Let β be any behavior of A. Then $\beta|A_i$ is a behavior of A_i for every $i \in I$.*

Lemma 2.2.5 Paste Lemma *Let $\{A_i, i \in I\}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Let β be a behavior sequence such that each action in β is an external action of A. If $\beta|A_i$ is a behavior of A_i for every $i \in I$, then β is a behavior of A.*

Finally, there is a hiding operator on automaton A with respect to some subset Σ of the actions of A. The result of this operation is to to create a new automaton that is identical to A except that all actions in Σ are reclassified as internal actions.

2.2.2 Useful Notation

In all the automata that we will describe in this thesis, the state of an automaton consists of values assigned to a set of variables. We use record notation to extract the values of specific variables in the state. We say that a variable x has a value v in state s if $s.x = v$. We sometimes omit the state s if it is clear by context which s we mean.

When we refer to the state s_i in execution α we mean the i-th state in the sequence α. An *interval* of an execution α is a contiguous subsequence of α that starts and ends with a timed state. The *duration* of an interval $[s_i, s_j]$ is $s_j.time - s_i.time$. An *interval* of a behavior β is a contiguous subsequence of β.

A *predicate* of automaton A is a subset of the states of A. A predicate S is described by a Boolean formula on variables; a state $s \in S$ iff the values of the variables of A in state s satisfy the Boolean formula. Thus if S is $x = 3$ then $s \in S$ iff $s.x = 3$. We also say that S holds in state s or S is true in state s to mean $s \in S$. We say that a predicate S remains true for time t after state s_i in execution α if for all states s_j that occur within time t of s_i in α, $s_j \in S$.

2.2.3 Modelling Asynchronous Protocols

In this thesis we will study "asynchronous" protocols. We wish such protocols to work regardless of timing assumptions. This is typically done using "fairness assumptions" instead of timing assumptions. However, we can model essentially the same thing by *ensuring that our protocols work regardless of the value of t_c assigned to any class c.*

The advantage of using parameterized upper bounds for classes comes in measuring time complexity. In the standard approach, after first proving correctness using "fair" executions, time complexity is then measured using the the assumption that the class times t_c are constants like 1 or 0. Often the time complexity arguments are extremely similar to the liveness arguments used in the proof of correctness. In our approach there is no need for this double effort; we replace liveness arguments by showing time

bounds on certain events occurring. These time bounds are parameterized in terms of the class times t_c. Thus to obtain time complexity measures, we simply substitute particular values for t_c.

Chapter 3

Stabilization: Definitions and Properties

In this chapter we will describe the basic definitions of stabilization we will use for the rest of the thesis. We begin in the first section with a state-based definition of stabilization that corresponds to the standard definitions in the literature. In the next section we describe a new definition of stabilization in terms of external behaviors. In Section 3.4 we describe a two stage proof technique for proving stabilization properties. Then in Section 3.5 we describe a modularity result. This result shows that (under certain conditions) we can prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts.

3.1 Definitions of Stabilization based on Executions

All the existing definitions of stabilization are in terms of the states and executions of a system. We will begin with a definition of stabilization that corresponds to the standard definitions (for example, that of Katz and Perry [KP90]). In the next section, we will describe another definition of stabilization in terms of external behaviors. We believe that the definition of behavior stabilization is appropriate for large systems that require modular proofs. However, the definition of execution stabilization given below is essential in order to *prove* results about behavior stabilization. We begin with the definition of execution stabilization since it is also the definition that most readers

are familiar with.

Suppose we define the correctness of an automaton in terms of a set C of legal executions. For example, recall that for the token passing system of Chapter 2 we defined the legal executions to be those in which there is exactly one token in every state, and in which every process periodically receives a token.

What do we mean when we say that an automaton A stabilizes to the executions in set C in time t? Intuitively, we mean that within time t, all executions of A begin to "look like" an execution in set C. For example, suppose C is the set of legal executions of a token passing system. Then in the initial state of A there may be zero or more tokens. However, the definition requires that within time t of the start of any execution of A, there is exactly one token in any state.

To formalize this, we begin with the definition of a t-suffix of an execution α. Intuitively, this is a suffix of α whose first element occurs no more than time t after the start of α. Although we will apply this definition only to executions, we will state the definition in terms of an arbitrary sequence of timed elements. Recall from Chapter 2 that a timed element is a tuple (x, t) where x is either a state or an action, and t is a time.

Definition 3.1.1 *Consider any two sequences of timed elements α and α'. We say that α' is a t-suffix of execution α if:*

- $\alpha'.start - \alpha.start \leq t$ *and*
- α' *is a suffix of* α.

We can now formally define execution stabilization to a set of executions:

Definition 3.1.2 *Let C be a set of sequences of timed elements. We say that automaton A stabilizes to the executions in C in time t if for every execution α of A there is some t-suffix of execution α that is in C.*

We also make the accompanying definition of execution stabilization to another automaton:

Definition 3.1.3 *We say that automaton A stabilizes to the executions of automaton B in time t if for every execution α of A there is some t-suffix of execution α that is an execution of B.*

So far, we have not made any assumptions about the automaton A that stabilizes to the executions specified by a set C or another automaton B. However, recall that in Dijkstra's original definition, an automaton is "self-stabilizing" if regardless of what state it starts in, an automaton "eventually" produces "legal" executions. Our definitions are more general than Dijkstra's definitions. To see this, we now extend Dijkstra's notion of "self-stabilization" to our timed setting.

As a stepping stone, for any automaton A we define $U(A)$ (which can be read as the unrestricted version of A) to be the automaton that is identical to A except that any state of A can be a start state of $U(A)$.

Definition 3.1.4 *For any automaton A, we let $U(A)$ denote the automaton that is identical to A except that $start(U(A)) = states(A)$.*

Next, we can say that an automaton A "self-stabilizes" in time t if the following holds: even if we start A in a state other than one of it's start states, the resulting execution will begin to "look like" a properly initialized execution of A within time t. Formally:

Definition 3.1.5 *We say that an automaton A self-stabilizes in time t if $U(A)$ stabilizes to the executions of A in time t.*

The following simple lemma shows that execution stabilization is transitive. This is an important lemma because it allows to prove execution stabilization in several stages.

Lemma 3.1.6 *If automaton A stabilizes to the executions of automaton B in time t_1 and B stabilizes to the executions of automaton C in time t_2, then A stabilizes to the executions of C in time $t_1 + t_2$.*

3.2 Definitions of Stabilization based on External Behavior

In Chapter 2, we argued that a major theme of the I/O Automaton model [LT89] is the focus on external behaviors of an automaton. For instance, the correctness of an automaton is specified in terms of its external behaviors. In Chapter 2 for instance, we showed how to specify the correctness of a token passing system without any reference to the state of the system. We did this by specifying the ways in which token delivery and token return actions can be interleaved

Thus it natural to look for a definition of stabilization in terms of external behaviors. We would also hope that such a definition would allow us to modularly "compose" results about the stabilization of parts of a system to yield stabilization results about the whole system.

Typically, the correctness of an IOA is specified by a set of legal behaviors P. An IOA A is said to *solve* P if the behaviors of A are contained in P. For stabilization, however, it is reasonable to weaken this definition and ask only that an automaton exhibit correct behavior after some finite time. In most of this thesis, we will use the behavior stabilization definitions for *specifying* the stabilization properties of a system.

As in the case of execution stabilization, we begin with the definition of a t-suffix of a behavior β. Intuitively, this is a portion of β that starts at time no more than t after the start of β. However, this is not as easy as defining a t-suffix of an execution. Recall that a behavior $\beta = (t_0, \gamma)$ consists of two components: a start time t_0 and a sequence of timed actions γ. Thus we cannot simply define a t-suffix of β to be a suffix of β as we did in the case of an execution. We would also like a t-suffix of β to be a behavior sequence: thus the t-suffix must have a start time as well as a sequence of timed actions.

Figure 3.1 is a pictorial view of a behavior β. The second row represents the sequence of actions of the behavior and the first row represents the times corresponding to each action as well as the start time of the behavior. The dashed line to the right of the start time represents an instant of time that occurs no more than time t after the start of the behavior. Now consider the behavior that starts at the time corresponding to the dashed line and consisting of all actions to the right of the dashed line. We will call such a behavior a t-suffix of behavior β. Intuitively, as we said before, this

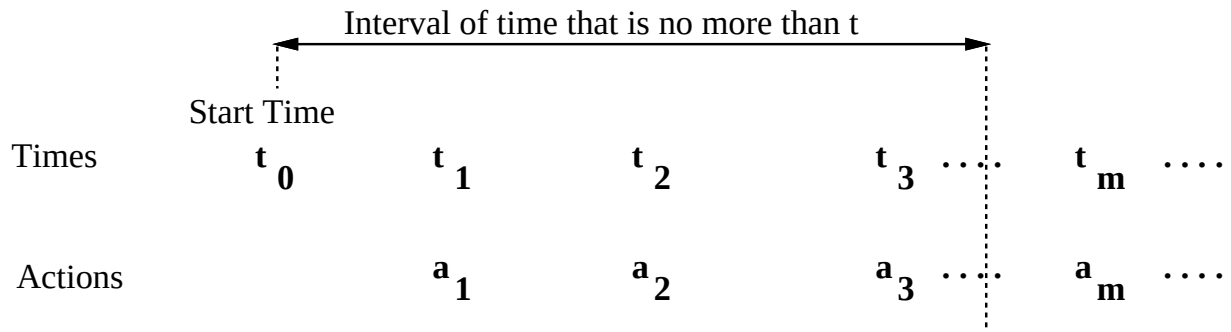


Figure 3.1: A pictorial view of a t-suffix of a behavior

represents a portion of β. However the portion starts at time no more than t after the start of β. Formally:

Definition 3.2.1 *Consider any two behavior sequences $\beta = t_0, \gamma$ and $\beta' = t'_0, \gamma'$. We say that β' is a t-suffix of behavior β if:*

- $\beta'.start - \beta.start \leq t$.
- *γ' is a suffix of γ containing all actions in β that occur at times strictly greater than $\beta'.start$.*

Note that the definition allows γ' to contain some, none, or all of the actions in β that occur at times equal to $\beta'.start = t'_0$. Note that the definition is similar but yet different from the definition of a t-suffix of an execution. Using this, we can now define behavior stabilization analogous to execution stabilization:

Definition 3.2.2 *Let P be a set of behavior sequences. An IOA A stabilizes to the behaviors in P in time t if for every behavior β of A there is a t-suffix of behavior β that is in P.*

An automaton is said to *solve* another automaton B if every behavior of A is a behavior of B. Similarly, we can specify that A stabilizes to the behaviors of some other automaton B.

Definition 3.2.3 *An automaton A is said to stabilize to the behaviors of another automaton B in time t if for every behavior β of A there is a t-suffix of behavior β that is a behavior of B.*

The following lemmas are "obvious" in that they are what we expect to be true.

Lemma 3.2.4 *Every automaton stabilizes to its own behaviors in time* 0.

Lemma 3.2.5 *If every behavior of automaton A is a behavior of automaton B, then A stabilizes to the behaviors of B in time* 0.

The next lemma is a transitivity result for behavior stabilization. It allow us to prove behavior stabilization results in stages.

Lemma 3.2.6 *If automaton A stabilizes to the behaviors of automaton B in time t_1 and B stabilizes to the behaviors of automaton C in time t_2, then A stabilizes to the behaviors of C in time $t_1 + t_2$.*

Another obvious consequence of our definition is:

Lemma 3.2.7 *If automaton A stabilizes to the behaviors of an automaton B in time t and $\tilde{t} \geq t$ then A stabilizes to the behaviors of B in time $\tilde{t}$.*

The previous lemma motivates a natural complexity measure called the *stabilization time* from A to B. Intuitively, this is the smallest time after which we are guaranteed that A will behave like B. However, since we are dealing with a potentially infinite set of behaviors we have to be a little more careful.

Definition 3.2.8 *The stabilization time from A to B is the infimum of all t such that A stabilizes to the behaviors of B in time t.*

The next lemma is simple but important because it ties together the execution and behavior stabilization definitions. It states that execution stabilization implies behavior stabilization. In fact, the only method we know to *prove* a behavior stabilization result is to first prove a corresponding execution stabilization result, and then use this lemma. Thus the behavior and execution stabilization definitions complement each other in this thesis: the former is typically used for *specification* and the latter is often used for *proofs*.

Lemma 3.2.9 *If automaton A stabilizes to the executions of automaton B in time t then automaton A stabilizes to the behaviors B in time t.*

Proof: Let β be any behavior of A. Let α be some execution of A corresponding to β. From the hypothesis, there is some α' that is a t-suffix of execution α and is also an execution of B. Consider the behavior β' of B corresponding to execution α' of B. From the definitions, we can verify that that β' is a t-suffix of behavior β. ∎

3.3 Discussion on the Stabilization Definitions

First, notice that we have defined what it means for an arbitrary IOA to stabilize to some target set or automaton. For most of the thesis we will be interested in proving stabilization properties only for a special kind of automata: unrestricted automata. An unrestricted IOA (see Section 3.5) is one in which all states of the automaton are also start states. Such an IOA models a system that has been placed in an arbitrary initial state by an arbitrary initial fault. However, (this important observation is due to Arora and Gouda [AG92]), we might also be interested in modelling the response of a system to more restricted kinds of initial faults. Such restricted faults initially place a system A in some subset L of the states of A. After the initial fault, we would like A to behave like some other automaton B. *Thus our general definitions of stabilization are applicable to other, more restricted forms of initial faults.* While we will not mention this explicitly from this point on, many of the techniques developed in this thesis can also be applied to more restricted initial fault models.

Next, it is reasonable to ask whether our definitions are sufficient to cover all cases that arise in practice? They do cover all the examples in this thesis. There are two places we can weaken our definitions. The first is that instead of requiring that *all* behaviors (or executions) of A stabilize in time t, we might only that certain "well-formed" behaviors (or executions) of A stabilize. A "well-formed" behavior (execution) is a behavior (execution) with certain restrictions on the input actions. For instance, if A can pass a token to the external environment E, we might only require stabilization for those behaviors (executions) in which E returns the token to A in bounded time.

The second possible modification (which is sometimes needed, though again not in this thesis) is to only require (in Definitions 3.2.2 and 3.1.2) that the t-suffix of

a behavior (execution) be a *suffix* of a behavior (execution) of the target set. One problem with this modified definition is that we know of no good proof technique to prove that the behaviors (executions) of an automaton are *suffixes* of a specified set of behaviors (executions).[1] By contrast, it is much easier to prove that every behavior of an automaton has a suffix that is in a specified set.

Thus we prefer to use the simpler definitions for this thesis.

3.4 Proof Technique

We begin by defining an extremely useful piece of notation that is used extensively in this thesis. This notation allows us to specify an IOA that is identical to another IOA except for start states. This is roughly the inverse of the $U(A)$ notation that creates an unrestricted version of automaton A without start states.

Definition 3.4.1 *For any automaton A and any subset L of states(A), we denote by $A|L$ the automaton that is identical to A except that start$(A|L) = L$.*

There has been a great deal of work in designing ordinary automata that have specific start states to solve specific problems. It would be nice to gain leverage from this existing body of work. Suppose we are given an IOA A that solves a set of behaviors P and we now wish to design an IOA B that stabilizes to the behaviors in P. Our goals are:

- We would like to use as much of the design of A as possible to design B.
- We would like to use as much of the proof that A solves P as possible to prove that B stabilizes to the behaviors in P.

We now describe one way in which these goals can be achieved. The following lemma is immediate from the definitions.

[1] Such proofs seem to involve arguments about reachable states. Familiar inductive proof techniques (such as invariant arguments, progress metrics etc.) do not seem to suffice for this purpose.

Lemma 3.4.2 *Consider an automaton A, $L \subseteq states(A)$, and a problem P. Suppose the following conditions are true:*

- *A stabilizes to the behaviors of $A|L$ in time t.*
- *Any behavior of $A|L$ is in P.*

Then A stabilizes to the behaviors in P in time t.

In the next two subsections, we describe the techniques used in this thesis for proving the two items on the list.

3.4.1 Proving that an Automaton Solves a Problem

To prove that any behavior of some automaton A is a behavior contained in some problem P, it suffices to prove that every behavior of A is a behavior of some other automaton B, and that every behavior of B is in P.

There are well-known techniques (e.g., [LT89]) to show any behavior of an automaton A is a behavior of another automaton B. A commonly used technique is a refinement mapping. The basic idea is to establish a suitable mapping f between a state of A and a state of B. Given an execution α of A, we use f to construct a mapped execution of B that has the same external behavior as α.

Theorem 3.4.3 Refinement Mapping: *Let A and B be automata with the same set of external actions. Let t_c be the upper bound on the time to perform actions in any class c of B. Let f be a mapping from the states of A to the states of B such that:*

1. *For any start state s of A, $f(s)$ is a start state of B.*
2. *For all transitions $(s, \pi, \tilde{s})$ of A, either*
   - *π is an action of B and $(f(s), \pi, f(\tilde{s}))$ is a transition of B OR*
   - *π is not an action of B and $f(s) = f(\tilde{s})$.*
3. *For any class c of B and any execution α of A, suppose some action in c is enabled in $f(s)$. Then within t_c time of s in α either:*

- *Some action in class c occurs OR*
- *Some state $\tilde{s}$ occurs such that no action in class c is enabled in $f(\tilde{s})$.*

Then every behavior of A is a behavior of B

Proof: Consider any execution α of A. We extend the function f to map executions as well as states in the following way. Let $f(\alpha)$ be the sequence formed from α by:

- Removing every timed action in α that is not a timed action of B and the timed state following such an action.
- Replacing every timed state (s,t) in the remaining sequence by $(f(s),t)$.

Since we retain every action of B, the behavior corresponding to $f(\alpha)$ is the behavior corresponding to α.

Next, we verify that $f(\alpha)$ is an execution of B. To do so we check the four conditions in Definition 2.2.1. Let $\alpha = (s_0,t_0),(a_1,t_1),(s_1,t_1)\ldots$. Clearly, by construction, $f(\alpha)$ is an alternating sequence of timed states and actions of B. Let $f(\alpha) = (s'_0,t'_0)(a'_1,t'_1)(s'_1,t'_1)\ldots$.

We check the four conditions in turn:

1. First $s'_0 = f(s_0) \in start(B)$ by hypothesis. Next consider (s'_i,a'_{i+1},s'_{i+1}) for all $i \geq 0$. Let a_k be the $i+1$'st action of B in α. Intuitively, a_k is the action in α that generated a'_{i+1} in $f(\alpha)$. Clearly, $a_k = a'_{i+1}$.

   Next, consider the smallest $j < k$ such that all actions between s_j and s_{k-1} in α are not actions of B. Then by construction, $f(s_j) = s'_i$ and $f(s_k) = s'_{i+1}$. Also by the hypothesis, $f(s_j) = f(s_{k-1})$ and $(f(s_{k-1}),a_k,f(s_k))$ is a transition of B. Thus (s'_i,a'_{i+1},s'_{i+1}) is a transition of B.

2. The second condition follows trivially from the construction.

3. The third condition follows because any subsequence of a time sequence is a time sequence.

4. Suppose some action in some class c of B is enabled in some state s'_i of $f(\alpha)$. Then, by construction there is a corresponding state s_j in α such that if γ is the suffix of α starting with s_j, then $f(\gamma)$ is the suffix of $f(\alpha)$ starting with s'_i. Then, by hypothesis, either:

   - Some action in class c of B occurs within t_c time of s_j in α. But by construction, the same action occurs within t_c time of s'_i in $f(\alpha)$.
   - Some state s_k occurs within t_c time of s_j in α and such that all actions in class c are disabled in $f(s_k)$. Let k be the smallest index such that this is true for s_k. Now if a_k is not an action of B, then since $f(s_{k-1}) = f(s_k)$ this contradicts the assumption that k is the smallest index with this property. Thus a_k is an action of B and hence, by construction, $(f(s_k), s_k.time)$ occurs in $f(\gamma)$. Thus there is a state that occurs within t_c time of s'_i in $f(\alpha)$ and such that no action of c is enabled in this state.

∎

3.4.2 Proving that an Automaton Stabilizes to another Automaton

We give one such technique in the following definition and theorem. The technique is similar to techniques used for proving liveness properties (e.g., [OL82, MP91]) of concurrent programs. Our theorem is a small generalization of a theorem for proving stabilization properties that was previously proposed in [GM90].

Let L be a closed predicate of automaton A – once L is true in an any execution of A, L remains true for the rest of the execution. We would like to prove that in any execution of A, L becomes (and stays) true in bounded time. This implies that A stabilizes to the executions of $A|L$ in bounded time. We will describe a proof rule for this purpose. Intuitively, instead of proving directly that the goal L eventually holds we prove that a number of subgoals L_i (each of which is a predicate of A) become and stay true. The L_i are chosen so that L is true if all the individual L_i are true.

Each subgoal L_i can be intuitively thought of as "depending" on some other set of subgoals. This dependence relation is formalized by a partial order $<$. If $j < i$, then (intuitively) L_i "depends" on L_j. Suppose we could show that if all the subgoals that

L_i depend on become and stay true, then L_i becomes and stays true. Then we can concude that eventually all subgoals (and hence L) become and stay true. Intuitively, this follows because the dependency relation is *acyclic* since it is formalized using a partial order.

We can extend this idea to a timed setting to show that L will become true in time proportional to the maximum length of a "dependency chain". A dependency chain is formalized using the standard concept of a *chain* in a partially ordered set. Consider a set $\{L_i, i \in I\}$ where $(I, <)$ is a finite partially ordered set. A chain is simply a sequence $L_1 < L_2 < \ldots < L_i$.

The preceding discussion motivates the following definition and theorem.

Definition 3.4.4 *We say that an automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t if:*

1. *$\mathcal{L} = \{L_i, i \in I\}$ of sets of states of A, where $(I, <)$ is a finite, partially ordered index set. We let height($\mathcal{L}$) denote the maximum length chain in the partial order.*
2. *$\cap_{i \in I} L_i \subseteq L$.*
3. *For all $i \in I$ and for all steps $(s, \pi, \tilde{s})$ of A, if s belongs to $\cap_{j \leq i} L_j$, then $\tilde{s}$ belongs to L_i.*
4. *For every $i \in I$ and every execution α of A and every state s in α the following is true. Suppose that either $s \in \cap_{j<i} L_j$ or there is no $L_j < L_i$. Then there is some state $\tilde{s} \in L_i$ that occurs within time t of s in α.*

The first condition says there is a partial order on the predicates in $\mathcal{L}$. The second says that L becomes "true", when all the predicates in $\mathcal{L}$ become true. The third is a stability condition. It says that any transition of A leaves a predicate L_i true, if all the predicates less than or equal to L_i are true in the previous state. Finally the last item is a liveness condition. It says that if all all the predicates *strictly* less than L_i are true in a state, then within time t after this state, L_i will become true.

We define *height*(L_i), the height of a predicate $L_i \in \mathcal{L}$, to be the maximum length of a chain that ends with L_i in the partial order. The value of *height*($\mathcal{L}$) is, of course,

the maximum height of any predicate $L_i \in \mathcal{L}$. By the liveness condition, within time t all predicates with height 1 become true; these predicates stay true for the rest of the execution because of the third stability condition. In general, we can prove by induction that within time $i \cdot t$ all predicates with height i become and stay true. This leads to a simple but useful theorem:

Theorem 3.4.5 Execution Convergence: *Suppose that automaton A is stabilized to predicate L using predicate set $\mathcal{L}$ and time constant t. Then, A stabilizes to the executions of $A|L$ in time $height(\mathcal{L}) \cdot t$.*

Proof: By induction on $h, 0 \leq h \leq height(\mathcal{L})$, in the following inductive hypothesis.

Inductive Hypothesis: There is some state s that occurs within time $h \cdot t$ of the start of α such that $s \in L_i$ for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h$.

The inductive hypothesis implies that there is some state s that occurs within time $h \cdot height(\mathcal{L})$ of the start of α and such that $s \in L$. The theorem follows from this fact taken together with Lemma 2.2.3.

Base Case, $h = 0$: Follows trivially since there is no $L_i \in \mathcal{L}$ with $height(L_i) = 0$.

Inductive Step, $0 \leq h \leq height(\mathcal{L}) - 1$: Assume it is true for h. Then there is some state s_i that occurs within time $h \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h$, $s_i \in L_i$. Consider any $L_j \in \mathcal{L}$ with $height(L_j) = h + 1$. By the fourth condition in Definition 3.4.4, we know there must some state $s_{f(j)} \in L_j$ that occurs within time t of s_i in α. Let $k = Max\{f(j) : height(L_j) = h + 1\}$. Then from the third condition in Definition 3.4.4, we see that s_k occurs within time $(h + 1) \cdot t$ of the start of α and such that for all $L_i \in \mathcal{L}$ with $height(L_i) \leq h + 1$, $s_i \in L_i$. ∎

3.5 Modularity Theorem

We will mostly deal with stabilization properties of a special class of automata called unrestricted automata or UIOA. Intuitively, a UIOA models a system that can start in an arbitrary state.

Definition 3.5.1 *A UIOA A is an automaton such that $start(A) = states(A)$ (i.e., all states are start states).*

However, we will often show that a UIOA A stabilizes to the behaviors of a second special kind of automaton called a Closed I/O Automaton or a CIOA . We define the *reachable states* of an automaton A to be the states that can occur in executions of A.

Definition 3.5.2 *A CIOA is an automaton such that every reachable state is also a start state.*

It is easy to see that:

Lemma 3.5.3 *Every UIOA is a CIOA .*

The two following lemmas are extremely convenient and are used often below without explicit reference. It is the reason we allow executions and behaviors to start with arbitrary values of time. Also, the next two lemmas depend crucially on the fact that there are no lower bounds on the time between actions.

Lemma 3.5.4 *Consider any execution α of a CIOA A. Any suffix of α that starts with a timed state is also an execution of A.*

Proof: Follows directly from Lemma 2.2.3 and the definition of a CIOA. ∎

Suppose we begin to view an automaton after it has "run for a while" and the resulting behavior is indistinguishable from an ordinary behavior of the automaton. Then, intuitively, we say that the automaton is *suffix-closed.* More formally:

Definition 3.5.5 *We say that an automaton A is suffix-closed if for every behavior β of A and every time $t \geq 0$, every t-suffix of behavior β is a behavior of A.*

A remarkable number of interesting automata we will study in this thesis are suffix-closed. This fact is explained by the following lemma:

Lemma 3.5.6 *Any CIOA A is suffix-closed.*

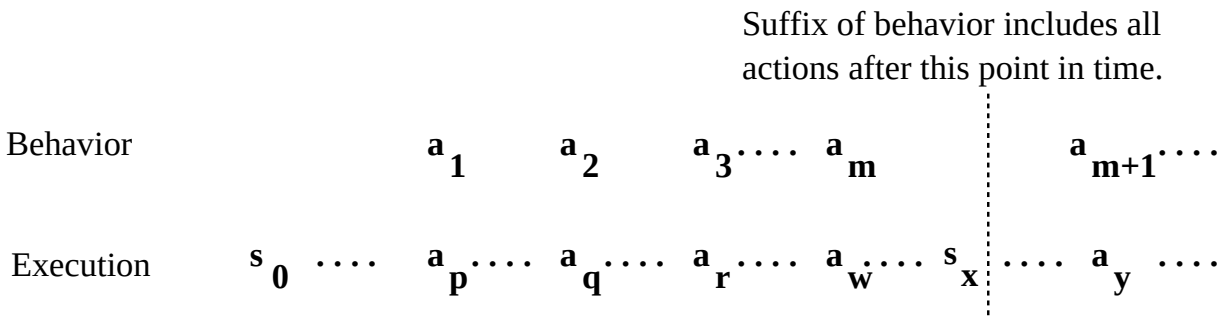


Figure 3.2: Obtaining the suffix of an execution corresponding to a t-suffix of the behavior of the execution.

Proof: We will only sketch the main idea of the proof. Consider any behavior β of A and any $t \geq 0$. Let β' be any t-suffix of behavior β. Consider any execution α of A such that the behavior of α is β. The proof consists of using α to construct another execution α' of A such that the behavior of α' is b'; α' is essentially a suffix of α whose start time is adjusted to match the start time of β'.

The behavior β and corresponding execution α are sketched in Figure 3.2. By definition, for every action in β there is a corresponding external action in α which occurs at the same time. This is sketched by drawing the action in the behavior directly above the corresponding action in the execution. (However, since the execution will, in general, have internal actions not included in the behavior, the indices of the actions will not necessarily match. Thus in the figure a_1 in β corresponds to a_p in α.)

The t-suffix β' can be sketched using a line that contains all actions in β occuring to the right of the line (see Figure 3.2). The line is drawn between two actions in β because the start time of β' may occur in between the times of two actions in β.

We need a suffix α' of α whose behavior is equal to β'. Thus we look for a state s_x in α corresponding to the vertical time line drawn in Figure 3.2. But we may not have a state in α whose time is equal to the start time of β'. So (intuitively) we choose s_x to be the first state that occurs to the "left" of the vertical time line. Then we choose α' to be the suffix of α starting with s_x and with the time of s_x adjusted to be equal to $\beta'.start$. This works for two reasons. First, by definition of a CIOA, s_x is a start state of A. Second, we have no lower bounds on the time between actions in A. Thus increasing the time of the initial state of an execution (and such that the resulting time is no greater than the time of the first action) still leaves us with a legal

execution. ∎

The suffix-closed property is not just a interesting curiousity. It also provides the basis for the following important Modularity Theorem that we discuss next.

Our Modularity Theorem about the stabilization of composed automata may seem "obvious". We would expect that if each piece A_i of a composed system stabilizes to the behaviors of say B_i, then the composition of the A_i should stabilize to the composition of the B_i. Sadly, this is not quite true. There is a counterexample described in Section 3.5.1 which shows that if we allow some of the B_i to be arbitrary automata, then this statement is false.

The main problem is that for a given behavior of the system A, the component automata A_i may stabilize at different times. But if each of the A_i begin to "look like" the corresponding B_i at different times, then it may not be possible to paste the resulting behavior into a behavior that "looks like" a behavior of B. However, this problem does not arise if each of the B_i is suffix-closed. Thus we have the following result.

Theorem 3.5.7 Modularity: *Let I be a finite index set. Let $A_I = \{A_i, i \in I\}$ be a compatible set of automata and $B_I = \{B_i, i \in I\}$ be a second set of compatible, suffix-closed automata. Suppose also that for all $i \in I$, A_i stabilizes to the behaviors of B_i in time t. Let $A = \Pi_{i \in I} A_I$ and $B = \Pi_{i \in I} B_I$. Then A stabilizes to the behaviors of B in time t.*

Proof: The proof relies on the Cut Lemma (Lemma 2.2.4) that allows us to dissect a behavior of a system into its component behaviors, and the Paste Lemma (Lemma 2.2.5) that allows us to paste component behaviors into a system behavior.

Consider any behavior β of A. Consider the β' that is a t-suffix of behavior β and such that:

- $\beta'.start = t + \beta.start$.
- Any actions in β' occur at times strictly greater than t.

We can verify that such a β' exists from the definition of a t-suffix of a behavior. Intuitively, β' is chosen so that all component behaviors are guaranteed to have stabilized in β'.

Now consider any $i \in I$. By the Cut Lemma (Lemma 2.2.4), $\beta|A_i$ is a behavior of A_i. But because A_i stabilizes to the behaviors of B_i in time t, there must be some $t' \leq t$ and some β_i such that:

- β_i is a t'-suffix of $\beta|A_i$ and is also a behavior of B_i.
- $\beta_i.start = t' + \beta.start$.

Next consider $\beta'|A_i$. It can be verified that $\beta'|A_i$ is a t''-suffix of β_i for some t''. Thus by the fact that B_i is suffix-closed, $\beta'|A_i$ is a behavior of B_i. Thus $\beta'|A_i$ is a behavior of B_i for all i. Hence by the Paste Lemma (Lemma 2.2.5), β' is a behavior of B. The theorem follows since β' is a t-suffix of behavior β. ∎

3.5.1 Discussion of the Modularity Theorem

In the hypothesis of the modularity theorem, we assumed that each of the B_i was suffix-closed. We show a counterexample to show that if the B_i are allowed to be arbitrary automata, then the theorem is false. Consider the specification of automaton B_i shown in Figure 3.3. Let A_i be a UIOA which is identical to B_i except that the start states of A_i are unrestricted (i.e., the initial value of $count_i$ in A can be any value in the range $\{0, \ldots, 2\}$).

It is easy to see that A_i stabilizes to the behaviors of B_i in time $3t$ because within that time the value of $count_i$ must reach 0. After such a state, any behavior of A_i is a behavior of B_i. Now consider an index set $I = \{1, 2\}$. Consider A which is the composition of A_1 and A_2 and B which is the composition of B_1 and B_2. We claim that A does not stabilize to B in time $3t$ (or in fact in any finite time).

To see this, we start with the following observation. In any behavior of B in which the actions of B_1 and B_2 strictly alternate, the counter values output in such a behavior will be of the form $0, 0, 1, 1, 2, 2, 0, 0, \ldots$. Now consider the behavior corresponding to an execution of A in which $count_1 = 0$ initially and $count_2 = 2$ initially and the actions of A_1 and A_2 strictly alternate starting with A_1. Then the counter values output in such a execution will be of the form $0, 2, 1, 0, 2, 1, 0, 2 \ldots$. From the earlier observation, it follows that there is no suffix of this behavior of A that is a behavior of B.

The state of B_i consists of an integer variable $count_i \in \{0, \ldots, 2\}$

The initial value of $count_i = 0$ (* i.e., B is not a UIOA *)

INCREMENT$_i(k)$ (*output action, outputs counter value using parameter k*)
  Precondition: $k = count_i$
  Effect: $count_i := (count_i + 1) \bmod 3$

Any INCREMENT$_i$ action is in a separate class with upper bound t.

Figure 3.3: Specification for Automaton B_i

Suppose we weakened the definition of stabilization to allow any t-suffix of A to be a suffix of a behavior of B. With this weaker definition, the counterexample disappears.[2] Thus the suffix-closed requirement may be a consequence of our (stronger) stabilization definition. However, we did not find a way to prove the modularity theorem using the weaker definition. Without the suffix-closed requirement it is difficult to "paste" together behavior suffixes of the B_i's to create a behavior of B. A possible research direction would be to look for weaker conditions (than the B_i being suffix-closed) under which the modularity theorem would work. Another research direction would be to extend these results to systems with non-zero lower bounds on the time between actions.

3.6 Summary

The two main contributions of this chapter are the definitions of stabilization in terms of external behaviors (Definitions 3.2.2 and 3.2.3) and the modularity theorem (Theorem 3.5.7).

The definitions of stabilization in terms of external behaviors are different from

[2] I am grateful to Robert Gallager and Victor Luchanko of MIT for pointing this out.

previous definitions that are in terms of the states and executions of the underlying automaton. The external behavior definition allow us to define that automaton A stabilizes to another automaton B even though A and B have different state sets. This is most useful when A is a low level model (e.g., an implementation) and B is a high level model (e.g., a specification).

The modularity theorem gives us a formal basis for a building-block approach: it allows us to prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts. The requirement that each of the target automata be suffix-closed may seem restrictive. However, in a stabilizing setting this is not the case. Most interesting specifications (for problems such as message delivery, routing, and scheduling) are either suffix-closed or can be rephrased so they are suffix-closed.

We have already seen that any UIOA (an automaton for whom every state is a start state) is suffix-closed. Similarly, any CIOA (an automaton for whom every reachable state is a start state) is suffix-closed. In a stabilizing setting the basic building blocks are UIOAs since they model systems that can start in an arbitrary state. The methods developed in this thesis, on the other hand, tend to construct automata that are CIOAs.[3] Thus we can build stabilizing solutions modularly in several stages. In the first stage we compose s set of UIOAs to yield a CIOA. In the next stage, the resulting CIOAs are composed with other UIOAs to yield more CIOAs. This process can be repeated indefinitely to build a complex stabilizing solution to a problem. Since all the pieces used are suffix-closed, the modularity theorem can be used at each stage. As an example, the stabilizing spanning tree protocol of Chapter 8 is constructed using the stabilizing reset protocol of Chapter 7 which in turn can be constructed using a stabilizing Data Link implementation. Thus, despite its restrictions, the modularity theorem is extremely useful in this thesis.

We also have a definition of stabilization in terms of executions that corresponds to the standard definition (Definition 3.1.2). This definition in terms of executions is used in this thesis for two reasons. First, it is sometimes useful in its own right when the alternative would entail adding many superfluous actions.[4] Second, the definition

[3]Our methods construct automata that stabilize to a specification automaton of the form $A|L$. If L is a closed predicate of A – i,e., no transition of A can falsify L – then $A|L$ is a CIOA.

[4]For example, the correctness of a spanning tree protocol is easily defined in terms of a *parent* variable at each node. For correctness, we could specify that the graph induced by the *parent* variables be a spanning tree of the network. An external behavior specification would require additional output

in terms of executions is *essential for proving* results about stabilization of behaviors because our main tool for proving such results is Lemma 3.2.9.

The definitions we use give us several nice properties that we believe any definition of stabilization should provide. The properties we believe to be important are: transitivity for both behavior and execution stabilization (Lemma 3.2.6 and Lemma 3.1.6), the fact that execution stabilization implies behavior stabilization (Lemma 3.2.9), the fact that stabilization in time t implies stabilization in time greater than t (Lemma 3.2.7), and the Modularity theorem (Theorem 3.5.7).

The index contains pointers to the definitions given in the last two chapters. The appendix also contains a list of commonly used notation for easy reference.

actions to report the value of the *parent* variables at each node.

Chapter 4

Local Checking and Correction in a Shared Memory Model

In Dijkstra's [Dij74] model, a network protocol is modelled using a graph of finite state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system such atomic communication is impossible. Typically communication has to proceed through channels. Such channels must be be modelled explicitly as state machines that can store messages sent from one node to another. Also, in message passing models, the channel state machine is essentially fixed (with actions to send and deliver packets) but the node state machines can be arbitrarily specified by the protocol designer. However, in Dijkstra's model *all* state machines are node state machines and can be arbitrarily specified by the protocol designer.

While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dij74] the concept of stabilization without undue complexity. For this chapter only, we will use Dijkstra's original model to *introduce* the method of local checking and correction. In later chapters, we will use a more realistic message passing model. Thus the goals of this chapter are:

- To describe some simple examples of local checking and correction that are more interesting than than the trivial examples given in Chapter 1.

- To show that existing work in [Dij74] and [AG90] can be understood very succinctly using the framework of local checking and correction.

The main result of the chapter is a theorem (Theorem 4.3.1) that states that any locally checkable protocol on a tree can be efficiently stabilized. To motivate this theorem, we begin in Section 4.2 with a reset protocol [AG90] due to Arora and Gouda. We examine the behavior of the Arora-Gouda protocol in a good state and conclude that the protocol is in a good state when all link subsystems are in a good state. Then we show how to add correction actions to the protocol such that if a link subsystem is in a bad state, it can be corrected to a good state. We also determine an order in which link subsystems can be corrected so as to ensure that the correction process converges.

In Section 4.3 we generalize the procedure followed in Section 4.2 to obtain Theorem 4.3.1. Then in Section 4.4 we show how one of Dijkstra's protocols [Dij74] can easily be understood using Theorem 4.3.1.

4.1 Modelling Shared Memory Protocols

We will use the version of the timed I/O Automaton model [MMT91] described in Chapter 2. How can we map Dijkstra's model into this model? Suppose each node in Dijkstra's model is a separate automaton. Then in the Input/Output automata model, it is not possible to model the simultaneous reading of the state of neighboring nodes. The solution we use is to dispense with modularity and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modelled using a "demon", is naturally a part of our model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

4.2 A Reset Protocol on a Tree

Before describing the reset protocol due to Arora and Gouda [AG90], we first describe the network reset problem.

Recall that we have a collection of nodes that communicate by reading the state of their neighbors. The interconnection topology is described by an arbitrary graph. Assume that we are given some application protocol that is being executed by the nodes. We wish to superimpose a reset protocol over this application such that when the reset protocol is executed the application protocol is "reset" to some "legal" global state. A "legal" global state is allowed to be any global state that is reachable by the application protocol after correct initialization. The problem is called distributed reset because reset requests may arrive at any node.

A simple and elegant network reset protocol is due to Finn [Fin79]. In this protocol each node i running the application protocol has a session number. When the reset protocol is not running, the session numbers at every node are the same. When a node receives a reset request, it resets the local state of the application (to some prespecified initial state) and increments its session number by 1. When a node sees that a neighbor has a higher session number, it changes its session number to the higher number and resets the application. Finally, the application protocol is modified so that a node cannot make a move until its session number is the same as that of its neighbors. This check prevents older instances of the application protocol from "communicating" with newer instances of the protocol. This protocol is shown to be correct [Fin79] if all the session numbers are initially zero and the session numbers are allowed to grow without bound.

We rule out the use of unbounded session numbers as unrealistic. Also, in a stabilizing setting, having a "large enough" size for a session number does not work. This is because the reset protocol can be initialized with all session numbers at their maximum value. Thus, we are motivated to search for a reset protocol that uses bounded session numbers. Suppose we could design a a reset protocol with unbounded numbers in which *the difference between the session numbers at any two nodes is at most one in any state.* Suppose also that for any pair of neighboring nodes u and v that compare session numbers, the session number of one of the nodes (say u) is always no less than the session number of the other node. Then, since the session numbers are only used for comparisons, it suffices to replace the session numbers by a single bit that we call $sbit_i$. This is the first idea in Arora and Gouda's reset protocol [AG90].

To realize this idea, we cannot allow a node to increment its session number as soon as it gets a reset request. Otherwise, multiple reset requests at the same node will cause the difference in session numbers to grow without bound. Thus nodes must

coordinate before they increment session numbers.

In Arora and Gouda's reset protocol [AG90], the coordination is done over a rooted tree. Arora and Gouda first show how to build a rooted tree in a stabilizing fashion. In what follows we will assume that the tree has already been built. Thus every node i has a pointer called $parent(i)$ that points to its parent in the tree and the parent of the root is a special value *nil*.

Given a tree, an immediate idea is to funnel all reset requests to the root. On receipt of a request, the root could send reset grants down the tree. Nodes could increment their session number on receiving a grant. Unfortunately, this does not work either because a node A in the tree may send a reset request and receive a grant before some other node B in the tree receives a grant. After getting its first grant, A may send another request and receive a second grant before B gets its first grant. Assuming that the session numbers are unbounded, the difference in the session numbers of A and B can grow without bound.

Instead, the reset task is broken into three phases. In the first phase, a node sends a reset request up the tree towards the root. In the second phase, the root sends a reset wave down the tree. In the third phase, the root waits until the reset wave has reached every node in the tree before starting a new reset phase. This ensures that after the system stabilizes,[1] the use of three phases will guarantee that a single bit $sbit_i$ is sufficient to distinguish instances of the application protocol.

The three phases are implemented by a mode variable $mode_i$ at each node i. The mode at node i has one of three possible values: *init*, *reset*, and *normal*. All nodes are in the *normal* mode when no reset is in progress. To initiate a reset, a node i sets $mode_i$ to *init* (this can be done only if both i and its parent are in *normal* mode). A reset request is propagated upwards by the action PROPAGATE_REQUEST$_i$ which sets the mode of the parent to *init* when the mode of the child is *init*. A reset wave is begun by the root by the action START_RESET which sets the mode of the root to *reset*. The reset wave propagates downwards by PROPAGATE_RESET$_i$ which sets the mode of a child to *reset* if the mode of the parent is *reset*. When a node changes its mode to *reset*, it flips its session number bit, and resets the application protocol. Finally, the completion wave is propagated by the action PROPAGATE_COMPLETION$_i$ which sets a node's mode to *normal* when *all* the node's children have *normal* mode.

[1] in this chapter, we will always use the execution stabilization definitions

The automaton code for this implementation is shown in Figure 4.1 and Figure 4.2. Notice that besides the actions we have already described, there is a CORRECT$_i$ action in Figure 4.2. This action was used in an earlier version [AG90] to ensure that the reset protocol was stabilizing.

Informally, the reset protocol is stabilizing if after bounded time, any reset requests will cause the application protocol to be properly reset. The correction action in Figure 4.2 [AG90] ensures stabilization in a very ingenious way. However, the proof of stabilization is somewhat difficult and not as intuitive as one might like. The reader is referred to [AG90] for details. Instead, we will use local checking and correction to describe *another* correction procedure that is very intuitive. As a result, the proof of stabilization becomes transparent.

We start by writing down the "good" states of the reset system in terms of link predicates $L_{i,j}$. We say that the system is in a good state if for all neighboring nodes i and j, the predicate $L_{i,j}$ holds, where $L_{i,j}$ is the conjunction of the two predicates:

- If $(parent(i) = j)$ and $(mode_j \neq reset)$ then $(mode_i \neq reset)$ and $(sbit_i = sbit_j)$
- If $(parent(i) = j)$ and $(mode_j = reset)$ then either:
  - $(mode_i \neq reset)$ and $(sbit_i \neq sbit_j)$ OR
  - $(sbit_i = sbit_j)$

The predicates can be understood intuitively as describing states that occur when the reset system is working correctly. The first predicate says that if the parent's mode is not *reset*, then the child's mode is not *reset* and the two session bits are the same. This is true when the system is working correctly because of two reasons. First, the child enters *reset* mode only when its parent is in that mode, and the parent does not leave *reset* mode until the child has left *reset* mode. Second, if the parent changes its session bit, the parent also goes into *reset* mode; and the child only changes its session bit when the parent's mode is *reset*.

The second predicate describes the correct states during the second and third phases of the reset until the instant that the completion wave reaches j. It says that if the parent's mode is *reset*, then there are two possibilities. If the child has not "noticed" that the parent's state is *reset*, then the child's bit is not equal to the

The state of the system consists of two variables for every process in the tree:
$mode_i \in \{init, normal, reset\}$
$sbit_i$, a bit

PROPAGATE_REQUEST$_i$ (*internal action to propagate a reset request upwards *)
Preconditions:
$mode_i = normal$
$i = parent(j)$ and $mode_j = init$
Effects: $mode_i := init$

START_RESET$_i$ (*internal action at root to start a reset wave*)
Preconditions: $mode_i = init$ and $parent(i) = nil$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_RESET$_i$ (*internal action to propagate reset downwards*)
Preconditions:
$mode_i \neq reset$
$j = parent(i)$ and $mode_j = reset$ and $sbit_j \neq sbit_i$
Effects:
$mode_i := reset$; (*also reset application state at this node*)
$sbit_i :=\sim sbit_i$; (*flip bit*)

PROPAGATE_COMPLETION$_i$ (*internal action to propagate completion wave upwards *)
Preconditions:
$mode_i = reset$
For all children j of i: $mode_j = normal$ and $sbit_i = sbit_j$
Effects: $mode_i := normal$

Every action is in a separate class with upper bound equal to
t_n

Figure 4.1: Normal Actions at node i in Arora and Gouda's Reset Protocol.[AG90]

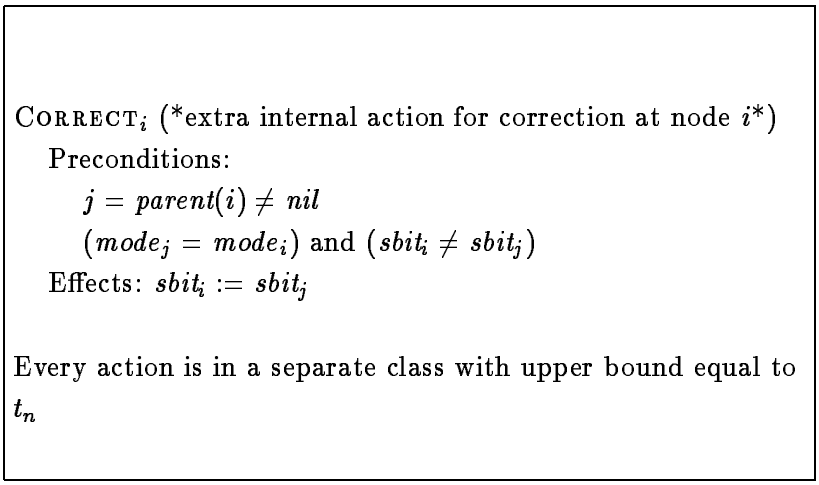


Figure 4.2: Original correction action in Arora and Gouda's Reset Protocol [AG90].

parent's bit. (This follows because when the parent changes its mode to *reset*, the parent also changes its bit; and just before such an action the second predicate assures us that the two bits are the same.) On the other hand, if the child has noticed that the parent's state is *reset*, then the two bits are the same. (This follows because when the child notices that the parent's mode is *reset*, the child sets its bit equal to the parent's bit and does not change its bit until the parent changes its mode.)

Suppose that in some state s these link predicates hold for all links in the tree. Then [AG90] show that the system will execute reset requests correctly in any state starting with s. This is not very hard to believe. But it means that all we have to do is to add correction actions so that all link predicates will become true in bounded time.

The tree topology once again suggests a simple strategy. We remove the old action CORRECT$_i$ in Figure 4.2 and add a new action CORRECT_CHILD$_i$ as shown in Figure 4.3. Basically, CORRECT_CHILD$_i$ checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that $L_{i,j}$ becomes true. Notice that CORRECT_CHILD$_i$ leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD$_i$ will leave $L_{j,k}$ true if $L_{j,k}$ was true in the previous state.

Thus we have an important stability property: correcting a link does not affect the

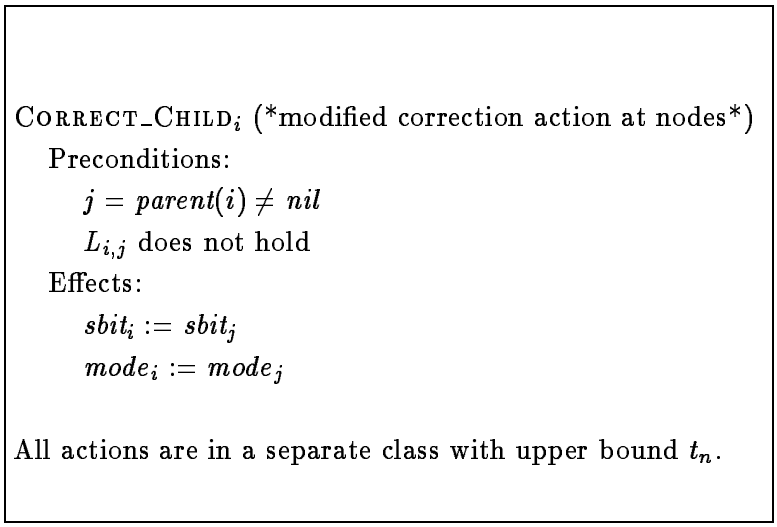


Figure 4.3: Modified Correction action for Arora and Gouda's Reset Protocol. All other actions are as in Figure 4.1.

correctness of links above it in the tree. Using this we can show that in bounded time, all links will be in a good state and so the system is in a good state. Rather than prove that this modified automaton stabilizes, we will prove a more general result in the next section: that *any* locally checkable tree automaton can be locally corrected into a good global state.

We will return to the network reset problem in Chapter 7. Our stabilizing reset protocol is more efficient than the reset protocol of [AG90] and is also designed to work in a message passing model.

4.3 Tree Correction for Shared Memory Systems

In the last section, we described informally the problem of stabilizing a reset protocol on a tree. We also suggested a technique of adding correction actions to every node. That example motivates us to ask whether there is a general result for trees. To describe and prove such a general result, we start with the following definitions.

We will continue to model a network as a single automaton in which a node can

read and write the state of its neighbors in a single move using an internal action. Formally:

A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.
- All actions of $\mathcal{N}$ are internal actions and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$
- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.
- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all $v \neq u$.

Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

A shared memory *tree automaton* is a shared memory network automaton where G is a rooted tree. Thus for any node i in a tree automaton, we assume there is a value *parent*(i) that points to the parent of node i in the tree. There is also a unique root node r that has *parent*$(r) =$ *nil*. For our purposes, it is convenient to model the *parent* values as being part of the code at each node. More generally, the parent pointers could be variables that are set by a stabilizing spanning tree protocol as shown in [AG90].

In this chapter, we will often use the phrase "tree automaton" to mean a "shared memory tree automaton" and the phrase "network automaton" [2] to mean a "shared memory network automaton".

[2] In all subsequent chapters, the terms tree and network automaton have different meanings.

A *closed predicate* [3] of an automaton A is a predicate L such that for any transition $(s, \pi, \tilde{s})$ of A, if $s \in L$ then $\tilde{s} \in L$.

A *link subsystem* of a tree automaton is an ordered pair (u, v), such that u and v are neighbors in the tree. To distinguish states of the entire automaton from the states of its subsystems we will sometimes use the word *global state* to denote a state of the entire automaton. For any global state s of a network automaton, we define $(s|u, s|v)$ to be the state of the (u, v) link subsystem. Thus the state of the (v, u) link subsystem in global state s is $(s|v, s|u)$.

A *local predicate* $L_{u,v}$ of a tree automaton is a subset of the states of a (u, v) link subsystem.

A link predicate set $\mathcal{L}$ for a tree automaton is a set that contains exactly one predicate for every link subsystem in the tree and which satisfies the following symmetry condition: for each pair of neighbors u and v, if $(a, b) \in L_{u,v}$, then $(b, a) \in L_{v,u}$. (i.e., while a link predicate set has two link predicates for each pair of neighbors, these two predicates are identical except for the order in which the states are written down.) We will also assume that every link predicate set is *non-trivial* in that there is at least one global state s such that $(s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.

A tree automaton is *locally checkable* for predicate L if there is some link predicate set $\mathcal{L} = \{L_{u,v}\}$ such that:

$L \supseteq \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u, v) in the tree.$\}$

In other words, the global state of the automaton satisfies L if every link subsystem (u, v) satisfies $L_{u,v}$.

Recall the definition of stabilization in terms of executions and the definition of an unrestricted automaton from Chapter 3. Recall that we use $A|L$ to denote the automaton that is identical to A except that the start states of $A|L$ are the states in set L. For any rooted tree T, we let *height*(T) denote the maximum length of a path between the root and a leaf in T.

We can now state a simple theorem.

[3] This is often called a stable predicate. We avoid this phrase because of potential confusion with stabilization.

Theorem 4.3.1 Tree Correction in Shared Memory Systems: *Consider any tree automaton $\mathcal{T}$ for tree T that is locally checkable for predicate L. Then there exists an unrestricted tree automaton $\mathcal{T}^+$ for T such that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time proportional to height(T).*

Thus after a time proportional to the height of the tree, any execution of the new automaton $\mathcal{T}^+$ will "look like" an execution of $\mathcal{T}$ that starts with a state in which L holds. To prove this theorem we first describe how to construct $\mathcal{T}^+$ from $\mathcal{T}$ and then show that $\mathcal{T}^+$ satisfies the requirements of the theorem.

Assume that $\mathcal{T}$ is *locally checkable* for predicate L using link predicate set $\mathcal{L} = \{L_{u,v}\}$. We start by defining the set of global states that satisfy all local predicates. Let $L' = \{s : (s|u, s|v) \in L_{u,v}$ for all link subsystems (u,v) in the tree.$\}$. Clearly $L \supseteq L'$. Also because of the non-triviality of the link predicate set, L' is not the empty set. To construct $\mathcal{T}^+$ from $\mathcal{T}$ we do the following:

- We first *normalize* all node states in $\mathcal{T}$. Intuitively, we remove all states in the state set of a node u that are not part of a global state that satisfies L'. Thus $S_u(\mathcal{T}^+) := \{s|u : s \in L'\}$. The state set of $\mathcal{T}^+$ is just the cross-product of the normalized state sets of all nodes. Intuitively, this rules out useless node states that never occur in global states that satisfy all local predicates.

- We retain all the actions of $\mathcal{T}$ but we add an extra precondition (i.e., an extra guard) to each action $a_u \in A_u$ of $\mathcal{T}$ as shown in Figure 4.4. Intuitively, this extra guard ensures that a normal action of $\mathcal{T}$ is not taken at node u unless all links adjacent to u are in "good states". All actions of $\mathcal{T}$ remain in the same classes in $\mathcal{T}^+$.

- We add an extra correction action CORRECT$_u$ for every node u in the tree that is not the root. CORRECT$_u$ is also described in Figure 4.4. Intuitively, this extra action "corrects" the link between node u and its parent if this link is not in a "good" state. Each CORRECT$_u$ action is put in a separate class with upper bound equal to t_n.

We outline a proof of the theorem by a series of lemmas. The first thing a careful reader needs to be convinced about is that the code in Figure 4.4 is realizable.

The state of $\mathcal{T}^+$ is identical to $\mathcal{T}$ except that the state set of each node is normalized to $\{s|u : s \in L'\}$

MODIFIED ACTION $a_u, a_u \in A_u$ (*modification of action a_u in $\mathcal{T}$*)
  Preconditions:
    Exactly as in a_u except for the additional condition:
    For all neighbors v of u: $(s|u, s|v) \in L_{u,v}$
  Effects:
    Exactly as in a_u

CORRECT$_u$ (*extra correction action for all nodes except the root*)
  Preconditions: $(parent(u) = v)$ and $((s|u, s|v) \notin L_{u,v})$
  Effects:
    Let a be any state in $S_u(\mathcal{T}^+)$ such that $(a, s|v) \in L_{u,v}$
    Change the state of node u to a

Each CORRECT$_u$ action is in a separate class with upper bound t_n

Figure 4.4: Augmenting $\mathcal{T}$ to create $\mathcal{T}^+$

The careful reader will have noticed that we made two assumptions. First, in the CORRECT$_u$ action, we assumed that for any link subsystem (u,v) of $\mathcal{T}^+$ and any state b of node v there is some a such that $(a,b) \in L_{u,v}$. Second, we assumed that when a modified action a_u is taken at node u, the resulting state of node u has not been removed as part of the normalization step.

We will begin with a lemma showing that the first assumption is a safe one. We show that the second assumption is safe later.

Lemma 4.3.2 *For any link subsystem (u,v) of $\mathcal{T}^+$ and for any state a of node u there is some b such that $(a,b) \in L_{u,v}$.*

Proof: We know that for any state a of v there is some state $s \in L'$ such that $s|u = a$. This follows because all node states have been normalized and because L' is not empty. Then we choose $b = s|v$. ∎

The next lemma shows a *local extensibility* property. It says that if any node u and its neighbors have node states such that the links between u and its neighbors are in good states, then this set of node states can be extended to form a good global state.

Lemma 4.3.3 *Consider a node u and some global state s of $\mathcal{T}^+$ such that for all neighbors v of u, $(s|u, s|v) \in L_{u,v}$. Then there is some global state $s' \in L'$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u.*

Proof: We create a global state s' by assigning node states to each node in the tree such that for every link subsystem (u,v), the state of the subsystem is in $L_{u,v}$. Start by assigning node state $s|u$ to u and $s|v$ to all neighbors v of u. At every stage of the iteration we will label a node x that has not been assigned a state and is a neighbor of a node y that has been assigned a state. But, by Lemma 4.3.2, we can do this such that the state of the subsystem containing x and y is in $L_{x,y}$. Eventually we label all nodes in the tree and the resulting global state is in L'. Once again, this is because for every link subsystem (u,v), the state of the (u,v) subsystem is in $L_{u,v}$. The labelling procedure depends crucially on the fact that the topology is a tree. ∎

To prove the theorem, we will use the Execution Convergence Theorem (3.4.5). However, to apply that theorem we have to work with predicates of $\mathcal{T}^+$ (i.e., sets of

states of $\mathcal{T}^+$) and not link predicates of $\mathcal{T}^+$ (i.e., sets of states of link subsystems of $\mathcal{T}^+$). This is just a technicality that we deal with as follows. For each link subsystem (u, v), we define the predicate $L'_{u,v} = \{s : (s|u, s|v) \in L_{u,v}\}$. Clearly, $L' = \cap L'_{u,v}$

Next consider some u, v, w such that $v = \mathit{parent}(u)$ and $w = \mathit{parent}(v)$. We assume that $v \neq \mathit{nil}$. (But v may be the root in which case w is *nil*.) The next lemma states an important stability property. It states that if $L'_{u,v}$ holds in some global state s of $\mathcal{T}^+$ it will remain true in any successor state of s if either:

- v is the root OR
- $L'_{v,w}$ is also true in s.

Lemma 4.3.4 *Consider some u, v, w such that $v = \mathit{parent}(u) \neq \mathit{nil}$ and $w = \mathit{parent}(v)$. Suppose there is some global state s of $\mathcal{T}^+$ such that $s \in L'_{u,v}$ and $(w \neq \mathit{nil}) \rightarrow s \in L'_{v,w}$. Then for any transition $(s, \pi, \tilde{s})$, $\tilde{s} \in L'_{u,v}$.*

Proof: It suffices to consider all possible actions π that can be taken at either u or v in state s. It is easy to see that we don't have to consider correction actions because, by assumption, neither the CORRECT$_u$ or the CORRECT$_v$ action is enabled in state s.

Consider a modified action a_u of $\mathcal{T}^+$ that is taken at node u. Suppose action a_u occurs in state s and results in a state $\tilde{s}$. By the preconditions of action a_u, for all children x of u, $(s|x, s|u) \in L_{x,u}$. But in that case by Lemma 4.3.3 there is some other global state $s' \in L'$ such that: $s'|u = s|u$, $s'|v = s'|v$ and $s'|x = s|x$ for all children x of u. Thus by the third property of a network automaton, the action a_u is also enabled in s' and, if taken in s', will result in some state say $\tilde{s}'$. But since L' is closed, $\tilde{s}' \in L'$ and hence $(\tilde{s}'|u, \tilde{s}'|v) \in L_{u,v}$. But by the third property of a network automaton, $\tilde{s}|u = \tilde{s}'|u$ and $\tilde{s}|v = \tilde{s}'|v$. Thus $\tilde{s} \in L'_{u,v}$.

The case of a modified action at v is similar. ∎

The previous lemma also shows that our second assumption is safe. If a modified action is taken at a node u, resulting in state $\tilde{s}$ then $\tilde{s} \in L'_{u,v}$ for some v. Thus by Lemma 4.3.3 there is some other state $\tilde{s}' \in L'$ such that $\tilde{s}'|u = \tilde{s}|u$. Thus $\tilde{s}|u$ cannot have been removed as part of the normalization step.

The next lemma states an obvious liveness property. If $L'_{u,v}$ does not hold in some global state of $\mathcal{T}^+$, then after at most t_n time units we will eventually reach some

global state $\tilde{s}$ in which $L'_{u,v}$ holds. Clearly this is guaranteed by the correction actions (either CORRECT_u or CORRECT_v depending on whether u is the child of v or vice versa) and by the timing guarantees.

Lemma 4.3.5 *For any any (u,v) link subystem and any any execution α of $\mathcal{T}^+$ and any state s_i of α, if $s_i \notin L'_{u,v}$ then there is some later state $s_j \in L'_{u,v}$ that occurs within t_n time units of s_i.*

Proof: Suppose not. Then either CORRECT_u (if u is the child of v) or CORRECT_v (if v is the child of u) is continuously enabled for t_n time units after s_i. But then by the timing guarantees, either CORRECT_u or CORRECT_v must occur within t_n time after s_i, resulting in a state in which $L'_{u,v}$ (and, of course, $L'_{v,u}$) holds. ∎

We now return to the proof of the theorem. First we define a natural partial order on the predicates $L'_{u,v}$. For any link subsystem (u,v), define the *child node* of the subsystem to be u if $\mathit{parent}(u) = v$ and v otherwise. Define the ordering $<$ such that $L'_{u,v} < L'_{w,x}$ iff the child node of the (u,v) subsystem is an ancestor (in the tree T) of the child node of of the (w,x) subsystem.

Using this partial order and Lemmas 4.3.4 and 4.3.5, we can apply the Execution Convergence Theorem (Theorem 3.4.5), to show that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}^+|L'$ in time $\mathit{height}(T) \cdot t_n$. But any execution α of $\mathcal{T}^+|L'$ is also an execution of $\mathcal{T}|L$. This follows from three observations. First, since L' is closed for $\mathcal{T}$, L' is closed for $\mathcal{T}^+$. Second, if L' holds in all states of an execution α of $\mathcal{T}^+|L'$, then no correction actions can occur in α. Third, any execution of $\mathcal{T}|L'$ is also an execution of $\mathcal{T}|L$ because $L \supseteq L'$. Thus we conclude that $\mathcal{T}^+$ stabilizes to the executions of $\mathcal{T}|L$ in time $\mathit{height}(T) \cdot t_n$.

This theorem can be used as the basis of a design technique. We start by designing a tree automaton T that is locally checkable for some L. Next we use the construction in the theorem to convert T into $\mathcal{T}^+$. $\mathcal{T}^+$ stabilizes to the executions of $T|L$ even when started from an arbitrary state.

4.3.1 Weakening the Fairness Requirement

In the previous subsection, we assigned each CORRECT_u action to a separate class. Actually the theorem only requires a property we call *eventual correction*: if a CORRECT_u action is continuously enabled, then a CORRECT_u action occurs within bounded time.

It is interesting that the protocols in [Dij74, AG90] only require that some enabled action in the entire network occur in bounded time. In other words, all the actions in the entire automaton can be placed in a single class. How can we ensure the eventual correction property in a model in which the only guarantee is that some enabled action (in the entire network) will occur in bounded time? To make sure the eventual correction property holds in such a model, we need to show that it is impossible to remain for an unbounded amount of time in a state in which $L_{u,v}$ does not hold and in which some other action other than CORRECT_u is enabled. This property can be established[4] quite easily for the protocols in [Dij74] and [AG90].

4.4 Rediscovering Dijkstra's Protocols

In this section, we will begin by reconsidering the second example in [Dij74]. This protocol is essentially a token passing protocol on a line of nodes with process indices ranging from 0 to $n-1$. Imagine that the line is drawn vertically so that process 0 is at the bottom of the line (and hence is called "bottom") and Process $n-1$ is at the top of the line (and called "top"). This is shown in Figure 4.5. The down neighbor of Process i is Process $i-1$ and the up neighbor is Process $i+1$. Process $n-1$ and Process 0 are not connected.

Dijkstra observed that it is impossible (without randomization) to solve mutual exclusion in a stabilizing fashion if all processes have identical code. To break symmetry, he made the code for the "top" and "bottom" processes different from the code for the others.

Dijkstra's second example is modelled by the automaton $D2$ shown in Figure 4.6. Each process i has a boolean variable up_i, and a bit x_i. Roughly, up_i is a pointer at node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 4.5 shows a state of this protocol when it is working correctly. First, there can be at most two consecutive nodes whose up pointers differ in value and the token is at one of these two nodes. If the two bits at the two nodes are different (as in the figure) then the token is at the upper node; else the token is at the lower node.

[4] I am grateful to Anish Arora for pointing this out to me.

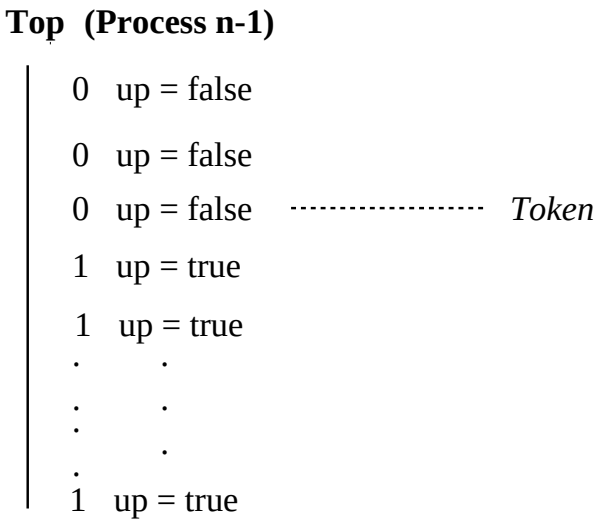


Figure 4.5: Dijktra's protocol for token passing on a line

For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = \mathit{false}$ for all processes other than process 0. We will remove the need for such initialization below. We start by understanding the correct executions of this protocol when it has been corectly initialized.

A process i is said to have the token when any action at Process i is enabled. As usual the system is correct when there is at most one token in the system. Now, it is easy to see that in the initial state only MOVE_UP$_0$ is enabled. Once node 0 makes a move, then MOVE_UP$_1$ is enabled followed by MOVE_UP$_2$ and so on as the "token" travels up the line. Finally the token reaches node $n-1$, and we reach a state s in which $x_i = x_{i+1}$ for $i = 0 \ldots n-3$ and $x_{n-1} \neq x_{n-2}$. Also in state s, $up_i = \mathit{true}$ for $i = 0 \ldots n-2$ and $up_{n-1} = \mathit{false}$. Thus in state s, MOVE_DOWN$_{n-1}$ is enabled and the token begins to move down the line by executing MOVE_DOWN$_{n-2}$ followed by MOVE_DOWN$_{n-3}$ and so on until we reach the initial state again. Then the cycle continues. Thus in correct executions, the "token" is passed up and down the line.

In the good states for Dijktra's second example, the line can be partitioned into two bands as shown in Figure 4.7. All bit and pointer values within a band are equal. (If a node within a band has $up_i = \mathit{false}$ we sketch it as a pointer that points downwards). All nodes within the upper band point downwards and all nodes within the lower band

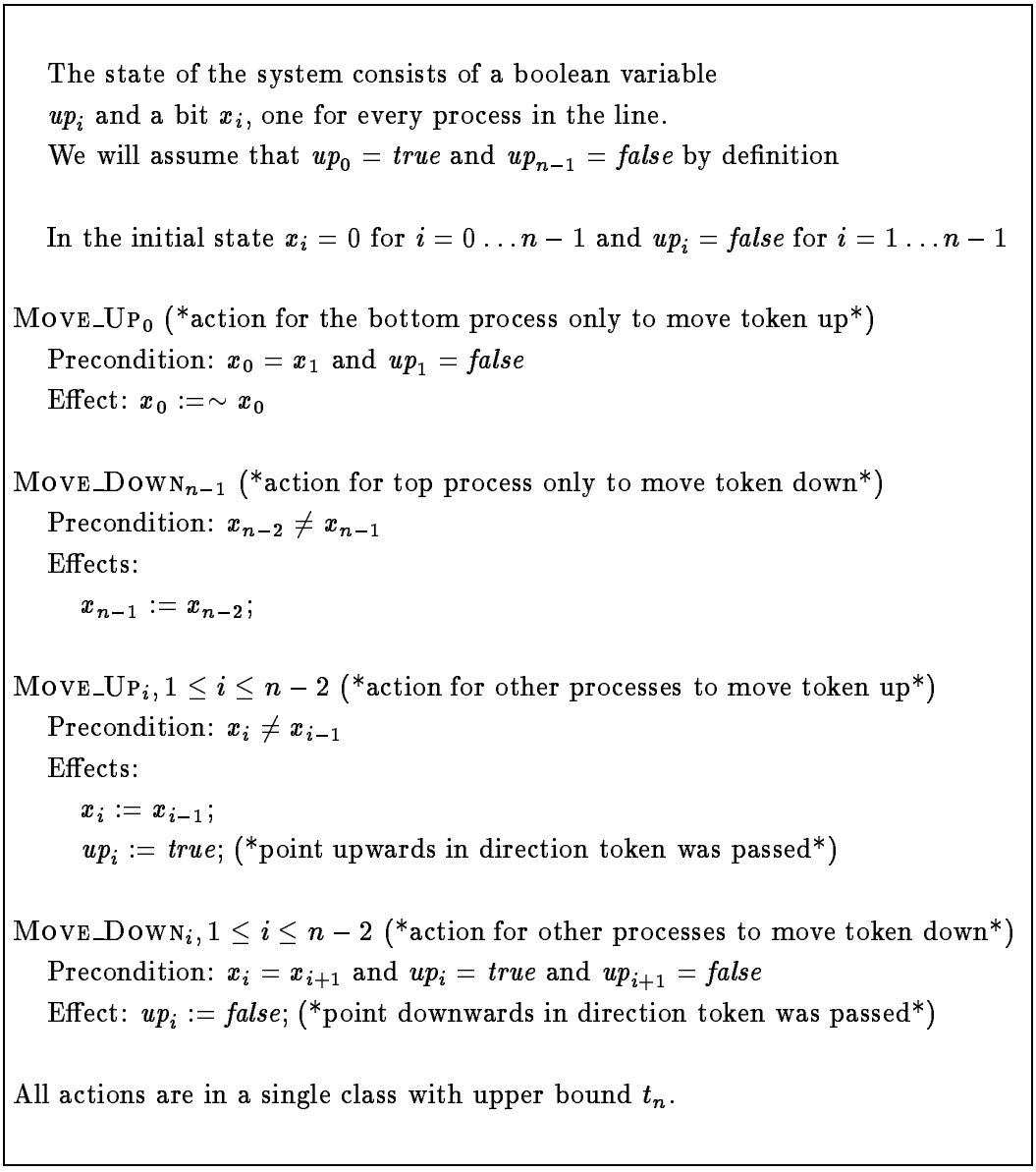


Figure 4.6: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most 4 states.

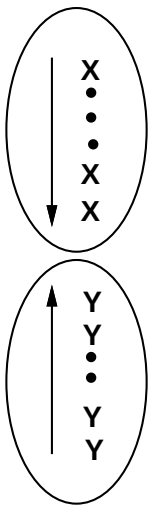


Figure 4.7: In the good states for Dijktra's second example, the line can be partitioned into 2 bands with the token at the boundary.

point upward. The token is at the boundary between the two bands. If the bit value X of the upper band is equal to the bit value Y of the lower band, the token is moving downwards; if the two bit values are unequal the token is moving downwards.

We describe these "good states" of $D2$ (that occur in correct executions) in terms of local predicates. In the shared memory model, a local predicate is any predicate that only refers to the state variables of a pair of neighbors. Intuitively, we see that if two neighboring nodes have the same pointer value, then their bits are equal; also if a node is pointing upwards, then so is its lower neighbor. Thus in a good state of $D2$, two properties are true for any Process i other than 0:

- If $up_{i-1} = up_i$ then $x_{i-1} = x_i$.
- If $up_i = true$ then $up_{i-1} = true$.

First, we prove that if these two local predicates hold for all $i = 1 \ldots n-1$, then there is exactly one action enabled. Intuitively, since $up_{n-1} = false$ and $up_0 = true$, we can start with process $n-1$ and go down the line until we find a pair of nodes i and $i-1$ such that $up_i = false$ and $up_{i-1} = true$. Consider the first such pair. Then the second predicate guarantees us that there is exactly one such pair. The first predicate then

guarantees that all nodes $j < i-1$ have $x_j = x_{i-1}$ and all nodes $k > i$ have $x_k = x_i$. Thus only one action is enabled. If $x_i = x_{i-1}$ and $i-1 \neq 0$ then only MOVE_DOWN$_{i-1}$ is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$ then only MOVE_UP$_0$ is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$ then only MOVE_UP$_i$ is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$ then only MOVE_DOWN$_{n-1}$ is enabled.

A similar argument shows that if there is exactly one action enabled, then both local predicates hold for all $i = 1 \ldots n-1$.

Let L be the predicate of $D2$ which consists of the states of $D2$ in which exactly one action is enabled. It is easy to see that $D2$ is a tree automaton that is locally checkable for L. Then we can use Theorem 4.3.1 to convert $D2$ into a new automaton $D2^+$ which stabilizes to executions in which there is exactly one token in each state.

The correction actions we add are once again different from the original actions in [Dij74]. However, the corrections actions we add (and consequently the proofs) are much more transparent than the original version.

Dijkstra also described two more stabilizing mutual exclusion protocols, one using K-state machines, where K is greater than the number of processes, and a solution using 3-state machines. In both solutions the topology is assumed to be a ring (i.e., the bottom and top processes are connected). The first protocol is easily seen to be locally checkable. Unfortunately it is no longer a tree automaton and hence Theorem 4.3.1 does not apply directly. However, with some extra work it is possible to derive the final protocol as a basic protocol augmented with correction actions that ensure that each link predicate becomes true. An even simpler way is to derive the K-state protocol using an idea that we call counter flushing (see Chapter 10 and Appendix E).

Dijkstra's three state protocol uses a different idea altogether. The protocol is not even locally checkable. This protocol seems extremely specific to the ring topology used. There are two main ideas. First, tokens are passed up and down a line in normal operation just as in the second example. Thus if there is more than one token and tokens must keep moving, the tokens must eventually "collide" at some node. This node can then destroy one of the tokens. This idea seems limited to a line/ring topology. The first idea, however, is not sufficient to detect a situation in which there are no tokens. The second idea is to exploit the neighbor relation between the top and bottom processes to detect the presence of no tokens. The states of the system are such that if there are no tokens, then all processes will have the same state. In

normal operation, the states of the top and bottom nodes are always different. Thus the absence of tokens can be "suspected" locally if the state of the bottom node is not equal to that of the top node. In that case, a token is manufactured. The actual protocol can be understood using these ideas.

4.5 Summary

Much of the initial work in self-stabilization was done in the context of Dijkstra's shared memory model of networks. Later, the work on local checking and correction was introduced [APV91b] in a message passing model. The main contribution of this chapter is to show that existing work in the shared memory model can be understood crisply in terms of local checking and correction. Protocols that appeared to be somewhat *ad hoc* are shown to have a common underlying principle.

However, as we have argued at the beginning of this chapter, we believe that message passing models are more useful and realistic. For the rest of the thesis we will concentrate on message passing models. *The definitions of network automata, local predicates, local checkability, local correctability, and link subsystems that we used in this chapter are specific to shared memory systems.* In the next chapter (Chapter 5) we will introduce definitions of these concepts for networks in which nodes communicate by message passing. The definitions in Chapter 5 will be used for the remainder of the thesis.

The main theorem in this chapter states that any locally checkable protocol that uses a tree topology can be efficiently stabilized. As the reader might expect, there is a corresponding Tree Correction theorem for message passing systems that is described at the end of Chapter 6.

Chapter 5

Local Checking and Correction for Network Protocols

In Chapter 4 we introduced a method of local checking and correction using a shared memory model taken from [Dij74]. Recall that in such a model, nodes communicate with their neighbors by reading the state of all neighbors in one atomic action. Thus there is no need to model channels between nodes. The shared memory model allowed us to introduce local checking and correction in a fairly simple way. However, it is not very realistic because of the high degree of atomicity that it assumes. In this chapter, and for the rest of the thesis, we will model communication between nodes by explicit message passing through links. However, we will restrict ourselves to a special type of link called a *Unit Storage Data Link* or UDL.

We begin in Section 5.1 by describing our model of a network protocol. We do so by modelling the network topology, the links between nodes by Unit Storage links, and the nodes themselves. Our model of a Unit Storage Link is new, and so in Section 5.2 we argue that such links can be implemented in real-life networks. Section 5.3 introduces the important concept of *locality* in network protocols: some key concepts such as *local subsystems, local checkability, and local correctability* are defined in this section. While many of the ideas are similar to the ideas in Chapter 4 (that were developed for shared memory systems), the new definitions are slightly more complex because of the presence of channels between nodes. The definitions in this chapter are used for the remainder of the thesis.

In Section 5.4 we state the main result of this chapter, the *Local Correction Theo-*