# Self-Stabilization by Local Checking and Correction

by

George Varghese

B.Tech, Electrical Engineering
Indian Institute of Technology, Bombay
(1981)
M.S., Computer Studies
North Carolina State University
(1983)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 1992

© Massachusetts Institute of Technology 1992

Signature of Author .....
Department of Electrical Engineering and Computer Science
October 20, 1992

Certified by .....
Baruch Awerbuch
Associate Professor
Thesis Supervisor

Certified by .....
Nancy A. Lynch
Professor
Thesis Co-supervisor

Accepted by .....
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

**Self-Stabilization by Local Checking and Correction**

by

George Varghese

Submitted to the Department of Electrical Engineering and Computer Science
on October 20, 1992, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

## Abstract

A *self-stabilizing protocol* is one that begins to behave correctly in bounded time, no matter what state the protocol is started in. Self-stabilization abstracts the ability of a protocol to tolerate *arbitrary* faults that stop. We investigate the power and applicability of *local checking and correction* for the design of stabilizing network protocols.

A link subsystem is a pair of neighboring nodes and the two links between them. Intuitively, a protocol $P$ is *locally checkable* if whenever $P$ is in a bad state, some link subsystem is also in a bad state. A protocol $P$ is *locally correctable* if $P$ can be corrected to a good state by locally correcting link subsystems.

We present four general techniques for designing stabilizing protocols. We first show that every locally checkable and correctable protocol can be stabilized in time proportional to the height of an underlying partial order. Second, we show that every locally checkable protocol on a tree can be stabilized in time proportional to the height of the tree. Third, we show that every locally checkable protocol can be stabilized in time proportional to the number of network nodes. The third result shows that we can dispense with the need for local correctability or the need for the underlying topology to be a tree as long as we are willing to pay a higher price in stabilization time. Fourth, we show that any deterministic synchronous protocol $\pi$ can be converted to an asynchronous, stabilizing version of $\pi$. The fourth technique is useful because there are network tasks for which a synchronous protocol exists but for which no asynchronous, locally checkable solution is known.

We also present two useful heuristics. The first heuristic, that of *removing unexpected packet transitions*, can often be used to transform a protocol into a locally checkable equivalent. A number of existing protocols work in a *dynamic network* model where links can fail and recover. The second 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.

Together our techniques cover a broad range of networking tasks. We use our general techniques to construct new or improved stabilizing solutions to many specific for Mutual Exclusion, Network Resets, Spanning Trees, Topology Update, Min Cost Flows etc. Many of our solutions are practical and can be applied to real networks without appreciable loss in efficiency. For example, the messages required for local checking can easily be piggybacked on the "keep-alive" traffic sent between neighbors in real networks.

Our techniques also help in succinctly understanding existing stabilizing protocols. We

define a special case of local checking called *one-way checking*. We show that many existing protocols implicitly use one-way checking together with two other methods that we call *counter flushing* and *timer flushing*.

In the past, papers on stabilization have avoided message passing models of communication because of the problems caused by unbounded storage Data links. In a stabilizing setting, such links can be initialized with an unbounded number of fictitious packets. Thus almost any non-trivial network task is impossible in a stabilizing setting in which the links have unbounded storage and the nodes are restricted to be finite state machines. We avoid this problem by using the standard asynchronous message passing model of a computer network except that each link is what we call a Unit Storage Data Link (UDL) that can store at most one packet. Our UDL model can be implemented over real physical channels. Our UDL model also generalizes easily to a Bounded Storage Data Link which can store a constant number of packets.

We introduce a new definition of stabilization in terms of the external behavior of a system. The definition allows us to define that an automaton $A$ stabilizes to another automaton $B$ even though $A$ and $B$ have different state sets. The definition also allows a clean statement of a useful *Modularity Theorem*. This theorem allows us to prove that a large system is stabilizing by proving that each of its pieces is stabilizing.

**Keywords:** Self-Stabilization, Fault-Tolerance, Network Protocols, Distributed Algorithms, Local Checking and Correction.

Thesis Supervisor: Baruch Awerbuch
Title: Associate Professor

Thesis Co-supervisor: Nancy A. Lynch
Title: Professor

## Acknowledgments

I have been fortunate to be helped by a huge number of talented people. Basically, I have had two sets of advisors – one set from DEC where I work, and another set from MIT.

I thank Radia Perlman for being my advisor at DEC. I owe so much to her. My interest in the field of self-stabilization began after watching Radia design two elegant and widely used stabilizing protocols. After Radia went back to MIT to complete her doctorate, she encouraged me to do the same. She plotted my course load, provided feedback, directed my application process, cajoled, consoled, and occasionally scolded. Recently, she was selected by Data Communications Magazine as one of twenty people who have changed the field of networking. But to me she is something much better, a good and loyal friend.

I thank Alan Kirby of DEC for being a wonderful manager and boss. By trusting and believing in me, he has helped me to do things that I did not think I was capable of. Besides being a fine manager, Alan also has a great deal of technical vision. All the projects he has assigned me to work on have provided seed ideas for this thesis.

I thank Baruch Awerbuch of MIT who supervised my thesis together with Nancy Lynch. Baruch is a restless visionary who is constantly conquering new areas. This thesis began with Baruch's conviction that the so called end-to-end problem had a stabilizing solution. That problem led us to invent local checking and correction. Still later, Baruch was convinced that it was "easy" to construct an efficient and stabilizing reset protocol. It is typical of his approach that he began with problems that other people (especially me!) considered hard.

I thank Nancy Lynch of MIT for being a superb thesis supervisor and advisor. Nancy had to struggle with initial drafts of this thesis that were truly terrible. Under her guidance, the formalism gradually emerged. Her idea of using a specification of stabilization based on external behavior has been particularly fruitful. Working with her has been a true education, as I have begun to learn a little of what scholarship and rigor means.

I thank Robert Gallager of MIT for his inspiration, and his patience in reading this thesis. His insistence on clarity has shamed me into rewriting parts of the thesis. I will always remember the kindness which he, a great and brilliant man, shows to his students.

I thank Mark Tuttle of DEC Cambridge Research Lab for his painstaking reading of this thesis. Time and again, Mark spotted errors and helped clean up important definitions. Even

his most critical comments were leavened with humor and encouragement. If this thesis is reasonably free of errors, it is because of the hard work of Nancy and Mark. Mark also helped me to understand the big picture; the introduction and Chapter 10 have improved because of his efforts.

I thank Boaz Patt who worked together with Baruch and myself on many of the ideas in this thesis. I thank Boaz especially for his hard work on an initial version of the reset protocol, and for initiating the use of an external behavior specification for the reset protocol. I owe Boaz many things, including the succinct title of this thesis.

I thank the DEC Graduate Engineering Program (GEEP) for their incredible generosity in allowing me to go to MIT on full salary for two years. I especially thank Shirley Stahl and Terry Sarandrea of GEEP for their ability to cut through administrative procedures and to help me when I needed help. I also thank Joanne Talbot at MIT for her administrative support at MIT.

This thesis is dedicated to my father, who would have loved to leaf through it if he were alive; and to my mother and sister, for their patience and love.

Finally, I'd like to thank God and his son Jesus, from whom I believe all good gifts come – even this wonderful set of people that it has been such a joy to live and work with.

# 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.

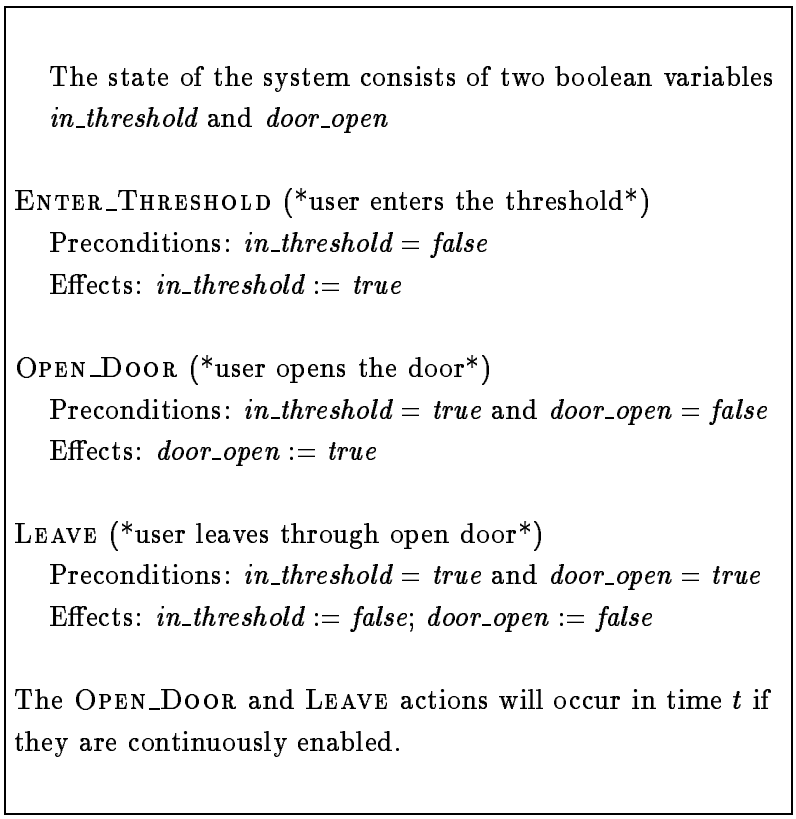



Figure 1.1: Door Closing System

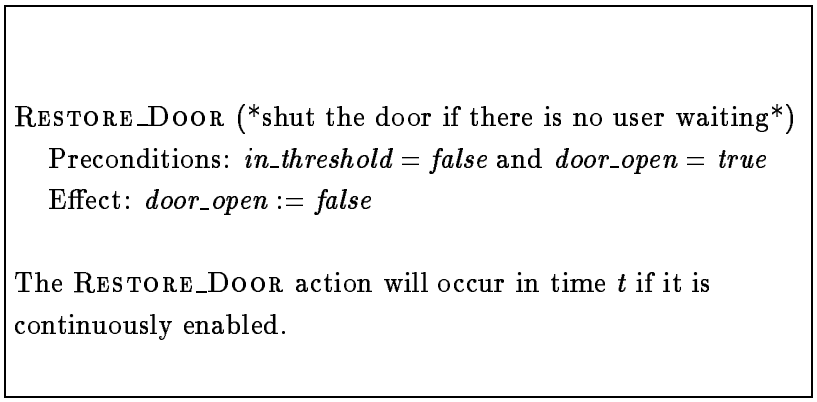



Figure 1.2: Extra action that models automatic door closing in a Door Closing System.

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

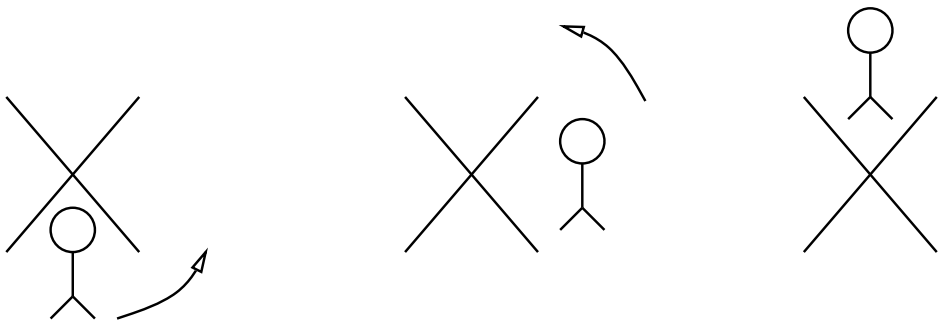



Figure 1.3: Exiting through a revolving door.

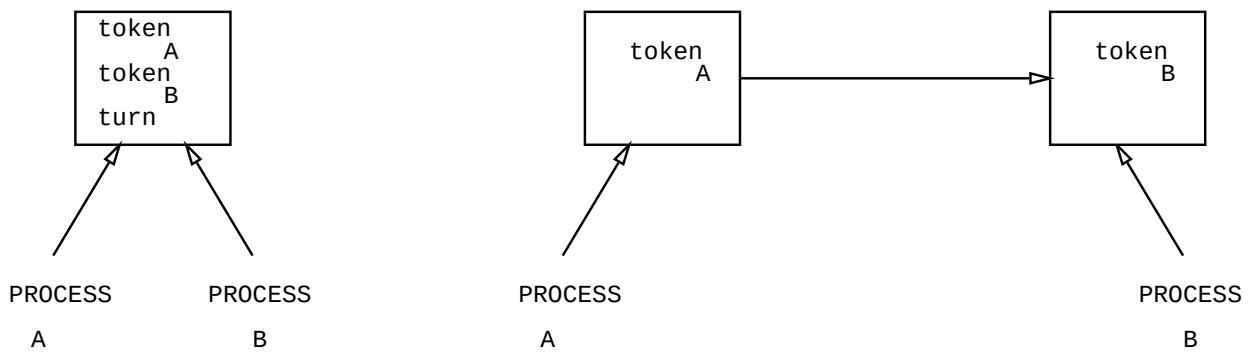
$A$ and $B$. A process can "run" only when it has the token.

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

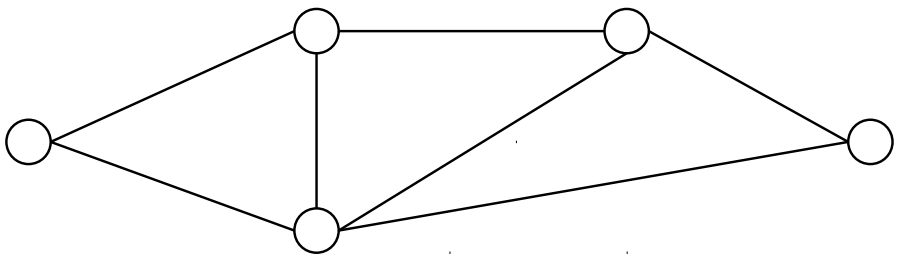


Figure 1.5: A typical mesh Network

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

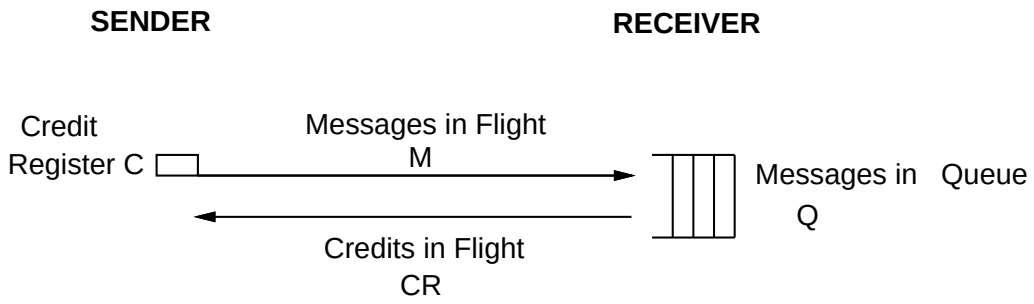



Figure 1.6: Credit Based Flow Control between a Sender and a Receiver

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

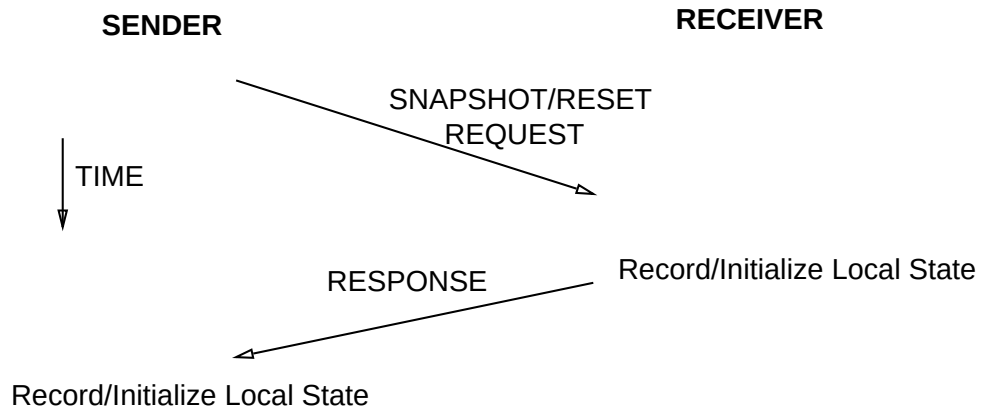



Figure 1.7: A Single Phase of Checking/Correction using Snapshots/Resets

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 $\mathit{Max}$. 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.

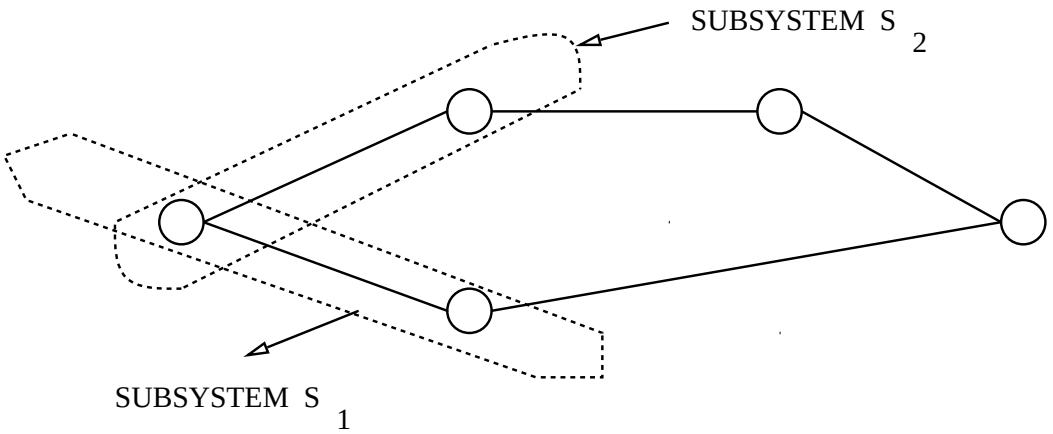



Figure 1.8: An Example of Two Overlapping Link Subsystems in a Network

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

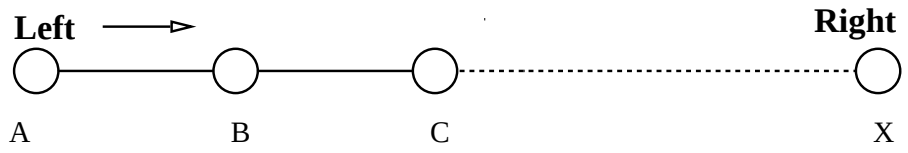



Figure 1.9: Token Passing in an Oriented Line Graph is not Locally Checkable

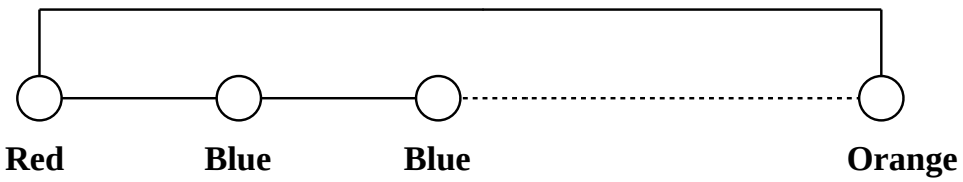



Figure 1.10: Coloring a Cycle is Locally Checkable but not Locally Correctable

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 $\pi$ into a self-stabilizing asynchronous version of $\pi$. 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

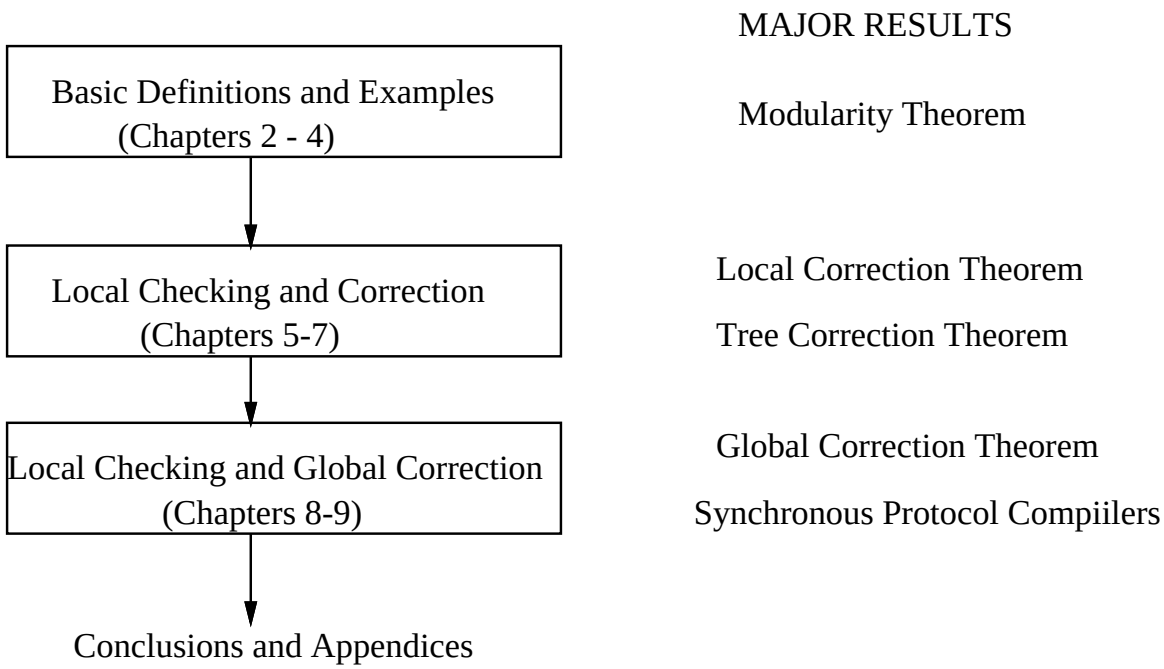



Figure 1.11: Thesis Organization

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 $\pi$ into a self-stabilizing asynchronous version of $\pi$. The main compiler, the *Resynchronizer*, works by first applying the synchronizer protocol of [Awe85] to create an asynchronous version of $\pi$. 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 $\pi$. 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.

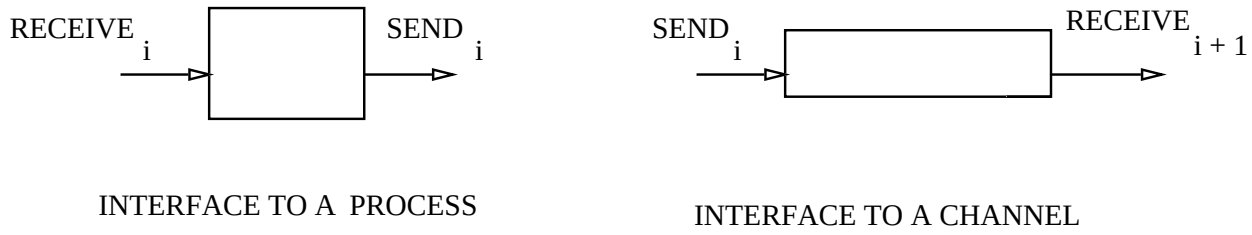



Figure 2.1: The Interface to Process $i$ and the Channel between Process $i$ and Process $i+1$ in a token passing system

### 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

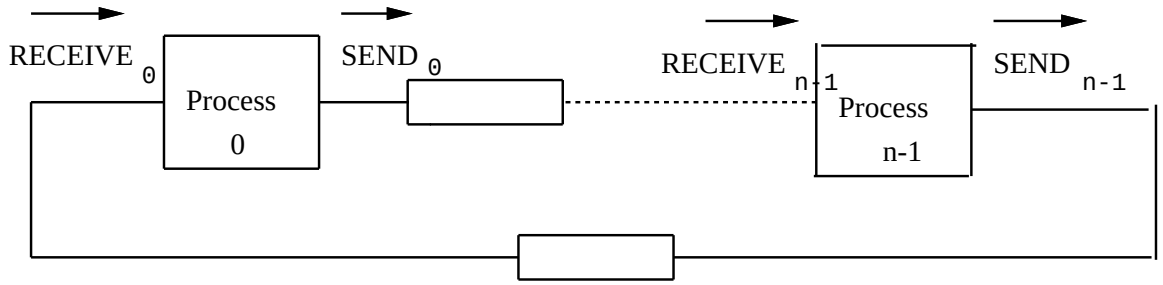



Figure 2.2: The token passing system formed by composition of processes and channels

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 $\pi$ 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 $\pi$. This in turn implies that action $\pi$ 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 $\alpha$ of $S$:

- There is exactly one token in any state of $\alpha$.
- Within a bounded time after any state of $\alpha$, 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.

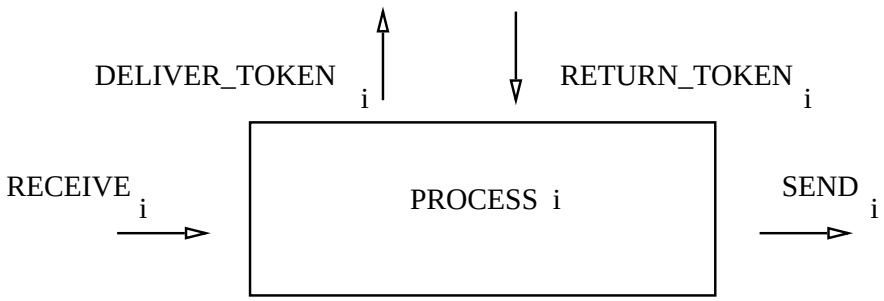



Figure 2.3: Process $i$ with additional interfaces to an external client

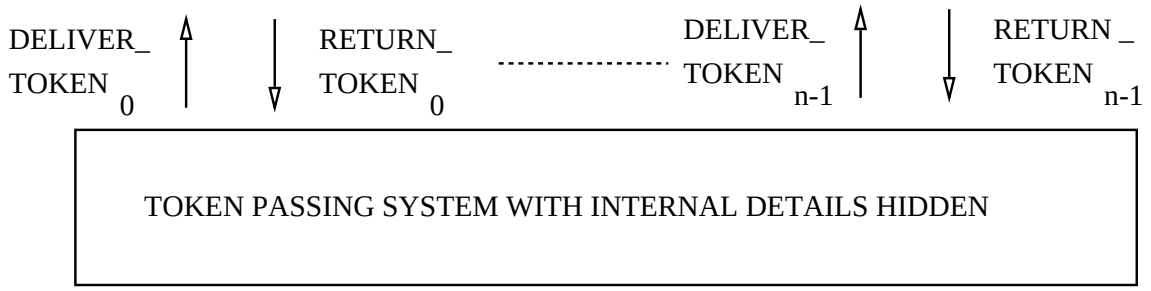



Figure 2.4: A modular token passing system and its external interface to clients

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 $\alpha$ is the subsequence of $\alpha$ 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 $\alpha$. A sequence $\beta$ is said to be a behavior of automaton $A$ if $\beta$ 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 $\beta$ of $S$ the following two properties hold in the sequence $\beta$:

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

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 $\alpha$ *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 $\alpha$ 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 $\alpha$ (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 $\alpha$ of $A$. Let $\gamma$ be the subsequence of $\alpha$ consisting of timed external actions, and let $t_0$ be the start time of $\alpha$. The* behavior $\beta$ *corresponding to $\alpha$ 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* $\beta$ 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 $\beta$; $\beta.start$ can be read as the start time of behavior sequence $\beta$. 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 $\alpha$ is the sequence $s_0, a_1, s_1, a_2, s_2, \ldots$. For brevity, we will frequently describe execution $\alpha$ 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 $\alpha$ 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 $\beta$ is the sequence $b_1, b_2, \ldots$. We will often describe behavior $\beta$ using the corresponding untimed behavior $b_1, b_2, \ldots$. Once again by our notation, the start time of $\beta$ is $\beta.start$ and the time associated with any action $b_j$ in $\beta$ 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 $\alpha$ of an automaton $A$. Any suffix of $\alpha$ 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 $\pi$ is an input action of $A$ if $\pi$ 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 $\beta$ of $A$ on to some constituent automaton $A_i$. The projection is the subsequence of $\beta$ containing actions of $A_i$. We also assume that $\beta|A_i$ inherits the times of $\beta$ in the natural way. Thus the time associated with any action in $\beta|A_i$ is the time associated with the corresponding action in $\beta$, and the start time of $\beta|A_i$ is the start time of $\beta$.

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 $\beta$ 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 $\beta$ be a behavior sequence such that each action in $\beta$ is an external action of $A$. If $\beta|A_i$ is a behavior of $A_i$ for every $i \in I$, then $\beta$ is a behavior of $A$.*

Finally, there is a hiding operator on automaton $A$ with respect to some subset $\Sigma$ 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 $\Sigma$ 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 $\alpha$ we mean the $i$-th state in the sequence $\alpha$. An *interval* of an execution $\alpha$ is a contiguous subsequence of $\alpha$ 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 $\beta$ is a contiguous subsequence of $\beta$.

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 $\alpha$ if for all states $s_j$ that occur within time $t$ of $s_i$ in $\alpha$, $s_j \in S$.

### 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 $\alpha$. Intuitively, this is a suffix of $\alpha$ whose first element occurs no more than time $t$ after the start of $\alpha$. 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 $\alpha$ and $\alpha'$. We say that $\alpha'$ is a $t$-suffix of execution $\alpha$ if:*

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

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 $\alpha$ of $A$ there is some $t$-suffix of execution $\alpha$ 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 $\alpha$ of $A$ there is some $t$-suffix of execution $\alpha$ 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 $\beta$. Intuitively, this is a portion of $\beta$ that starts at time no more than $t$ after the start of $\beta$. 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 $\gamma$. Thus we cannot simply define a $t$-suffix of $\beta$ to be a suffix of $\beta$ as we did in the case of an execution. We would also like a $t$-suffix of $\beta$ 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 $\beta$. 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 $\beta$. Intuitively, as we said before, this

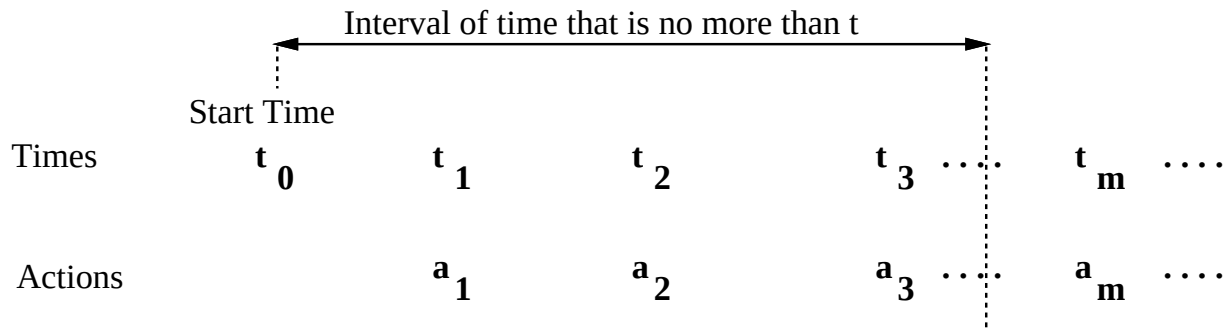



Figure 3.1: A pictorial view of a $t$-suffix of a behavior

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

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

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

Note that the definition allows $\gamma'$ to contain some, none, or all of the actions in $\beta$ 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 $\beta$ of $A$ there is a $t$-suffix of behavior $\beta$ 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 $\beta$ of $A$ there is a $t$-suffix of behavior $\beta$ 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 $\beta$ be any behavior of $A$. Let $\alpha$ be some execution of $A$ corresponding to $\beta$. From the hypothesis, there is some $\alpha'$ that is a $t$-suffix of execution $\alpha$ and is also an execution of $B$. Consider the behavior $\beta'$ of $B$ corresponding to execution $\alpha'$ of $B$. From the definitions, we can verify that that $\beta'$ is a $t$-suffix of behavior $\beta$. ∎

## 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 $\alpha$ of $A$, we use $f$ to construct a mapped execution of $B$ that has the same external behavior as $\alpha$.

**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*
   - *$\pi$ is an action of $B$ and $(f(s), \pi, f(\tilde{s}))$ is a transition of $B$ OR*
   - *$\pi$ is not an action of $B$ and $f(s) = f(\tilde{s})$.*
3. *For any class $c$ of $B$ and any execution $\alpha$ of $A$, suppose some action in $c$ is enabled in $f(s)$. Then within $t_c$ time of $s$ in $\alpha$ 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 $\alpha$ 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 $\alpha$ by:

- Removing every timed action in $\alpha$ 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 $\alpha$.

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 $\alpha$. Intuitively, $a_k$ is the action in $\alpha$ 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 $\alpha$ 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 $\alpha$ such that if $\gamma$ is the suffix of $\alpha$ 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 $\alpha$. 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 $\alpha$ 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 $\alpha$ of $A$ and every state $s$ in $\alpha$ 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 $\alpha$.*

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 $\alpha$ 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 $\alpha$ 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 $\alpha$ 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 $\alpha$. 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 $\alpha$ 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 $\alpha$ of a CIOA $A$. Any suffix of $\alpha$ 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 $\beta$ of $A$ and every time $t \geq 0$, every $t$-suffix of behavior $\beta$ 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.*

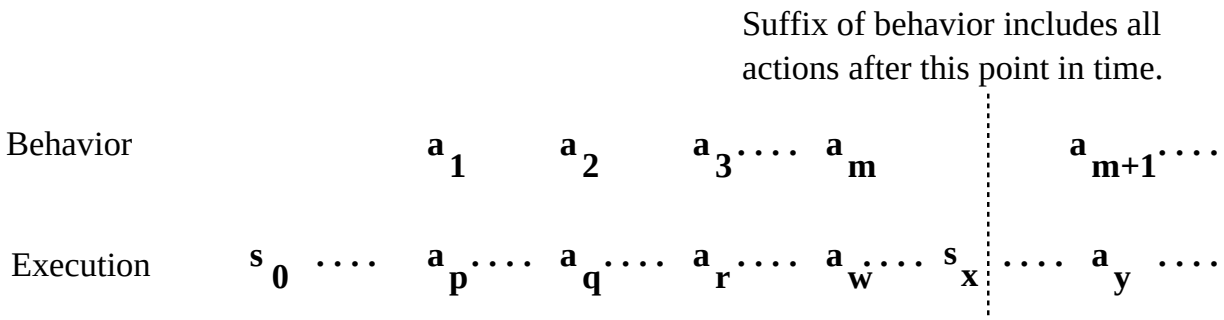



Figure 3.2: Obtaining the suffix of an execution corresponding to a $t$-suffix of the behavior of the execution.

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

The behavior $\beta$ and corresponding execution $\alpha$ are sketched in Figure 3.2. By definition, for every action in $\beta$ there is a corresponding external action in $\alpha$ 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 $\beta$ corresponds to $a_p$ in $\alpha$.)

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

We need a suffix $\alpha'$ of $\alpha$ whose behavior is equal to $\beta'$. Thus we look for a state $s_x$ in $\alpha$ corresponding to the vertical time line drawn in Figure 3.2. But we may not have a state in $\alpha$ whose time is equal to the start time of $\beta'$. So (intuitively) we choose $s_x$ to be the first state that occurs to the "left" of the vertical time line. Then we choose $\alpha'$ to be the suffix of $\alpha$ 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 $\beta$ of $A$. Consider the $\beta'$ that is a $t$-suffix of behavior $\beta$ and such that:

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

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

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 $\beta_i$ such that:

- $\beta_i$ 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 $\beta_i$ 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), $\beta'$ is a behavior of $B$. The theorem follows since $\beta'$ is a $t$-suffix of behavior $\beta$. ∎

### 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]

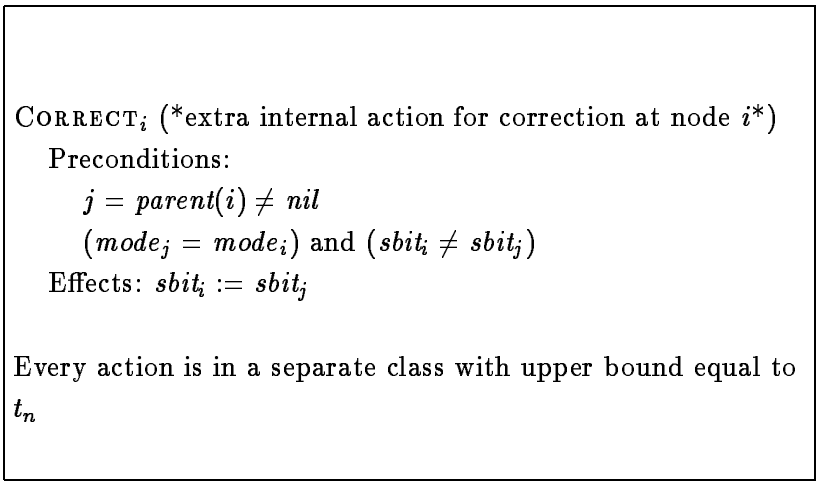



Figure 4.2: Original correction action in Arora and Gouda's Reset Protocol [AG90].

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

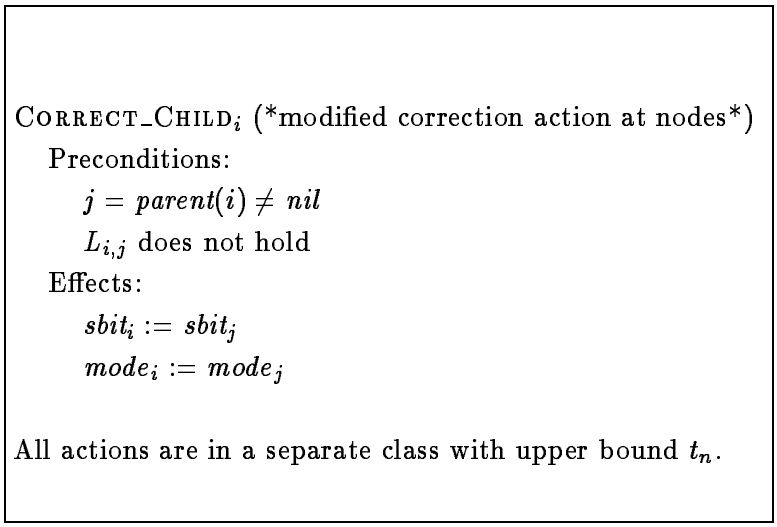



Figure 4.3: Modified Correction action for Arora and Gouda's Reset Protocol. All other actions are as in Figure 4.1.

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 $\pi$ 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 $\pi$ 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 $\pi$ 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 $\text{CORRECT}_u$ or $\text{CORRECT}_v$ 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 $\alpha$ of $\mathcal{T}^+$ and any state $s_i$ of $\alpha$, 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 $\text{CORRECT}_u$ (if $u$ is the child of $v$) or $\text{CORRECT}_v$ (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 $\text{CORRECT}_u$ or $\text{CORRECT}_v$ 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 $\alpha$ 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 $\alpha$ of $\mathcal{T}^+|L'$, then no correction actions can occur in $\alpha$. 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 $\text{CORRECT}_u$ action to a separate class. Actually the theorem only requires a property we call *eventual correction*: if a $\text{CORRECT}_u$ action is continuously enabled, then a $\text{CORRECT}_u$ 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 $\text{CORRECT}_u$ 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.

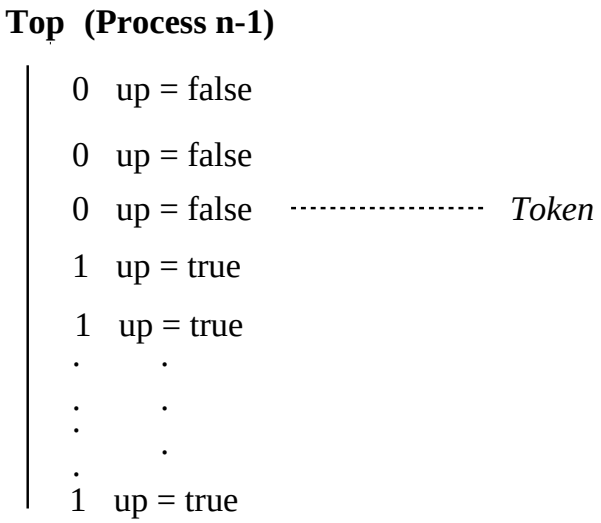



Figure 4.5: Dijktra's protocol for token passing on a line

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

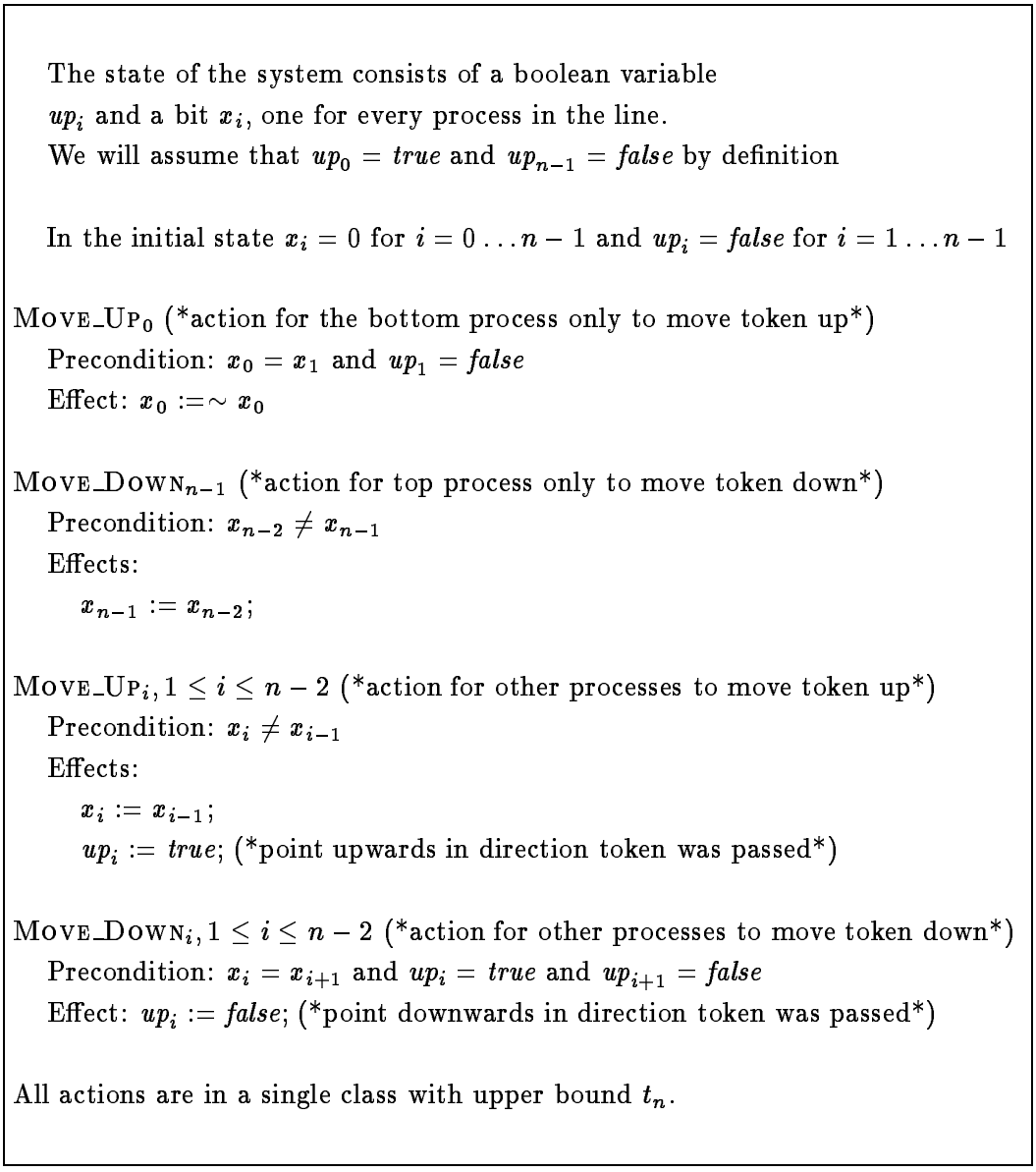



Figure 4.6: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most 4 states.

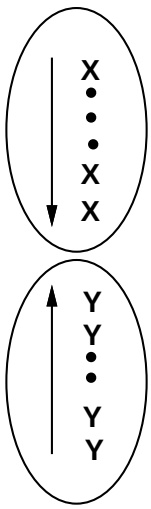



Figure 4.7: In the good states for Dijktra's second example, the line can be partitioned into 2 bands with the token at the boundary.

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-*

*rem.* In essence, this result states that any locally correctable protocol can be stabilized using a simple transformation. The transformation involves the addition of extra actions to do local checking and correction. Section 5.4 also contains a formal description of the transformation. The next two sections contains a proof of the Local Correction Theorem. We first provide an intuitive "proof" that presents the main ideas and then present a formal proof. The formal proof is important because it shows how we use the proof techniques of Chapter 4 to formally prove stabilization results. The proofs of the Tree and Global Correction theorems in later chapters are much more intuitive; the formal proof in this chapter provides an important example of how such intuitive proofs can be formalized.

Finally, Section 5.7 argues that the method of local checking and correction (that is formalized by the Local Correction Theorem) is practical, and can be added to real networks without an appreciable loss in efficiency.

## 5.1 Modelling Network Protocols

For the rest of this thesis, we will restrict ourselves to proving stabilization properties for network protocols and network automata. To model a network protocol, we need to model the network topology, the links between nodes, and the nodes themselves. Our model is essentially the standard asynchronous message passing model except for three differences:

- The major difference is that links are restricted to store at most one packet at a time.
- The nodes are restricted to use a certain stylized discipline for sending packets on unit storage links.
- We assume that for every pair of neighbors, there is some *a priori* way of assigning one of the two nodes as the "leader" of the pair.

We will argue that even with these differences our model can be implemented in real networks.

### 5.1.1 Modelling Network Topology

We use a directed graph $G$ to specify the network topology. For any two neighboring nodes $u$ and $v$, there are two edges $(u,v)$ and $(v,u)$. The network automaton will have a undirectional channel corresponding to each directed edge.

In addition, we require that $G$ satisfies the following property: there is a function that for any pair of neighboring nodes $u$ and $v$ in $G$ assigns one of the two nodes as the "leader" of the edges $(u,v)$ and $(v,u)$. Thus we are really requiring a way to break symmetry between neighboring nodes in $G$.

It is possible to remove this assumption (of having a leader function) at the cost of some increased complexity in the protocols. However this assumption is not restrictive in practice. In a real implementation, if every node has a unique ID then a simple stabilizing protocol can elect the minimum ID node as leader. Each node can periodically transmit its ID to its neighbor and both nodes choose the minimum ID. If the nodes do not have IDs then an equally simple randomized protocol can elect a leader on every link. We prefer to encode the leader function directly in the graph $G$ instead of presenting these simple protocols explicitly.

In summary, the nodes and edges of $G$ correspond to the actual physical topology of the network, while the leader function describes a way to break symmetry between neighboring nodes. Formally:

We will call a directed graph $(V,E)$ *symmetric* if for every edge $(u,v) \in E$ there is an edge $(v,u) \in E$.

**Definition 5.1.1** *A* topology graph $G = (V,E,l)$ *is a symmetric, directed graph* $(V,E)$ *together with a leader function* $l$ *such that for every edge* $(u,v) \in E$, $l(u,v) = l(v,u)$ *and either* $l(u,v) = u$ *or* $l(u,v) = v$.

We use $E(G)$ and $V(G)$ to denote the set of edges and nodes in $G$. If it is clear what graph we mean we sometimes simply say $E$ and $V$. As usual, if $(u,v) \in E$ we will call $v$ a neighbor of $u$.

The following definition of a leader edge is useful later. Of the two possible edges between two neighbors, it produces the edge directed away from the leader.

**Definition 5.1.2** *We call* $(u,v)$ *a* leader edge *of* $G$ *if* $(u,v) \in E$ *and* $l(u,v) = u$.

### 5.1.2 Modelling Network Links

Traditional models of a data link have used what we call *Unbounded Storage* Data Links that can store an unbounded number of packets. Now, real physical links do have bounds on the number of stored packets. However, the unbounded storage model is a useful abstraction in a non-stabilizing context.

Unfortunately, this is no longer true in a stabilizing setting. *If the link can store an unbounded number of packets, it can have an unbounded number of "bad" packets in the initial state.* It has been shown [DIM91a] that almost any non-trivial task is impossible in such a setting. Thus in a stabilizing setting it is necessary to define Data Links that have bounded storage.

A *network automaton* for topology graph $G$ consists of a node automaton for every vertex in $G$ and one channel automaton for every edge in $G$. We will restrict ourselves to a special type of channel automaton, a unit storage data link or UDL for short. Intuitively, a UDL can only store at most one packet at any instant. Node automata communicate by sending packets to the UDLs that connect them. In the next section, we will argue that a UDL can be implemented over real physical channels.

We fix a packet alphabet $P$. We assume that $P = P_{data} \cup P_{control}$ consists of two disjoint packet alphabets. These correspond to what we call data packets and control packets. The specification for a UDL will allow both data and control packets to be sent on a UDL.

**Definition 5.1.3** *We say that $C_{u,v}$ is the UDL corresponding to ordered pair $(u,v)$ and with link delay $t$ if $C_{u,v}$ is the UIOA defined in Figure 5.1.*

By the convention we have established, $C_{u,v}$ is a UIOA since we have not defined any start states for $C_{u,v}$. The external interface to $C_{u,v}$ includes an action to send a packet at node $u$ ($\text{SEND}_{u,v}(p)$), an action to receive a packet at node $v$ ($\text{RECEIVE}_{u,v}(p)$), and an action $\text{FREE}_{u,v}$ to tell the sender that the link is ready to accept a new packet. The state of $C_{u,v}$ is simply a single variable $Q_{u,v}$ that stores a packet or has the default value of *nil*.

Notice two points about the specification of a UDL. The first is that if the UDL has a packet stored, then any new packet sent will be dropped. Second, the FREE action is enabled continuously whenever the UDL does not contain a packet.

Each $p$ belongs to the packet alphabet $P$ defined above.
The state of the automaton consists of a single variable $Q_{u,v} \in P \cup nil$.

SEND$_{u,v}(p)$ (*input action*)
Effect:
If $Q_{u,v} = nil$ then $Q_{u,v} := p$;

FREE$_{u,v}$ (*output action*)
Precondition: $Q_{u,v} = nil$
Effect: None

RECEIVE$_{u,v}(p)$ (*output action*)
Precondition: $p = Q_{u,v} \neq nil$
Effect: $Q_{u,v} := nil$;

The FREE and RECEIVE actions are in separate classes with an upper bound called the *link delay* which is equal to $t$ for both classes.

Figure 5.1: Unit Storage Data Link automaton

### 5.1.3 Modelling Network Nodes

Next we specify node automata. We do so using a set that contains a node automaton for every node in the topology graph. For every edge incident to a node $u$, a node automaton $N_u$ must have interfaces to send and receive packets on the channels corresponding to that edge. However, we will go further and require that nodes obey a certain stylized convention in order to receive feedback from and send packets on links.

In the specification for a UDL if a packet $p$ is sent when the UDL already has a packet stored, then the new packet $p$ is dropped. We will prevent packets from being dropped by requiring that the sending node keep a corresponding *free* variable for the link that records whether or not the link is free to accept new packets. The sender sets the *free* variable to *true* whenever it receives a FREE action from the link. We require that the sender only send packets on the link when the *free* variable is *true*. Finally, whenever the sender sends a packet on the link, the sender sets its *free* variable to *false*.

We wish the interface to a UDL to stabilize to "good behavior" even when the sender and link begin in arbitrary states. Suppose the sender and the link begin in arbitrary states. Then we can have two possible problems. First, if $free = true$ but the UDL contains a packet, then the first packet sent by the sender can be dropped. However, it is easy to see that all subsequent packets will be accepted and delivered by the link. This is because after the first packet is sent, the sender will never set *free* to *true* unless it receives a FREE notification from the link. But a FREE notification is delivered to the sender only when the link is empty. The second possible problem is deadlock. Suppose that initially $free = false$ but the channel does not contain a packet. To avoid deadlock, the UDL specification ensures that the FREE action is enabled continuously whenever the link does not contain a packet.

Thus we will require that each sending node $u$ keep a corresponding $free_u[v]$ variable for each neighbor $v$. By our convention, $u$ can only send packets to $v$ when $free_u[v] = true$. Thus we will also require that $u$ enqueue packets that it wants to send to $v$ in an outbound queue for the link called $queue_u[v]$. When $free_u[v]$ becomes *true*, the packet at the head of the outbound queue is sent on the link. The use of the queuing and free disciplines is quite natural for a UDL; more importantly, these conventions make it easy to transform a node automaton to do local checking. This will become clearer in a few sections.

The preceding paragraphs motivate the following formal definition.

**Definition 5.1.4** *We say that an automaton $N$ is a node automaton for node $u$ in graph $G = (V, E, l)$ and with node delay $t$ if:*

- *$N$ has an output action* $\text{SEND}_{u,v}(p)$ *and input actions* $\text{RECEIVE}_{v,u}(p)$ *and* $\text{FREE}_{u,v}$ *for each $p \in P_{data}$ and for each neighbor $v$ of $u$.*
- *$N$ has a boolean variable $free_u[v]$ and a queue variable $queue_u[v]$ for every $v$ such that $(u, v) \in E$. The queue variable $queue_u[v]$ is a queue of bounded size consisting of packets drawn from the data packet alphabet $P_{data}$.*[1]
- *Actions of $N$ other than* $\text{SEND}_{u,v}(p)$ *can only change $queue_u[v]$ by adding packets (drawn from alphabet $P_{data}$) to the tail of $queue_u[v]$. Of course, such actions can leave the queue unchanged.*
- *The code for the output action* $\text{SEND}_{u,v}(p)$ *and the input action* $\text{FREE}_{u,v}$ *at node $N$ is as shown in Figure 5.2.*
- *Each* $\text{SEND}_{u,v}(p)$ *action in $N$ is in its own class with upper bound called the* node delay *that is equal to $t$ for all classes.*

In particular, note that every $\text{SEND}_{u,v}(p)$ action at a node is a locally controlled action. Also, in all the network automata described in this thesis, the transitions in automaton $N_u$ will only depend on the state of $u$, the leader function $l$, and the identities of the neighbors $v$ of $u$. In other words, $N_u$ will use only local information available to $u$ about the graph $G$. We prefer not to formalize this requirement as part of the definition of a network automaton. Instead, we will use this as an informal criterion to rule out trivial solutions (to some problems) in which $N_u$ encodes the entire graph.

[1]The reader may be puzzled by the fact that the links accept both data and control packets but the node automata only send data packets. In a few sections, we will create *augmented node automata* by adding actions to ordinary node automata. These extra actions are used to send and receive control packets for the purposes of local checking/correction.

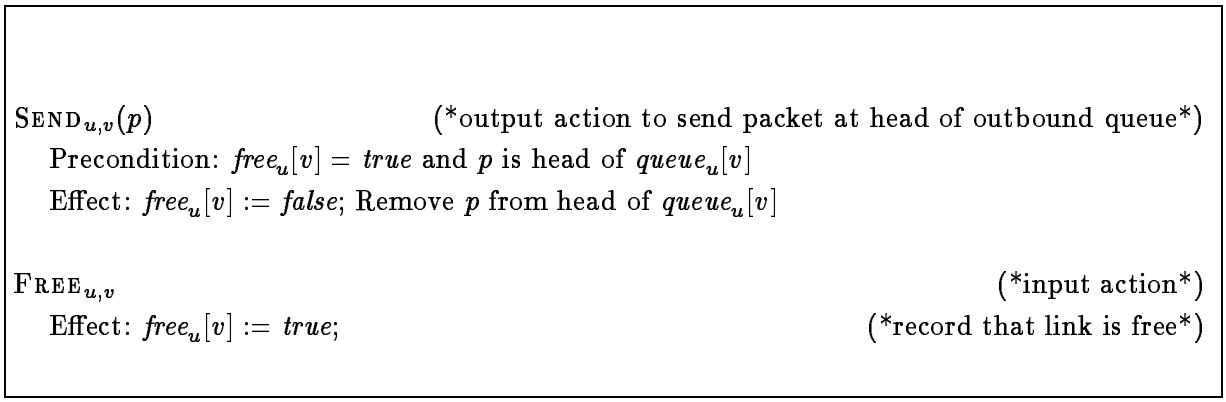



Figure 5.2: Code at a node automaton to send a data packet and to respond to a free signal from a link

### 5.1.4 Network Automata

Now we are ready to define a network automaton. Naturally, it is the composition of a set of node and channel automata.

**Definition 5.1.5** *Let $G = (V, E, l)$ be a topology graph. Let $N$ be a set containing exactly one node automaton $N_u$, for every $u \in V$ and such that each $N_u$ has node delay $t$. Let $C_{u,v}$ be a UDL for each $(u,v) \in E$ such that every UDL has link delay $\tilde{t}$. Then $Net(G, N, t, \tilde{t})$, the network automaton with node delay $t$ and link delay $\tilde{t}$, is the composition of the automata $N_u$ for all $u \in V$ with the automata $C_{u,v}$ for all $(u,v) \in E$.*

For the most part we will deal with network automata in which the node and link delays are fixed. Thus we let $t_n$ denote the *default node delay* and $t_l$ denote the *default link delay*. If we do not explicitly mention the node and link delays, then the node and link delays are assumed to be $t_n$ and $t_l$ respectively. Thus we will use $Net(G, N)$ to denote $Net(G, N, t_n, t_l)$. We will sometimes say that a time $t$ is a *constant* if $t = c_1 t_n + c_2 t_l$, where $c_1$ and $c_2$ are some real scalar constants. We use "constant time" to emphasize that the time does not depend on the size of the network but only on the node and link delays.

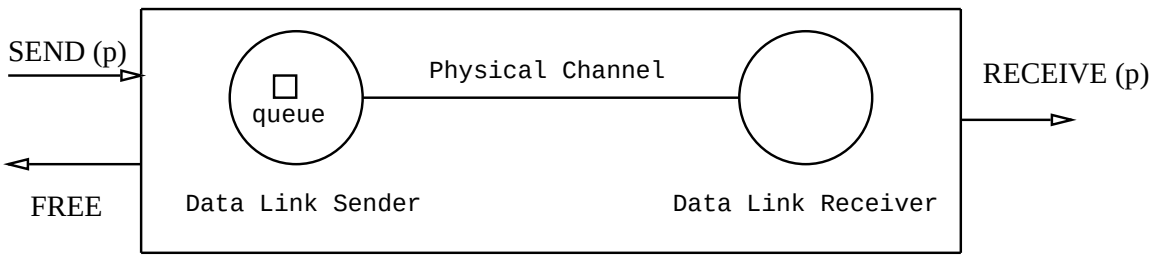



Figure 5.3: Implementing a UDL over a physical channel

## 5.2 Implementing Our Model in Real Networks

In a real network implementation, the physical channel connecting any two neighboring nodes would typically not be a UDL. For example, a telephone line connecting two nodes can often store more than one packet. The physical channel may also not deliver a free signal. Instead, an implementation can *construct* a Data Link protocol on top of the physical channel such that the resulting Data Link protocol *stabilizes* to the behaviors of a UDL (e.g. [AB89], [Spi88a]).

Figure 5.3 shows the structure of such a Data Link protocol over a physical link. The sender end of the Data Link protocol has a queue that can contain a single packet. When the queue is empty, the FREE signal is enabled. When a SEND($p$) arrives and the queue is empty, $p$ is placed on the queue; if the queue is full, $p$ is dropped. If there is a packet on the queue, the sender end constantly attempts to send the packet. When the receiving end of the Data Link receives a packet, the receiver sends an ack to the sender. When the sender receives an ack for the packet currently in the queue, the sender removes the packet from the queue.

If the physical channel is initially empty and the physical channel is FIFO (i.e., does not permute the order of packets), then a standard stop and wait or alternating bit protocol [BSW69] will implement a UDL. However, if the physical channel can initially store packets, then the alternating bit protocol is not stabilizing [Spi88a]. There are two approaches to creating a stabilizing stop and wait protocol. Suppose the physical channel can store at most $X$ packets in both directions. Then [AB89]

suggest numbering packets using a counter that has at least $X + 1$ values. Suppose instead that no packet can remain on the physical channel for more than a bounded amount of time. [Spi88a] exploits such a time bound to build a stabilizing Data Link protocol. The main idea is to use either numbered packets or timers to "flush" the physical channel of stale packets.

A stop and wait protocol is not very efficient over physical channels that have a high transmission speed and/or high latency. It is easy to generalize a UDL to a *Bounded Storage Data Link* or BDL that can store more than one packet. For instance, the FREE signal for a BDL should be modified to include the number of packets currently stored in the BDL. It is also easy to implement a BDL over a physical channel with either bounded storage or bounded delay using the techniques described in [AB89] and [Spi88a]. We prefer to use a UDL for the rest of this thesis as it provides a simple and elegant interface. However, the reader concerned about efficiency should be aware that all the protocols in this thesis can be modified (slightly) to work with BDLs.

Finally, there is one last concern about UDLs. We have seen that real implementations will use Data Links that *stabilize* to a UDL. However, in our model every link is assumed to actually be a UDL. Let $S$ be a network in which each link *stabilizes* to the behaviors of a UDL and such that every node automaton is a UIOA. Let $\tilde{S}$ be the same network except that every link is replaced by a UDL. Now a UDL is a UIOA and hence is suffix-closed. Thus a nice consequence of Theorem 3.5.7 is that if $\tilde{S}$ stabilizes to the behaviors in some problem $P$, then so does $S$. Thus, to prove a stabilization result about $S$ it suffices to prove the stabilization result about $\tilde{S}$. This is an example of why the modularity theorem (Theorem 3.5.7) is important.

## 5.3 Locality

In this section, we reconsider the definitions of local checkability that we introduced in Chapter 4. Our new definitions will be slightly more complex because of the presence of channels between nodes. *These new definitions will be used for the rest of the thesis.*

### 5.3.1 Link Subsystems and Local Predicates

Consider a network automaton with graph $G$. Roughly speaking, a property is said to be local to a subgraph $G'$ of $G$ if the truth of the property can be ascertained by examining only the components specified by $G'$. For now we will concentrate on *link subsystems* that consist of a pair of neighboring nodes $u$ and $v$ and the channels between them. In Chapter 10, we will discuss how our methods can be generalized to arbitrary subsystems.

In the following definitions, we fix a network automaton $\mathcal{N} = Net(G, N)$.

**Definition 5.3.1** *We define the* $(u, v)$ *link subsystem of* $\mathcal{N}$ *as the composition of* $N_u$, $C_{u,v}$, $C_{v,u}$, *and* $N_v$.

For any state $s$ of $\mathcal{N}$: $s|u$ denotes $s$ projected on to node $N_u$ and $s|(u, v)$ denotes $s$ projected onto $C_{u,v}$. Thus when $\mathcal{N}$ is in state $s$, the state of the $(u, v)$ subsystem is the 4-tuple: $(s|u, s|(u, v), s|(v, u), s|v)$.

A predicate $L$ of $\mathcal{N}$ is a subset of the states of $\mathcal{N}$. Let $(u, v)$ be some edge in graph $G$ of $\mathcal{N}$. A *local predicate* $L_{u,v}$ of $\mathcal{N}$ for edge $(u, v)$ is a subset of the states of the $(u, v)$ subsystem in $\mathcal{N}$. We use the word "local" because $L_{u,v}$ is defined in terms of the $(u, v)$ subsystem.

The following definition provides a useful abbreviation. It describes what it means for a local property to hold in a state $s$ of the entire automaton.

**Definition 5.3.2** *We say that a state* $s$ *of* $\mathcal{N}$ satisfies *a local predicate* $L_{u,v}$ *of* $\mathcal{N}$ *iff* $(s|u, s|(u, v), s|(v, u), s|v) \in L_{u,v}$.

We will make frequent use of the concept of a closed predicate. Intuitively, a property is closed if it remains true once it becomes true. In terms of local predicates:

**Definition 5.3.3** *A local predicate* $L_{u,v}$ *of network automaton* $\mathcal{N}$ *is closed if for all transitions* $(s, \pi, \tilde{s})$ *of* $\mathcal{N}$, *if* $s$ *satisfies* $L_{u,v}$ *then so does* $\tilde{s}$.

The following definitions provide two more useful abbreviations. The first gives a name to a collection of local predicates, one for each edge in the graph. The second,

the conjunction of a collection of "local properties", is the property that is true when all local properties hold at the same time. As in Chapter 4, we will require that the conjunction of the local properties is non-trivial – i.e., there is some global state that satisfies all the local properties.

**Definition 5.3.4** *$\mathcal{L}$ is a* link predicate set *for $\mathcal{N} = Net(G, N)$ if for each $(u, v) \in G$ there is some $L_{u,v}$ such that:*

- *If $(a, b, c, d) \in L_{u,v}$ then $(d, c, b, a) \in L_{v,u}$. (i.e., $L_{u,v}$ and $L_{v,u}$ are identical except for the way the states are written down.)*
- *$\mathcal{L} = \{L_{u,v}, (u, v) \in G\}$*
- *There is at least one state $s$ of $\mathcal{N}$ such that $s$ satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}$.*

**Definition 5.3.5** *The conjunction of a link predicate set $\mathcal{L}$ is the predicate $\{s : s$ satisfies $L_{u,v}$ for all $L_{u,v} \in \mathcal{L}\}$. We use $Conj(\mathcal{L})$ to denote the conjunction of $\mathcal{L}$.*

Note that $Conj(\mathcal{L})$ cannot be the null set by the definition of a link predicate set.

### 5.3.2 Local Checkability

Suppose we wish a network automaton $\mathcal{N}$ to satisfy some property. An example would be the property "all nodes have the same color". We can often specify a property of $\mathcal{N}$ formally using a predicate $L$ of $\mathcal{N}$. Intuitively, $\mathcal{N}$ can be locally checked for $L$ if we can ascertain whether $L$ holds by checking all link subsystems of $\mathcal{N}$. The motivation for introducing this notion is performance: in a distributed system we can check all link subsystems in parallel in constant time. We formalize the intuitive notion of a locally checkable property as follows.

**Definition 5.3.6** *A network automaton $\mathcal{N}$ is locally checkable for predicate $L$ using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{L}$ is a link predicate set for $\mathcal{N}$ and $L \supseteq Conj(\mathcal{L})$.*
- *Each $L_{u,v} \in \mathcal{L}$ is closed.*

The first item in the definition requires that $L$ holds if a collection of local properties all hold. The second item is perhaps more surprising. It requires that each local property also be closed.

We add this extra requirement because in an asynchronous distributed system it appears to be impossible to check whether an arbitrary local predicate holds *all* the time. What we can do is to "sample" the local subsystem periodically to see whether the local property holds. Suppose the network automaton consists of three nodes $u$, $v$ and $w$ and such that $v$ is the neighbor of both $u$ and $w$. Suppose the property $L$ that we wish to check is the conjunction of two local predicates $L_{u,v}$ and $L_{v,w}$. Suppose further that exactly one of the two predicates is always false, and the predicate that is false is constantly changing. Then whenever we "check" the $(u,v)$ subsystem we might find $L_{u,v}$ true. Similarly whenever we "check" the $(v,w)$ subsystem we might find $L_{v,w}$ true. Then we may never detect the fact that $L$ does not hold in this execution. We avoid this problem by requiring that $L_{u,v}$ and $L_{v,w}$ be closed.

### 5.3.3 Local Correctability

The motivation behind local checking was to efficiently ensure that some property $L$ holds for network automaton $\mathcal{N}$. We would also like to *efficiently* correct $\mathcal{N}$ to make the property true. We have already set up some plausible conditions for local checking. Can we find some plausible conditions under which $\mathcal{N}$ can be *locally corrected*?

To this end we define a local reset function $f$. This is a function with three arguments: the first argument is a node say $u$, the second argument is any state of node automaton $N_u$, and the second argument is a neighbor $v$ of $u$. The function produces a state of the node automaton corresponding to the first argument. Let $s$ be a state of $\mathcal{N}$; recall that $s|u$ is the state of $N_u$. Then $f(u, s|u, v)$ is the state of $N_u$ obtained by applying the local reset function at $u$ with respect to neighbor $v$. We will abuse notation by omitting the first argument when it is clear what the first argument is. Thus we prefer to write $f(s|u, v)$ instead of the more cumbersome $f(u, s|u, v)$.

We will insist that $f$ meet two requirements so that $f$ can be used for local correction (Definition 5.3.7).

Assume that the property $L$ holds if a local property $L_{u,v}$ holds for every edge $(u,v)$. The first requirement is that if any $(u,v)$ subsystem does not satisfy $L_{u,v}$, then

applying $f$ to both $u$ and $v$ should result in making $L_{u,v}$ hold. More precisely, let us assume that by some magic we have the ability to simultaneously:

- Apply $f$ to $N_u$ with respect to $v$;
- Apply $f$ to $N_v$ with respect to $u$;
- Remove any packets stored in channels $C_{u,v}$ and $C_{v,u}$.

Then the resulting state of the $(u,v)$ subsystem should satisfy $L_{u,v}$. Of course, in a real distributed system such simultaneous actions are clearly impossible. However, we will achieve essentially the same effect by applying a so-called "reset" protocol to the $(u,v)$ subsystem. We will describe a stabilizing local reset protocol for this purpose in the next section.

The first requirement allows nodes $u$ and $v$ to correct the $(u,v)$ subsystem if $L_{u,v}$ does not hold. But other subsystems may be correcting at the same time! Since subsystems overlap, correction of one subsystem may invalidate the correctness of an overlapping subsystem. For example, the $(u,v)$ and $(v,w)$ subsystems overlap at $v$. If correcting the $(u,v)$ subsystem causes the $(v,w)$ subsystem to be incorrect, then the correction process can "thrash". To prevent thrashing, we add a second requirement. In its simplest form, we might require that correction of the $(u,v)$ subsystem leaves the $(v,w)$ subsystem correct *if* the $(v,w)$ subsystem was correct in the first place.

However, there is a more general definition of a reset function $f$ that turns out to be useful. Recall that we wanted to avoid thrashing that could be caused if correcting a subsystem causes an adjacent subsystem to be incorrect. Informally, let us say that the $(u,v)$ subsystem depends on the $(v,w)$ subsystem if correcting the $(v,w)$ subsystem can invalidate the $(u,v)$ subsystem. If this dependency relation is cyclic, then thrashing can occur. On the other hand if the dependency relation is acyclic then the correction process will eventually stabilize. Such an acyclic dependency relation can be formalized using a partial order $<$ on *unordered* pairs of nodes: informally, the $(u,v)$ subsystem depends on the $(v,w)$ subsystem if $\{v,w\} < \{u,v\}$.

Using this notion of a partial order, we present the formal definition of a local reset function:

**Definition 5.3.7** *We say $f$ is a local reset function for network automaton $\mathcal{N} = Net(G, N)$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$, if for any state $s$ of $\mathcal{N}$ and any edge $(u, v)$ of $G$:*

- **Correction:** $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$.
- **Stability:** *For any neighbor $w$ of $v$,*

  *If $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$ and $\{v,w\} \not< \{u,v\}$ then*
  $(s|u, s|(u,v), s|(v,u), f(s|v,w)) \in L_{u,v}$.

Notice that in the special case where all the link subsystems are independent, no edge is "less" than any other edge in the partial order.

Using the definition of a reset function, we can define what it means to be locally correctable.

**Definition 5.3.8** *A network automaton $\mathcal{N}$ is* locally correctable *to $L$ using link predicate set $\mathcal{L}$, local reset function $f$, and partial order $<$ if:*

- *$\mathcal{N}$ is locally checkable for $L$ using $\mathcal{L}$.*
- *$f$ is a local reset function for $\mathcal{N}$ with respect to $\mathcal{L}$ and $<$.*

Intuitively, if we have a reset function $f$ with partial order $<$ we can expect the local correction to stabilize in time proportional to the maximum chain length in the partial order. Recall that a chain is a sequence $a_1 < a_2 < a_3 \ldots < a_n$. Thus the following piece of notation is useful.

**Definition 5.3.9** *For any partial order $<$, height$(<)$ is the length of the maximum length chain in $<$.*

## 5.4 Local Correction Theorem

### 5.4.1 Overview

In the previous section, we set up plausible conditions under which a network automaton can be locally corrected to achieve a property $L$. We claimed that these conditions

could be exploited to yield local correction. In this section we make these claims precise. We show how to take a network automaton $\mathcal{N}$ that can be locally corrected to $L$, and *transform* it into a new automaton $\mathcal{N}^+$. The new automaton $\mathcal{N}^+$ has the property that in all its executions, property $L$ holds after a bounded amount of time. More precisely, $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}|L$ in a bounded amount of time. The next subsection contains a formal statement of this result.

To transform $\mathcal{N}$ into $\mathcal{N}^+$ we will add actions and states to $\mathcal{N}$. These actions will be used to send and receive *snapshot packets* (that will be used to do local checking on each link subsystem) and *reset* packets (that will be used to do local correction on each link subsystem). For every link $(u,v)$, the leader $l(u,v)$ initiates the checking and correction.[2]

### 5.4.2 Precise Statement of the Result

To state the result formally, we need the following definitions. First, when we augment a network automaton the resulting automaton should have the same topology and also be an unrestricted automaton (UIOA) that can start in any state. The topology restriction rules out trivial "centralized" solutions. We also require that the links remain UDLs. To formalize these requirements, we define a new type of automaton.

**Definition 5.4.1** *Let $G = (V,E,l)$ be a topology graph. An automaton for graph $G$ is the composition of an automaton for each $u \in V$, together with $C_{u,v}$ for each $(u,v) \in E$. We assume that all automata being composed are compatible.*

Notice that any network automaton for graph $G$ is also an automaton for graph $G$. However, an automaton for graph $G$ need not be a network automaton because a network automaton has additional constraints (such as having outbound queues and free variables for each link) on the node automata.

A UIOA for graph $G$ is an automaton for graph $G$ that is also a UIOA. Recall that we used $A|L$ to denote the automaton identical to $A$ except that its start states belong to set $L$. The following piece of shorthand is useful for a concise statement of the theorem.

[2] Without this assumption, we have to complicate the code and proof to deal with simultaneous checking and correction actions by both ends of a link. This can actually be done, thereby getting rid of the requirement for a leader on links. But it isn't worth the increased complexity.

**Definition 5.4.2** *Let $\mathcal{N}$ denote a network automaton. We will use $\mathcal{N}(t)$ to denote the automaton that is identical to $\mathcal{N}$ except that the link and node delays in $\mathcal{N}(t)$ are equal to $t$.*

Now (finally!) we can state our theorem. Intuitively, it states that if $\mathcal{N}$ is locally correctable to $L$ using local reset function $f$ and partial order $<$, then we can transform $\mathcal{N}$ into $\mathcal{N}^+$ such that $\mathcal{N}^+$ satisfies the following property: in time proportional to $height(<)$, every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which $L$ holds and in which the node and link delays are increased by some constant factor.

**Theorem 5.4.3 Local Correction:** *Consider any network automaton $\mathcal{N} = Net(G, N)$ that is* locally correctable *to $L$ using link predicate set $\mathcal{L}$, local reset function $f$, and partial order $<$. Then there exists some $\mathcal{N}^+$ that is a UIOA for graph $G$ and constants $c$ and $\tilde{c}$ such that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$ in time $\tilde{c} \cdot height(<)$.*

### 5.4.3 Overview of the Transformation Code

For those familiar with snapshot protocols, the structure of our local snapshot protocol is slightly different from the well-known Chandy-Lamport snapshot protocol [CL85]. It is easy to show that the Chandy-Lamport scheme cannot be used without modifications over unit storage links. Briefly, the reason is as follows. The correctness proof of the algorithm in [CL85] is based on reordering executions while preserving causality constraints. The only causality constraint for a link in [CL85] is that any action that sends a packet $p$ on link $L$ cannot be reordered to come after an action that receives $p$ on link $L$. However, a UDL has an additional causality constraint. A free signal delivered by link $L$ after delivering packet $p$ cannot be reordered to come before the action that delivers packet $p$. The Chandy-Lamport scheme was not designed to incorporate this extra causality constraint. As a result, if the Chandy-Lamport scheme is used unmodified over a network with UDLs, the snapshot may (incorrectly) return a state in which there is more than one packet on a UDL!

Our local snapshot/reset protocol works roughly as follows. Consider a $(u, v)$ subsystem. Assume that $l(u, v) = u$ – i.e., $u$ is the leader on link $(u, v)$. A single snapshot or reset phase has the structure shown in Fig 5.4.

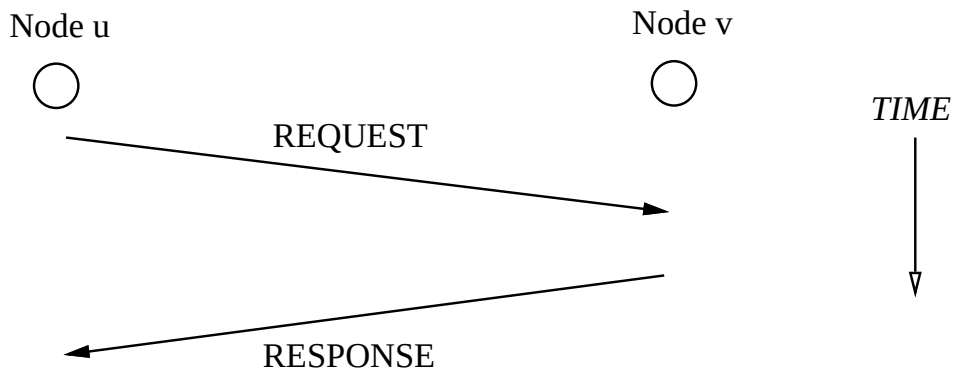



Figure 5.4: The structure of a single phase of the local snapshot/reset protocol

A single phase of either a snapshot or reset procedure consists of $u$ sending a request that is received by $v$, followed by $v$ sending a response that is received by $u$. During a phase, node $u$ sets a flag ($phase_u[v]$) to indicate that it is checking/correcting the $(u, v)$ subsystem. While this flag is set, *no packets other than request packets can be sent on link* $C_{u,v}$. Since a phase completes in constant time, this does not delay the data packets by more than a constant factor.

In what follows, we will use the basic state at a node $u$ to mean the part of the state at $u$ "corresponding" to automaton $N_u$. To do a snapshot, node $u$ sends a snapshot request to $v$. A snapshot request is identified by a *mode* variable in the request packet that carries a *mode* of *snapshot*. If $v$ receives a request with a *mode* of *snapshot*, Node $v$ then records its basic state (say $s$) and sends $s$ in a response to $u$.

When $u$ receives the response, it records its basic state (say $r$). Node $u$ then records the state of the $(u, v)$ subsystem as $x = (r, nil, nil, s)$. If $x \notin L_{u,v}$ (i.e., local property $L_{u,v}$ does not hold) then $u$ initiates a reset.

To do a reset, node $u$ sends a reset request to $v$. A reset request is identified by a *mode* variable in the request packet that carries a *mode* of *reset*. Recall that $f$ denotes the local reset function. After $v$ receives the request, $v$ changes its basic state to $f(v, s, u)$, where $s$ is the previous value of $v$'s basic state. Node $v$ then sends a response to $u$. When $u$ receives the response, $u$ changes its basic state to $f(u, r, v)$, where $r$ is the previous value of $u$'s basic state.

Of course, the local snapshot and reset protocol must also be stabilizing. However, the protocol we just described informally may fail if requests and responses are not

properly matched. This can happen, for instance, if there are spurious packets in the initial state of $\mathcal{N}^+$. To make the snapshot and reset protocols stabilizing, we number all request and response packets. Thus each request and response packet carries a number *count*. Also, the leader $u$ keeps a variable $count_u[v]$ that $u$ uses to number all requests sent to $v$ within a phase. At the end of the phase, $u$ increments $count_u[v]$. Similarly, the responder $v$ keeps a variable $count_v[u]$ in which $v$ stores the number of the last request it has received from $u$. Node $v$ weeds out duplicates by only accepting requests whose number is not equal to $count_u[v]$.

Clearly the *count* values can be arbitrary in the initial state and the first few phases may not work correctly. However, numbering and a few easy checks ensure that in constant time a response will be properly matched to the correct request. Because the links are unit storage, we will see that a space of 4 numbers is sufficient. Our use of numbering is taken from the stabilizing global snapshot protocol of [KP90]. However, our protocol is simpler and more efficient because we are restricted to a single link subsystem.

Besides properly matching requests and responses, we must also avoid deadlock when the local snapshot/reset protocol begins in an arbitrary state. To do so, when $phase_u[v]$ is *true* (i.e., $u$ is in the middle of a phase), $u$ continuously sends requests. Since $v$ weeds out duplicates this does no harm and also prevents deadlock. Similarly, $v$ continuously sends responses to the last request $v$ has received. Once the responses begin to be properly matched to requests, this does no harm, because $u$ discards such duplicate responses.

An irritating issue that we have to deal with in creating $\mathcal{N}^+$ is the issue of scheduling packets to be sent on links. Notice that the checking and correction protocols are going on concurrently with the protocol corresponding to $\mathcal{N}$. To make Theorem 5.4.3 work, we need to ensure that any data packets that are placed on the queue for channel $C_{u,v}$ are sent in constant time. On the other hand, the checking process also needs to send request and response packets that are encoded as control packets.

We build a simple stabilizing scheduler that ensures fair access to the link for each of three packet types: requests, responses and data packets. First we notice that at the leader end of a link, only requests and data packets need to be sent. At the other end, only responses and data packets need to be sent.

Consider the leader end of a link first. Suppose $l(u,v) = u$. We know that data

packets should never be sent while a snapshot or reset phase is in progress. Now, we have a variable $phase_u[v]$ at $u$ which is set to true whenever a phase is in progress. The phase ends when a matching response is received and $phase_u[v]$ is set to false. At the end of a phase, the oldest data packet is sent, and a new phase is begun after the data packet is sent. The scheduler is stabilizing because if there is no data packet waiting to be sent, we allow a new phase to begin immediately after the previous phase ends. The net effect is that if there are data packets waiting, we send one data packet between consecutive checking/correction phases. If a problem is discovered during a snapshot phase, we do not do a reset until the next phase, after sending any waiting data packet.

Now consider the end of a link that is not the leader. Suppose $l(u,v) = v$. To give "fair turns" to the response packets we use a variable $turn_u[v]$ that has only two values *data* (for data packets), and *response* (which is the value of *turn* for response packets). No packet can be sent until either its turn arrives or there is no packet of the other type. After a packet of a particular type is sent, the turn is "toggled" to the other type.

### 5.4.4 Constructing Augmented Automata: Formal Description

To transform $\mathcal{N}$ into $\mathcal{N}^+$ we will show how to transform each node automaton $N_u$ in $\mathcal{N}$ into a new, augmented node automaton $N_u^+$. Finally, $\mathcal{N}^+$ is the composition of the new node automata and the (unchanged) channel automata.

Assume that network automaton $\mathcal{N} = Net(G,N)$ can be *locally corrected* to $L$ using link predicate set $\mathcal{L} = \{L_{u,v}\}$ and local reset function $f$. We create $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$ by adding states and actions to each node automaton $N_u$ as follows:

- We add the following new variables and their domain specifications, to the state of $N_u$, one for each neighbor $v$ of $u$ as shown below:.
  - $count_u[v] \in \{0 \ldots 3\}$ (used to number request and response packets to ensure proper matching. The magic number 3 arises because a link subsystem can store at most three distinct counter values on the sending link, receiving link, and at the receiver node.)

- $mode_u[v] \in \{reset, snapshot\}$ (used to keep track of whether the current phase is a snapshot or reset phase.)
- $phase_u[v]$ is a Boolean (set to true during a reset or snapshot phase. It is used by the leader node to inhibit data packets from being sent during a phase.)
- $freeq_u[v]$ is a Boolean (set to true after *any* packet is sent and set to false after any packet is removed from the link. The original variable $free_u[v]$ will be set to true after a data packet is sent and set to false after a data packet is removed from the link. We could have optimized by using just one free variable but keeping two variables makes the projection and the proof easier.)
- $turn_u[v] \in \{response, data\}$ (used to keep track of whether a data packet or a response packet has the next turn to be sent.)

- We add two new packet types to $\mathcal{N}^+$. Recall that there are two basic types of packets, data packets and control packets. We will encode request and response packets as control packets. We will use the symbols $p_{data}$, $p_{req}$, and $p_{resp}$ to denote data, request and response packets respectively. The format of a data packet is defined by automaton $\mathcal{N}$. The encoding of the other two packets is:

  **Request** : $(Control, Request, count, mode)$, where *count* is an integer from $0 \ldots 3$, and *mode* is either *reset* or *snapshot*.

  **Response** : $(Control, Response, count, mode, node_state)$, where *count* is an integer from $0 \ldots 3$, *mode* is either *reset* or *snapshot*, and *node_state* is a state of a node automaton $N_u$ in $\mathcal{N}$.

  As usual, we use record notation to extract fields of a packet. Thus $p_{req}.count$ is the *count* field in a request packet.

- We modify the $\text{SEND}_{u,v}(p)$ (for data packets $p$) and $\text{FREE}_{u,v}$ actions of the original automaton $N_u$ as shown in Figure 5.5. This code contains modifications for both the leader and responder ends in one piece of code; however, some parts of the code applies only to leaders and some parts only apply to responders.

  If node $u$ is the leader on link $(u, v)$, then we enable sending data packets only when $phase_u[v] = false$ indicates a phase is not in progress. After the data packet

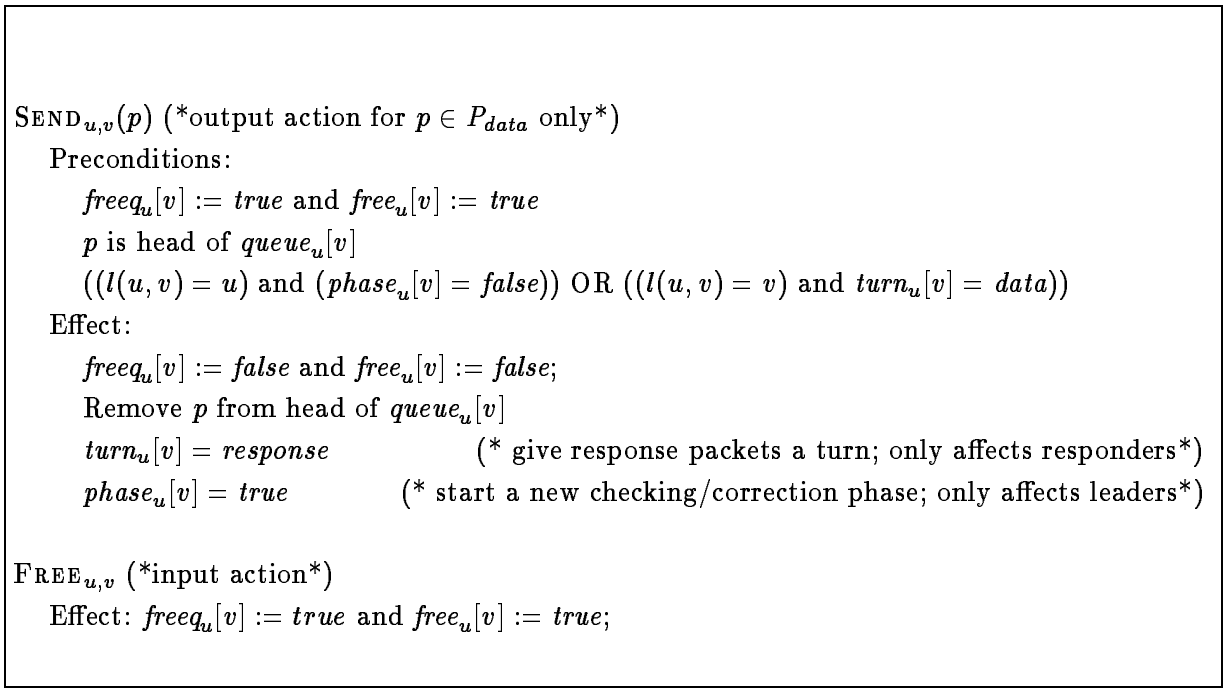



Figure 5.5: Code for the modified SEND$_{u,v}(p)$ actions at a modified node $N_u^+$.

is sent, we set $phase_u[v] = true$. If node $u$ is not the leader on link $(u,v)$, then we enable sending data packets only when $turn_u[v] = data$. Also immediately after sending a data packet, we set $turn_u[v] = response$, which allows the response packets to get a fair turn. We use the $freeq_u[v]$ variable to keep track of whether there is some packet (either data or control) on $C_{u,v}$ while $free_u[v]$ keeps track of whether there is a data packet on $C_{u,v}$. The code appears to have some redundant checks (for example, the code checks both free variables before sending a data packet) but these extra checks do make the proof easier.

- We add the actions shown in Figures 5.6 and 5.7 to $N_u$ for each neighbor $v$ of $u$. These actions only apply to control packets. We use the following notation. Let $s$ denote the current state of $\mathcal{N}^+$ when an action is performed. Let $s|u$ denote the current state of $\mathcal{N}_u^+$ *projected onto the original automaton* $N_u$. In order to project $s$ to $s|u$ we do the following. All variables of $N_u$ take the same values as the corresponding variables in $N_u^+$.

$\text{SEND}_{u,v}(p_{req})$ (*output action: $u$ repeatedly sends a request till it gets a response*)
Precondition:
$l(u,v) = u$ (*$u$ is the leader of link subsystem*)
($phase_u[v] = true$) or ($queue_u[v]$ is empty) (*phase in progress or no data packets waiting*)
$freeq_u[v] = true$ (* no packet in transit on link to $v$ *)
$p_{req}.count = count_u[v]$; (*count in packet is count of phase*)
$p_{req}.mode = mode_u[v]$; (*mode in packet is mode of phase*)
Effect:
$freeq_u[v] = false$ (* set to false until link says it is free*)
$phase_u[v] := true$; (*remains true until matching response returns*)

$\text{RECEIVE}_{v,u}(p_{req})$ (*input action, receive request at $u$ from $v$*)
Effect:
If $p_{req}.count \neq count_u[v]$ and $l(u,v) = v$ then (* not a duplicate or invalid packet*)
$count_u[v] := p_{req}.count$; (*remember count*)
$mode_u[v] := p_{req}.mode$; (*remember mode*)

Figure 5.6: Code to send and receive request packets at node $u$.

When we apply $f$ to the projected state by setting $s|u$ to $f(s|u,v)$ we affect only the projected variables. Thus, for instance, the value of $freeq_u[v]$ remains unchanged.

- We add two extra classes for every neighbor $v$ of $u$ to $N_u$. Each output action of the form $\text{SEND}_{u,v}(Control, Request, *)$ is added to a *new* partition class. Similarly, each output action of the form $\text{SEND}_{u,v}(Control, Response, *)$ is added to a *new* partition class. The time associated with all new classes is still the node delay $t_n$.

- We hide all actions of $\mathcal{N}^+$ that are not actions of $\mathcal{N}$.

$\textsc{Send}_{u,v}(p_{resp})$ (*output action: $u$ repeatedly sends a response to last request*)

Precondition:

$l(u,v) = v$ (*$u$ is not the leader of link subsystem*)

($turn = response$) or ($queue_u[v]$ is empty) (*response's turn or no data packets waiting*)

$freeq_u[v] = true$ (* no packet in transit on link to $v$ *)

$p_{resp}.count = count_u[v]$;

If $mode_u[v] = snapshot$ then $p_{resp}.node_state = s|u$ else $p_{resp}.node_state := f(s|u,v)$

Effect:

If $mode_u[v] = reset$ then $s|u := f(s|u,v)$ (*reset node $u$'s state locally*)

$mode_u[v] := snapshot$ (* return to default mode of snapshot*)

$turn_u[v] := data$ (*give data packets a turn*)

$freeq_u[v] := false$ (* set to false until link says its free*)

$\textsc{Receive}_{v,u}(p_{resp})$ (*input action to receive response at $u$ from $v$*)

Effect:

If ($count_u[v] = p_{resp}.count$) and ($phase_u[v] := true$) and ($l(u,v) = u$) then

If $mode_u[v] = snapshot$ then

If $(s|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$ then $mode_u[v] := reset$

Else if $mode_u[v] = reset$ then $s|u = f(s|u,v)$ (*reset node $u$'s state locally*)

$phase_u[v] := false$; (*end of phase*)

$count_u[v] := (count_u[v] + 1) \bmod 4$;

Figure 5.7: Code to send and receive response packets at node $u$.

## 5.5 Intuitive Proof of Local Correction Theorem

In the previous section we described how to transform a locally correctable automaton $\mathcal{N}$ into an augmented automaton $\mathcal{N}^+$ We have to prove that in time proportional to the height of the partial order every behavior of $\mathcal{N}^+$ (described in the last section) is a behavior of $\mathcal{N}$ in which all local predicates hold.

The basic intuition behind the proof is sketched in Figure 5.8 and Figure 5.9. Consider some $(u, v)$ subsystem in which $u$ is the leader. We describe the intuition behind the use of a counter to ensure proper request-response matching, and the intuition behind the local snapshot and reset procedures.

### 5.5.1 Intuition Behind Counter Based Matching

Recall that in the augmented automaton, both local snapshots and local responses are implemented using a request-response protocol. As we will see below, both the local snapshot and reset procedures will only work correctly if the response from $v$ is sent following the receipt of the request at $u$. The diagram on the left of Figure 5.8 shows a scenario in which requests are matched incorrectly to "old" responses that were sent in previous phases. Thus we need each phase to eventually follow the structure shown in the right of Figure 5.8.

The code of the augmented automaton ensures that correct matching will occur after at most 5 phases by numbering requests and responses. Thus the code uses a counter in the range 0..3. The significance of the number 3 will be seen below. The sender $u$ keeps a counter $count_u[v]$ to number requests and the receiver $v$ keeps a counter $count_v[u]$ to number responses. Node $v$ accepts a new request numbered $c$ only if $c \neq count_v[u]$. When $v$ accepts this request, $v$ also sets $count_v[u]$ to be equal to $c$. Finally, node $u$ accepts a response numbered $c$ only if $c = count_u[v]$. When $u$ accepts this response, $u$ also increments $count_u[v]$ mod 4. This is implemented in the code and illustrated in the diagram on the right of Figure 5.8.

In the first two phases, packets sent by $u$ and $v$ may be dropped by the links because the links may have packets stored in the initial state. However, it is easy to see from the properties of a UDL, that after at most two phases, the links in both directions are *drop-free* – i.e., any packets sent from that point on will be delivered. Thus we need

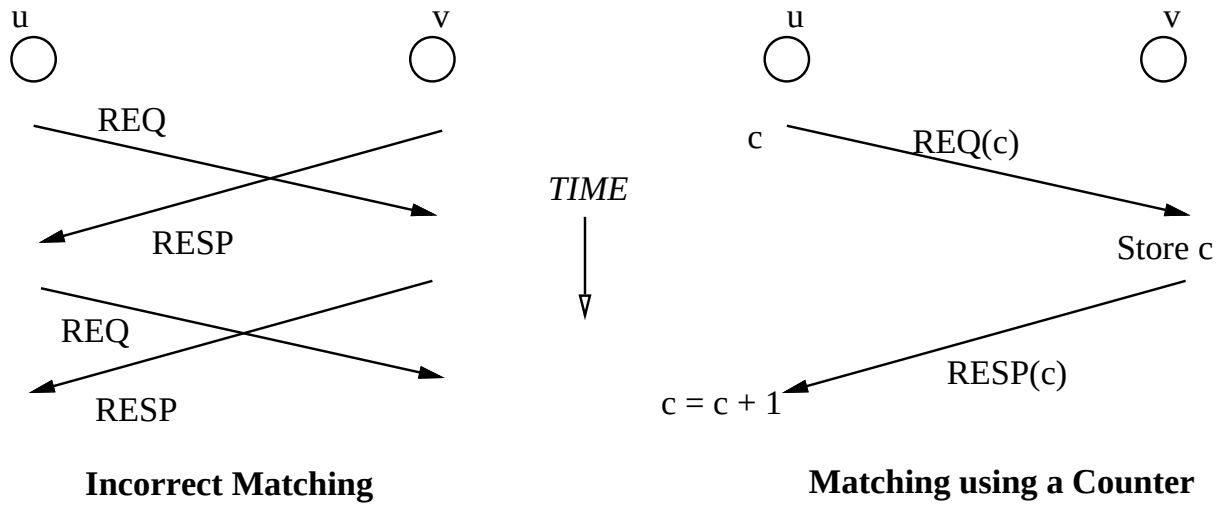



Figure 5.8: Using counter flushing to ensure that request-response matching will work correctly within a small number of phases.

to show that within the next three phases, the request-response matching will begin to work correctly. This follows from a simple paradigm that we call *counter flushing*.

Counter flushing is a general technique that is quite versatile. Chapter 10 describes some more applications of counter flushing. In our case, the counter flushing argument runs as follows:

- There can be at most 3 counter values stored in the two links (i.e., the link from $u$ to $v$ and the link from $v$ to $u$) and the receiver. This is the significance of the number 3.
- The sender retransmits till it gets a response and so the sender counter will keep being incremented.
- Within 3 increments of the sender counter, the sender counter will reach a "fresh" counter value that is not present in the links and the receiver. This is because the counter space has 4 values, and new counter values can only be created by the sender.
- Suppose the sender sends a request numbered $c$ where $c$ is a fresh value that is not present in the receiver or on the two links. Then when a later response numbered

$c$ is received, this response is a matching response because no aliasing is possible. Also, by the time the sender receives the matching response, the only counter value stored in the two links and the receiver is $c$. In other words, a freshly numbered request and its matching response will "flush" the $(u, v)$ subsystem of outdated counter values. Hence the term counter *flushing*.

- After all old counter values have been flushed, we say that the $(u, v)$ subsystem is *clean*. It is easy to show that all subsequent phases follow the structure shown in the diagram on the right of Figure 5.8.

### 5.5.2 Intuition Behind Local Snapshots

The diagram on the left of Figure 5.9 shows why a snapshot works correctly if the response from $v$ is sent following the receipt of the request at $u$. Let $a'$ and $b$ be the state of nodes $u$ and $v$ respectively just *before* the response is *sent*. Let $a$ and $b'$ be the state of nodes $u$ and $v$ respectively just *after* the response is delivered. This is sketched in Figure 5.9.

From the code we know that node $u$ does not send any data packets to $v$ during a phase. Also $v$ cannot send another data packet to $u$ from the time the response is sent until the response is delivered. This is because the link from $v$ to $u$ is a UDL that will not give a free indication until the response is received. Recall that *nil* denotes the absence of any packet on a link. Thus the state of the $(u, v)$ subsystem just before the response is sent is $(a', nil, nil, b)$. Similarly, the state of the $(u, v)$ subsystem just after the response is delivered is $(a, nil, nil, b')$.

We claim that it is *possible* to construct some other execution of the $(u, v)$ subsystem which starts in state $(a', nil, nil, b)$, has an intermediate state equal to $(a, nil, nil, b)$ and has a final state equal to $(a, nil, nil, b')$. This is because we could have first applied all the actions that changed the state of node $u$ from $a'$ to $a$, which would cause the $(u, v)$ subsystem to reach the intermediate state. Next, we could apply all the actions that changed the state of node $v$ from $b$ to $b'$, which will cause the $(u, v)$ subsystem to reach the final state. Note that this construction is only possible because $u$ and $v$ do not send data packets to each other between the time the response is sent and until the time the response is delivered.

Thus the state $(a, nil, nil, b)$ recorded by the snapshot is a *possible* successor of the

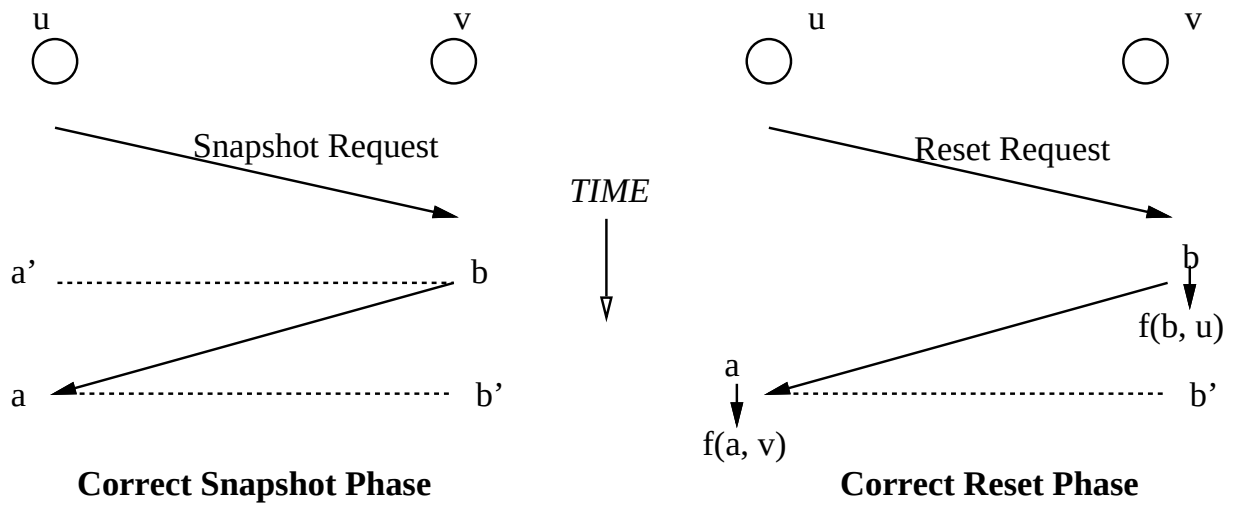



Figure 5.9: Local Snapshots and Resets work correctly if requests and responses are properly matched.

state of $(u, v)$ subsystem when the response is sent. The recorded state is also a a *possible* predecessor of the state of $(u, v)$ subsystem when the response is delivered. But $L_{u,v}$ is a closed predicate – it remains true once it is true. Thus if $L_{u,v}$ was true just before the response was sent, then the state recorded by the snapshot must also satisfy $L_{u,v}$. Similarly, if $L_{u,v}$ is false just after the response is delivered, then the state recorded by the snapshot *does not* satisfy $L_{u,v}$. Thus the snapshot detection mechanism *will not produce false alarms* if the local predicate holds at the start of the phase. Also the snapshot mechanism *will detect a violation* if the the local predicate does not hold at the end of the phase.

### 5.5.3 Intuition Behind Local Resets

The diagram on the right of Figure 5.9 shows why a local reset works correctly if the response from $v$ is sent following the receipt of the request at $u$. Let $b$ be the state of node $v$ just *before* the response is *sent*. Let $a$ and $b'$ be the state of nodes $u$ and $v$ respectively just *before* the response is delivered. This is sketched in Figure 5.9.

The code for an augmented automaton will ensure that just after the response is sent, node $v$ will locally reset its state to $f(b, u)$. Similarly, immediately after it receives the response, node $u$ will locally reset its state to $f(a, v)$. Using similar

arguments to the ones used for a snapshot, we can show that there is some execution of the $(u, v)$ subsystem which begins in the state $(f(a, v), nil, nil, f(b, u))$ and ends in the state $(f(a, v), nil, nil, b')$. But the latter state is the state of the $(u, v)$ subsystem immediately after the response is delivered. But we know, from the correction property of a local reset function, that $(f(a, v), nil, nil, f(b, u))$ satisfies $L_{u,v}$. Since $L_{u,v}$ is a closed predicate, we conclude that $L_{u,v}$ holds at the end of the reset phase.

### 5.5.4 Intuition Behind Local Correction Theorem

We can now see intuitively why the augmented automaton will ensure that all local predicates hold in time proportional to the height of the partial order. Consider a $(u, v)$ subsystem where $\{u, v\} \not< \{w, x\}$ for any pair of neighbors $w, x$ – i.e., $\{u, v\}$ is a minimal element in the partial order. Then, within 5 phases of the $(u, v)$ subsystem the request-response matching will begin to work correctly. If the sixth phase of the $(u, v)$ subsystem is a snapshot phase, then either $L_{u,v}$ will hold at the end of the phase or the snapshot will detect a violation. But in the latter case, the seventh phase will be a reset phase which will cause $L_{u,v}$ to hold at the end of the seventh phase.

But once $L_{u,v}$ remains true, it remains true. This is because $L_{u,v}$ is a closed predicate of the original automaton $\mathcal{N}$ and the only extra actions we have added to $\mathcal{N}^+$ that can affect $L_{u,v}$ are actions to locally reset a node using the reset function $f$. But by the stability property of a local reset function, any applications of $f$ at $u$ with respect to some neighbor other than $v$ cannot affect $L_{u,v}$. Similarly, any applications of $f$ at $v$ with respect to some neighbor other than $u$ cannot affect $L_{u,v}$. Thus in constant time, the local predicates – corresponding to link subsystems that are minimal elements in the partial order – will become and remain true.

Now suppose that the local predicates for all subsystems with height $\leq i$ hold from some state $s_i$ onward. By similar arguments, we can show that in constant time after $s_i$, the local predicates for all subsystems with height $i + 1$ become and remain true. Once again, the argument depends crucially on the stability property of a local reset function. The intuition is that applications of the local reset function to subsystems with height $\leq i$ do not occur after state $s_i$. But these are the only actions that can falsify the local predicates for subsystems with height $i + 1$. The net result is that all local predicates become and remain true within time proportional to the height of the partial order $<$.

## 5.6 Formal Proof of Local Correction Theorem

The reader who is satisfied with the "intuitive proof" given above should skip this section. However, there are a number of details glossed over in the intuitive proof that are spelled out in the formal proof. The formal proof also provides a good example of the proof techniques of Chapter 3. Formal proofs for other major theorems in this thesis like the Tree Correction theorem (Chapter 6) and the Global Correction Theorem (Chapter 8) can be constructed along similar lines.

### 5.6.1 Overview of Formal Proof

We wish to prove the local correction theorem, Theorem 5.4.3. Thus we have to prove that in time proportional to $height(<)$, every behavior of $\mathcal{N}^+$ will "look like" a behavior of $\mathcal{N}$ in which $L$ holds and in which the node and link delays are increased by some constant factor.

The formal proof is based on the proof technique described in Chapter 3 in Lemma 3.4.2. Thus the proof consists of two major parts:

- We first define a predicate $Q$ (more details below) and show that any execution of $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$. We prove this using the Execution Convergence theorem, Theorem 3.4.5.
- We show that any behavior of $\mathcal{N}^+|Q$ is also a behavior of $\mathcal{N}(c)|L$, for some constant $c$. We prove this using the Refinement Mapping theorem, Theorem 3.4.3

We now present a more detailed roadmap of each part of the formal proof.

The first part of the proof is described in Sections 5.6.2 to 5.6.4. To show that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ we use another two step process:

- In Section 5.6.3 we formally define the concept of a clean link alluded to earlier. Intuitively, a link is clean if the snapshot numbering scheme is working correctly, meaning that requests and responses will properly be matched. We use the predicate $C$ to denote the fact that all links are clean. At the end of Section

5.6.3, in Theorem 5.6.14 we prove that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$ in constant time. The proof is based on the counter flushing intuition we have described earlier.

- In Section 5.6.4 we define another important concept called a quiet link. Recall that our intent is to make the local predicate $L_{u,v}$ hold for every link. Intuitively, a link $(u,v)$ is quiet if two properties hold. First, $L_{u,v}$ holds. Second, if all links less than $(u,v)$ in the partial order are also quiet, then there will be no more reset actions on that link. We use the predicate $Q$ to denote the fact that all links are quiet. At the end of Section 5.6.4, in Theorem 5.6.22 we prove that $\mathcal{N}^+|C$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time proportional to the height of the partial order.

The second part of proof (Section 5.6.5, (Lemma 5.6.23)) shows that any behavior of $\mathcal{N}^+|Q$ is also a behavior of $\mathcal{N}(c)|L$, for some constant $c$. This is done using the Refinement Mapping theorem, Theorem 3.4.3. In order to use this theorem we must derive a projected state of $\mathcal{N}$ from a state of $\mathcal{N}^+$. We have already seen how to derive a projected state $s|u$ of a node $N_u$ from a state $s$ of $N_u^+$. To complete our job, we need to state how to project the state of the channels in $\mathcal{N}^+$ to $\mathcal{N}$.

Consider the problem of projecting the channel state. Intuitively, in $\mathcal{N}^+$, the original automaton $\mathcal{N}$ is "sharing" each channel with request and response packets. Let's look at the behavior of $\mathcal{N}^+$. When a control packet is on the channel $C_{u,v}$, we can pretend that, as far $\mathcal{N}$ goes, the channel is really empty. There are two consequences of this. First, if the projected $N_u$ has a data packet to send, it can take longer before the data packet is sent. Second, it can take longer before a free signal is delivered by the link. This can happen if there is a control packet on the link. We model this by saying that in the projected behavior *the node and link delays are increased by a constant factor*. This is the source of the increased link and node delays in Theorem 5.4.3.

**Definition 5.6.1** *For any state $s$ of $\mathcal{N}^+$, we define $Proj(s)$, the state of $\mathcal{N}^+$ projected onto $\mathcal{N}$, as follows:*

- *For any two neighboring nodes $u$ and $v$: if $s.Q_{u,v} \in P_{data}$ then $Proj(s).Q_{u,v} = s.Q_{u,v}$; else $Proj(s).Q_{u,v} = nil$. (i.e., if there is a data packet $p$ in channel $C_{u,v}$*

*in the original state, then p is also present in the projected state; if not, $C_{u,v}$ is considered empty in the projected state.)*

- *All other variables of $\mathcal{N}$ have the same values in state $Proj(s)$ as in state $s$.*

We complete the second part of the proof (Lemma 5.6.23) using the Refinement Mapping theorem and using the mapping function *Proj*. Notice that is not as simple as it might appear. If all actions of $\mathcal{N}^+$ that change the projected state are actions of $\mathcal{N}$, then it would be quite easy. Unfortunately we have a complication. There are actions of $\mathcal{N}^+$ that can cause a local reset. An example is the receipt of a $p_{resp}$ packet with $p_{resp}.mode = reset$. Such actions change the projected state but are not actions of $\mathcal{N}$. However, we will show that such actions cannot appear in executions of $\mathcal{N}^+|Q$. This is because $Q$ is a strong enough predicate to ensure not only that property $L$ holds in the projected state, but also that no local reset actions are enabled.

We continue to let $s|u$ denote the current state of $\mathcal{N}_u^+$ *projected onto the original automaton* $N_u$. We also use $s|(u,v)$ to denote the state of $C_{u,v}$ *projected onto the original automaton* $N$. *In other words* $s|(u,v) = Proj(s).Q_{u,v}$.

### 5.6.2 Phases

We formally define a phase (see Figure 5.4) on a link, as an interval during which the checking/correction procedure works on that link. As we have seen, the checking/correction procedure works by setting $phase_u[v] = true$, and attempting to send out a numbered request packet. If a response is received with a matching number then $phase_u[v]$ is set to false. We make this more precise below.

**Definition 5.6.2** *A $(u,v)$ phase for execution $\alpha$ is any interval $\gamma$ of $\alpha$ such that*

- $u = l(u,v)$
- *$\gamma$ begins with a state $s_i$ of $\alpha$ such that $s_i.phase_u[v] = false$ and $s_{i+1}.phase_u[v] = true$.*
- *$\gamma$ ends with the first state $s_j, j > i$ of $\alpha$ in which $s_j.phase_u[v] = false$.*

Notice that a $(u,v)$ phase is only defined for a leader edge $(u,v)$.

The definition of a phase allows the last state of a phase to overlap the first state of the next phase. Clearly we can divide an execution $\alpha$ into consecutive $(u,v)$ phases as well as intervals that lie between $(u,v)$ phases. Thus we can speak of the $i$-th $(u,v)$ phase in $\alpha$ in this division.

Our first lemma states that in between phases, at most one data packet is sent on $C_{u,v}$. Thus we can think of an execution (from the point of view of any link $C_{u,v}$) as alternating between a phase during which requests are sent and responses are received, followed by a period where at most one data packet is sent on $C_{u,v}$.

**Lemma 5.6.3** *Between two consecutive $(u,v)$ phases on link $(u,v)$ at most one $\text{SEND}_{u,v}(p_{data})$ event can occur.*

**Proof:** The first $\text{SEND}_{u,v}(p_{data})$ event after the $i$-th $(u,v)$ phase will set $phase_u[v] = true$ which begins the $i+1$-st phase. ∎

The proofs of the next two lemmas are quite tedious and are relegated to the appendix. The first lemma states that a single phase completes in constant time. Intuitively, the time taken to complete a phase consists of the time to start a phase (which may involve the sending of a data packet) followed by the time to send a request and receive a matching response. The second lemma shows that data packets are sent out on a link in a bounded time after they are placed on the queue for the link. Intuitively, this is because a phase takes constant time to complete and if the data queue is non-empty, the code sends at least one data packet between consecutive phases.

Before we state these lemmas, we define a quantity $t_p$. Intuitively, $t_p$ is the the time it takes to complete a phase.

**Definition 5.6.4** *We let $t_p = 6t_n + 12t_l$.*

The major components of the time to complete a phase are sketched in Figure 5.10. Since the free variables may be incorrect in the initial state, the first packet sent on a link can be dropped. However, in constant time, the links stop dropping packets. Then, after the possible sending of data packet, a request is sent out by $u$ in constant time. Next, after the possible sending of a data packet, a response is sent by $v$ in constant time. The appendix contains more details.

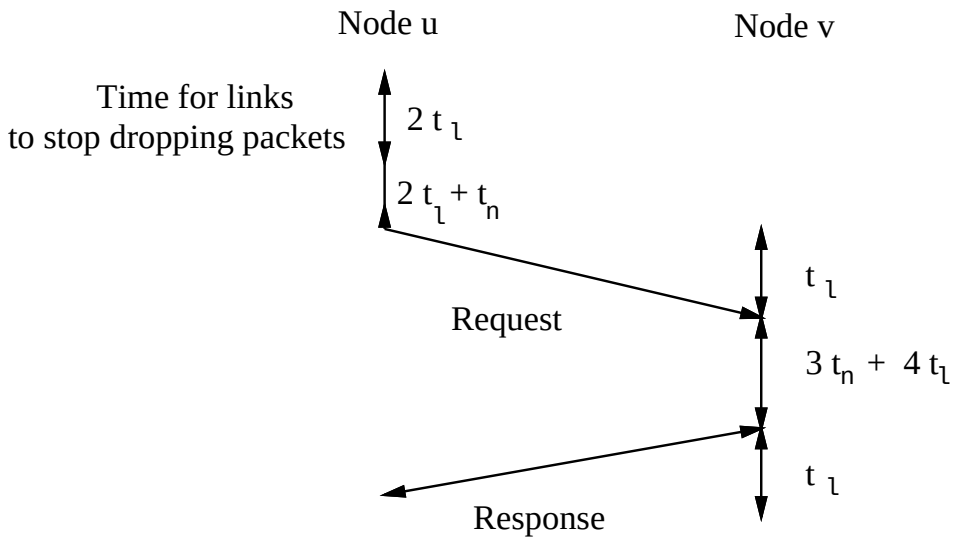



Figure 5.10: The major components of the time required to complete a phase.

**Lemma 5.6.5 Phase Rate:** *For all $\alpha$ and any leader edge $(u,v)$ and any integer $x$, at least $x$ $(u,v)$ phases will have completed within $(x+1)\cdot t_p$ time units after the start of $\alpha$.*

**Proof:** From Lemma B.3.4 and Lemma B.3.5 in Section B.3 of the appendix. The reason why we need $(x+1)\cdot t_p$ time (instead of $x \cdot t_p$ time) to complete $x$ phases is as follows. Suppose in the initial state of $\alpha$, $phase_u[v] = true$; it may take $t_p$ time before $phase_u[v]$ becomes *false*. Thus, the first $(u,v)$ phase in $\alpha$ may be a "partial phase". However, the definition of a phase does not allow us to consider a "partial phase" as a phase. ∎

**Lemma 5.6.6 Data Packet Rate:** *For any $\alpha = s_0, a_1, \ldots$ and any $(u,v)$, if $s_0.|queue_u[v]| > 0$ then a $\text{SEND}_{u,v}(p_{data})$ occurs within $t_p$ time units after $s_0$.*

**Proof:** At the end of Section B.3 in appendix. ∎

### 5.6.3 Clean Phases

We know that a link can drop packets if a packet is sent when the link already has a packet. To be sure that packets will not be dropped, the sender must never think that

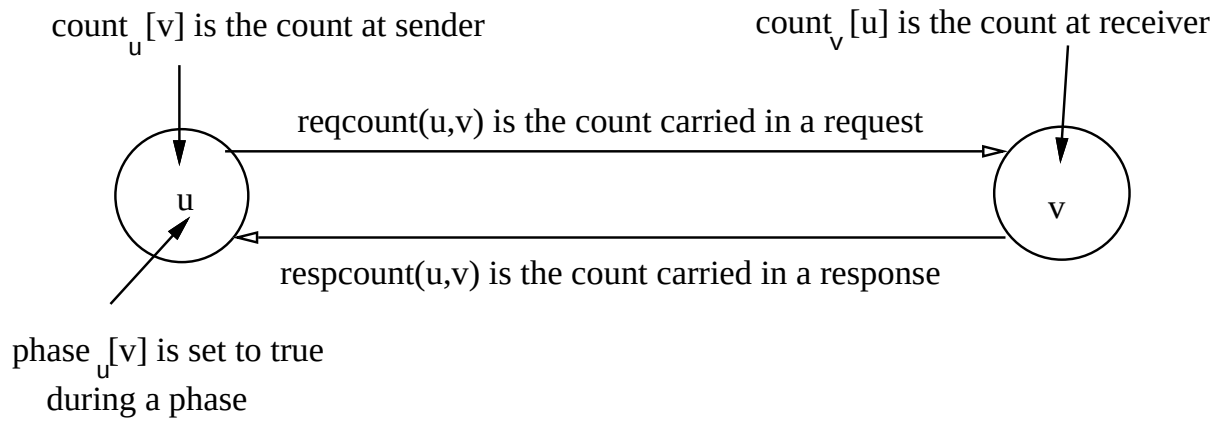



Figure 5.11: The key variables used in the definition of a clean link: $countset(u,v)$ is the union of the count at the receiver and any count values in request and response packets. A clean link ensures that between phases, the sender count value is not in $countset$.

a link is free when it isn't. In this case, we say the link is "drop-free".

**Definition 5.6.7** *Let $F_{u,v}$ denote the predicate of $\mathcal{N}^+$ defined by: $(freeq_u[v] = true) \rightarrow (Q_{u,v} = nil)$. We also say that $(u,v)$ is drop-free in state $s$ of $\mathcal{N}^+$ if $s \in F_{u,v}$.*

In the appendix, we show that once a link is drop-free, it remains drop-free. Intuitively, this is because the sender will not record the link as free unless it receives a free signal from the link, which means that there is no packet on the link. In the appendix, we also show a link $(u,v)$ becomes drop-free in constant time – in fact, after the first packet sent on the link. This follows because after the first packet is sent on the link, the sender records the link as being busy, and this trivially satisfies the drop-free predicate.

Before studying the structure of phases, we introduce some notation to denote the set of counter values in a $(u,v)$ subsystem. These include counters in any request or response packets, and the counter stored at node $v$. These variables are sketched in Figure 5.11.

**Definition 5.6.8** *For any state $s$ we define the derived variables $reqcount(u,v)$, $respcount(u,v)$ and $countset(u,v)$ as follows:*

1. *If $Q_{u,v} = p_{req}$ then $reqcount(u,v) = p_{req}.count$; otherwise $reqcount(u,v) = undefined$*

2. *If* $Q_{v,u} = p_{resp}$ *then* $respcount(u,v) = p_{resp}.count$*; otherwise* $respcount(u,v) =$ *undefined.*

3. $countset(u,v)$ *is the set formed by the union of the values in* $respcount(u,v)$, $reqcount(u,v)$ *and* $count_v[u]$

In order for a $(u,v)$ phase to work correctly, we need the phase to follow the structure describe in Figure 5.4. For this to happen, when a numbered request is first sent during a phase, the number of the request should not already present at the receiver or in the channels. If this is not the case, it is possible for an incorrect response to be accepted in the phase. We formalize this notion of a link and a phase being "clean" using five conditions. The second condition is the crucial condition; the other four are supporting conditions required to ensure that the clean predicate is closed.

**Definition 5.6.9** *We say that a leader edge* $(u,v)$ *is clean in state* $s$ *of* $\mathcal{N}^+$ *iff all the following predicates are true in* $s$*:*

1. $(v,u)$ *and* $(u,v)$ *are drop-free.*

2. *If* $phase_u[v] = false$ *then* $count_u[v] \notin countset(u,v)$

3. *If* $phase_u[v] = true$ *then* $reqcount(u,v) = count_u[v]$ *or* $reqcount(u,v) = undefined$

4. *If* $phase_u[v] = true$ *and* $respcount(u,v) = count_u[v]$ *then* $respcount(u,v) = count_v[u]$.

5. *If* $count_v[u] = count_u[v]$ *then* $Q_{u,v} \notin P_{data}$.

The first condition ensures that the links in both directions are drop-free. The second condition ensures that when a numbered request is first sent during a phase, the number of the request is not already present at the receiver or in the channels. This is the important property of a clean link.

The third condition states that during a phase, any requests must carry the sender's number. The fourth condition states that during a phase if there is a matching response on the channel from the receiver to the sender, then the receiver has the same number as the sender. The fifth condition says that if during a phase the sender and receiver

numbers are the same, then there can be no data packet in transit from the sender to the receiver. The fifth condition (as we will see below) follows from the fact that the sender will not send a data packet during a phase.

**Definition 5.6.10** *A $(u,v)$ phase $p$ is clean if $(u,v)$ is clean in the first state of $p$.*

Intuitively, a clean phase will contain an action to send a request packet at, $u$ followed by the receipt of this packet at $v$ followed by the sending of a response by $u$ followed by the receipt of the response at $u$. The receipt of the matching response ends the phase. This is shown in Figure 5.4. Thus a clean phase will ensure that the response received at node $u$ will correspond to the requesting information at node $u$.

A nice property is that once an edge becomes clean, it remains clean.

**Lemma 5.6.11** *For any transition $(s,\pi,\tilde{s})$ and any leader edge $(u,v)$, if $(u,v)$ is clean in $s$ then $(u,v)$ is clean in $\tilde{s}$.*

**Proof:** See Section B.4 in the appendix. However, it is not hard to see informally see why this is true. First, as we have seen before, once a link is drop-free, it remains drop-free. Next, we know from the fourth condition that a matching response can only be received if the receiver has the same number as the sender. Also from the second condition, any requests present during a phase must have the same number as the sender. Now the sender always increments its number after receiving a matching response. Thus after a phase is over, the number of the sender must be different from any of the numbers present in the receiver or channels: this is the second condition. The third condition follows from the fact that a phase begins by sending a request with the sender's number, and subsequent retransmissions of requests carry the same number. Next, if a matching response is present, it must have been sent by the receiver during the phase. But, by the third condition, the receiver cannot change its number after it sent the response. Thus if there is a matching response, the number of the receiver must be the same as the sender, which is the fourth condition.

The fifth condition follows because the sender never sends a data packet during a phase. By the second condition, at the start of the phase the receiver number is not equal to the sender number; thus the two numbers become the same only after the receiver has received a request. But the receipt of this request "flushes" out any

data packets that were in transit from sender to receiver at the start of the phase. This (taken together with the fact that the sender never sends a data packet during a phase) is what makes the fifth condition hold. ∎

The following lemma explains why the request-response matching procedure is stabilizing.

**Lemma 5.6.12** *A leader edge $(u,v)$ is clean in all states after the fifth $(u,v)$ phase in any execution $\alpha$.*

**Proof:** (Idea) We first show that after at most two phases $(v,u)$ and $(u,v)$ are both drop-free. This follows (see Claim B.2.2 in the appendix) since some packet (i.e., at least one request and at least one response) must have been sent on either link by this time. Next consider the end of the second phase. In this state, there must be some $c \in \{0,\ldots,3\}$ such that $c \notin \mathit{countset}(u,v)$. This follows because, by definition, $\mathit{countset}(u,v)$ has a maximum of three elements. Now consider the first time after the end of the second phase that $\mathit{count}_u[v] = c$. This must occur at the end of a phase because $\mathit{count}_u[v]$ is only incremented at the end of a phase. Also this must occur at or before the end of the fifth phase because $\mathit{count}_u[v]$ increases by 1 mod 4 at the end of each phase.

It is easy to see using Claim B.4.1 that when $\mathit{count}_u[v]$ first becomes equal to $c$, $c \notin \mathit{countset}(u,v)$. Thus at or before the end of the fifth phase, $\mathit{phase}_u[v] = \mathit{false}$ and $\mathit{count}_u[v] \notin \mathit{countset}(u,v)$. Thus at or before the end of the fifth phase, edge $(u,v)$ is clean. Finally, by Claim 5.6.11, $(u,v)$ is clean in all subsequent states of $\alpha$. ∎

Now we have reached the goal of this subsection. First we define:

**Definition 5.6.13** *Let $C$ be the predicate of $\mathcal{N}^+$ such that for all leader edges $(u,v)$, $(u,v)$ is clean in all states in $C$.*

Recall that $\mathcal{N}^+|C$ is the automaton that is identical to $\mathcal{N}^+$ except that all leader edges are clean in any initial state.

**Theorem 5.6.14** *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$ in time $6t_p$.*

**Proof:** Follows directly from Lemma 5.6.11, Lemma 5.6.12 and Lemma 5.6.5 and the Execution Convergence theorem, Theorem 3.4.5. Lemma 5.6.11 shows that each leader edge becomes clean after at most 5 phases which by Lemma 5.6.5 takes at most $6t_p$ time. Lemma 5.6.11 shows that once a leader edge is clean it remains clean. Then the Execution Convergence theorem (Theorem 3.4.5) shows that all leader edges become clean in at most $6t_p$ time. ∎

### 5.6.4 Quiet Links: Establishing Link Predicates

Clean phases are only useful because they allow local predicates to be established as we show in this section. We know from the last section that every execution has a suffix that is clean. When we say that $L_{u,v}$ holds in a state $s$ of $\mathcal{N}^+$, we mean that $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$.

Recall that $<$ is the partial order on edges associated with reset function $f$. Consider a link $(u,v)$ such that there is no other link $\{w,x\} < \{u,v\}$. We will show that if in the first state of a clean $(u,v)$ phase $mode_u[v] = snapshot$ then at the end of the phase, a true snapshot of the $(u,v)$ subsystem is obtained. In particular, it is possible at the end of such a phase to determine whether $L_{u,v}$ holds at the end of the phase. If it does not hold, $mode_u[v]$ is changed to *reset*. In a similar fashion, we can show that any clean phase whose initial state has $mode_u[v] = reset$ will guarantee that $L_{u,v}$ holds at the end of the phase. The net effect is that $L_{u,v}$ holds by end of the second $(u,v)$ phase of a "clean" execution.

However, we want to show not only that $L_{u,v}$ holds by end of the second $(u,v)$ phase but also that no more "reset" actions can occur after this point so that $L_{u,v}$ will remain true. This motivates the following definitions of a quiet link. Please refer to Figure 5.12 for an intuitive explanation.

**Definition 5.6.15** *We say that a leader edge $(u,v)$ is quiet in state $s$ of $\mathcal{N}^+$ iff the following predicates hold in $s$:*

1. *$(u,v)$ is clean.*
2. *$(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$*
3. *$(mode_u[v] \neq reset)$ and $(mode_v[u] \neq reset)$.*

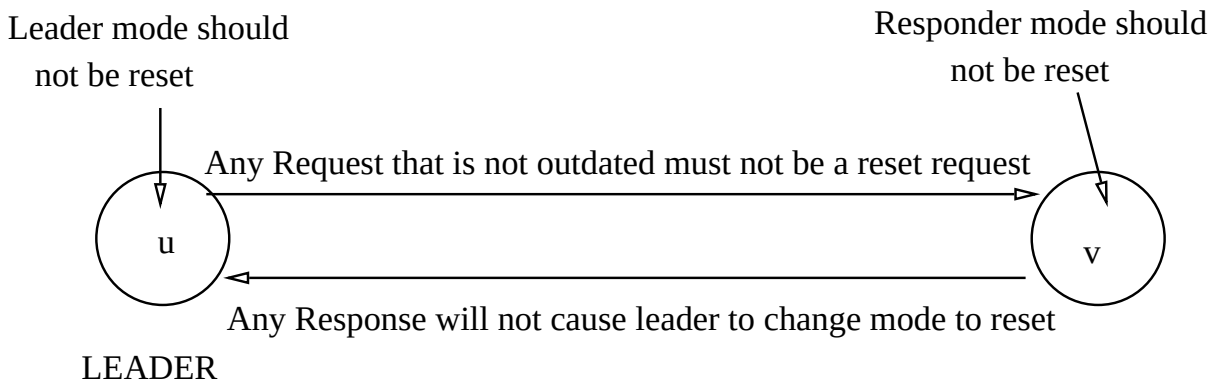



Figure 5.12: A leader edge is quiet if it clean, its local predicate holds, and it satisfies the conditions sketched in the figure.

4. *If* $Q_{u,v} = p_{req}$ *and* $p_{req}.mode = reset$ *then* $p_{req}.count = count_v[u]$.

5. *If* $Q_{v,u} = p_{resp}$ *and* $p_{resp}.count = count_u[v]$ *then* $(s|u, nil, nil, p_{resp}.node_state) \in L_{u,v}$

Notice that the fourth condition above ensures that there is no reset request on the link that could be accepted by the receiver at a later state. The fifth condition ensures that any snapshot information in a matching response will not cause the sender to change its mode to *reset*.

Our goal is to show that eventually all links are quiet.

**Definition 5.6.16** *Let $\mathcal{Q}$ be the predicate set consisting of the following predicate for each leader edge in $G$. The predicate for each leader edge $(u,v)$ is that $(u,v)$ is quiet. Let $Q$ be the predicate that is the intersection of all predicates in $\mathcal{Q}$.*

We will extend the partial order $<$ (which was defined on undirected pairs of nodes) to leader edges by assuming that that leader edge $(u,v) <$ leader edge $(w,x)$ iff $\{u,v\} < \{w,x\}$. Next, we will use the Execution Convergence theorem (Theorem 3.4.5) to show that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$. We know that $\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|C$. Thus it is sufficient to show that $\mathcal{N}^+|C$ is stabilized to $Q$ using predicate set $\mathcal{Q}$ and partial order $<$. To show this, we will first show two lemmas:

First we state the required stability condition: that a leader edge $(u,v)$ will remain quiet as long as all leader edges less than or equal to $(u,v)$ are quiet.

**Lemma 5.6.17** *Consider a leader edge $(u,v)$. If every leader edge $(w,x) \leq (u,v)$ is quiet in some state $s$ of $\mathcal{N}^+|C$, then for any transition $(s,\pi,\tilde{s})$, $(u,v)$ is quiet in state $\tilde{s}$.*

**Proof:** In Section B.5 in the appendix. However, the basic idea is simple. We have already seen that the clean predicate is stable. We also know from the definition of local checkability that $L_{u,v}$ is closed for all the actions of the original automaton $\mathcal{N}$. However, in $\mathcal{N}^+$ we added additional actions to send requests and responses. It is easy to verify that the sending of snapshot requests and responses do not affect $L_{u,v}$. However, $L_{u,v}$ is affected by actions that send reset requests and responses on edges to neighbors that are less than $(u,v)$ in the partial order. (If such "reset" actions occur on edges to neighbors that are *not* less than $(u,v)$ in the partial order, then the definition of the local correction function ensures that $L_{u,v}$ remains true.) But such "reset actions" cannot occur on edges less than $(u,v)$ in the partial order, because such edges are quiet by hypothesis.

Next, if $L_{u,v}$ holds, and the snapshot information in matching responses is always correct, the sender will never change its mode to *reset*. But if the sender never changes its mode to *reset*, the sender will never send a reset request. And if the receiver never receives a reset request, the receiver will never change its mode to *reset*. ∎

Next we state the required liveness condition: that a leader edge $(u,v)$ will become quiet in bounded time if all leader edges less than $(u,v)$ are already quiet.

**Lemma 5.6.18** *Consider a leader edge $(u,v)$ and some execution $\alpha$ of $\mathcal{N}^+|C$. If every leader edge $(w,x) < (u,v)$ is quiet in some state $s_i$ of $\alpha$, then $(u,v)$ is quiet in some state that occurs within $3 \cdot t_p$ of $s_i$ in $\alpha$.*

**Proof:** In In Section B.5 in the appendix. However, the basic idea is simple. Consider the first complete $(u,v)$ phase after $s_i$. If it is a snapshot phase, and $L_{u,v}$ does not hold at the end of the phase, this will be detected and the next phase will become a reset phase. Thus either the first or second complete phase after $s_i$ will be a reset phase. At the end of the reset phase, $(u,v)$ will become quiet because all the links that $(u,v)$

depends on are already quiet. The upshot is that within two complete phases, $(u, v)$ becomes quiet. But this can take up to $3t_p$ time (where $t_p$ is the time to complete a single phase) because the first phase after $s_i$ can be an incomplete phase. ∎

The last two lemmas immediately give us:

**Lemma 5.6.19** *$\mathcal{N}^+|C$ is stabilized to $Q$ using predicate set $\mathcal{Q}$ and time constant $3t_p$.*

Thus applying the Execution Convergence Theorem (3.4.5) and using the last lemma, we get:

**Lemma 5.6.20** *$\mathcal{N}^+|C$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time height$(f) \cdot 3t_p$.*

For convenience we define a quantity $t_q$ which intuitively can be thought of as the time after which any execution of $\mathcal{N}^+$ becomes quiet.

**Definition 5.6.21** *We let $t_q = 6t_p + height(f) \cdot 3t_p$;*

Thus we have the major result of this section:

**Lemma 5.6.22** *$\mathcal{N}^+$ stabilizes to the executions of $\mathcal{N}^+|Q$ in time $t_q$.*

**Proof:** Follows directly from Lemmas 5.6.14 and 5.6.19 and transitivity (Lemma 3.1.6). ∎

### 5.6.5 Projecting Behaviors of $\mathcal{N}|Q$

In this subsection, we prove that if all leader edges are quiet in the initial state of some execution $\alpha$ of $\mathcal{N}^+$, then the external behavior corresponding to $\alpha$ is a behavior of $\mathcal{N}|L$. Intuitively, this is not very hard to believe. This is because for every leader edge $(u, v)$, $L_{u,v}$ holds in every projected state of $\alpha$. We also know that there will be no reset transitions in $\alpha$. Formally:

**Lemma 5.6.23** *Every behavior of $\mathcal{N}^+|Q$ is a behavior of $\mathcal{N}(t_p)|L$*

**Proof:** Follows by using a refinement mapping (Theorem 3.4.3) using the mapping function *Proj* (Definition 5.6.1).

First, for any state $s$ of $\mathcal{N}^+|Q$, and any leader edge $(u,v)$, we know (from the definition of $Q$) that $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$. Thus $Proj(s) \in L$. Thus for any state $s$ of $\mathcal{N}^+$, $Proj(s)$ is a start state of $\mathcal{N}(t_p)|L$

Next consider any transition, $(s, \pi, \tilde{s})$ of $\mathcal{N}^+|Q$. We know from the definition of $Q$ that there are no reset transitions in $\mathcal{N}^+|Q$. Suppose $\pi$ is a $\text{SEND}_{u,v}(p_{req})$ , $\text{RECEIVE}_{u,v}(p_{req})$ , $\text{SEND}_{v,u}(p_{resp})$ , or $\text{RECEIVE}_{v,u}(p_{resp})$ for any $(u,v)$. Then it is easy to see that $Proj(s) = Proj(\tilde{s})$. But such actions are not actions of $\mathcal{N}$.

If $\pi$ is a $\text{SEND}_{u,v}(p_{data})$ action for some edge $(u,v)$, then it must be that $p_{data}$ is at the the head of $s.queue_u[v]$ and $s.free_u[v] = true$. Since $Proj(s).queue_u[v] = s.queue_u[v]$ and $Proj(s).free_u[v] = s.free_u[v]$, the $\text{SEND}_{u,v}(p_{data})$ event is also enabled in $Proj(s)$. Also, $\tilde{s}.Q_{u,v} = Proj(\tilde{s}).Q_{u,v} = p_{data}$, since $(u,v)$ is drop-free in $s$. Thus $(Proj(s)$, $\text{SEND}_{u,v}(p_{data})$ , $Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Similarly if $\pi$ is a $\text{RECEIVE}_{u,v}(p_{data})$ action for some edge $(u,v)$, then it must be that $s.Q_{u,v} = p_{data}$ and $\tilde{s}.Q_{u,v} = nil$. Thus $Proj(s).Q_{u,v} = p_{data}$ and $Proj(\tilde{s}).Q_{u,v} = nil$. and $(Proj(s)$, $\text{RECEIVE}_{u,v}(p_{data})$ , $Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$. Next if $\pi$ is a $\text{FREE}_{u,v}$ action for some edge $(u,v)$, then it must be that $s.Q_{u,v} = nil$ and $\tilde{s}.Q_{u,v} = nil$. Thus $Proj(s).Q_{u,v} = nil$ and $Proj(\tilde{s}).Q_{u,v} = nil$ and $(Proj(s), \text{FREE}_{u,v}, Proj(\tilde{s}))$ is a transition of $\mathcal{N}(t_p)|L$.

Finally suppose that $\pi$ is any other action of $\mathcal{N}(t_p)|L$. (For example, this category would include internal actions of $\mathcal{N}(t_p)|L$.) Then such actions remain unmodified and the values of all node variables are identical in $s$ and $Proj(s)$. Thus $(Proj(s), \pi, Proj(\tilde{s}))$ is a transition of $\mathcal{N}|L(t_p)$.

Finally consider the timing properties. Suppose a $\text{SEND}_{u,v}(p_{data})$ event is enabled in $Proj(s)$ for any edge $(u,v)$. Then $s.|queue_u[v]| > 0$. Thus by Lemma 5.6.6, a $\text{SEND}_{u,v}(p_{data})$ event occurs within $t_p$ time after $s$. Suppose a $\text{RECEIVE}_{u,v}(p_{data})$ event is enabled in $Proj(s)$ for any edge $(u,v)$. Then $s.Q_{u,v} = p_{data} \neq nil$. From the timing properties of $\mathcal{N}^+|Q$ we know that a $\text{RECEIVE}_{u,v}(p_{data})$ will occur in $t_l$ time, and $t_l \leq t_p$.

Consider a $\text{FREE}_{u,v}$ action for any edge $(u,v)$. Within $t_l$ time units after $s$, either a $\text{FREE}_{u,v}$ action will occur or there is a state $s'$ such that $s'.Q_{u,v} \neq nil$. Consider the second case. From Claim B.3.1, within $t_l$ time units after $s'$, there is a state $s''$ such that $s''.Q_{u,v} = nil$. Since $(u,v)$ is drop-free in all states of $\alpha$, $s''.freeq_u[v] = false$. Thus $Q_{u,v} = nil$ will remain true until a $\text{FREE}_{u,v}$ action occurs within time $t_l$ after $s''$.

Thus a $\text{FREE}_{u,v}$ action will occur within $3t_l$ time units after state $s$. In particular, this means that a $\text{FREE}_{u,v}$ action will occur within $t_p$ time units of any state $s$ such that $\text{FREE}_{u,v}$ is enabled in $Proj(s)$.

Finally suppose that $\pi$ is any other locally controlled action of $\mathcal{N}(t_p)|L$. Thus $\pi$ is an internal action of $\mathcal{N}(t_p)|L$. Notice that $\pi$ is in the same class say $c$ (with upper bound equal to $t_c$) in both $\mathcal{N}^+|Q$ and $\mathcal{N}(t_p)|L$. Suppose that $\pi$ is enabled in $Proj(s)$. Then $\pi$ is enabled in $s$. Then in $t_c$ time of $s$ in $\alpha$ either some action in class $c$ occurs or there is some state $\tilde{s}$ such that no action in class $c$ is enabled in $Proj(\tilde{s})$. This follows because if no action in class $c$ is enabled in $\tilde{s}$, then no action in class $c$ is enabled in $Proj(\tilde{s})$, and vice versa. ∎

### 5.6.6 Tying up the Proof

We now return to the proof of the Local Correction Theorem, Theorem 5.4.3. It follows from Lemma 5.6.23 and Lemma 5.6.22 and Theorem 3.4.2.

## 5.7 Implementing Local Checking in Real Networks

In this chapter, we described a stabilizing snapshot and reset protocol for link subsystems. This protocol was used to transform an automaton $\mathcal{N}$ that was locally correctable to some predicate $L$ into a UIOA $\mathcal{N}^+$ that stabilizes to the behaviors of $\mathcal{N}(c)|L$ for some constant $c$. To make the snapshot/reset protocol stabilizing we numbered snapshot requests and responses; we also relied on the fact that each link was a UDL. In practice, however, there is an even simpler way of making the snapshot/reset protocol stabilizing. This can be done using timers.

Suppose there is a known bound on the length of time a packet can be stored in a link and a known bound on the length of time between the delivery of a request packet and the sending of a matching response packet. Then by controlling the interval between successive snapshot/reset phases it is easily possible to obtain a stabilizing snapshot protocol. The interval is chosen to be large enough such that all packets from the previous phase will have disappeared at the end of the interval. This solution was advocated by us in [APV91a] and was implemented in a trial implementation on the Autonet [MAM+90].

To keep our model simple, however, we have assumed that all lower bounds (on the time between events) are zero. Thus we have no way to model timers, which are needed to control the interval between phases. While we will not describe the timer based scheme formally, the reader interested in practical applications should be aware of the simplicity of a timer based scheme. Notice also that the maximum value of the timer need only be some small multiple of the worst-case round trip delay on *a single link.* We call such timers *local timers.*

In most real networks, each node sends "keep-alive" packets periodically on every link in order to detect failures of adjacent links. If no keep-alive packet arrives before a local timer expires, the link is assumed to have failed. Thus, it is common practice to assume time bounds for the delivery and processing of packets. Note also that the snapshot and reset packets used for local checking can be "piggy-backed" on these keep-alive packets without any appreciable loss in efficiency.

## 5.8 Summary

The two main contributions of this chapter are the definition of Unit Storage Data Links in Section 5.1.2, and the notion of local checking and correction (Section 5.3 and Local Correction theorem, Theorem 5.4.3).

In a stabilizing setting it is necessary to define Data Links that have bounded storage. First, such models correspond to physical reality. Second, they avoid the theoretical problems with unbounded storage links. We have chosen unit storage links (UDLs) because they are practical (see Section 5.2) and they can be modelled elegantly. We have also defined a stabilizing interface to a UDL. This is done by having the link periodically deliver a free signal (to avoid deadlock) and by having the sender keep a variable that indicates whether the link is free. We hope the UDL model will be used by others.

Intuitively, a protocol is locally checkable if whenever the protocol is in a bad state, some pair of neighbors can detect this fact. Intuitively, a protocol is locally correctable if the protocol can be corrected to a good global state by independently correcting the states of each link subsystem. A link subsystem is just a pair of neighboring nodes and the links between them.

Local checkability is not a very new or surprising idea. For example, many authors have proposed local methods for detecting termination and deadlocks. In a stabilizing setting, the intuitive notion of local checkability was first referred to in a paper by [AKY90]. However, their reference to this concept (during the description of a spanning tree protocol) was brief and intuitive.

Our contribution has been to make precise the notion of local checkability, and to show that this is a useful and pervasive concept. Our definition (Definition 5.3.6) has some subtle aspects: for example, the requirement that each local predicate be closed may not be immediately obvious. Also, we have implemented local checking by doing a snapshot of each link subsystem. Now, a number of practical, self-stabilizing protocols (e.g., [Per85]) do what essentially amounts to local checking in the following way. Periodically each node sends its state to all its neighbors. However, (as we will show in Chapter 8) such periodic sending of state is not always sufficient to do local checking. Periodic sending is sufficient if all local predicates can be separated into what we call (see Chapter 8) one-way predicates. In the general case, a snapshot is required.

The idea of local correction is more unusual. It is perhaps surprising that there are non-trivial protocols with this property. As we will see in Chapter 6, an easy way to ensure local correction is to first build a spanning tree of the network and then do local correction using the tree. Luckily, there is another class of protocols that are locally correctable: these are protocols that work in dynamic networks in which links can fail and recover. We will see an important example of such protocols in Chapter 7.

Local checking and correction is practical. In most real networks, each node sends "keep-alive" packets periodically on every link. The packets used for local checking and correction can be "piggy-backed" on these keep-alive packets.

# Chapter 6

# Stabilizing Mutual Exclusion and Tree Correction

The main result of Chapter 5 was the Local Correction theorem, Theorem 5.4.3. In this chapter we will consider a simple application of the Local Correction theorem to the problem of *mutual exclusion*. Section 6.1 gives an overview of our stabilizing mutual exclusion protocol. Section 6.2 formally defines our mutual exclusion protocol as a network automaton. Section 6.3 describes how local checking and correction is added to the automaton to create a stabilizing solution to the mutual exclusion solution.

The last two sections of this chapter extract some general principles from the example of stabilizing mutual exclusion. In Section 6.4, we discuss a weaker notion of local checkability called *weak local checkability*. We show that in certain cases, a simple heuristic of removing *unexpected packet transitions* can be used to transform a protocol that is weakly locally checkable into a locally checkable protocol. This heuristic has proved to be quite useful, and is used in later chapters.

Finally, in Section 6.5, we show (informally) an important result, the Tree Correction theorem. This theorem states that any locally checkable protocol on a tree topology can be efficiently stabilized. In other words, if the underlying topology is a tree we can dispense with the need for the (stronger) local correctability condition. This theorem is the counterpart to a similar theorem proved in Chapter 4 for shared memory systems.

## 6.1 Overview of Token Passing Protocol

Token passing is one way of ensuring mutual exclusion in a network. If there is only one token in a network, a node can go into the critical region when it receives a token. In the I/O automaton model this is typically modelled by adding an output action to every node by which the node can give permission to some external user to go into the critical section, and adding an input action that tells the node when the user has finished with the critical section. For simplicity we ignore this extra piece of modelling which is important for providing a modular interface to other subsystems.

Token protocols that stabilize in time proportional to the height of the tree have been described before by [DolevIM90] in a shared memory setting. However, we believe our protocol is simpler and more transparent. While the mutual exclusion protocol is very simple, its simplicity makes it a good candidate to understand how the general method of local correction developed in Chapter 5 can be applied.

Stabilizing spanning tree protocols have been well studied [AKY90],[AG90],[AV91]. Thus we can reduce the problem of token passing on an arbitrary connected network to that of token passing on a tree. Instead of modelling the interface to the network automaton that computes the tree, we will (for simplicity) assume that the network graph is a tree. Thus we are dealing with a network automaton of the form $\mathcal{N} = Net(G, N)$ where $G$ is a tree graph. Formally:

**Definition 6.1.1** *A tree graph $T$ is a topology graph such that:*

- *$T$ is a rooted tree. (i.e, there is a distinguished node called the root and there is a unique path between any node and the root.)*
- *For every edge $(u, v)$ in $G$, the leader function $l(u, v) = u$ if $u$ is the parent of $v$ in the tree, and $l(u, v) = v$ if $v$ is the parent of $u$ in the tree.*

**Definition 6.1.2** *A tree automaton is a network automaton whose topology graph is a tree graph.*

For a tree graph we will refer to the leader function $l$ as *parent* for obvious reasons.

The simplest mutual exclusion algorithm is to pass a token along some tour (e.g., DFS) of the tree; a node can go "critical" when it has the token. But such a protocol

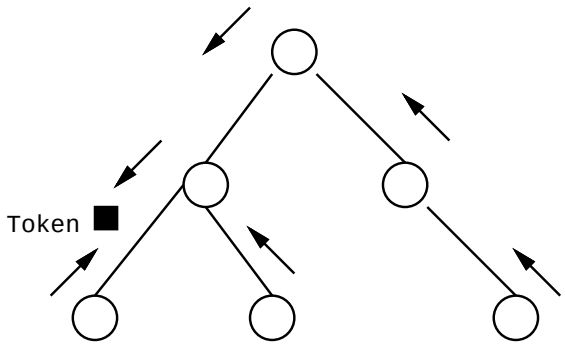



Figure 6.1: Adding a pointer to each node makes token passing on a tree locally checkable.

is not locally checkable; if there are two tokens at two links, each link subsystem may not locally detect a problem.

Once we see this, it is easy to add a small amount of state to make the token protocol locally checkable. We add a pointer, $pointer_u$ to each node $i$ such that $pointer_u$ points to where the token is in the tree; also if the token is at node $u$, then $pointer_u = nil$. This is shown in Figure 6.1.

## 6.2 Specification of Token Passing Protocol

Our stabilizing token passing protocol is based on this idea and is specified by $\mathcal{N} = Net(T, N)$ where:

- $T$ is a tree graph.
- Each $u \in V$ is a UIOA of the form described in in Figure 6.2.

We assume that there is a function $neighborSet(u)$ that lists the set of neighboring nodes of node $u$. (Recall that we allowed node automata to depend on the set of neighboring nodes and the leader function.) We assume there is some fixed circular ordering on $neighborSet(u)$ and that there is a function $Succ_u$ which when given a neighbor $v$ as its argument produces the next neighbor in the ordering. The only packet sent by this automaton is a token packet $token \in P_{data}$. We use the following additional variables:

- $pointer_u \in neighborSet(u) \cup nil$ (*pointer to token*)
- $last_u \in neighborSet(u)$ (*last neighbor $u$ received token from*)
- $queue_u[v]$ (*queue that either contains exactly one *token* packet or is empty.*)
- $free_u[v]$ : *boolean* (*true if no packet in transit on link to $v$ *).

Note that we can use domain restriction to ensure that the outbound queue for a link either contains a token or is empty. This can be done by using a single value to encode the queue that is either *token* or *nil*. We use a queue to keep the definition of $N_u$ compatible with the definition of a node automaton in Chapter 5. To make the token passing automaton a node automaton, we also have to pass the token by first enqueueing the token in the outbound queue for a link. Then, a second separate action is required to send the token from the queue to the link. However, by making $N_u$ a node automaton, we can apply the Local Correction Theorem of Chapter 5.

Notice the code we have given does not specify start states because each $N_u$ is a UIOA .

**Lemma 6.2.1** *$N = \{N_u, u \in T\}$ is a set of node automata for $T$ with $P_{data} = \{token\}$.*

. **Proof:** Simple checking of the definitions given earlier. ∎

## 6.3 Adding Local Checking and Correction

To add local checking and correction, we need to define a link predicate set $\mathcal{L}$ for $\mathcal{N} = Net(T, N)$. And to define $\mathcal{L}$ we need to define a link predicate, $L_{u,v}$, for each edge $(u, v)$ in $T$. Let $havetoken(u, v)$ be the boolean condition that is true iff there is a token in either $queue_u[v]$, $Q_{u,v}$, $queue_v[u]$, or $Q_{v,u}$. (Informally, $havetoken(u, v)$ is true iff there is a token stored on any of the links between $u$ and $v$ or on the outbound queues for such links.)

Intuitively, $L_{u,v}$ should describe the legal states of the $(u, v)$ subsystem when $\mathcal{N}$ is behaving properly and there is exactly one token in the system. Consider such a good state and suppose the pointer at $u$ is pointing to $v$. Then either the token is in transit between $u$ and $v$ OR the pointer at $v$ is pointing away from $u$.

ENQUEUE$_{u,v}$ (*output action to enqueue token in queue for neighbor $v$*)
  Preconditions:
    $pointer_u = nil$; (* $u$ has token?*)
    $v = Succ_u(last_u)$; (* $v$ is the next neighbor after $last$ ?*)
  Effect:
    $pointer_u := v$ (* point towards where token is sent*)
    Add $token$ to $queue_u[v]$ (*store token in outbound queue*)

SEND$_{u,v}$($token$) (*output action to send token to neighbor $v$*)
  Preconditions:
    $free_u[v] = true$ (* link to $v$ free?*)
    $token$ is head of $queue_u[v]$
  Effect:
    $free_u[v] = false$ (* set to false until link says its free*)
    Remove $token$ from head of $queue_u[v]$ (*empty outbound queue*)

FREE$_{u,v}$ (*link to $v$ says it is free, input action*)
  Effect:
    $free_u[v] = true$

RECEIVE$_{v,u}$($token$) (*input action, token received from neighbor $v$*)
  Effect:
    If $pointer_u = v$ then (* token received on expected link?*)
      $pointer_u := nil$ (* accept token*)
      $last_u := v$ (* update $last$*)

  All actions are in a separate class with upper bound $t_n$.

Figure 6.2: Actions for a node $u$ with respect to a neighbor $v$ in the stabilizing token passing protocol.

**Definition 6.3.1 Local Predicates for Mutual Exclusion:** *We define the local predicate $L_{u,v}$ of $\mathcal{N}$ to hold iff all the following conditions hold in the $(u,v)$ subsystem:*

- *Exactly one of $(pointer_u \neq v)$ or $havetoken(u,v)$ or $(pointer_v \neq u)$ is true.*
- *There can be at most one token packet in the combination of $queue_u[v]$, $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$.*

*Let $\mathcal{L}$ be the link predicate set that consists of $L_{u,v}$ for each edge $(u,v)$ in $T$. Let $L$ denote $Conj(\mathcal{L})$.*

Then we can show that:

**Lemma 6.3.2** *The network automaton $\mathcal{N} = Net(T,N)$ is locally checkable for $L$ using link predicate set $\mathcal{L}$.*

**Proof:** By definition $L = Conj(\mathcal{L})$. We also need to show that each $L_{u,v}$ is a closed predicate. Recall that we say that a state $s$ of $\mathcal{N}$ *satisfies* $L_{u,v}$ iff $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$. Thus we need to show that for any transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if $s$ satisfies $L_{u,v}$, then so does $\tilde{s}$. So assume that $s$ satisfies $L_{u,v}$.

Clearly we need only consider actions at nodes $u$ and $v$ since only such actions can affect variables of the $(u,v)$ subsystem. Also we need only consider actions at node $u$, because the argument for actions at node $v$ is symmetrical.

Suppose $\pi$ is a $\text{ENQUEUE}_{u,x}$ event. Then in $s$, $pointer_u = nil$. Thus we can infer that in $s$, $havetoken(u,v) = false$ and $pointer_v = u$. If $x = v$, then in $\tilde{s}$, $havetoken(u,v) = true$ and $pointer_u = v$ and $pointer_v = u$. Also in $\tilde{s}$ there is no token in $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$. On the other hand, if $x \neq v$, then in $\tilde{s}$, $havetoken(u,v) = false$ and $pointer_u \neq v$ and $pointer_v = u$. In either case, $\tilde{s}$ satisfies $L_{u,v}$.

Suppose $\pi$ is a $\text{SEND}_{u,x}(token)$ event. If $x \neq v$, then this action does not affect any of the concerned variables. So suppose $x = v$. Then in $s$, $token$ is the head of $queue_u[v]$. Thus we can infer that in $s$, $pointer_u = x$, $havetoken(u,v) = true$, and $pointer_v = u$. Also in $s$ there is no token in $Q_{u,v}$, $queue_v[u]$ and $Q_{v,u}$. All this action does is to move the token from $queue_u[v]$ to $Q_{u,v}$. Thus $\tilde{s}$ satisfies $L_{u,v}$.

```
Empty queue_u[v]                                  (*remove any token stored in queue*)
If u is the parent of v then
   If pointer_u = v then pointer_u = nil          (* take away token from child subtree*)
Else                                                          (*v is parent of u*)
   pointer_u = v                                          (* point towards parent*)
```

Figure 6.3: Code for $f(s|u,v)$, the reset function applied at node $u$ with respect to neighbor $v$

Suppose $\pi$ is a $\text{FREE}_{u,x}$ event. Then this action does not change the concerned variables and hence $\tilde{s}$ satisfies $L_{u,v}$.

Suppose $\pi$ is a $\text{RECEIVE}_{v,u}(token)$ event. Then in $s$, $Q_{v,u} = token$. Thus we can infer that in $s$, $pointer_u = v$, $havetoken(u,v) = true$ and $pointer_v = u$. Also in $s$ there is no token in $Q_{u,v}$, $queue_v[u]$ and $queue_u[v]$. Thus in $\tilde{s}$, the only change is that $Q_{v,u} = nil$, $havetoken(u,v) = false$, and $pointer_u = nil$.

The most interesting case is if $\pi$ is a $\text{RECEIVE}_{x,u}(token)$ event with $x \neq v$. We consider two cases. Suppose in $s$, $pointer_u \neq x$. Then, since the code will not accept the token in this case, it is easy to see that in $\tilde{s}$, the values of $pointer_u$,$pointer_v$,$Q_{u,v}$, $Q_{v,u}$, $queue_v[u]$ and $queue_u[v]$ are identical to their values in $s$. Hence $\tilde{s}$ satisfies $L_{u,v}$. Suppose in $s$, $pointer_u = x$. Then we can infer that in $s$, $pointer_v = u$ and $havetoken(u,v) = false$. Also in $\tilde{s}$, $pointer_u = nil$ and $pointer_v = u$ and $havetoken(u,v) = false$. Thus $\tilde{s}$ satisfies $L_{u,v}$. ∎

The next thing we have to do is to specify a local reset function $f$ to correct each $(u,v)$ subsystem. Our idea is very simple. Let us define a partial order on pairs of neighbors in $T$ such that "any edge $e$ is less than any edge below $e$ in $T$". More precisely, we let $\{u,v\} \prec \{v,w\}$ iff $u$ is the parent of $v$ and $v$ is the parent of $w$. Also, $\{u,v\} \not\prec \{w,x\}$ if the two pairs do not have a node in common. We let $<$ be the transitive closure of $\prec$. Thus, the partial order directly reflects the structure of $T$.

To allow $f$ to be a reset function using $<$ we must ensure that applying $f$ to the state of $v$ with respect to child $w$ does not affect the stability of $L_{u,v}$. This can be achieved by the following reset function $f$ described in Figure 6.3.

**Lemma 6.3.3** *The function $f$ defined in Fig. 6.3 is a local reset function for network automaton $\mathcal{N} = Net(T, N)$ with respect to link predicate set $\mathcal{L}$ and partial order $<$.*

**Proof:** Consider any edge $(u, v)$ and suppose that $u$ is the parent of $v$; the reverse case is symmetrical. We check the two conditions in Definition 5.3.7.

- **Correction:** We need to show that $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$. Consider $f(s|u,v)$. From the code in Figure 6.3 we see that in this node state, $pointer_u \neq v$ and $queue_u[v]$ is empty. Consider $f(s|v,u)$. From the code in Figure 6.3 we see that in this node state, $pointer_v = u$ and $queue_v[u]$ is empty. Thus $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$.
- **Stability:** We only need to check the stability condition for links less than $(u,v)$ in the partial order. Since $v$ is the child of $u$ it suffices to show the following: for any neighbor $w$ of $v$, if $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$ then $(s|u, s|(u,v), s|(v,u), f(s|v,w)) \in L_{u,v}$. But if we change the state of node $v$ from $s|v$ to $f(s|v,w)$, $queue_v[u]$ remains unchanged. Next, if $pointer_v \neq u$ in $s|v$, then $pointer_v \neq u$ in $f(s|v,w)$. Similarly, if $pointer_v = u$ in $s|v$, then $pointer_v = u$ in $f(s|v,w)$. Taken together, these facts imply the stability condition.

∎

Let $height(T)$ denote the height of tree $T$ which in turn is the length of the longest path from the root to a leaf. Clearly $height(f) = height(T)$.

The next lemma that follows directly from the previous lemmas and the definitions:

**Lemma 6.3.4** *The network automaton $\mathcal{N} = Net(T, N)$ is locally correctable to $L$ using link predicate set $\mathcal{L}$ and the reset function $f$ defined in Fig. 6.3.*

The following theorem follows directly by applying the Local Correction theorem (Theorem 5.4.3).

**Theorem 6.3.5** *Let $\mathcal{N}$ be the set of node automata defined in Figure 6.2, $\mathcal{L}$ be the local predicate set defined in Definition 6.3.1, and $f$ be the local reset function defined in Figure 6.3. Let $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$, Then $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(t_p)|L$ in time $t_q \cdot height(T)$, where $t_q$ and $t_p$ are constants.*

Let the symbol $*$ denote any node in the tree. We define the "correct" set of behaviors of a token passing system using a set $B$. Informally, $B$ is the set of behaviors consisting of phases in which a token is received at a node, then enqueued at the same node, and then passed to a neighbor of the node. Also, we require that a token will be received periodically at every node.

**Definition 6.3.6** *We define the set $B$ of correct behaviors of a token passing system as follows. $B$ is the set containing any behavior $\beta$ that only has actions of $\mathcal{N}$ and such that:*

- *For any $u$, after any* $\text{RECEIVE}_{*,u}(token)$ *event in $\beta$ the next event other than a* $\text{FREE}_{*,*}$ *event is an* $\text{ENQUEUE}_{u,*}$ *event.*
- *For any $u,v$, after any* $\text{ENQUEUE}_{u,v}(token)$ *event in $\beta$ the next event other than a* $\text{FREE}_{*,*}$ *event is a* $\text{SEND}_{u,v}(token)$ *event.*
- *For any $u,v$, after any* $\text{SEND}_{u,v}(token)$ *event in $\beta$ the next event other than a* $\text{FREE}_{*,*}$ *event is a* $\text{RECEIVE}_{u,v}(token)$ *event.*
- *For any $u$, and any suffix $\gamma$ of $\beta$, a* $\text{RECEIVE}_{*,u}(token)$ *will occur in $c \cdot n$ time after the start of $\gamma$, where $c$ is some constant and $n$ is the number of nodes in $T$.*

We first argue informally that the behaviors of $\mathcal{N}(t_p)|L$ are in $B$.

To make a verbal argument, we introduce some intuitive terminology. Let us say that a token is in transit between nodes $u$ and $v$ if $havetoken(u,v) = true$. We say that a token is at node $u$ if $pointer_u = nil$.

We first see that for any $s \in L$, there is at least one token in $s$. An intuitive explanation for this is as follows. We will define a search procedure to find a token in state $s$. Start at any node $u$ in the tree. If $pointer_u = nil$ then there is a token at $u$. If $pointer_u = v$ then from the local predicates, either there is a token in transit between $u$ and $v$ or $pointer_v \neq u$. If $pointer_v \neq u$ we continue the search procedure recursively at node $v$. Since we never backtrack, the search procedure cannot continue indefinitely without encountering some leaf node $w$ such that $pointer_w \neq x$, where $x$ is the parent of $w$. But if $w$ is a leaf node and $pointer_w \neq x$ then, $pointer_w = nil$ and hence the token is at $w$.

Thus we know that there is at least one token in state $s$. Suppose this token is at node $u$. Then by induction on the length of the path between $u$ and any node $w \neq u$, it is easy to see that $pointer_w \neq nil$. Similarly for any edge $(w,x)$, by induction on the length of the path between $u$ and $(w,x)$ we can show that the token is not in transit between $w$ and $x$. A similar argument shows that if in $s$ the token is in transit between $u$ and $v$, then the token is not at any node $x$, nor is it in transit on any other edge. $(w,x)$. Hence there is exactly one token in state $s$.

Once we know that there is exactly one token in any state $s$ of $\mathcal{N}(t_p)|L$, it is quite easy to prove that the behavior corresponding to any execution $\alpha$ of $\mathcal{N}(t_p)|L$ is in $B$.

For example consider any execution $\alpha$ and any state $s_i$ in $\alpha$ immediately following a $\text{RECEIVE}_{*,u}(token)$ event for some node $u$. Then it is easy to see that in $s_i$, $pointer_u = nil$. This predicate will continue to hold until a an $\text{ENQUEUE}_{u,*}$ event occurs. But if $pointer_u = nil$ in a state $s$, then (since there can be only one token in state $s$) the only actions enabled are $\text{ENQUEUE}_{u,*}$ or a $\text{FREE}_{*,*}$ event. Similar arguments can be used to show the behavior of $\alpha$ satisfies the next two properties that characterize a behavior in $B$ However, we also need to show the fourth property of a behavior: that for any $u$, and any suffix $\gamma$ of $\alpha$, a $\text{RECEIVE}_{*,u}(token)$ will occur in $c \cdot n$ time after the start of $\gamma$, where $c$ is some constant. This can be shown by an inductive argument using the properties of the *Succ* function.

Thus we can show:

**Theorem 6.3.7** *Let $\mathcal{N}^+$ be the automaton defined in Theorem 6.3.5. Then $\mathcal{N}^+$ stabilizes to the behaviors in problem $B$ in time $t_q \cdot height(T)$, where $t_q$ and $t_p$ are constants.*

## 6.4 Removing Unexpected Packet Transitions

Consider the following modification of the token passing protocol described in Figure 6.2. The modification is shown in Figure 6.4. The only difference is that the routine to receive a token at a node $u$ from node $v$ has been changed. The only change is that we no longer check whether $pointer_u = v$ before accepting the token. Let us call the resulting tree automaton $\mathcal{N}^*$.

Assume, however, that we continue to use the definitions of $\mathcal{L}$, $L$, and $L_{u,v}$ from Definition 6.3.1. Then it is quite easy to prove that $\mathcal{N}^*$ *is* not *locally checkable with*

```
MODIFIEDRECEIVE_{v,u}(token)                    (*token is received from neighbor v*)
  Effect:
    pointer_u := nil                                          (* accept token*)
    last_u := v                                               (* update last*)

  All external actions are in a separate class with upper bound t_n.
```

Figure 6.4: Modified Code for a node $u$ in a token passing protocol. The remaining code is identical to the code in Figure 6.2

*respect to $L$ and $\mathcal{L}$.* This follows from the fact that $L_{u,v}$ is not a closed predicate of $\mathcal{N}^*$.

Despite this it is not hard to prove that the behaviors of $\mathcal{N}^*|L$ are exactly the behaviors of $\mathcal{N}|L$. In fact, if we are allowed the luxury of specifying initial states, $\mathcal{N}^*|L$ is a "natural" IOA to solve the token passing problem. Suppose a protocol designer has started with $\mathcal{N}^*|L$ and now wishes to construct a UIOA that stabilizes to the behaviors specified by the token passing problem. It is interesting to note that this can be done by the following two step process:

- Transform $\mathcal{N}^*$ into $\mathcal{N}$ by adding the extra check shown in Figure 6.2.
- Transform $\mathcal{N}$ into $\mathcal{N}^+$ as shown earlier.

We would like to abstract this process. First, note that the extra check added in going from $\mathcal{N}^*$ into $\mathcal{N}$ essentially amounts to the following heuristic: *if we receive an "unexpected" packet $p$ at node $u$ from node $v$, then we do not process $p$.* Notice that in $\mathcal{N}$, a token received from $v$ when $pointer_u \neq v$ (see Figure 6.5) is certainly "unexpected". We will formalize this intuitive notion of an "unexpected" packet below. However, for the present it is important to intuitively understand why such checks for unexpected packets are useful. Consider a transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$ such that $s$ satisfies $L_{u,v}$ but not $L_{u,w}$. Then it is quite possible that there is an "unexpected" packet on channel $Q_{w,u}$ in state $s$. By adding checks for such packets in $\mathcal{N}$, we can ensure that $\tilde{s}$ satisfies $L_{u,v}$. Also note that these checks do not affect the "correct" executions of $\mathcal{N}|L$.

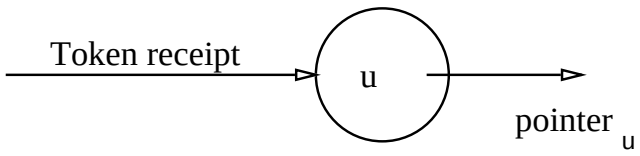



Figure 6.5: Receiving a token on a link that is not being pointed to is an unexpected packet transition. In general, an unexpected packet transition is a packet reception that could never have occurred if the receiving link subsystem was in a good state.

We now formalize these observations. First, we define the notion of weak local checkability. Intuitively, we remove the requirement that that each $L_{u,v}$ be a closed predicate and instead require only that $L$ is a closed predicate.

**Definition 6.4.1** *A network automaton $\mathcal{N}$ is weakly checkable for predicate $L$ using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{L}$ is a link predicate set for $\mathcal{N}$ and $L \supseteq Conj(\mathcal{L})$.*
- *For any transition $(s, \pi, \tilde{s})$ of $\mathcal{N}$, if $s \in L$ then $\tilde{s} \in L$*

Ideally, we would like to prove that any weakly checkable protocol can be transformed into an "equivalent" locally checkable protocol. While we do not know how to do this in general, we can can obtain such a result if the automaton is also *locally extensible*. Intuitively, an automaton is locally extensible with respect to a link predicate set if any pair of "good" adjacent link subsystem states can be extended to form a "good" state of the entire automaton.

**Definition 6.4.2** *A network automaton $\mathcal{N} = Net(G, N)$ is locally extensible with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ if the following condition is true:*

*For any two adjacent edges $(u, v)$ and $(v, w)$ in $G$, if $x \in L_{u,v}$ and $y \in L_{v,w}$ then there is some state $s \in Conj(\mathcal{L})$ such that $x = (s|u, s|(u, v), s|(v, u), s|v)$ and $y = (s|v, s|(v, w), s|(w, v), s|w)$.*

To transform a locally extensible and weakly checkable protocol $\mathcal{N}^*$ into an "equivalent" locally checkable protocol, we will add checks to $\mathcal{N}^*$ for "unexpected" packets. We formalize this notion of an "unexpected packet" (see Figure 6.5) as follows:

**Definition 6.4.3** *Consider a network automaton $\mathcal{N}$ with link predicate set $\mathcal{L} = \{L_{u,v}\}$ and some pair of neighbors $u, v$ in $\mathcal{N}$. We say that a transition $(s, \pi, \tilde{s})$ is an unexpected packet transition at $u$ with respect to $v$ if:*

*$\pi$ is a* RECEIVE$_{v,u}(p)$ *event and there is no $a, b$ such that $(s|u, a, p, b) \in L_{u,v}$.*

For example, in Figure 6.2, a $(s, \text{RECEIVE}_{v,u}(token), \tilde{s})$ transition with $s.pointer_u \neq v$ is an unexpected packet transition at $u$ with respect to $v$. We can now state a simple theorem.

**Theorem 6.4.4** *Consider a network automaton $\mathcal{N}^* = Net(G, N)$ that is weakly checkable for predicate $L$ with respect to predicate set $\mathcal{L}$. Suppose further that $\mathcal{N}$ is locally extensible with respect to $\mathcal{L}$. Then it is possible to construct another automaton $\mathcal{N}$ such that:*

- *$\mathcal{N}$ is an automaton for graph $G$ and the executions of $\mathcal{N}|L$ are identical to the executions of $\mathcal{N}^*|L$*
- *$\mathcal{N}$ is locally checkable for predicate $L$ with respect to predicate set $\mathcal{L}$.*

**Proof:** We transform $\mathcal{N}^*$ into $\mathcal{N}$ by replacing all unexpected packet transitions $(s, \pi, \tilde{s})$ in $\mathcal{N}^*$ by the null transition $(s, \pi, s)$. Then we use the local extensibility property to show each $L_{u,v}$ is a closed local predicate of $\mathcal{N}$. ∎

## 6.5 Tree Correction Theorem

The reader who has read Chapter 4 might suspect that any locally checkable protocol on trees can be made locally correctable. Thus for tree automata it seems plausible that we do not need the stronger hypothesis that the tree automaton be locally correctable. This is indeed true. Compare the following theorem to Theorem 4.3.1 (but be aware that Theorem 4.3.1 applies to shared memory tree automata) and the Local Correction theorem (Theorem 5.4.3).

**Theorem 6.5.1 Tree Correction:** *Consider any tree automaton $\mathcal{T} = Net(T, N)$ that is* locally checkable *for $L$ using link predicate set $\mathcal{L}$. Then there exists some $\mathcal{N}^+$ that is a UIOA for graph $G$ and constants $c$ and $\tilde{c}$ such that $\mathcal{N}^+$ stabilizes to the behaviors of $\mathcal{N}(c)|L$ in time $\tilde{c} \cdot height(T)$.*

The proof of this theorem is extremely similar to that of Theorem 4.3.1 of Chapter 4. However, since we can no longer do snapshots and resets atomically, we need to use the local snapshot and reset protocols defined in Chapter 5. Thus we have to add actions for local checking and correction as in the proof of the Local Correction Theorem, Theorem 5.4.3. However, there are two differences. We will assume that for every $(u, v)$ subsystem in which $v$ is the parent of $u$, the child $u$ performs the checking/correction. The local snapshot protocol is identical to the protocol of Chapter 5, but the local reset protocol is a little different. This is sketched in Figure 6.6. The figure should be compared with the right hand diagram in Figure 5.9.

The basic idea is that the reset response carries the state $b$ of the parent at the instant the response was sent. When the child gets the response, the child sets its state to $f(b)$, where $f$ is a function that we describe next. Basically, $f$ is chosen such that for every state $b$ of the parent node $v$, $(f(b), nil, nil, b) \in L_{u,v}$. In other words, $f$ is chosen so that we can reset the link subsystem to a good state by only changing the state of the child node. Of course, that is the basic idea in the proof of Theorem 4.3.1. The only tricky part is to argue that we can find such a function $f$. The proof is again similar to the proof of Theorem 4.3.1: we first normalize the original protocol $\mathcal{T}$ to get rid of "useless" node states that can never occur in global states in which all local predicates are true. We then show that the required function $f$ exists for the normalized protocol.

The Tree Correction theorem is not a corollary of the Local Correction theorem. Recall in Chapter 5 that when we applied a local reset function to the state of leader node $u$ with respect to node $v$, the resulting state of node $u$ can only depend on the previous state of node $u$. However, in the proof of the Tree Correction theorem we require the the resulting state of node $u$ to depend on the previous state of node $v$!

Finally, we note that we could have derived the stabilizing mutual exclusion protocol by showing that it was locally checkable and then using the Tree Correction theorem directly.

## 6.6 Summary

In Chapter 4, we showed (in a shared memory model) that any locally checkable protocol on a tree could be stabilized in time propostional to the height of the tree.

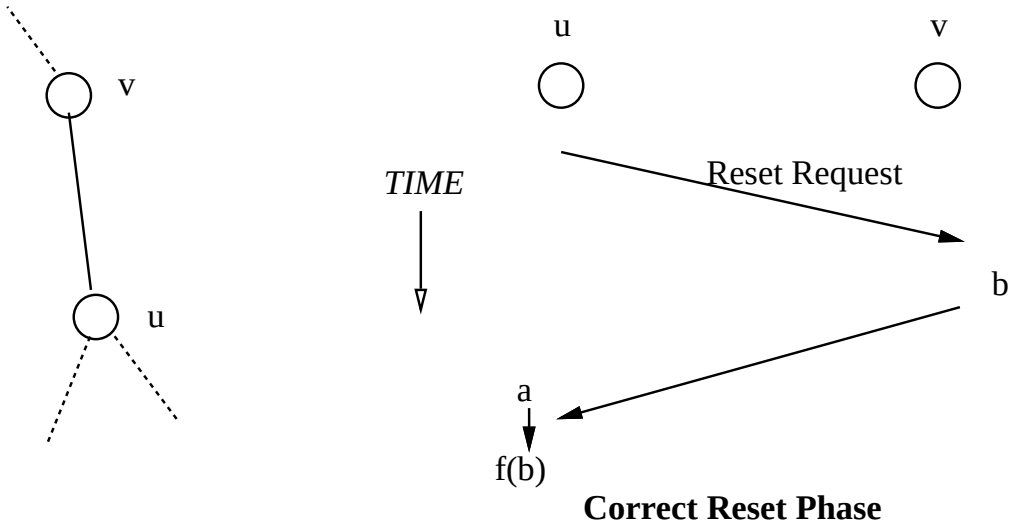



Figure 6.6: Sketch of a reset phase used for Tree Correction. Node $v$ is the parent of node $u$ in the tree.

This chapter shows that this theorem can be extended to the message passing model (Tree Correction theorem, Theorem 6.5.1). We also described a simple application – the problem of token passing on a tree. A stabilizing solution to this problem can be derived using either the Local Correction or Tree Correction theorems.

The token passing problem suggests a simple strategy for stabilizing protocols. First, we try to add a small amount of state to make the protocol locally checkable. Recall that we added a pointer to each node for this purpose in the token passing protocol. Then we combine the original protocol with another protocol that computes a spanning tree. Finally, we do local correction on the resulting spanning tree. Although we have not done so, it is important to formally describe how an arbitrary protocol $P$ could be composed with a spanning tree protocol so that the net effect is the same as if $P$ were running on the final tree.

Local checkability requires that each local predicate be a closed predicate. Removing unexpected packet transitions (Figure 6.5) is a useful heuristic that is often sufficient to ensure that each local predicate is closed.

The token passing protocol on a tree can be generalized to passing a constant number of tokens on a tree. In this case, we replace the pointer variable $pointer_u$ at each node $u$ by a variable $token_count_u[v]$ (one for each neighbor $v$) that keeps track of the number of tokens that are in the direction of neighbor $v$. The local predicates

have also to be suitably modified.

# Chapter 7

# Stabilizing Network Reset

The stabilizing network reset protocol described in this chapter is the link between the previous two chapters (which were about Local Correction) and the last two chapters of the thesis (which are about Global Correction).

The major service provided by a network reset protocol is *synchronization* of the nodes in a network. In the first section of this chapter, we informally introduce the concept of synchronization, and discuss why this service is useful. In Section 7.2 we review existing reset protocols. Then in Section 7.3 we specify the reset problem. Previous specifications of reset protocols have been state-based. Our specification of the reset problem is novel in that it is based on external behaviors. In the next section (Section 7.4), we give an overview of our reset protocol. Then in Section 7.5, we formally specify our reset protocol using a reset automaton.

Sections 7.6 and 7.7 are devoted to showing that the reset automaton is a *stabilizing* solution to the reset problem. We do this using the Local Correction theorem developed in Chapter 5. We show that the reset automaton is locally checkable by exhibiting a set of closed local predicates for the reset automaton. Then we show that the reset automaton is locally correctable by showing that the local predicates are independent – i.e., the partial order that formalizes the dependency relation between the predicates is the trivial partial order. Thus the reset protocol stabilizes in constant time. Next, (in Section 7.7), we show that once the reset automaton is in a state that satisfies all local predicates, then the behaviors exhibited by the reset automaton are indeed the behaviors specified by the reset problem. This completes the proof that the reset automaton is a stabilizing solution to the reset problem.

The last two sections of the chapter try to abstract some general principles based on the example of the stabilizing reset protocol. The reset protocol in this chapter is based on an existing non-stabilizing reset protocol [AAG87] that works in networks where links can fail and recover. Section 7.8 suggests that this is no accident – locally checkable protocols that work in networks where links can fail and recover are likely to also be locally correctable.

Because this chapter is very long, we offer two suggestions for reading. First, constantly consult the roadmaps at the beginning of the chapter and each long section. Second, it is hard to appreciate the specification of the reset problem until one sees why it is useful. Chapters 8 and 9 describe important applications of the reset protocol. Readers may prefer to read Chapter 8 after reading the specification in Section 7.3 and before reading the rest of Chapter 7.

## 7.1 Synchronization

A reset protocol is used to *synchronize* all the nodes in a network. Before we describe what we mean by synchronizing *all* the nodes in a network, it is helpful to understand a form of synchronization between *a pair* of nodes in a network. Such synchronization is provided by a Data Link protocol [Spi88a]. We will then see that in essence a network reset protocol generalizes the guarantees of a Data Link protocol to *multiple nodes* in a network.

### 7.1.1 Data Link Synchronization

Consider a pair of neighboring nodes in a network, say $u$ and $v$. Suppose the physical link between two nodes in a network can periodically crash and recover. The Data Link protocol is responsible for providing notification (as to whether the link is up or down) to the users at $u$ and $v$. Thus at each node, the Data link protocol issues Link up and Link down actions to signify that the link is operational or not operational respectively. If the network is asynchronous, it is impossible to provide a Link down event (or Link up event) at exactly the same instant at both $u$ and $v$. But if Link up and down events are reported independently (and possibly at different times) at both ends, the Data Link must provide some additional functions to *synchronize* $u$ and $v$.

The synchronization requirements for a Data Link protocol can be stated elegantly [Spi88a] as follows. First, for node $u$ (or node $v$) we can define an *operating interval* at node $u$ (or at node $v$) to be the time from a Link up event at that node until the next Link down event at that node. If an operating interval does not end with a Link down event we say that the interval is a *final interval.* Thus each execution of the Data Link protocol induces a set of operating intervals at both nodes. Then for synchronization, we require that there is a symmetric relation between intervals at the two nodes (called a *mating* relation in [AE86]) such that:

- An operating interval can be mated to at most one other operating interval.
- Suppose an operating interval $\alpha$ at $u$ is mated to an operating interval $\beta$ at $v$. Then the sequence of messages received by $v$ in $\beta$ must be a prefix of the sequence of messages sent by $u$ in $\alpha$. (This is often called the prefix property of Data Link protocols.) Also, if $\alpha$ is a final interval, then so is $\beta$ and in this case the sequence of messages received by $v$ in $\beta$ must be identical to the sequence of messages sent by $u$ in $\alpha$.
- Suppose an operating interval $\alpha$ at $u$ is mated to an operating interval $\beta$ at $v$ and an operating interval $\alpha'$ at $u$ is mated to an operating interval $\beta'$ at $v$. Then if $\alpha'$ occurs later than $\alpha$ then $\beta'$ occurs later than $\beta$.

The mating relation for two nodes $u$ and $v$ is sketched using the time-space diagram shown in Figure 7.1. We have depicted Link up events by horizontal lines. The Link down events between Link down events are depicted by crosses. An arrow from $v$ to $u$ depicts a packet sent by $v$ that is successfully delivered at $u$, and vice versa. An arrow from $v$ that does not reach $u$ is a packet sent by $v$ that is *not* delivered at $u$. In the figure, the second operating interval at $u$ mates to the second operating interval at $v$. Also, the third operating interval at $u$ mates to the fourth operating interval at $v$ and the two intervals are final intervals. Notice that the sequence of packets received by $u$ in its second interval is a prefix of the sequence of packets sent by $v$ in its second interval. Also notice that all packets sent in the two final intervals are delivered.

Why does this mating relation provide a useful form of synchronization? It does so because the synchronization relation guarantees that the behavior of a node during an operating interval is *what might have occurred in some asynchronous execution of the*

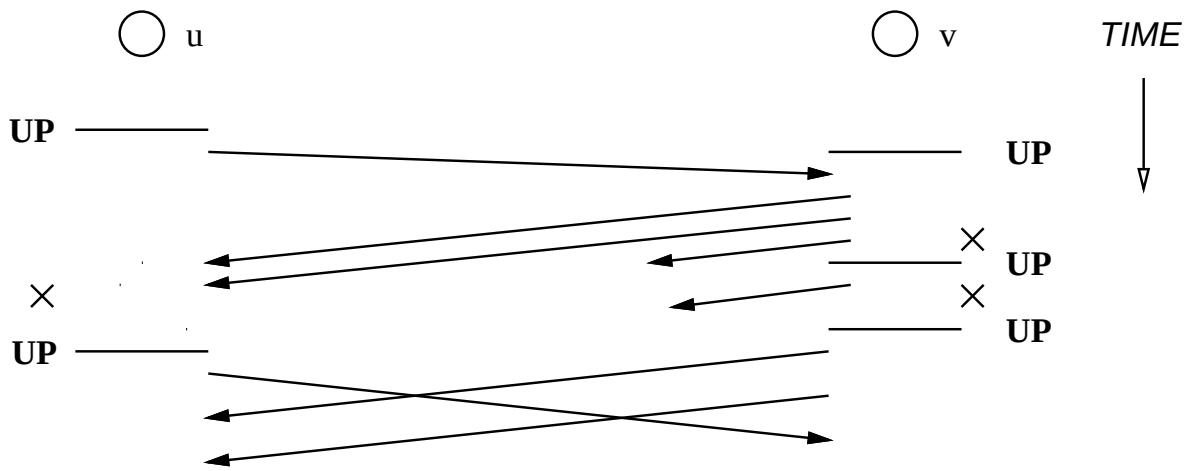



Figure 7.1: Illustrating the Mating Relation

*Data Link protocol in which there were no link failures.* This is a crucial abstraction that allows users of the Data Link protocol to deal with failures in a simple way.

Almost identical forms of synchronization are provided by virtual circuit protocols ([Spi88a]) and transport protocols. Thus all these protocols essentially synchronize two nodes in a network.

### 7.1.2 Network Synchronization

The synchronization provided by a network reset protocol is a generalization of the synchronization guarantees of a Data link. A reset protocol synchronizes *all* the nodes in the network. Informally, the reset problem [Fin79] is to design a reset service that can be superimposed on any other distributed protocol $P$. The reset may be invoked at any node, and its effect is to output a signal at all the nodes of the system in a consistent way. We use $\Sigma$-messages to denote the messages sent and received by protocol $P$.

By consistent, we mean the following. As in the Data Link protocol, the reset protocol induces signal intervals at each node (i.e., intervals between consecutive signal events at a node). Then for each pair of neighbors $u$ and $v$ we require that the signal intervals at $u$ and $v$ can be mated as described earlier. For example, if we replaced the Link up events with Signal events in Figure 7.1 and ignored Link down events, then

Figure 7.1 could equally well describe the pairwise mating relation provided by a reset protocol. Notice, however, that in a Data Link the Link up events occur independently on each link and so the the operating intervals on each link adjacent to a node $u$ can be different. However, in a Reset protocol, the Signal events induce signal intervals on a per node basis.

For example this means that the sequence of $\Sigma$-messages *sent* by any node $u$ to any neighbor $v$ after the last signal at $u$ must be equal to the sequence of $\Sigma$-messages *received* by $v$ after its last signal.

Why is this called a reset service? Suppose $S_u$ defines the set of "legal" start states for every node $u$. Suppose the "legal executions" of $P$ are those executions in which the initial state of every node $u$ belongs to $S_u$ and the initial state of every channel is a state in which the channel contains no $\Sigma$-messages. It's natural to define the legal states of $P$ as those states that occur in legal executions of $P$. Suppose we now superimpose a reset service over $P$. Suppose also that whenever a node $u$ receives a signal, we locally reset the state of node $u$ (i.e., the local state of protocol $P$ at node $u$) to some state in $S_u$. Then after every node has received its last signal, protocol $P$ is in a legal state. In other words, the signals provide a consistent time point for every node $u$ such that we can *globally reset* protocol $P$ to a legal state by *locally resetting* each node $u$ at its time point. The time point for each node is the instant it receives a signal.

Thus a global reset service is much like the reset button on a computer. After the reset button is pushed, the computer is restored to a "good" state. However, globally resetting a network to a "good" state is much more challenging than resetting a single node. In a network, reset requests can arrive at any node and the signals must be delivered at every node in a consistent fashion. Ideally, we would like the signals to be delivered to every node at the same instant. However, since this seems to be impossible in a distributed system, we settle for delivering the signal in a consistent manner (see above).

Why is a reset service useful? It was introduced in [Gal76, Fin79] as a tool for converting any protocol that works in a so-called *static network* to work in a so-called *dynamic network*. A static network, as the name suggests, is a network in which the topology of the network remains fixed during the execution of the algorithm. A dynamic network, by contrast, is a network in which nodes and links can crash, thereby changing the topology. However, it is assumed that topology changes eventually stop

and that some node in the final topology can detect the fact that there has been a topology change. Roughly, the idea behind [Fin79] is that any node that detects a topology change makes a reset request. If successful, the reset request results in restarting the static protocol. This methodology is quite practical. For instance, the Autonet local area network uses a version of [Fin79] to cope with failures.

Besides its use in dynamic networks, a reset protocol is also useful in a stabilizing setting. As we show in Chapters 8 and 9, *a stabilizing reset protocol is an important tool for the design of other stabilizing protocols.* Notice that in order to use the reset service in dynamic networks [Fin79], some node must detect the last topological change. More generally, suppose that any bad state of a network protocol can be detected locally by some node. This corresponds to what we have called a locally checkable protocol in Chapter 5. As we will see in Chapter 8, any locally checkable protocol can be stabilized using a stabilizing reset protocol. Intuitively, our idea is similar to that of [Fin79]. We add actions to each node to make a reset request if the node locally detects a bad state of the network.

The technique of using a reset protocol to stabilize a locally checkable protocol is quite different from the techniques developed in Chapter 5 and 6. In Chapter 5, a locally checkable and correctable protocol is stabilized by doing *independent local resets* of each link subsystem. In Chapter 6, a locally checkable protocol on a tree is also stabilized by doing *independent local resets* of each link subsystem.

By contrast, we can stabilize a locally checkable (but perhaps not locally correctable) protocol by doing a *coordinated global reset* of the entire network. As one might guess, there is a performance penalty in using a network reset. The stabilization time of a solution that uses local correction is proportional to the height of the underlying partial order (see Local Correction Theorem, Theorem 5.4.3). The stabilization time of a solution that uses tree correction is proportional to the height of the tree (see Tree Correction Theorem, Theorem 6.5.1).

However, the stabilization time of a solution that uses global correction is proportional to the number of nodes in the network. Because the height of the partial order (or the height of a tree) typically is smaller than the number of nodes, we have the following rule of thumb. If a protocol is locally correctable or runs on a tree, then it pays to use the techniques of Chapter 5 or 6. However, if a locally checkable protocol cannot be shown to be locally correctable, then the network reset approach provides a (perhaps less efficient) stabilizing solution. It is perhaps elegant that the network

reset protocol is itself stabilized using the local reset approach of Chapter 5.

In this chapter we introduce the most efficient known stabilizing network reset protocol. We do so by applying the method of local checking and correction of Chapter 5 to an existing reset protocol described in [AAG87] The space overhead of the protocol is logarithmic. Our reset protocol stabilizes in constant time.

## 7.2 Existing Solutions

In Chapter 4, we described a stabilizing reset protocol due to Arora and Gouda [AG90]. Their protocol was described in terms of a shared memory model but it appears that it can be adapted to work in our message passing model. [AG90] also describes a stabilizing protocol to build a spanning tree of the network. For the spanning tree protocol, it is assumed that processes have unique identifiers, and that there is some *a priori* bound $K$ on the number of nodes in the network. The IDs and $K$ cannot be corrupted by transient errors. The stabilization time of the spanning tree protocol is $O(K)$, where $K$ is a non-volatile bound on the number of nodes in the network. Their protocol will also work correctly if $K$ is an upper bound on the diameter of the final network. However, in a network in which the topology can change arbitrarily, often the only reasonable bound on the diameter of the network is a bound on the number of nodes in the network. Secondly, as we will see in Chapter 8, their spanning tree protocol is based on a routing algorithm that works poorly in practice.

Katz and Perry [KP90] describe a stabilizing reset protocol. Their approach requires the election of a leader and the stabilization time of their approach is $O(n^2)$ where $n$ is the number of nodes in the network.

On the other hand, the reset protocol we describe does not require node IDs, and takes $O(n)$ time to stabilize, where $n$ is the *actual* number of nodes in the network. Our protocol does not require the computation of a spanning tree. Thus it can be used to build a stabilizing spanning tree protocol as we show in the Chapter 8.

## 7.3 Specifying the Desired Behaviors of a Reset Protocol

This section is divided into four subsections. First, we describe the external interface to the reset protocol. Then, we give an overview of the specification and some of its difficulties. After this motivation, we formally specify the reset problem and then briefly discuss alternative specifications.

### 7.3.1 Interface to Reset Protocol

We first describe the external interface to the reset protocol.

A reset service is modelled as a network automaton $\mathcal{R} = Net(G, N)$. The external interface for any node $u$ in the network is shown in Figure 7.2. We have the usual interfaces to send and receive *packets* between neighbors as described in Chapter 5. However, in addition each node also has interfaces to send and receive *messages* on behalf of external users of the reset service. We assume that every message $m$ is drawn from some message alphabet $\Sigma$, and that $\Sigma \subset P_{data}$, where $P_{data}$ is the data packet alphabet used by the network automaton. Intuitively, the messages sent by users to the reset service will be relayed between nodes of the network using packets.

Thus node $u$ also has an input action $\text{SENDM}_{u,v}(m)$ by which an external user can send a message to neighbor $v$. Similarly node $u$ has an output action $\text{RECEIVEM}_{v,u}(m)$ by which the reset service can deliver a message $m$ from the user at node $v$ to the user at node $u$. There is also a $\text{FREEM}_{u,v}$ output action that is used by the reset service to indicate that it is ready to accept another message from node $u$ to node $v$. Thus the external interface between a reset service and its users essentially mimics the interface offered by a UDL (see Chapter 5) except that packets are replaced by messages. This is an important property that will be exploited later.

However, the interface between a user and a reset service is richer than the interface between a user and a set of UDLs. This is because the reset service at node $u$ offers two additional actions: an input action $\text{REQUEST}_u$ and an output action $\text{SIGNAL}_u$. Intuitively, the $\text{REQUEST}_u$ action is used by the user at node $u$ to request a network reset, and the $\text{SIGNAL}_u$ action is used to inform the user at node $u$ that a reset has

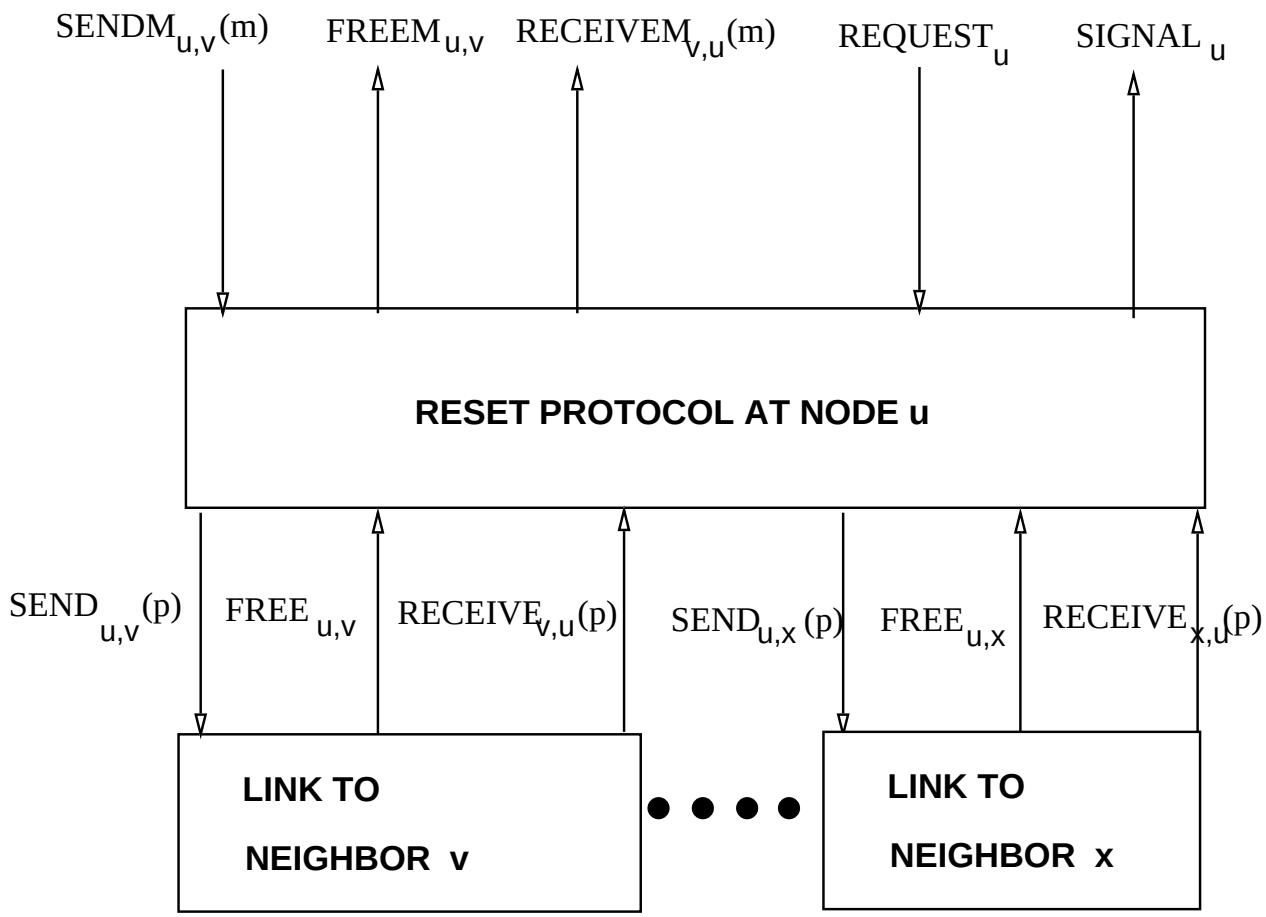



Figure 7.2: Interface specification for reset service

been completed. We will refer to any event of the form $\text{SIGNAL}_u$ as a *signal event* and any event of the form $\text{REQUEST}_u$ as a *request event.*

### 7.3.2 Difficulties in Specifying the Reset Problem

The ideal behaviors of a non-stabilizing reset protocol can be specified in terms of three properties: *timeliness, consistency, and causality.* Intuitively, a behavior is timely if, in the absence of reset requests, free events are delivered in constant time and sent messages are delivered in constant time. A behavior is consistent if there is a symmetrical *mating* relation between signal intervals at neighbors. Finally, a behavior is causal if reset signals are only caused by reset requests and reset requests result in reset signals.

As in a UDL, we cannot guarantee that any message sent by a user from say $u$ to $v$ will be received at $v$ unless (see Figure 7.2) this message is sent after $u$ performs a $\text{FREEM}_{u,v}$ action. If this is true (and no other message is sent from $u$ to $v$ between the free action and the send action), then we will say that the send action is *safe*. We will only require delivery of messages corresponding to safe message send events. The specification will allow other messages to be dropped.

Any specification of the behaviors of a stabilizing reset protocol has to take into account three anomalies that do not occur in the specification of a non-stabilizing reset protocol:

- The first message sent on any link may not be *safe* in that it may not be preceded by a free event.
- Some of the initial messages that are delivered may be *abnormal* in that they do not correspond to any messages sent. It appears that no stabilizing reset protocol can guarantee that there is some suffix of every behavior that contains no abnormal message delivery.[1]
- Some of the initial signal actions may not be "caused" by any reset request.

[1] A stabilizing reset protocol can begin in a state in which all links can have arbitrary messages stored. There can be executions that contain no state in which all links are empty of messages. Any suffix of such an execution will have abnormal message deliveries.

We handle these difficulties as follows. We add the following condition to the timeliness property: within linear time of the start of any behavior of the reset protocol, all received messages are *normal* – i.e., correspond to some message sent. We weaken the mating relation so that it is possible to receive abnormal messages in a signal interval – however, normal messages received in a signal interval must have been sent in a mated interval. Finally, we relax the consistency property and ask only that causality holds in linear time after the start of a behavior. Precise definitions are given in the next subsection.

### 7.3.3 Formal Specification of Reset Problem

Recall the following notation. When we say that an event $a_j$ occurs within time $t$ in $\beta$ we mean that $a_j.time \leq \beta.start + t$. When we say that a time $t$ is a constant, we mean that $t = ct_l + c't_n$ where $c, c'$ are some scalar constants and $t_n$ and $t_l$ are the default node and link delays respectively. In this and following chapters, we will also use the following notation borrowed from complexity theory. When we say that $t = O(n)$, we mean that $t \leq cn$, where $c$ is some constant time and $n$ is the number of nodes in the graph $G$. This reflects a linear dependence on the size of the input, if we consider the input to be the network graph. If we say that an event $a_i$ occurs within $O(n)$ time in a behavior $\beta$, we mean that $a_i.time - \beta.start = O(n)$. If we say that an event $a_j$ occurs within $O(n)$ time after an earlier event $a_i$ in behavior $\beta$, we mean that $a_j.time - a_i.time = O(n)$; in this case, we will also say that event $a_i$ occurs within $O(n)$ time *before* event $a_j$ in behavior $\beta$. Sometimes we will say linear time to mean an interval of time that is $O(n)$ in duration.

As in a UDL, we cannot guarantee that any message sent by a user from say $u$ to $v$ will be received at $v$ unless (see Figure 7.2) this message is sent after $u$ performs a $\text{FREEM}_{u,v}$ action. If this is true (and no other message is sent from $u$ to $v$ between the free action and the send action), then we will say that the send action is *safe*. We formalize this restriction by defining what it means for a message send event to be *safe*.

**Definition 7.3.1** *Consider any behavior $\beta$ of a reset protocol that has the interface shown in Figure 7.2. We say that an action $a_j = \text{SENDM}_{u,v}(m)$ is safe in $\beta$ if:*

- *There is some $\text{FREEM}_{u,v}$ action in $\beta$ before $a_j$ and*

- *No other action of the form* $\text{SENDM}_{u,v}(*)$ *occurs between* $a_j$ *and the* $\text{FREEM}_{u,v}$ *action.*

Clearly we would eventually like every message that is *received* at $u$ from say $v$ to correspond to some message *sent* from $v$ to $u$. We will require that there is at most one message in transit from $v$ to $u$. This will make it easy to make a correspondence between received and sent packets. Thus:

**Definition 7.3.2** *We say that event* $a_k = \text{RECEIVEM}_{v,u}(m)$ *in* $\beta$ *is a normal receive at* $u$ *from* $v$ *iff:*

- *There is some* $a_j = \text{SENDM}_{v,u}(m)$ *in* $\beta$ *that occurs before* $a_k$ *in* $\beta$ *and*
- *There is no* $\text{RECEIVEM}_{v,u}(*)$ *event between* $a_j$ *and* $a_k$ *in* $\beta$.

*We will refer to the earliest* $\text{SENDM}_{v,u}(m)$ *in* $\beta$ *that occurs before* $a_k$ *(and satisfies the above properties) as the send corresponding to* $a_k$.

Notice that if the sender ignores the free notification and sends a message $m$ several times before it is received, we make a correspondence between the receive and the earliest send event.

### Specifying Timeliness

Next, we formalize the timeliness property. We we say that a behavior is timely if it satisfies four conditions for every pair of neighbors $u$ and $v$. First all receive events that occur after linear time in $\beta$ are normal. In other words, after at most $O(n)$ time, each packet received corresponds to some packet sent. Also any normal receive event occurs within $O(n)$ time after the corresponding send event. This is shown in Figure 7.3.

The second condition is that either the user at $u$ periodically receives a free action indicating that the link to $v$ is free or else a signal event occurs periodically at either $u$ or $v$. Third, any message sent by $v$ to $u$ is either delivered in constant time or else a signal event occurs (at either $u$ or $v$) after the message is sent. The essence of the second and third conditions is that, in the absence of signal events, free events are

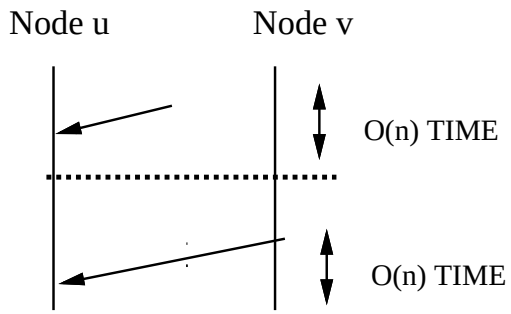



Figure 7.3: Normal message receipt after linear time: There is a send action at $v$ corresponding to any message received at $u$ after $O(n)$ time. Also any normal receive event occurs within $O(n)$ time after it is sent.

delivered periodically and sent messages are delivered in constant time. Intuitively, signal events are caused by reset requests; if reset requests are made continuously, then the reset protocol cannot guarantee periodic free events or the delivery of messages.

The fourth timeliness condition is important (for example in Chapter 8) in applications. It says that signals cannot keep occurring at $u$ without also occurring at a neighbor $v$. More precisely, if a signal event occurs at $u$ then a signal event must occur at $v$ in linear time before or after the signal event at $u$. However, because the reset protocol can start in an arbitrary state, we relax this requirement and only ask that this property hold after a linear amount of time.

**Definition 7.3.3** *We say that a behavior $\beta$ is timely if for every pair of neighbors $u, v$:*

1. **Normal Receipt of Messages**: *There is some constant $c$ such that every every receive event that occurs at time greater than $\beta.start + c \cdot n$ is normal. Also if $a_j$ is any normal receive event and $a_i$ is the send corresponding to $a_j$, then $a_j$ occurs within $O(n)$ time after $a_i$.*

2. **Periodic Free Events**: *Consider any $t$-suffix $\gamma$ of behavior $\beta$. Then in $\gamma$ either a $\text{FREEM}_{u,v}$ occurs within constant time, or a signal event occurs at $u$ within $O(n)$ time, or a signal event occurs at $v$ within $O(n)$ time.*

3. **Timely Message Delivery**: *Suppose $a_j$ is a safe $\text{SENDM}_{u,v}(m)$ event in $\beta$. Then after $a_j$ either a $\text{RECEIVEM}_{u,v}(m)$ occurs within constant time, or a signal*

*event occurs at $u$ within $O(n)$ time, or a signal event occurs at $v$ within $O(n)$ time.*

4. **Signals at a Node induce Signals at Neighbors**: *There is some constant $c$ such that for every* SIGNAL$_u$ *event $a_j$ that occurs at time greater than $\beta.start + c \cdot n$ there is a* SIGNAL$_v$ *event that occurs in linear time before or after $a_j$.*

### Specifying Consistency

Before we formalize the consistency property, we formalize the notion of a signal interval at a node in a behavior.

**Definition 7.3.4** *Consider a behavior $\beta$. A signal interval at node $u$ in $\beta$ is a contiguous subsequence of $\beta$ that:*

- *Begins with either the start of $\beta$ or a* SIGNAL$_u$ *event and*
- *Ends with either the first* SIGNAL$_u$ *event that occurs after the start of the interval or (if there is no such* SIGNAL$_u$ *event) the interval ends with the end of $\beta$. In the latter case we call the signal interval a final interval.*

Thus any behavior induces signal intervals at each node. We now specify the consistency condition by requiring a mating relation between signal intervals. However, the mating relation is weaker than the relation for Data Links because it does not require the prefix property. The third property is a little tricky. Consider Figure 7.4. Suppose the send at $v$ is safe and is followed by a free action at $v$ that is in the same signal interval at $v$. Then we require that the sent message is delivered and the delivery event occurs between the send and the free events. In essence, this states that all messages (except possibly the last message) sent safely in a signal interval at $v$ are delivered at $u$.

**Definition 7.3.5** *We say that a behavior $\beta$ satisfies the consistency property if for every pair of neighbors $u, v$ there is a symmetric relation (called a* mating *relation) between signal intervals at $u$ and $v$ such that:*

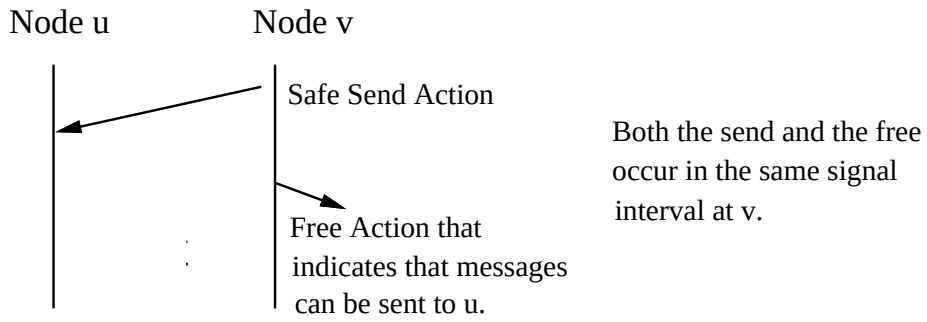



Figure 7.4: Successful sending of messages: between the sending of a message and the free action that indicates that the next message can be sent, either the message is delivered or a signal occurs at the sender.

1. **At most one mate**: *A signal interval at $u$ can be mated to at most one signal interval at $v$.*

2. **Signal Intervals that communicate are mates**: *Let $a_k$ be any normal receive event at $u$ from $v$ in $\beta$ and let $a_m$ be the send event corresponding to $a_k$. Then the signal interval at $u$ containing $a_k$ is mated to the signal interval at $v$ containing $a_m$.*

3. **Successful Sending of Messages**: *Consider any safe* $\text{SENDM}_{v,u}(m)$ *event $a_j$ and a later* $\text{FREEM}_{v,u}$ *event that occur in a signal interval at $v$. Then between these two events in $\beta$ there must be a normal* $\text{RECEIVEM}_{v,u}(m)$ *event $a_k$ such that $a_j$ is the send corresponding to $a_k$.*

4. **Mating of Final Signal Intervals**: *A final interval at $u$ can only mate to a final interval at $v$.*

5. **Mating Relation Preserves Temporal Ordering**: *Suppose a signal interval $S_u$ at $u$ is mated to a signal interval $S_v$ at $v$ and a signal interval $S'_u$ at $u$ is mated to a signal interval $S'_v$ at $v$. Then if $S'_u$ occurs later than $S_u$ then $S'_v$ occurs later than $S_v$.*

Suppose $S_u$ and $S_v$ are signal intervals at $u$ and $v$ respectively that are mates. Notice that as compared to a Data Link specification, we have weakened the requirements for a mating relation: we no longer require that the sequence of message received by $v$ from $u$ is a prefix of the sequence sent from $u$ to $v$. *However, if all received messages*

*are normal and all sent messages are safe, then the third consistency condition does imply the prefix property.* The third and fourth consistency conditions also imply that if $S_v$ is a final interval, and if all received messages are normal and all sent messages are safe, then the two sequences are *identical.*

The reader may wonder whether it is sufficient to specify consistent behavior only in final intervals. In that case, the consistency condition is much simpler. However, if the stabilizing reset protocol is to be used as a tool to build other stabilizing protocols (which is what we do in the next two chapters), then we claim that the reset protocol *must* make some guarantees during non-final intervals. A typical user of the reset protocol will be constantly doing some form of checking (see Chapter 8 for example) and will make a reset request if the checking detects a violation. But if the reset protocols exhibits arbitrary behavior during non-final intervals, then the user may always detect violations. These in turn lead to continuous reset requests which prevent the forming of a final interval. In other words, final intervals are only guaranteed if the user stops making reset requests; but users may only stop making reset requests if the reset protocol guarantees some form of consistency during non-final intervals. As another example, some user protocols may periodically make reset requests to start a new phase of the protocols. Such protocols (Chapter 9 describes an example of such a protocol) never stop making reset requests!

### Specifying Causality

A causal behavior satisfies two intuitive conditions: that signal events are only caused by reset requests and reset requests result in signal events. Ideally, any signal event must be preceded by a reset request that occurs a linear amount of time before the signal event. Notice that this guarantees that all signal events will disappear in linear time after the last reset request in a behavior. However, because the reset protocol can start in an arbitrary state, we relax this requirement and only ask that this property hold after a bounded amount of time. But a causal behavior should also ensure the flip side of the above condition: reset requests should result in signal events. A protocol that simply ignored reset requests would be useless. We specify the second condition by requiring that a reset request at a node $u$ is followed in linear time by a signal event at node $u$.

**Definition 7.3.6** *We say that a behavior $\beta$ is causal if:*

1. *There is some constant $c$ such that every signal event $a_k$ that occurs at time greater than $\beta.start + c \cdot n$ is preceded by a request event $a_j$ that occurs in linear time before $a_k$.*

2. *A* SIGNAL$_u$ *event occurs within linear time after any* REQUEST$_u$ *event.*

We are now ready to describe the behaviors that should be produced by a reset protocol.

**Definition 7.3.7** *We define the reset problem RP to be the behaviors $\beta$ that are timely, consistent and causal.*

The following lemma is useful because it tells us that every suffix of a behavior in $RP$ is also in $RP$.

**Lemma 7.3.8** *If a behavior $\beta$ is in RP, then any t-suffix of behavior $\beta$ is in RP.*

**Proof:** The proof consists mostly of looking at each property in the definition of $RP$ and showing that if a suffix of $\beta$ does not have the property then neither does $\beta$. The only tricky case is the property that all messages received after at most $O(n)$ time from the start of a behavior are normal. However, this can be deduced from the fact that the send event corresponding to a normal receive event in $\beta$ occurs at most $O(n)$ time before the receive event. ∎

### 7.3.4 Alternative Specifications of Reset Problem

Traditional definitions (i.e., [AAG87]) of a reset service define the correctness of a reset service in terms of the states of protocol $P$, the user of the reset service. Our definition is more modular because it focuses on the behavior of the reset subsystem without any reference to the states of the users of the reset service.

Next, one might like a reset protocol to satisfy a much stronger consistency property than the one specified above. In the stronger condition, the mating relation between signal intervals would also be *transitive.*. For instance, suppose $u$, $v$ and $w$ are connected in a cycle such that $u, v$, and $v, w$, and $w, u$ are all neighbors. Also suppose

that a signal interval $S_u$ at $u$ mates to a signal interval $S_v$ at $v$, $S_v$ mates to some $S_w$ at $w$, and $S_w$ mates to some interval $S'_u$ at $u$. Then our consistency specification allows $S_u$ to be different from $S'_u$. However, transitivity requires that $S_u = S'_u$. The stronger condition seems to capture the essence of network synchronization in that the signal events provide consistent time points across the entire network. However, we show in the appendix that our reset protocol (and the reset protocol of [AAG87]) *does not* satisfy the stronger condition. The applications in this thesis do not need the stronger condition.

*Note that the weaker condition does imply that there is a transitive mating relation between final signal intervals at all nodes in the network.*

Also the specification of the behaviors of a stabilizing reset protocol is complicated by anomalies that occur at the beginning of the behavior such as abnormal messages and signals that are not "caused" by any requests. Given this difficulty, we might be tempted to specify the reset problem using *suffixes* of the behaviors of a non-stabilizing reset protocol. This results in a more elegant definition. However, if we choose that definition, then we do not know any reasonably simple proof technique to show that a protocol stabilizes to behaviors that are *suffixes* of a specified set of behaviors.[2]

## 7.4 Overview of the solution

In this section, we will give an overview of a stabilizing reset protocol. Our solution basically consists of stabilizing the reset protocol of [AAG87] using the method of Chapter 5. In the first subsection, we describe a simple reset protocol that is not stabilizing. In the next subsection, the problems of the simple reset protocol are used to motivate the main ideas behind our reset protocol. Next, we give an overview of the code. Finally, we end this section with an example execution of our reset protocol. The next section contains a formal description of our reset protocol.

[2]Such proofs seem to involve showing that every initial state of an automaton $A$ is a reachable state of another automaton $B$. But familiar inductive proof techniques (such as invariant arguments, progress metrics etc.) do not seem to suffice to show that one state is reachable from another state.

### 7.4.1 Problems with a Simple Reset Protocol

Amazingly, the consistency condition for normal reset behaviors can be guaranteed by the following *Simple Reset Protocol* (SRP). This is a non-stabilizing protocol. So assume that this protocol begins in a state where all queues and links do not contain any messages.

In the absence of reset requests, nodes are in the so called *Ready* state. In this state, any message sent from (say $u$ to $v$) is queued by $u$ in an outbound queue for the link. When the message arrives at $v$, it is stored in a buffer which we will call $buffer_v[u]$. Eventually, the message stored in $buffer_v[u]$ is delivered to the user at $v$. Thus in the absence of reset requests, sent messages are delivered in FIFO order. In the following two paragraph description of the protocol, when we say "node $u$" we mean "the reset protocol at node $u$".

When node $u$ receives a reset request ($\text{REQUEST}_u$), the reset protocol at $u$ sends an ABORT packet to all its neighbors and goes into a special *Abort* mode where it waits until it gets an ABORT packet from all its neighbors. It does so by setting a boolean flag $ack_u[v]$ for all neighbors $v$ to indicate that it is expecting an ABORT packet from $v$. A node $u$ that receives an ABORT packet in the *Ready* mode behaves in almost the same way as a node that receives a reset request – i.e., $u$ sends an ABORT packet to all its neighbors. However, in this case $u$ sets $ack_u[v]$ for all neighbors $v$ *except* the neighbor $v$ from which it received the ABORT packet. If $u$ receives an ABORT packet from neighbor $v$ and $ack_u[v] = true$ then $u$ sets $ack_u[v] = false$. As soon as $ack_u[v] = false$ for all neighbors $v$, $u$ returns to the *Ready* mode and performs a $\text{SIGNAL}_u$ action.

The consistency condition is guaranteed by two additional rules. When an ABORT packet from $u$ arrives at $v$, $buffer_v[u]$ is emptied. Second, no packet in $buffer_v[u]$ is delivered while $v$ is in *Abort* mode and until $v$ has performed any outstanding $\text{SIGNAL}_v$ action. Intuitively, sending an ABORT packet on a link and waiting until another ABORT packet returns on the link effectively "flushes" any old messages that were sent in previous signal intervals. Essentially, we send an ABORT packet between any two SIGNAL events at a node and we delay delivery of packets until an outstanding signal has been performed. This ensures that a signal interval at $u$ "communicates" or mates with at most one signal interval in $v$. The Simple Reset Protocol (SRP) is similar to the Chandy-Lamport snapshot protocol [CL85] with ABORT packets replacing "markers".

The problem with SRP is that it can easily be placed in a state where it never

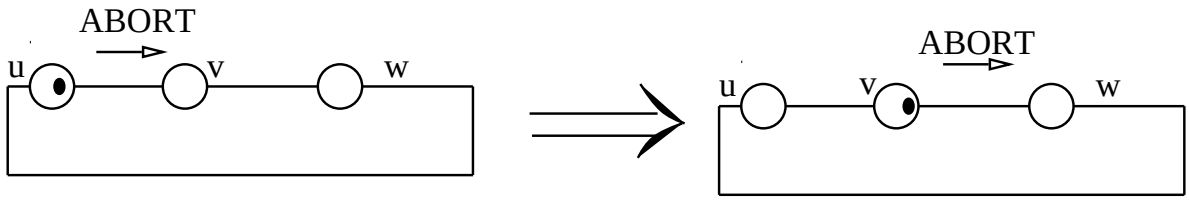



Figure 7.5: Two "bad" states of the Simple Reset protocol. The black dot before an edge indicates that a node is waiting for an ABORT on that edge.

terminates – i.e., violates the causality property. Consider the topology shown in Figure 7.5. Suppose in the initial state there is an ABORT packet in the channel from $u$ to $v$, and $u$ is waiting for an ABORT from $v$ (but not from $w$) before $u$ can exit from the *Abort* mode. Nodes $v$ and $w$ are in the *Ready* mode, and all other links are empty. Next, assume that the ABORT packet arrives at $v$ which causes $v$ to enter ABORT state and send an ABORT packet to $u$ and $w$. Assume that the ABORT packet sent to $u$ arrives quickly and causes $u$ to return to *Ready* mode. Notice that by the rules of the protocol, $v$ does not expect an ABORT from $u$. The resulting state is shown in Figure 7.5. By symmetry, it is clear that the execution can remain in a cycle of states where an ABORT packet continuously cycles through the network.

Now, since SRP is a non-stabilizing reset protocol, it may be possible to show that (after proper initialization) SRP will never enter such a "bad state". If the network is static, then this is indeed true. However, if network can have links that can fail and recover (a so called "dynamic network") then a series of link failures and recoveries can leave SRP in the "bad" state depicted in Figure 7.5.

More importantly for our purposes, SRP does not seem like a suitable point of departure for constructing a stabilizing reset protocol. Specifically, there does not appear any easy way to make SRP locally checkable. For instance, consider the topology of Figure 7.5. In a "good" state of SRP, if there is an ABORT packet in the cycle, there must be at least one other ABORT packet in the cycle, which is travelling in the "opposite" direction. This seems hard to check locally, even with the addition of a small amount of state. Instead, our point of departure is the AAG reset protocol of [AAG87]. This protocol works in dynamic networks and can be made locally checkable and correctable. We describe some more details of how the AAG protocol works and why it avoids the problems of the Simple Reset Protocol in the appendix. The ap-

pendix also contains a description of the changes we made to make the AAG protocol stabilizing. For the rest of this chapter, we will describe our reset protocol that is based on (but not identical to) the AAG protocol.

### 7.4.2 The basic idea behind the Reset protocol

Our reset protocol uses essentially the same idea as SRP for ensuring consistency – once again consistency is ensured by using ABORT packets to "flush" old packets (that were sent in a previous signal interval) from links. However, our protocol is much more conservative about allowing a node to to return to *Ready* mode.

A global (but approximate) summary is as follows. The protocol responds to reset requests in three phases. In the first phase, ABORT packets are sent out from nodes that receive reset requests. Nodes that receive ABORT packets and are in the *Ready* mode, broadcast ABORT packets to their neighbors. Thus the first phase consists of a wave of abort packets that spreads outwards from a reset request much the same way as in SRP. However, in our protocol the abort waves create an *abort forest* as they spread outwards. Consider any node $u$. Node $u$'s parent in the abort forest is the neighbor from whom $u$ last received an ABORT packet that caused $u$ to leave *Ready* mode. If node $u$ left *Ready* mode because $u$ received a reset request, then $u$ has no parent and $u$ is considered a *root* in the abort forest. Notice that the abort forest is a set of *ad hoc* abort trees that are created while reset requests are being processed.

In the second phase, a node sends an ack to its parent when the subtree rooted at that node has stopped expanding. Thus, in the second phase a wave of acks flow up the abort trees to the roots. The first and second phases work in much the same way as the Dijkstra-Scholten termination detection algorithm [DS80]. As in the Dijkstra Scholten algorithm, some nodes may be in the first phase (forward propagation of ABORT packets) while other nodes may be in the second phase (sending acks up to parents).

What distinguishes our protocol from the Dijkstra-Scholten protocol is that there is a third phase in our protocol. When a root of an abort tree receives acks from all its children, it starts a wave of READY packets that flows down the tree and allows all nodes in the tree to return to the *Ready* mode. Thus the crucial difference between SRP and the AAG protocol is as follows. In the former, nodes return to *Ready* mode

after communicating with their neighbors. However, in our protocol a node $u$ returns to *Ready* mode only after the root of the abort tree (that $u$ is part of) knows that the abort wave has stopped propagating, and has informed $u$ of this fact using a READY packet. Thus in the SRP protocol a node returns to *Ready* mode in constant time after it receives an ABORT packet. In our protocol a node may have to wait $O(n)$ time to return to *Ready* mode (potentially three worst case delays across the network, one for each phase).

The reader may wonder why three phases are needed. The appendix provides some intuition by describing why three phases were used in the original AAG protocol to avoid the problems of the Simple Reset Protocol.

### 7.4.3 Overview of the Code

The code of our reset protocol works as follows.

Each node $u$ has three interesting variables. First each node has a mode, $mode_u$ which is one of *Ready*, *Abort* or *Converge*. *Ready*, is the "normal" mode of a node when it is not processing a reset request. If the mode of $u$ is *Abort* or *Converge*, then this means that $u$ is processing an abort request.

Next, each node $u$ has an ack bit $ack_u[v]$ for each neighbor. If this bit is set, it indicates that $u$ is waiting for an ack from $v$. (Unlike the Simple Reset Protocol, our protocol uses explicit ACK packets.) Finally, $u$ has a parent variable $parent_u$ that points to the neighbor from which $u$ received the reset request that it is processing. If $u$ received a reset request through a $\text{REQUEST}_u$ action (i.e., a reset request directly at $u$ itself) then $u$ sets $parent_u = nil$. If $parent_u = nil$, we will say that $u$ is a root of an abort tree. A list of variables used by the code is shown in Figure 7.6.

In our code, the mode of a node is characterized by the state of the other variables at the node. Thus the mode of node $u$ is really a derived variable:

$$mode(u) = \begin{cases} Abort, & \text{if } \exists v \text{ such that } ack_u[v] = true \\ Converge, & \text{if } parent_u \neq nil \text{ and } \forall v\,(ack_u[v] = false) \\ Ready, & \text{if } parent_u = nil \text{ and } \forall v\,(ack_u[v] = false) \end{cases}$$

Notice that unlike the Simple Reset Protocol, we have a third mode called *Converge*.

State

$ack_u[v]$: Boolean Flag for each neighbor $v$ of $u$
$parent_u$: Either one of the neighbors of $u$ or $nil$
$dist_u$: integer in the range $0..n'$, where $n'$ is an upper bound on the number of nodes in the graph.
We also assume that for any (ABORT, $d$) packet, $d$ is an integer in the range $0..n'$.
$signalbit_u$: Boolean flag (*used to remember to do a signal event*)
$free_u[v]$: Boolean Flag for each neighbor $v$ of $u$ (*says whether link to $v$ is ready to accept a packet*)
$freem_u[v]$: Boolean Flag for each neighbor $v$ of $u$ (*says whether $v$ is ready to accept a message*)
$queue_u[v]$: queue of size 5 consisting of packets drawn from $P_{data}$ (*outbound queue for link to $v$ *)
$buffer_u[v]$: queue of size 1 that can contain a $\Sigma$-message only (*inbound message queue for link from $v$ *)

The following piece of code is used as a macro to propagate ABORT packets:

```
PROPAGATE_u(w, dist) ≡
  parent_u := w
  dist_u := dist
  For all neighbors v of u do                    (*broadcast abort packets to neighbors*)
    ack_u[v] := true
    enqueue (ABORT, dist + 1) on queue_u[v]
```

Figure 7.6: Variables and a Useful Macro

If $mode(u) = Converge$, this means that $u$ has received acks from all its children and is waiting for a READY packet from its parent.

There are also three interesting packets used by the protocol: abort packets (that are encoded as by tuples of the form $(\text{ABORT}, d)$ where $d$ is a integer representing distance from the root), ack packets (that are encoded simply as ACK), and ready packets (that are encoded simply as READY).

The protocol will deadlock if the protocol is placed in an initial state in which the *parent* pointers form a cycle. In order to be able to locally check that the subgraph induced by the *parent* pointers is acyclic, we maintain a distance variable at each node, such that a node's distance is one greater than that of its parent. Specifically, distance is initialized to 0 upon a reset request, and its accumulated value is appended to any abort packets. Thus we encode an abort packet as a tuple $(\text{ABORT}, d)$, where $d$ is a distance.

The code that implements most of the reset protocol is shown in Figure 7.7. For convenience, we have marked certain transitions in the figure with the labels $VR, VA, DA, IA, FA, RA$ and $RR$.

A $VR$ (for *Valid Request*) transition is a reset request that causes a node to change its *mode* to *Abort*. A $VA$ (for *Valid Abort*) transition is the receipt of an $(\text{ABORT}, d)$ packet with valid distance field that causes a node to change its *mode* to *Abort*. A $DA$ (for *Distance Invalid Abort*) transition is the receipt of an $(\text{ABORT}, d)$ packet such that the distance field $d$ is at the maximum value and such that its receipt causes a node to change its *mode* to *Abort*.

An $IA$ (for *Invalid Abort*) transition is the receipt of an $(\text{ABORT}, *)$ packet that *does not* cause a node to change its *mode* to *Abort*. A $FA$ (for *Final Ack*) transition is the receipt of an ACK packet that causes say node $u$ to send an ACK packet to its parent. It is not hard to see that the ack that was received must have been the last ack that node $u$ was waiting for. A $RA$ (for *Root Ack*) transition is the receipt of an ACK packet at a root node that causes the root node to change its *mode* to *Ready*. A $RR$ (for *Regular Ready*) transition is the receipt of an (READY) packet at a node that causes the node to change its *mode* to *Ready*.

Refer to these labels in Figure 7.7 when following the description below.

The code in Figure 7.7 uses a small macro called $\text{PROPAGATE}_u(v, d)$. This procedure is used to broadcast $(\text{ABORT}, *)$ packets and is shown in Figure 7.6. The first

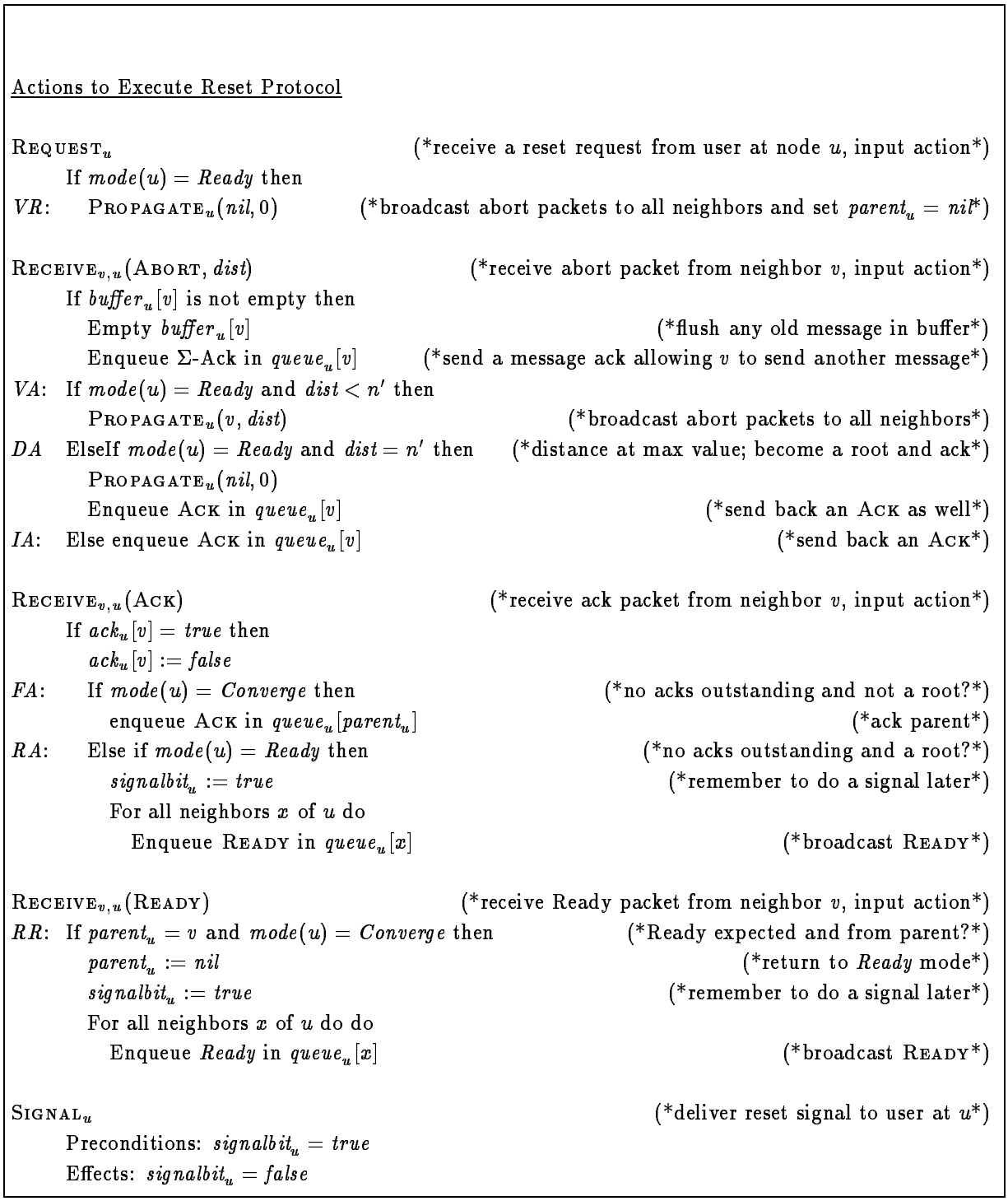



Figure 7.7: Actions at node $u$ to execute reset protocol functions with respect to any neighbor $v$.

parameter to the procedure specifies that $v$ is the new *parent* of $u$ and the second parameter specifies that $d$ is the distance from $u$ to root of $u$'s abort tree. The abort packets sent out as a result of this procedure (see Figure 7.6) will carry a distance value of $d + 1$. We add the distance variable to abort packets in order to make the protocol locally checkable.

When a reset request is made at some node $u$ while in *Ready* mode ($VR$), $u$ changes its mode to *Abort*, broadcasts an ABORT packet to all its neighbors, and sets its ack bits to *true* for all neighbors $v$. Node $u$ then waits until all the neighboring nodes send back an ack packet. If node $u$ receives an abort packet while in *Ready* mode ($VA$), it marks the neighbor from which the packet arrived as its parent, broadcasts ABORT, and waits for ACK packets to be received from all its neighbors. We also add a check to see whether the distance in the ABORT packet is less than $n'$, which is the maximum value of the distance variable at a node. In linear time after all local predicates hold, this condition will always hold. However, this check helps to ensure that the local predicates remain closed during initial periods. If the distance check fails, ($DA$) node $u$ becomes a root and sends out abort packets just as if it received a reset request; however, node $u$ also sends back an ACK to the ABORT packet it received. Finally, if the ABORT packet is received by a node not in *Ready* mode ($IA$), an ACK is sent back immediately.

When node $u$ receives an ACK from $v$ it sets the ack bit for $v$ to *false*. The action of node $u$ when it receives the last anticipated ack depends on the value of *$u$'s parent*. If $u$'s parent is not *nil* ($FA$), $u$'s mode is changed to *Converge*, and an ACK is sent to the parent. Notice that since $mode_u$ is a derived variable, $mode_u$ = *Converge* when $ack_u[v]$ = *false* for all neighbors $v$ of $u$. If $u$ is a root ($RA$), $u$ changes its mode to *Ready* (by setting $parent_u$ = *nil*), and broadcasts a ready packet to all neighbors. If node $u$ gets a READY packet from its parent while in *Converge* mode ($RR$), then $u$ changes its mode to *Ready* (by setting $parent_u$ = *nil*), and broadcasts a ready packet to all neighbors.

Finally, whenever node $u$ changes its mode from either *Abort* or *Converge* to *Ready* it sets a flag (which we call $signalbit_u$) to remind itself to later output a signal event. In other words, a $\text{SIGNAL}_u$ event is enabled whenever $signalbit_u$ is set; naturally, when this event occurs, the flag is cleared. For convenience, we introduce the following definition.

**Definition 7.4.1** *We say that node $u$ has a* status *of* on *whenever $signalbit_u = false$ and $mode_u = Ready$, and* off *otherwise.*

The code that implements the sending and receiving of $\Sigma$-messages is shown in Figure 7.8.

If the status of $u$ is on, $u$ relays $\Sigma$-messages between the user and the network. More specifically, when a $\text{SENDM}_{u,v}(m)$ event occurs and node $u$ has status on, $u$ queues $m$ on an outbound queue (called $queue_u[v]$ as in previous chapters) that contains packets to be sent to $v$. This queue can also contain an abort, ack or ready packet. Eventually, packets in this outbound queue are delivered to neighbor $v$. When $v$ receives a $\Sigma$-message $m$ from $v$, $v$ queues $m$ in an inbound message buffer called $buffer_v[u]$. Later, if $v$'s status is on, it will do a $\text{RECEIVEM}_{u,v}(m)$ event and remove $m$ from the buffer and deliver it to the user. See Figure 7.9 for a sketch of the inbound and outbound queues for a link.

If $u$'s status is not on, $u$ *discards* $\Sigma$ messages input by the user through $\text{SENDM}_{u,v}$ actions. Also, $u$, will not do a $\text{RECEIVEM}_{v,u}$ action unless its status is on. Recall that all $\Sigma$-messages from a neighbor $v$ are queued on $buffer_u[v]$. In order to separate packets that are sent during separate signal intervals at $v$, we use the same rule as the Simple Reset Protocol. *When a abort packet arrives from neighbor $v$, $buffer_u[v]$ is emptied.* Once again, this simple rule is really the key to ensuring the conditions required to satisfy the consistency property.

*Note that $buffer_u[v]$ can store at most one message.* Clearly, if user messages are not to be dropped, we have to rely on the FREEM action as a form of "flow control". Our scheme is as follows. We require that there is at most one $\Sigma$-message in transit from $u$ to $v$. Thus $u$ keeps a variable $freem_u[v]$. Whenever $v$ delivers (or destroys) a $\Sigma$-message from $u$, $v$ sends a $\Sigma$-Ack back to $u$. When $u$ receives a $\Sigma$-Ack from $v$, $u$ sets $freem_u[v]$ to *true*. This enables the $\text{FREEM}_u[v]$ action, which tells the user at $u$ that it can safely send a message to $v$. Thus all we are doing is using a $\Sigma$-Ack message to provide feedback to the sender that the buffer at the other end is empty.

### 7.4.4 Example

Figure 7.10 describes a sample execution of the reset protocol for a simple topology consisting of four node $u$,$v$, $w$ and $x$ connected as shown in the figure. The figure

Actions to send and receive $\Sigma$-messages

SENDM$_{u,v}(m)$, $m \in \Sigma$ (*input action by which user sends a message*)
  If $mode(u) = Ready$ and $signalbit_u = false$ and $freem_u[v] = true$ then
  (* accept message only if mode is ready, no outstanding signals, and message flow control says OK*)
    enqueue $m$ in $queue_u[v]$ (*enqueue $m$ in outbound queue to $v$ *)
    $freem_u[v] = false$ (*inhibit delivery of free event to user*)

SEND$_{u,v}(p)$ (*output action to send a packet on UDL link to $v$*)
  Preconditions:
    $p$ is the head of $queue_u[v]$ (*$p$ is head of outbound queue for link*)
    $free_u[v] = true$
  Effects:
    Delete $p$ from $queue_u[v]$
    $free_u[v] := false$

FREE$_{u,v}$ (*input action from link to $v$ to say it is free*)
  $free_u[v] := true$

RECEIVE$_{v,u}(m)$, $m \in \Sigma$ (*input action to receive a user message from $v$*)
  Enqueue $m$ in $buffer_u[v]$ (*store $m$ in inbound buffer for link*)

RECEIVE$_{v,u}(\Sigma$-Ack), (*input action to receive a $\Sigma$- message ack from $v$*)
  $freem_u[v] = true$ (*record that $v$ is ready to accept a new message*)

RECEIVEM$_{v,u}(m)$, $m \in \Sigma$ (*output action to deliver a user message from $v$ to the user*)
  Preconditions:
    $m$ is the head of $buffer_u[v]$
    $mode(u) = Ready$ and $signalbit_u = false$ (*mode is ready and no outstanding signals*)
  Effects:
    Remove $m$ from $buffer_u[v]$
    Enqueue $\Sigma$-Ack in $queue_u[v]$ (*send a message ack back to $v$*)

FREEM$_{u,v}$ (*output action to tell user that it can safely send another message to $v$*)
  Preconditions:
    $freem_u[v] = true$ and $signalbit_u = false$ and $mode(u) = Ready$
  Effects:
    None

Figure 7.8: Actions at a node $u$ to send and receive user messages to and from any neighbor $v$ of $u$.

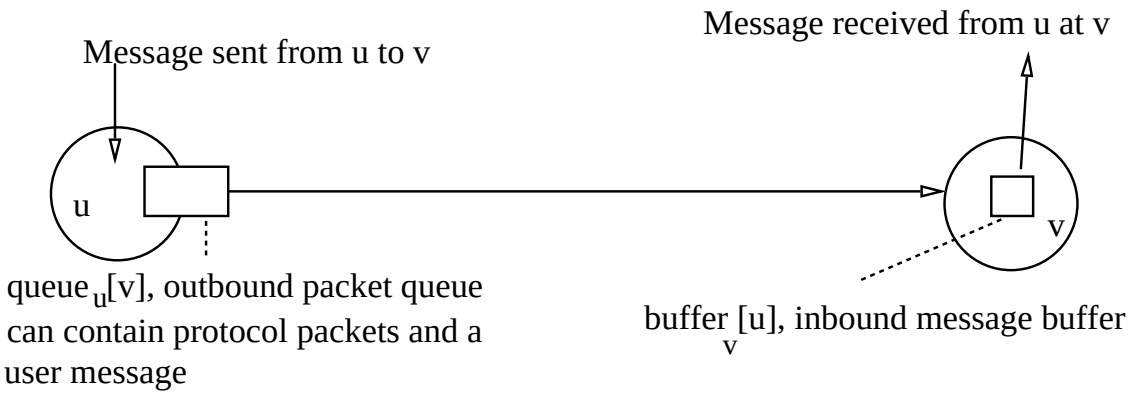



Figure 7.9: Sketch of the inbound and outbound queues for a link.

describes seven snapshots (F1-F7) taken during this sample execution.

The execution begins (F1) with nodes $u$ and $w$ receiving a snapshot request. Node $u$ and $w$ go into *Abort* mode and send abort packets to their neighbors. In F2, the abort packet from $u$ has arrived at $v$. Next, $u$ and $w$ receive each other's abort packet and (F3) send back acks to each other. We assume that the abort from $w$ to $x$ travels slowly which allows the abort packet sent from $v$ to arrive at $x$ earlier (F3). Thus in F3, the abort tree rooted at $u$ (which is shown using dotted lines) consists of $u$, $v$ and $x$.

Next, $x$ sends an abort to both $v$ and $w$ which are acked immediately. Then $x$ receives the "slow" abort from $w$ and sends back an ack. Once $x$ gets an ack back from both $v$ and $w$, $x$ sends back an ack to its parent $v$ (F4). When $v$ receives this ack, it sends back an ack to its parent $u$ (F5). Finally, in F6, $u$ sends a ready packet down to $v$ and then (F6), $v$ sends a ready packet to $x$. Note that the ready packet destroys the abort tree as it travels down the tree.

## 7.5 Reset Automaton

In describing the reset code, we use the following notation. As in any node automaton there is a outbound queue of packets ($queue_u[v]$) at any node $u$ for every neighbor $v$ of $u$. As usual, $queue_u[v]$ consists of packet waiting to be sent on $C_{u,v}$.

Each node automaton $N_u$ is specified given in Figures 7.6, 7.7, and 7.8. The code

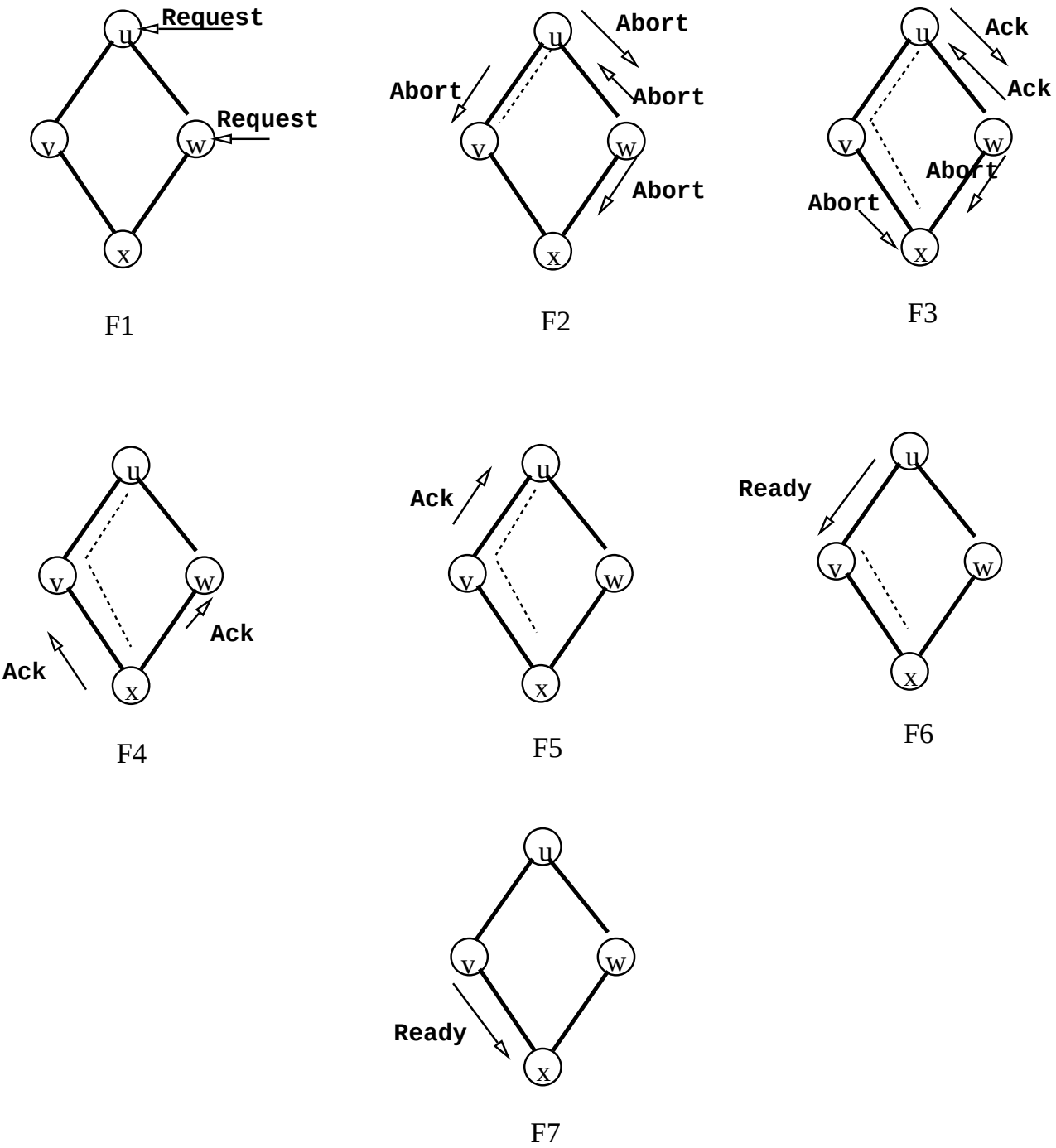



Figure 7.10: A sample execution for the reset protocol described in terms of seven snapshots.

uses the variables and macro specified in Figure 7.6. The piece of code that deals with implementing the main part of the reset protocol is described in Figure 7.7. The piece of code that deals with sending and receiving $\Sigma$-messages is described in Figure 7.8.

## 7.6 Reset Protocol is Locally Checkable and Correctable

Let $G$ be the topology graph that models the topology of the network on which the reset protocol works.

**Lemma 7.6.1** *$N = \{N_u, u \in G\}$ is a set of node automata for $G$ with $P_{data} = \Sigma \cup \{(\text{ABORT}, *), \text{ACK}, \text{READY}\}$.*

**Proof:** Simple checking of the definitions given earlier. ∎

### 7.6.1 Overview of Predicates

Next, we prove that the protocol is locally checkable, by describing a set of predicates in Figure 7.14 and showing that these predicates are closed. The description of the predicates uses the shorthand notation shown in Figure 7.13. Please refer to both these figures during the following discussion.

Recall that $Q_{u,v}$ is the queue corresponding to the single packet that can be stored in channel $C_{u,v}$. For convenience, we define $xqueue_u[v]$ (i.e., the extended queue between $u$ and $v$) as the queue formed by the concatenation of $Q_{u,v}$ and $queue_u[v]$. Thus in Figure 7.9, the extended queue between $u$ and $v$ is the queue formed by concatenating the outbound link queue and the link itself. Note that we do not include the inbound message buffer! If $Q_{u,v} \neq nil$, we assume that $Q_{u,v}$ is the head of $xqueue_u[v]$.

We informally describe the predicates. The first two predicates $\mathcal{A}$ and $\mathcal{B}$ deal with the ack flag $ack_u[v]$ at a node $u$. Intuitively, this bit is set if $u$ expects an ack from $v$. $\mathcal{A}$ states that if $u$ is expecting an ack from $v$ then one of three possibilities must be true: either there is an abort packet in transit from $u$ to $v$ (Case 1 in Figure 7.11), OR $v$ has received the abort packet and has chosen $u$ as its parent (Case 2 in Figure 7.11),

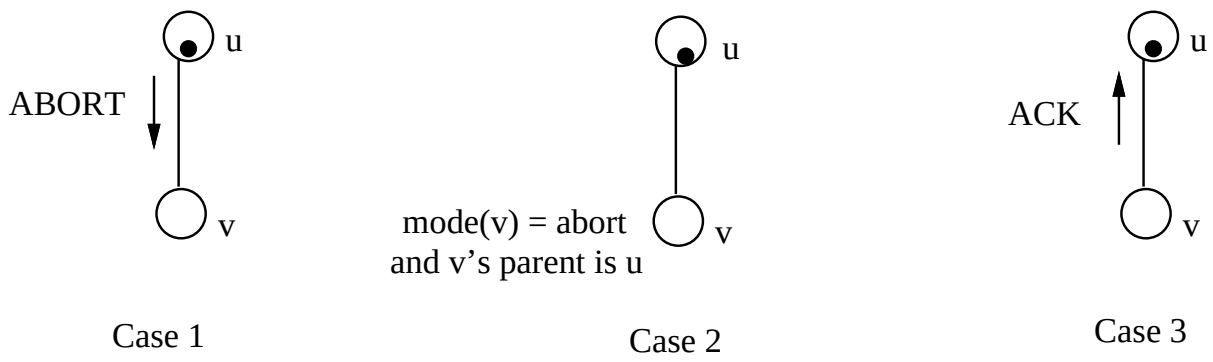



Figure 7.11: The Three Cases for the first and second predicates. The black dot before an edge indicates that a node is waiting for an Ack on that edge.

OR there is an ack packet in transit from $v$ to $u$ (Case 3 in Figure 7.11). Predicate $\mathcal{B}$ says that at most one of these three possibilities can be true at the same time. In some sense, $\mathcal{A}$ and $\mathcal{B}$ govern the first two phases of the reset protocol

Consider Figure 7.10. In that example execution, the first case is illustrated in F2, the second case in F3 and F4, and the third in F5.

The next two predicates $\mathcal{C}$ and $\mathcal{D}$ govern the second and third phases of the reset protocol. They deal with the $parent_u$ variable at a node $u$. Intuitively, $parent_u = v$ if $v$ is the parent of $u$ in the abort tree. $\mathcal{C}$ states that if $v$ is the parent of $u$ and the the mode of $u$ is *Converge* then one of three possibilities must be true: either there is an ack packet in transit from $u$ to $v$ (Case 1 in Figure 7.12), OR $v$ has received the ack packet and has cleared its ack bit for $u$ but has not changed its mode to *Ready* (Case 2 in Figure 7.12), OR there is a ready packet in transit from $v$ to $u$ (Case 3 in Figure 7.12). Predicate $\mathcal{D}$ says that if $v$ is $u$'s parent, then at most one of these three possibilities can be true at the same time.

As an example, consider nodes $x$ and $v$ in Figure 7.10, where $v$ is the parent of $x$. In that example execution, the first case is illustrated in F4, the second in F5 and F6, and the third in F7.

The next predicate $\mathcal{E}$ is crucial to the proof of termination of the protocol. It states that the distance of a child in the abort tree is one more than the distance of its parent. The only exception to this is if the parent has "abdicated" by sending a ready packet that is currently in transit to the child. Essentially, this predicate shows that abort trees are acyclic and have a maximum height of $n$, the number of nodes; the proof of

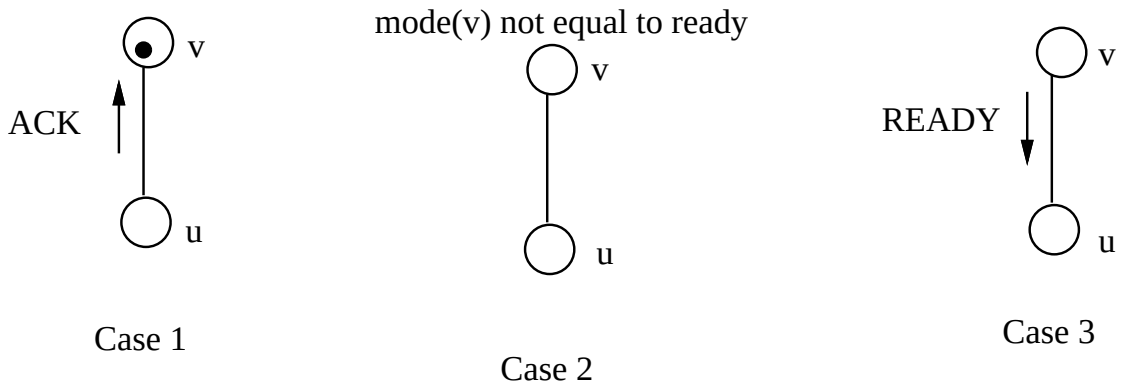



Figure 7.12: The Three Cases for the third and fourth predicates. Node $v$ is the parent of Node $u$ in all three cases.

termination will essentially consist of induction on the height of the abort trees. The predicate $\mathcal{F}$ is a supporting predicate required to prove that $\mathcal{E}$ is closed. It states that if $u$ sends an abort to $v$ then the distance in the abort packet is one more than $u$'s current distance.

Predicate $\mathcal{G}$ is is again a supporting predicate required to prove that some of the other predicates are closed. Suppose there is a packet $p$, that is either a ready packet or a user message or a $\Sigma$-Ack, in transit from $u$ to $v$. Then $p$ must have been sent when $u$ was in the *Ready* mode. Thus, it must be that either $u$ is still in the *Ready* mode or that $u$ has gone into *Abort* mode since the time $p$ was sent. But in the latter case there must be an abort packet behind $p$ in transit from $u$ to $v$.

Predicate $\mathcal{H}$ governs the flow control scheme that ensures at most one message in transit from $u$ to $v$. If $freem_u[v]$ is *true* then the $\text{FREEM}_{u,v}$ action is enabled and so there must be no message in transit from $u$ to $v$ and also no message acks ($\Sigma$-Ack's) in the reverse direction. On the other hand, if $freem_u[v]$ is *false* then either a message is in transit from $u$ to $v$, or a message ack is in transit in the reverse direction, but not both.

Predicate $\mathcal{Q}$ is the reason why we can get away with an outbound queue size for links (e.g., the size of $queue_u[v]$) of 5. It states that the outbound queue (even after concatenation with the channel queue) can contain at most one packet of each type: abort, ack, and ready. Since $\mathcal{H}$ tells us we can have at most one $\Sigma$-message and at most one $\Sigma$-Ack, it means that a queue size of 5 is sufficient.

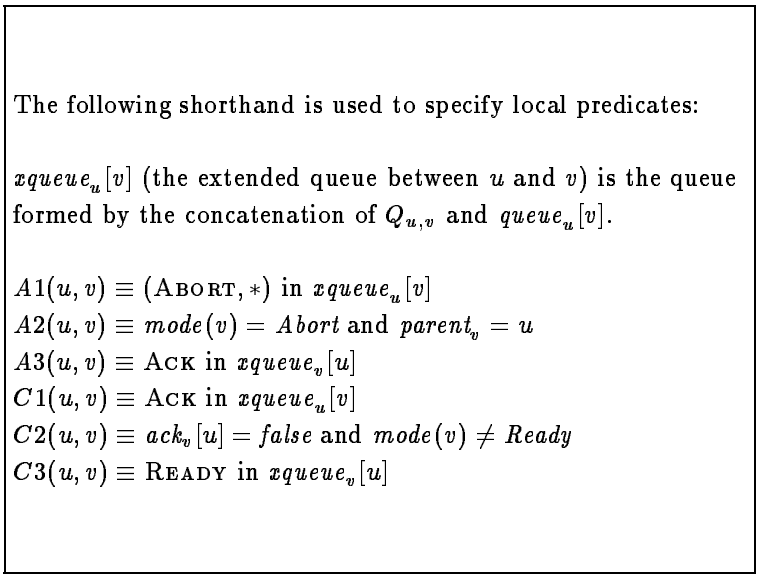



Figure 7.13: Shorthand Used to Define Local Predicates

We now show that the conjunction of these predicates is a closed predicate. The proofs, which we relegate to the appendix, consist basically of rigorous (and somewhat tedious) case-analysis.

### 7.6.2 Proving that the Local Predicates of the Reset Protocol are Closed

First, the the local predicates of Figure 7.14 are specific to a directed link $(u,v)$. Recall that for local checkability we need exactly one local predicate for each edge. Hence:

**Definition 7.6.2** *We let $G_{u,v}$ be the intersection of the local predicates in Figure 7.14 for any edge $(u,v)$. For any edge $(u,v)$, we let $L_{u,v}$ be the intersection of $G_{u,v}$ and $G_{v,u}$.*

**Definition 7.6.3** *Let $L_{u,v}$ be the local predicate defined in Definition 7.6.2. Let $\mathcal{L}$ be the link predicate set containing $L_{u,v}$ for every $(u,v)$ in $G$ and let $L = Conj(\mathcal{L})$.*

$\mathcal{A}$: $ack_u[v] = true$ iff
one of $A1(u,v)$, $A2(u,v)$, or $A3(u,v)$ holds.

$\mathcal{B}$: At most one of $A1(u,v)$, $A2(u,v)$, or $A3(u,v)$ holds.

$\mathcal{C}$: $parent_u = v$ implies
$mode(u) = Converge$ iff
one of $C1(u,v)$, $C2(u,v)$, or $C3(u,v)$ holds.

$\mathcal{D}$: $parent_u = v$ implies
at most one of $C1(u,v)$, $C2(u,v)$, or $C3(u,v)$ holds.

$\mathcal{E}$: $parent_u = v$ implies
one of the following holds:
$dist_u = dist_v + 1$ and $mode(v) \neq Ready$ OR
$C3(u,v)$.

$\mathcal{F}$: If $xqueue_u[v]$ contains a (ABORT, $d$) packet then $d = dist_u + 1$.

$\mathcal{G}$: If $p$ in $xqueue_u[v]$ and $p =$ READY or $p$ is a $\Sigma$-message or $p = \Sigma$-Ack then
Either $mode(u) = Ready$
Or there is an ABORT in $xqueue_u[v]$ after $p$.

$\mathcal{H}$: Let $M_{u,v}$ denotes the concatenation of $xqueue_u[v]$, and $buffer_v[u]$. Then:
If $freem_u[v] = true$ then
There is no $\Sigma$-message in $M_{u,v}$ and no $\Sigma$-Ack in $xqueue_v[u]$
Else one of the following holds:
There is exactly one $\Sigma$-message in $M_{u,v}$ OR
There is exactly one $\Sigma$-Ack in $xqueue_v[u]$

$\mathcal{Q}$: $xqueue_u[v]$ contains at most one (ABORT, $*$), ACK, or READY packet.

Figure 7.14: Reset Protocol: Local Predicates for edge $(u,v)$. Refer to the code given in Figure 7.13 for an explanation of the shorthand used.

The proof, which is in the appendix, consists of showing that each $L_{u,v}$ is a closed local predicate.

**Lemma 7.6.4** *For a leader edge $(u,v)$ and transition $(s,a,s')$ of $\mathcal{R}$, if $s$ satisfies $L_{u,v}$, then $s'$ satisfies $L_{u,v}$.*

**Proof:** By lemmas, D.1.4, D.1.5, D.1.6, D.1.7, D.1.8, and D.1.3 in Section D.1 of the appendix. The predicates are closed because of the code in [AAG87] and the heuristic of removing unexpected packet transitions.

We quickly sketch what is involved in such a proof. Consider predicate $\mathcal{A}$ as sketched in Figure 7.11. We essentially consider all states that satisfy this local predicate and show that no transition can cause this predicate not to hold in the next state. For example, consider Figure 7.11. If $u$ is not expecting an ack from $v$ in a state, then the only transitions that can cause this to happen is if $u$ gets a reset request or an ABORT packet while in *Ready* mode. But this causes $u$ to send an an ABORT packet to $v$ which leaves us in Case 1 of the predicate.

Rather than consider all possible transitions, we save some effort by first identifying the transitions that can affect key variables. Then we focus our attention on such transitions. This method is described in the appendix. ∎

The following theorem is immediate from the last lemma.

**Theorem 7.6.5** *The reset automaton $\mathcal{R}$ can be locally checked for $L$ using $\mathcal{L}$.*

### 7.6.3 Reset Protocol is Locally Correctable

Consider the function $f$ defined in Figure 7.15. Let $<$ be the trivial partial order such for all $u,v,w,x$ in $G$, $\{u,v\} \not< \{w,x\}$. (i.e., no pair of neighbors is less than any other pair of neighbors.) We claim that $f$ is a local reset function for $\mathcal{R}$ with respect to $\mathcal{L}$ and partial order $<$.

**Lemma 7.6.6** *The function $f$ defined in Figure 7.15 is a local reset function for network automaton $\mathcal{R}$ with respect to link predicate set $\mathcal{L} = \{L_{u,v}\}$ and partial order $<$.*

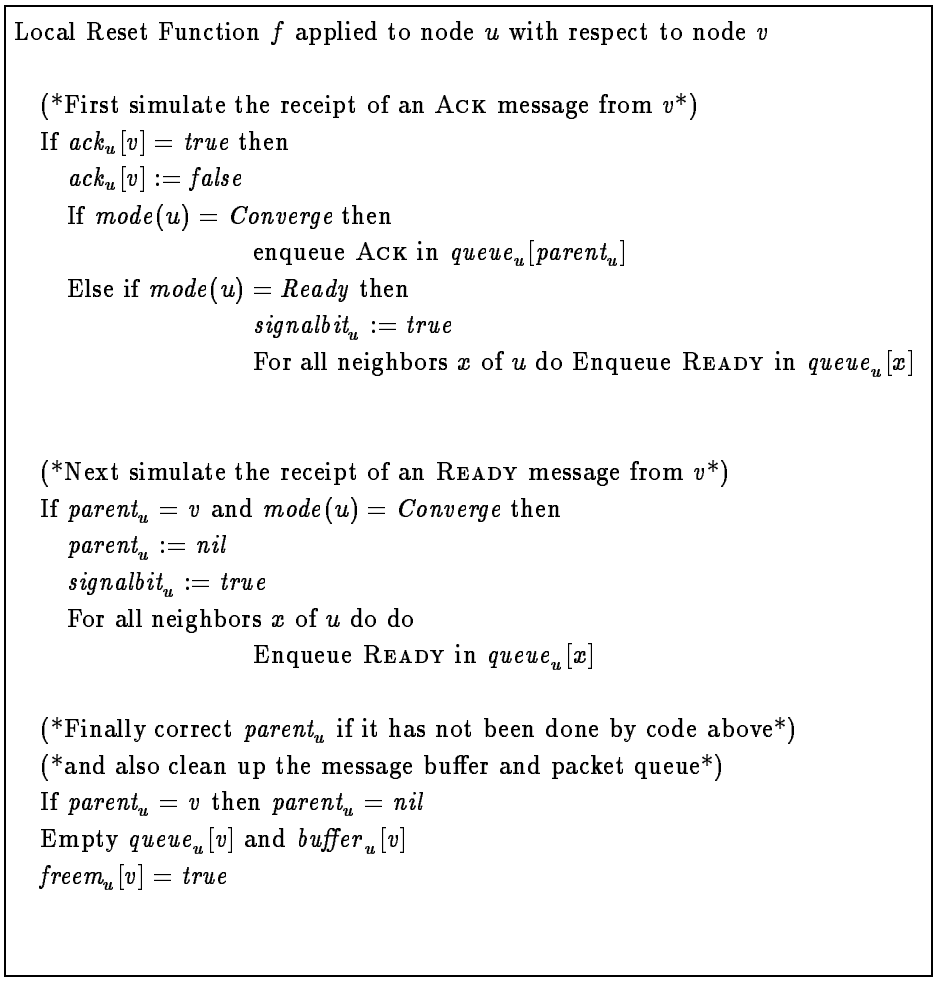



Figure 7.15: Reset Protocol. Local Reset Function

**Proof:** Consider any state $s$ of $\mathcal{R}$ and any leader edge $(u,v)$ of $G$: We show each of the properties required by a local correction function.

- **Correction:** In the state $f(s|u,v)$, it is easy to check that $parent_u \neq v$, $ack_u[v] = false$ and $queue_u[v]$ and $buffer_u[v]$ are empty. Also $freem_u[v] = true$. Similarly, in the state $(f(s|v,u)$, it is easy to check that $parent_v \neq u$, $ack_v[u] = false$ and $queue_v[u]$ and $buffer_v[u]$ are empty and $freem_v[u] = true$. Thus it follows that $(f(s|u,v), nil, nil, f(s|v,u)) \in L_{u,v}$.

- **Stability:** We need to show the following fact for any neighbor $w$ of $v$:
  If $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$ then $(s|u, s|(u,v), s|(v,u), f(s|v,w)) \in L_{u,v}$.

  But if we look at the code for $f(s|v,w)$ we see that $f(s|v,w)$ is the same state that would occur at $v$ in an execution in which the reset protocol starts in state $s$ and consisting of the following sequence of actions:

  - A $\textsc{Receive}_{w,v}(\textsc{Ack})$ action (see Figure 7.7) followed by
  - A $\textsc{Receive}_{w,v}(\textsc{Ready})$ action (see Figure 7.7) followed by
  - A hypothetical internal action that results in the execution of the last three lines of Figure 7.15 applied to node $v$ with respect to node $w$.

  Now we know that the state at $v$ resulting after the $\textsc{Receive}_{w,x}(\textsc{Ack})$ and $\textsc{Receive}_{w,x}(\textsc{Ready})$ events must satisfy the stability condition because we have already proved this in Lemma 7.6.5.

  Now consider the hypothetical internal action applied to $v$. Note that this internal action can only change $parent_v$ from $w$ to $nil$. Also a careful look will show that this internal action will never change the value of $mode(v)$ since it is only applied after the simulated processing of a $\textsc{Ready}$ packet from $w$. Now the predicates described in $L_{u,v}$ only depend on the value of $mode(v)$, $ack_v[u]$ and the predicate $parent_v = u$. But none of these values are affected by the hypothetical internal action. Thus $L_{u,v}$ is unaffected by the hypothetical internal action.

  Thus the result of executing all three actions must satisfy the stability condition.

∎

**Lemma 7.6.7** *Let $f$ be the reset function described in Figure 7.15. Then $\mathcal{R}$ is* locally correctable *to $L$ using link predicate set $\mathcal{L}$, reset function $f$, and partial order $<$.*

**Proof:** Follows from Lemma 7.6.6 and Lemma 7.6.5. ∎

**Theorem 7.6.8** *$\mathcal{R}^+$ stabilizes to the behaviors of $\mathcal{R}(t_p)|L$ in time $t_q$, where $t_q$ and $t_p$ are constants.*

**Proof:** Follows directly from the Local Correction theorem, Theorem 5.4.3 and Lemma 7.6.7. Notice that $height(<) = 1$ since $<$ is the trivial partial order. ∎

## 7.7 The behaviors of a reset protocol after it stabilizes

We have already shown that $\mathcal{R}^+$ stabilizes to the behavior of $\mathcal{R}(t_p)|L$ in constant time. Recall that $\mathcal{R}(t_p)|L$ is identical to the automaton $\mathcal{R}$ except that the node and link delays are increased by a constant factor, and all link predicates hold.

We now show that $\mathcal{R}(t_p)|L$ solves the reset problem $RP$. However, because the set of behaviors specified by $RP$ remains unchanged after scaling by constant factors, it suffices to show that $\mathcal{R}|L$ solves the reset problem $RP$. Thus in the remainder of this section and in the appendix, we show that every behavior $\beta$ of $\mathcal{R}|L$ is in $RP$.

We relegate the proof of this theorem to the appendix. However, in this section, we intuitively explain why any behavior $\beta$ of $\mathcal{R}|L$ is timely, consistent, and causal. The intuition provided in this section should help the reader understand the proof in the appendix.

In the proofs when we talk of a state $s$ or an execution $\alpha$, we mean a state or execution of $\mathcal{R}|L$. Recall that we defined a derived variable called $status(u)$ which is *on* if $mode(u) = Ready$ and $signalbit(u) = false$ and *off* otherwise. From the code, it is easy to see that $u$ will only send and receive messages when $status(u) = on$.

The first important lemma is what we call the *Termination Lemma*. This states the following. Consider any node $u$. Assume that $mode(u) \neq Ready$ in some state $s_i$ of any execution $\alpha$. The lemma states that $mode(u)$ will change to *Ready* in $O(n)$ time after $s_i$.

### 7.7.1 Why the Termination Lemma Works

Let $c$ be the worst-case time for a packet queued at node $u$ to reach neighbor $v$. It's not hard to see that $c$ is a constant because the outbound queue size at each node is at most 5 and because of the properties of a UDL. In the following, we will say that a node $u$ is a root in some state $s$ of the reset protocol if $mode(u) = Abort$ and $parent_u = nil$ in state $s$.

By assumption, $mode(u) \neq Ready$ in state $s_i$. Since $mode(u) \neq Ready$, $u$ must either be a root or have a parent (say $v$) in state $s_i$. By using invariants $\mathcal{A}$ and $\mathcal{C}$ repeatedly, we can obtain a chain of nodes starting with $u$ such that each node is the parent of the previous node. Also the chain must either end with a READY packet or must end with a root node $r$ such that $mode(r) = Abort$ in $s_i$. This is shown in Figure 7.16. We also know from invariant $\mathcal{E}$ that the distance of each node in the chain is one more than that of its parent. Thus no node can occur more than once in this chain and hence this chain can consist of at most $n$ nodes.

(The following is a more detailed argument that explains why each chain must end in a READY packet or a root $r$. If $mode(u) = Abort$, then either $u$ is a root or $u$ has a parent, say $v$. In the latter case by $\mathcal{A}$, $ack_v[u] = true$ and so $mode(v) = Abort$. If $mode(u) = Converge$ then $u$ has a parent, say $v$. By $\mathcal{C}$, either there is an ACK in transit from $u$ to $v$ (in which case by $\mathcal{A}$, $mode(v) = Abort$), or $mode(v) \neq Ready$, or there is a READY packet in transit from $v$ to $u$. Thus either $u$ is a root, or $u$ has a parent $v$ such that $mode(v) \neq Ready$ or there is a READY packet in transit from $v$ to $u$. We now repeat this argument until we either arrive at a root node or a READY packet.)

Consider the case where the chain ends with a READY packet. In this case, it is not hard to see that within $O(n)$ time after $s_i$, a READY packet will reach $u$ which will cause $u$ to go into *Ready* mode.

So consider the latter case where the chain ends with a root $r$. Suppose we can show that in $O(n)$ time after $s_i$, there is an action $a_j$ in which $r$ changes its mode to *Ready* and sends a READY packet to all its children. But if we can show this we are done in $O(n)$ time after $a_j$ using the arguments in the previous paragraph. So all we have to do is to show that in $O(n)$ time after $s_i$, $r$ will change its mode to *Ready*

If the mode of $r$ is *Abort*, then by definition $r$ must have some neighbor $v$ that it expects an ack from (i.e., $ack_r[v] = true$). Now by invariant $\mathcal{A}$, this means that (see

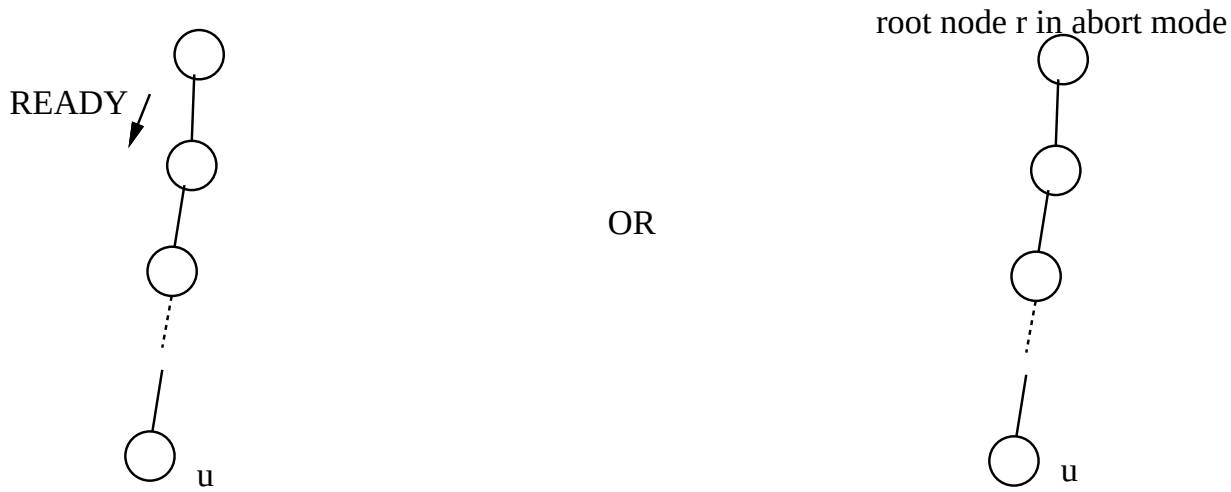



Figure 7.16: The two cases that can occur if $u$'s mode is not *Ready*. Each node in a chain is the parent of the node below it.

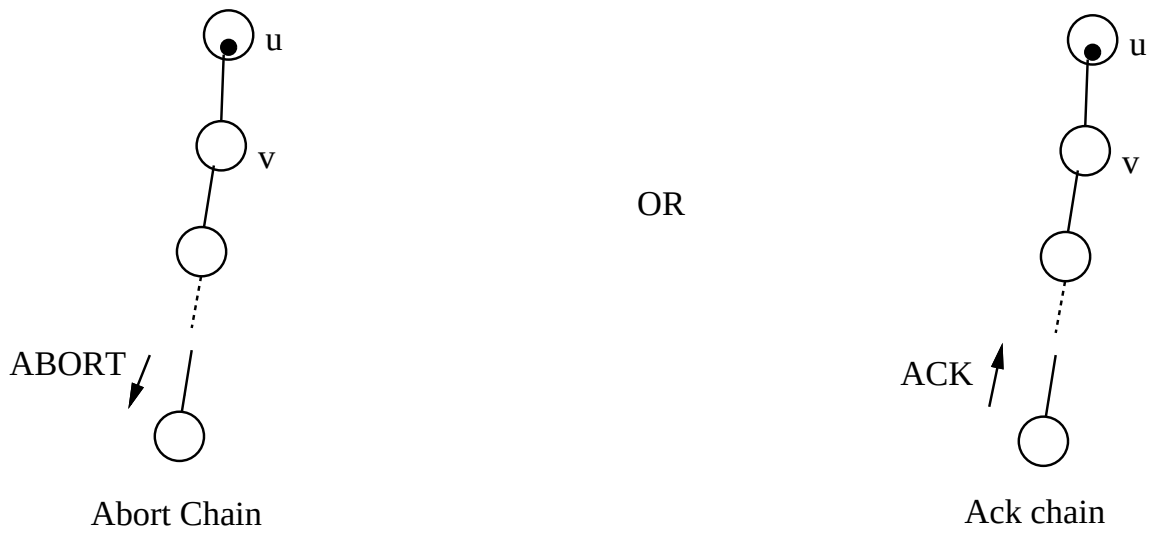



Figure 7.17: The two cases that can occur if node $u$ is expecting an ack on the link to $v$.

Figure 7.11) either there is an abort in transit from $r$ to $v$, or an ack in transit from $v$ to $r$, or $v$ is also in *Abort* mode and $r$ is the parent of $v$. But if $v$ is in in *Abort* mode, we can continue this argument inductively to produce a chain of nodes starting with $r$ such that each node is the parent of the next node. We also know from invariant $\mathcal{E}$ that this chain can consist of at most $n$ nodes. At the end of this chain (see Figure 7.17, there must be either an abort or an ack. We call a chain that ends with an abort packet an *abort chain* and we call a chain that ends with an ack packet an *ack chain*

Recall that $c$ is the constant that reflects the worst-case time for a packet queued at node to reach a neighbor. Now, observe that in $c$ units of time any abort chain must either increase in size by 1 or be converted into an ack chain. But since the size of an abort chain cannot increase beyond $n$ (the number of nodes), within $O(n)$ time any abort chain must have converted into an ack chain. Similarly in $c$ units of time, any ack chain must decrease in size by 1. Thus in $O(n)$ time any ack chain will disappear.

The upshot is that within $O(n)$ time, $ack_r[v]$ will become *false*. Since this happens for any child $v$ of $r$, within $O(n)$ time after $s_i$, $r$ will change its mode from *Abort* to *Ready* and we are done.

A formal proof can be made based on these arguments. The appendix contains a formal statement of the lemma (Lemma D.2.1) but omits a detailed formal proof.

### 7.7.2 Why behaviors of the reset automaton are timely, consistent, and causal

Recall that we defined a derived variable called $status(u)$ which is *on* if $mode(u) =$ *Ready* and $signalbit(u) =$ *false* and *off* otherwise. From the code, it is easy to see that $u$ will only send and receive messages when $status(u) = on$.

The first important tool is what we call the *Signal Lemma.* Consider any execution $\alpha$ of $\mathcal{R}|L$ and any node $u$ and any state $s_i$ in $\alpha$. The signal lemma basically says that if $status(u) =$ *off* in state $s_i$, then a $\text{SIGNAL}_u$ event occurs within $O(n)$ time after $s_i$ in $\alpha$. This follows because if $signalbit_u =$ *true* in $s_i$, then a $\text{SIGNAL}_u$ event must occur within constant time of $s_i$ by the timing conditions. On the other hand, if $mode(u) \neq$ *Ready* in $s_i$, then the Termination Lemma tells us that in $O(n)$ time after $s_i$, we reach a state $s_j$ in which $mode(u) =$ *Ready*. If $s_j$ is the first such state after $s_i$, then by the code, we

know that $u$ cannot change its mode to *Ready* without also setting $signalbit_u = true$. Then, as before, a SIGNAL$_u$ event must occur within constant time of $s_j$ in $\alpha$.

Now we consider an execution $\alpha$ of $\mathcal{R}|L$ and sketch why the behavior corresponding to $\alpha$ satisfies the timeliness, consistency, and causality properties.

### Proving the Timeliness Property

Consider first the timeliness property:

1. **Normal Receipt of Messages**: We need to show that there is some constant $c$ such that every every receive event that occurs at time greater than $\beta.start + c \cdot n$ in any execution $\alpha$ is normal. Also if $a_j$ is any normal receive event and $a_i$ is the send corresponding to $a_j$, then $a_j$ occurs within $O(n)$ time after $a_i$.

   This follows because that any message $m$ in transit from say $v$ to $u$ (i.e, stored either in the queue at $v$, the link from $v$ to $u$, or the buffer at $u$) cannot remain in transit for more than $O(n)$ time. If a message $m$ is stored either in the queue at $v$ or in the link from $v$ to $u$, then (by the properties of the link and the fact that the queue holds at most 5 packets), the message $m$ will be stored in the buffer at $u$ in constant time. Next, we argue that in $O(n)$ time after a state $s_i$ in which a a message $m$ is in the buffer at $u$, $m$ will either (see Figure 7.18) be delivered or be "flushed" by an ABORT packet sent from $v$.

   If $status(v)$ remains *on* for a constant amount of time after $s_i$, then the message $m$ will be delivered. Thus the only other possibility is that message $m$ remains in the buffer because $status(v)$ is *off* in constant time after $s_i$. But in that case by the Signal Lemma, there will be a SIGNAL$_u$ event in $O(n)$ time after $s_i$. With a little work (see appendix) we can show that such SIGNAL$_u$ events cannot keep occurring at node $u$ for a linear amount of time without causing $v$ to send an ABORT packet to $u$; this will flush the buffer at $u$.

2. **Periodic Free Events**: Consider any $t$-suffix $\gamma$ of execution $\alpha$. Then in $\gamma$ either a FREEM$_{u,v}$ occurs within constant time or a signal event occurs at $u$ within $O(n)$ time or a signal event occurs at $v$ within $O(n)$ time.

   This follows because of the following observation. Let $c$ be a sufficiently large constant time. Suppose either $status(u)$ or $status(v)$ is *off* in $c$ time after the

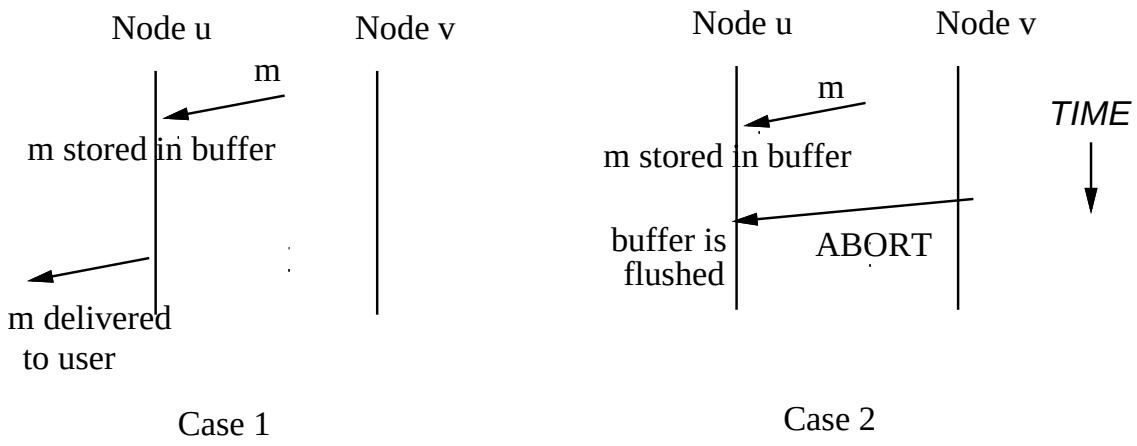



Figure 7.18: Two Cases for how a message is removed from a link

start of $\gamma$. Then we are done by the Signal Lemma. But if this is not true, and $c$ is sufficiently large, then any message $m$ in transit from $u$ to $v$ will be delivered in constant time; also any $\Sigma$-Ack message in transit from $v$ to $u$ will be delivered in constant time; this will result in $\mathit{freem}_u[v] = \mathit{true}$ in constant time. In constant time after $\mathit{freem}_u[v]$ is *true* (and assuming that $c$ is large enough so that $\mathit{status}(u)$ remains *on* in this interval), a $\text{FREEM}_{u,v}$ event will occur.

3. **Timely Message Delivery**: Suppose $a_j$ is a safe $\text{SENDM}_{u,v}(m)$ event in $\gamma$. Then either a $\text{RECEIVEM}_{u,v}(m)$ occurs within constant time after $a_j$ or a signal event occurs at $u$ within $O(n)$ time or or a signal event occurs at $v$ within $O(n)$ time.

   This follows because of a similar observation to the one used to show periodic free events. Let $c$ be a sufficiently large constant time. Suppose either $\mathit{status}(u)$ or $\mathit{status}(v)$ is *off* in $c$ time after $a_j$. Then we are done by the Signal Lemma. If not, by arguments similar to the ones above, we show that message $m$ will be accepted and stored at $u$ and then sent to $v$ where it will be delivered in constant time. We assume that $c$ is large enough so that $\mathit{status}(u)$ and $\mathit{status}(v)$ remain *on* in this interval.

4. **Signals at a Node induce Signals at Neighbors**: There is some constant $c$ such that for every $\text{SIGNAL}_u$ event $a_j$ that occurs at time greater than $\beta.\mathit{start}+c\cdot n$ there is a $\text{SIGNAL}_v$ event that occurs in linear time before or after $a_j$.

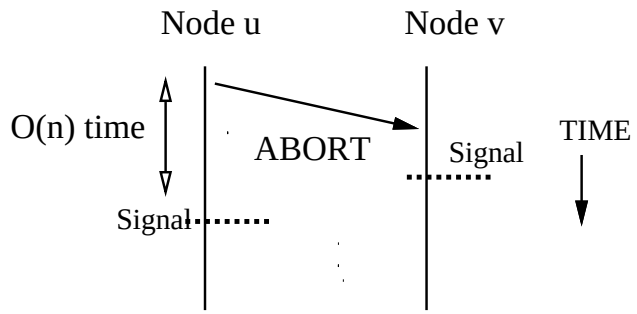



Figure 7.19: A signal event at $u$ that occurs sufficiently "late" must be preceded in linear time by the sending of an ABORT packet to $v$. This in turn causes a signal event to occur within linear time at $v$.

First, within linear time of the start of the execution $\alpha$ corresponding to $\beta$, there must be some state $s_h$ in which $mode = Ready$ (by the termination lemma). In constant time after $s_h$, there must be some state $s_i$ in which $signalbit_u = false$. This follows because $signalbit_u$ cannot remain *true* for constant time without a $\text{SIGNAL}_u$ action occurring, which sets $signalbit_u = false$. Thus any $\text{SIGNAL}_u$ action $a_j$ that occurs after state $s_i$ must have been "caused" by $u$ receiving a reset request or an ABORT packet while in *Ready* mode. Let this action be $a_{j'}$; by the termination lemma, $a_{j'}$ occurs in linear time before $a_j$. But as part of action $a_{j'}$, the code will also send an ABORT packet to neighbor $v$ as sketched in Figure 7.19. Thus in constant time after $a_{j'}$, this ABORT packet will arrive at $v$ and result in a state in which $status(v) = off$. Thus, by the Signal Lemma, in $O(n)$ time after $a_{j'}$, a $\text{SIGNAL}_v$ event occurs. Since $a_{j'}$ occurs within linear time before $a_j$, the $\text{SIGNAL}_v$ event occurs within linear time before or after $a_j$.

### Proving the Consistency Property

Consider now the consistency property. As in the Simple Reset Protocol, the consistency conditions follow from the sending of ABORT packets between signal intervals. *We define a signal interval $S_u$ at $u$ and a signal interval $S_v$ at $v$ to be mates iff a normal message sent in $S_u$ is received in $S_v$ or vice versa.* We can show that the mating relation is well-defined and symmetrical because of the following two properties.

The first property which we call *send consistency* states that messages sent in a signal interval at $v$ can be received in at most one signal interval at $u$; conversely

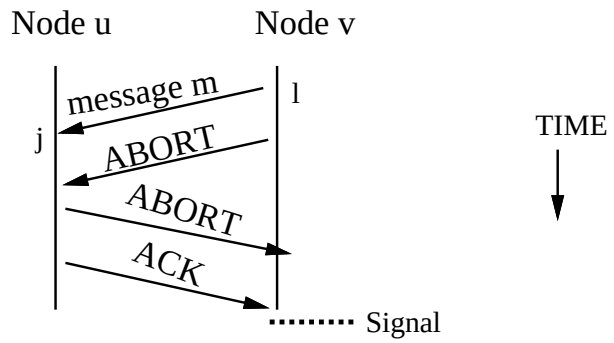



Figure 7.20: What happens between the sending of a message and a later Signal event.

messages received in a signal interval at $u$ could have been sent in at most one signal interval at $v$. This will help establish that each signal interval at $u$ can have at most one mated signal interval at $v$.

**Send Consistency**: Let $a_j$ and $a_k$ be any two normal receive events at $u$ from $v$ in $\beta$. Let $a_l$ and $a_m$ be the send events corresponding to $a_j$ and $a_k$ respectively. Then there is a SIGNAL$_v$ event between $a_l$ and $a_m$ iff there is a SIGNAL$_u$ event between $a_j$ and $a_k$.

Consider Figure 7.20. This figure shows that if there is a SIGNAL$_v$ event after $a_l$ then there must be an ABORT packet sent in between from $v$ to $u$. On receipt of this packet, all packets in the buffer at $u$ will be flushed and *status*($u$) must become *off*. Next, $u$ will not deliver any messages until it has performed a SIGNAL$_u$ and *status*($u$) is *on* again. But any messages sent by $v$ after the SIGNAL$_v$ event will arrive (by the FIFO property of the queue and link) after the ABORT packet arrives at $u$. This guarantees that such messages will not be delivered until $u$ performs its SIGNAL$_u$ event. Similar (but more complicated) arguments can be used to show the other side of this claim.

Next, we show a second property which states (in essence) that a signal interval at $u$ cannot send messages to and receive messages from different signal intervals at $v$. This will help show that the mating relation is symmetric.

**Send-Receive Consistency** Let $a_j$ be a normal receive event at $u$ from $v$ and let $a_m$ be a normal receive event at $v$ from $u$. Let $a_l$ and $a_k$ be the send events corresponding to $a_j$ and $a_m$ respectively. Then there is a SIGNAL$_v$ event between $a_l$ and $a_m$ iff there is a SIGNAL$_u$ event between $a_j$ and $a_k$.

Consider Figure 7.20. This figure shows that if there is a SIGNAL$_v$ event after $a_l$ then there must be an ABORT packet sent from $v$ to $u$ before $v$ performs its SIGNAL$_v$ event. When this packet arrives at $v$ there are two possibilities. Suppose the mode of $u$ is not equal to *Ready* at this point. If $u$ has sent any message $m$ to $u$ in the past, then $u$ must have sent an ABORT packet to $v$ after message $m$. On the other hand, if the mode of $u$ is *Ready* when the ABORT packet from $v$ arrives, then $u$ will immediately send an ABORT packet to $u$. In either case, the ABORT packet from $u$ to $v$ will arrive at $v$ before the ACK (see Figure 7.20 which shows the second case) from $u$ to $v$. But when the ABORT packet arrives at $v$, it will flush out any messages in transit from $u$ to $v$; it is only later that the SIGNAL$_v$ event can be performed.

This effectively means that any messages sent by $u$ after the ABORT is received at $u$ can only be delivered at $v$ after the SIGNAL$_v$ event. Similar arguments can be used to complete the proof of this claim.

Now we show the third property of a consistent behavior listed in Definition 7.3.5.

**Successful Sending of Messages**: Between any safe SENDM$_{v,u}(m)$ event and a later FREEM$_{v,u}$ event, there is either a RECEIVEM$_{v,u}(m)$ event or a SIGNAL$_v$ event.

Call the first SENDM$_{v,u}(m)$ event $a_i$. If in state $s_i$, $status(v) = off$, message $m$ will be dropped; but in this case by the Signal Lemma a SIGNAL$_v$ will occur after $s_i$. But in the period between $s_i$ and the SIGNAL$_v$ event, since $status(v) = off$, the code ensures that no FREEM$_{v,u}$ event can occur. So suppose $m$ is placed in $queue_v[u]$ in $s_i$. Now we return to Figure 7.18. We know that either (case 1) $m$ is delivered to $u$ or $m$ is destroyed by a later ABORT packet. In either case, we know from predicate $\mathcal{H}$, that $freem_v[u]$ will be *false* until $m$ is no longer in transit and so no FREEM$_{v,u}$ can occur in the interim. In Case 2, after the ABORT is sent by $v$, $status(v)$ will remain *off* until a SIGNAL$_v$ occurs. Thus in this period as well no FREEM$_{v,u}$ event can occur.

Next, we show the fourth property of a consistent behavior listed in Definition 7.3.5.

**Mating of Final Signal Intervals**: Let $a_j$ be a normal receive event at $u$ from $v$ in $\alpha$ and $a_l$ be the send corresponding to $a_j$. Then there is no SIGNAL$_u$ event after $a_j$ iff there there is no SIGNAL$_v$ event after $a_l$.

This follows from Figure 7.20. If there is a SIGNAL$_v$ after $a_l$ there must be an ABORT packet that will be received at $u$ after $a_j$. This will result in $status(u)$ becoming

*off* after $a_j$. The Signal Lemma now tells us that there will be a SIGNAL$_u$ event after $a_j$. The reverse argument is similar.

Finally, the fifth property of a consistent behavior (i.e., the mating relation preserves temporal ordering) follows in essence from the fact that the UDLs are FIFO links and the fact that ABORT packets sent between signal intervals flush the links and buffers of previously sent messages.

### Proving the Causality Property

To show that the behavior corresponding to execution $\alpha$ is causal, we prove the two properties of a causal behavior:

1. There is some constant $c$ such that every signal event $a_k$ in $\alpha$ that occurs at time greater than $\beta.start + c \cdot n$ is preceded by a request event $a_j$ that occurs in linear time before $a_k$.

   It is sufficient to show that there is some constant $c$ such that the following is true: if we consider any interval $[s_i, s_k]$ in which no reset request occurs and such that $s_k.time - s_i.time \leq cn$, then $a_k$ cannot be a signal event. The required property follows from this because it implies that if $a_k$ is a signal event, then a reset request must have occurred in linear time before $a_k$.

   If we choose $c$ large enough, then we can break the interval $[s_i, s_k]$ into four subintervals $[s_i, s_{i'}]$, $[s_{i'}, s_j]$, $[s_j, s'_j]$ and $[s_{j'}, s_k]$ such that:

   - Every node $u$ has had $mode(u) = \mathit{Ready}$ in the interval $[s_i, s_{i'}]$. (More precisely, for every node $u$ there is some state $s_l$ in the interval $[s_i, s_{i'}]$ such that $s_l.mode(u) = \mathit{Ready}$.)
   - Every node $u$ has had $mode(u) = \mathit{Ready}$ in the interval $[s_{i'}, s_j]$.
   - Every node $u$ has had $mode(u) = \mathit{Ready}$ in the interval $[s_j, s_{j'}]$.
   - For any $u$, any signal event enabled in $s_{j'}$ is guaranteed to occur in the interval $[s_{j'}, s_{k-1}]$.

   Intuitively, we choose the first three subintervals to be long enough, so that every node will have had a chance to go to *Ready* once in each subinterval. The termination lemma tells us that this can be done using subintervals whose duration

is $O(n)$. Finally, the fourth subinterval needs to be sufficiently long so that any signal events enabled at the start will occur before the end of the subinterval. By the timing conditions (see code), this can be done using a subinterval whose duration is a constant.

First, note that a node $u$ can become a root (i.e. $mode_u = Abort$ and $parent_u = nil$ only through two events: first, by a reset request at node $u$; and second by receiving an (ABORT, $n'$) packet with the distance variable at the maximum value. We call the second transition, a *spurious reset request*. We first claim that a spurious reset cannot occur in any state in which the following *bounded distance* predicate holds: for all nodes $u$, if $parent_u = nil$ then $dist_u = 0$. This follows because if there is an (ABORT, $n'$) in transit from say $v$ to $u$ we can apply $\mathcal{A}$ and $\mathcal{B}$ repeatedly to obtain a chain of nodes ending with a root node $r$. If $dist_r = 0$ then by $\mathcal{E}$ and $\mathcal{F}$ we can arrive at a contradiction in that any ABORT packet must carry a distance less than $n - 1$.

Next, we claim that all spurious reset requests disappear after the first subinterval. This is because the bounded distance predicate must hold after the first interval. If any node $u$ becomes a root after the first interval, since it was *Ready* at some time during the first interval, it must have become a root through either a reset request or a spurious reset request; but either of these transitions will also set $dist_u = 0$.

Next, we claim that for every node $u$, at the end of the third subinterval that $mode(u) = Ready$. This follows because of the following intuitive argument.

Recall that a root of an abort tree is a node $u$ with $parent_u = nil$ and $mode(u) = Abort$. Second, we claim that any roots of abort trees that existed at the start of the second subinterval can no longer exist by the end of the second subinterval. This is because (by choice of subinterval) any such root node must have changed its mode to *Ready* by the end of the second subinterval. But since no reset requests occur in the entire interval, and no spurious requests occur after the first subinterval, no new roots can be created after the first subinterval. Thus there are no roots by the end of the second subinterval.

Now consider any node $u$. But if there are no roots at the end of the second subinterval, $u$ cannot enter *Abort* mode in the third subinterval. This is because in any state $s$ in which $s.mode(u) =$ ABORT there must be a root node. (This in

turn follows by applying predicate $\mathcal{B}$ repeatedly.) But we know that $u$ changed its mode to *Ready* somewehere in the third subinterval. Also, $u$ cannot change its mode from *Ready* to *Converge* or *Abort* without first going into *Abort* mode. Thus, $u$ must be in *Ready* mode by the end of the third subinterval.

Finally, if no real or spurious reset requests occur in the fourth subinterval and every node is ready by the end of the third subinterval, then no signal event can occur at the end of the fourth subinterval. This is because we choose the fourth subinterval such that any signal actions enabled at the start of this subinterval would occur *before* the end of this subinterval.

2. A $\text{SIGNAL}_u$ event occurs within $c \cdot n$ time after any $\text{REQUEST}_u$ event in $\alpha$.

   This follows easily because immediately after the $\text{REQUEST}_u$ event, $status(u) = off$. The claim now follows from the Signal Lemma.

Thus, as we show more formally in Theorem D.2.28, every behavior of $\mathcal{R}|L$ is in $RP$. This can be used to show:

**Theorem 7.7.1** *$\mathcal{R}^+$ stabilizes to the behaviors in problem RP in constant time.*

**Proof:** Follows directly from Theorem 7.6.8 and Theorem D.2.28. ∎

## 7.8 Local Correctability and Dynamic Network Protocols

A *dynamic network* is a network in which faults are limited to link failures: links can crash and recover in arbitrary fashion. A dynamic protocol is a protocol that works correctly in dynamic networks. If we assume that the topology (and the list of neighbors of each node) eventually stabilizes, then any stabilizing protocol $P$ will eventually work correctly in a dynamic network. This is because any finite sequence of link failures can only leave $P$ in some arbitrary state.

A large number of protocols have been designed for dynamic networks. Dynamic protocols are useful because the most common faults in real networks are link and node crashes. Many of these existing protocols have not explicitly been designed to be

stabilizing. However, in this section, we conjecture that a number of dynamic protocols can be made locally correctable.

We start with the reset protocol described in [AAG87] on which the reset protocol described in this chapter is based. This protocol was originally designed for dynamic networks. Thus besides the actions described in this chapter, the protocol in [AAG87] had actions for link failure and recovery. Thus for every node $u$ and every neighbor $v$ of $u$, the protocol had an input action LINK_UP$_{u,v}$ (corresponding to the link to $v$ coming up at node $u$) and an input action LINK_DOWN$_{u,v}$ (corresponding to the link to $v$ coming down at node $u$).

Next, consider the reset function (Figure 7.15) used in this chapter for the reset protocol. The reset function applied to a node $v$ with respect to a neighbor $u$ is exactly:

- The code performed in [AAG87] for a LINK_DOWN$_{u,v}$ event, immediately followed by
- The code performed in [AAG87] for a LINK_UP$_{u,v}$ event.

In other words, we can obtain a local reset function by simulating a link failure immediately followed by a link recovery. Is this a coincidence?

We present a rough (but incomplete) argument as to why this might work. First, consider the stability condition for local correctability. Consider any neighbor $w$ of $u$. Suppose that the $(u, w)$ subsystem is in a good state – i.e., in a state that belongs to $L_{u,w}$. Now when the link to $v$ comes up or goes down, the original protocol had to preserve the stability of $L_{u,w}$. Thus the code for the LINK_UP$_{u,v}$ and LINK_DOWN$_{u,v}$ events preserves the stability of $L_{u.w}$.

Now consider the correction condition. Clearly it is possible in the dynamic protocol to have the link at $u$ go down simultaneously at both ends and then come up simultaneously at both ends. When the link comes up it should come up with no messages in the links and with both $u$ and $v$ in states that satisfy $L_{u,v}$.

Both arguments given above are incomplete. For example, consider our "proof" of the stability condition. We claimed that if the $(u, w)$ subsystem was in a good state, the LINK_DOWN$_{u,v}$ events would preserve the stability of $L_{u.w}$. To make a more careful argument we have to add the following local extensibility condition. For every

state $a \in L_{u,w}$, there is some valid global state $s$ of the network, in which the $(u,w)$ subsystem is in state $a$ and the link from node $u$ to node $v$ is considered to be up at node $u$. Since $s$ is a valid state of the original protocol, and the LINK_DOWN$_{u,v}$ can occur in state $s$, the result of taking the LINK_DOWN$_{u,v}$ action must result in a new valid state $s'$. But since $s'$ is a valid state of the original protocol it must satisfy all local predicates, including $L_{u,w}$.

It is possible to formalize the intuitive arguments by adding similar local extensibility conditions, and showing that the protocol in [AAG87] satisfies these conditions. We will not do so here.

## 7.9 Summary

The three main ideas in this chapter are as follows:

First, we have given a new definition of the correctness of a reset protocol in terms of its external behaviors. We have seen that the resulting definition is, in some sense, a generalization of the synchronization guarantees offered by a Data Link protocol. However, while a Data Link protocol synchronizes two nodes, a reset protocol synchronizes multiple nodes. Notice that our definition specifies the behavior of the reset protocol when reset requests are continuously being made and not just in the event that there is a last reset request. The definition we used in our original paper ([APV91b]) only specified the behaviors in the event that there is a final reset request. However, in trying to apply the reset protocol (for instance, in Chapter 8) we soon found a need for the present specification.

Second. we have applied the Local Correction theorem to stabilize a version of the reset protocol described in [AAG87]. We had to make some subtle changes to the original protocol to make it locally checkable and correctable.

Third, we have conjectured that many locally checkable protocols that work in dynamic networks can be made locally correctable. To obtain a reset function, we concatenate the code that the original protocol used for a link down event with the code used for a link up event. As another example, Spinelli [Spi88a] describes a *virtual circuit* protocol that works in dynamic networks. This protocol appears to be locally checkable and it appears that we can use the link up and link down code to create a reset function. Interestingly, Spinelli [Spi88a] makes his protocol stabilizing by sending

a message periodically from the source of the virtual circuit to the destination of the virtual circuit and back. He also uses a timer whose value is proportional to the maximum end-to-end delay in the network. If, as we conjecture, local checking and correction is applicable to Spinelli's protocol, the resulting protocol will be simpler and faster than the one presented in [Spi88a].

# Chapter 8

# Global Correction Theorem

The previous three chapters have been concerned with local checking and *local* correction. This chapter marks an important transition as we move to local checking and *global* correction. For the next two chapters, we will study the use of the stabilizing reset protocol of Chapter 7 for global correction. In this chapter, we will prove a Global Correction Theorem. This theorem states that any *locally checkable* protocol can be stabilized in time proportional to the number of network nodes using global correction. Thus global correction removes the need for the original protocol to be locally correctable but pays a price in terms of stabilization time. In the next chapter, we will apply global correction to a simple synchronizer protocol [Awe85].

The focus of this chapter is the Global Correction Theorem. The Global Correction theorem should be contrasted with the Local Correction Theorem (Theorem 5.4.3) of Chapter 5. The extra price paid for Global Correction is a stabilization time that is proportional to the *number of network nodes* instead of the *height of some underlying partial order.*

A second important idea contained in this chapter is the concept of one-way checkability. One-way checking is a special case of local checking that results in simpler and more efficient stabilizing protocols. We illustrate the two ideas in this chapter by using a simple spanning tree protocol. The spanning tree protocol is locally checkable and hence can be stabilized using the Global Correction Theorem. However, because the Spanning Tree protocol is also one-way checkable, we can create a simpler (and more efficient) stabilizing spanning tree protocol.

This chapter is organized as follows. In the first two sections, we state and prove the *Global Correction Theorem*. In Section 8.3 we describe a locally checkable protocol to compute a spanning tree. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network. Because it does not appear that the spanning tree protocol is locally correctable, the methods of Chapter 5 do not seem applicable. However, the Global Correction Theorem applies to this spanning tree protocol.

Next, in Section 8.4 we explain the concept of one-way checkability, and show that the spanning tree protocol is one-way checkable. We combine this observation with with the Global Correction Theorem to yield a simple stabilizing spanning tree protocol. Finally, in Section 8.6 we quickly sketch how Global Correction can also be applied to the design of a stabilizing protocol for topology maintenance in dynamic networks.

## 8.1 Statement of Global Correction Theorem

In this section and the next, we show that *any* locally checkable protocol $\mathcal{N}$ can be efficiently stabilized using a reset protocol. The basic idea in the proof of the Global Correction theorem is extremely similar to the transformation used in Chapter 5 (see Figure 5.6 and Figure 5.7) for the proof of the Local Correction theorem. As before, we augment the original automaton with actions to perform local snapshots on every $(u, v)$ subsystem. However, there are two crucial differences in the proof of Global Correction:

- First, we replace all links (UDL's) in the original protocol by a stabilizing reset protocol for graph $G$ as described in Chapter 7. In other words, each node $u$ now communicates with its neighbors using the interfaces provided by the reset subsystem. Now, the reset protocol offers an interface that is identical to a UDL except that packets are replaced by messages. Thus we also have to replace the actions that send and receive packets in the node automata with actions to send and receive messages.

- Second, when a violation is detected in the $(u, v)$ subsystem, a global reset request is made using the $\text{REQUEST}_u$ action offered by the reset protocol interface. Then

when a $\text{SIGNAL}_u$ action occurs, node $u$ will essentially do a "local restart" by initializing its state. This will guarantee that no more violations of the $(u, v)$ subsystem will be detected after both $u$ and $v$ have initialized their state.

Suppose we are given a locally checkable automaton $\mathcal{N}$. Then in the transformation outlined above, we need to replace the events to send and receive packets by events to send and receive messages. Thus we cannot guarantee that the transformed automaton stabilizes to the behaviors of $\mathcal{N}|L$ but to a renamed version of $\mathcal{N}|L$ in which action names have been suitably renamed. This motivates:

**Definition 8.1.1** *Consider a network automaton $\mathcal{N}$. We define $R(\mathcal{N})$, the message-renamed version of $\mathcal{N}$, to be the automaton that is identical to $\mathcal{N}$ except that for all $u, v$:*

- *Every* $\text{SEND}_{u,v}(*)$ *(i.e., packet send) event in $\mathcal{N}$ is renamed as a* $\text{SENDM}_{u,v}(*)$ *(i.e., message send) event.*
- *Every* $\text{RECEIVE}_{u,v}(*)$ *(i.e., packet receive) event in $\mathcal{N}$ is renamed as a* $\text{RECEIVEM}_{u,v}(*)$ *(i.e., message receive) event.*
- *Every* $\text{FREE}_{u,v}$ *(i.e., packet free notification) event in $\mathcal{N}$ is renamed as a* $\text{FREEM}_{u,v}$ *(i.e., message free notification) event.*

*The message-renamed version of a set of behaviors of a network automaton is defined similarly.*

Note that renaming does not really affect the services offered by an automaton because the users of the automaton can change their interface to accommodate the renaming.

With this definition, we can state the main theorem of this chapter. It states that any locally checkable network automaton $\mathcal{N}$ can be transformed into another automaton $\mathcal{N}^+$ such that $\mathcal{N}^+$ stabilizes to a version of $\mathcal{N}$ in which i) All local predicates hold ii) The node and link delays are increased by a constant factor iii) The packet events are renamed as message events.

**Theorem 8.1.2 Global Correction:** *Consider any network automaton* $\mathcal{N} = Net(G, N)$ *that is* locally checkable *for some predicate* $L$ *using link predicate set* $\mathcal{L}$. *Then there exists some* $\mathcal{N}^+$ *that is a message-renamed UIOA for graph* $G$ *and a constant* $c$ *such that* $\mathcal{N}^+$ *stabilizes to the behaviors of* $R(\mathcal{N}(c)|L)$ *in* $O(n)$ *time.*

## 8.2 Proof of the Global Correction Theorem

We present the main ideas in the proof of the Global Correction Theorem in the following subsections. We first sketch the construction of the augmented automaton $\mathcal{N}^+$. Then, we show that after linear time of the start of any execution of $\mathcal{N}^+$ all reset requests disappear. We use this to conclude that $\mathcal{N}^+$ stabilizes to a message-renamed version of $\mathcal{N}$ in linear time. Finally, we make some observations about modularity in the proof.

### 8.2.1 Construction of $\mathcal{N}^+$

Let $L_{u,v}$ be the local predicate for each leader edge $(u, v)$. We construct $\mathcal{N}^+$ as follows. As in Chapter 5, for every leader edge $(u, v)$ we add actions to nodes $u$ and $v$ to perform the local snapshot protocol on the $(u, v)$ subsystem. The local snapshot protocol is used to detect a violation of predicate $L_{u,v}$. When the leader $u$ detects a violation, $u$ *makes a reset request.* The code is derived from the code used for the proof of the Local Correction theorem in Chapter 5 (Figures 5.6, 5.7, and 5.5) by making the following changes:

- All sending and receiving of packets in the modified node automata $N_u^+$ is replaced by sending and receiving of messages so that $N_u^+$ can be a user of the reset subsystem. Thus instead of sending request and response packets, we send request and response messages. Similarly we have to replace the free packet notification events with free message notification events.
- There is no longer a need for a *mode* variable in $N_u^+$ because each phase is implicitly a snapshot phase. Whenever the mode is checked in the original code, we only follow the code path corresponding to $mode = snapshot$.
- We add a $requestbit_u$ variable to $N_u^+$ to remember to do a reset request.

- When a response is received at the leader and a local predicate violation is detected, $requestbit_u$ is set to *true* (instead of changing $mode_u[v]$ to *reset*).

- We compose the modified node automata $N_u^+$ with the reset protocol $\mathcal{R}|L$ of Chapter 7. Thus in the final automaton, $N_u^+$ has input action $\text{SIGNAL}_u$ and output action $\text{REQUEST}_u$.

- $N_u^+$ makes a reset request $\text{REQUEST}_u$ whenever $requestbit_u$ is *true*; on taking this action $requestbit_u$ is set to *false*.

- On receiving a $\text{SIGNAL}_u$ event, $N_u^+$ does a local restart. It first resets the basic state of the original automaton $N_u$ to some initial value $I_u$ ( we will discuss how $I_u$ is determined below). Also for each neighbor $v$ of $u$, $u$ will locally reset all the phase and free variables. More precisely, the $free_u[v]$, $freeq_u[v]$ and $phase_u[v]$ variables are all set to *false* for all neighbors $v$. The counter variable $count_u[v]$ is initialized to 1 (if $u$ is the leader) and to 0 (if $u$ is not the leader). This will ensure that after a signal event at $u$ and $v$, all $(u, v)$ phases are clean and all message send events at $u$ are safe.

- The definition of local checkability in Chapter 5 ensures that there is always at least one global state $s$ which satisfies all local predicates. Now if there are any messages in transit in global state $s$, we consider the state $s'$ that results when all these messages are received by their destinations. Because the local predicates are all closed predicates, $s'$ satisfies all local predicates as well. But in addition $s'$ has no messages in transit. Then we choose the initial state $I_u$ for each node $u$ to be equal to $s'|u$.

The modified code is described in Figure 8.1, Figure 8.2, and Figure 8.3. The message formats and timing partitions are as in the proof of the Local Correction Theorem. As before, we hide all actions of $\mathcal{N}^+$ that are not actions of $\mathcal{N}$.

### 8.2.2 All reset triggers disappear in Linear Time

In the proof we will make heavy use of the reset protocol specification given in Chapter 7 and some aspects of the proof of the Local Correction Theorem in Chapter 5. Refer to Chapter 7 for definitions of normal messages, the send corresponding to a receive,

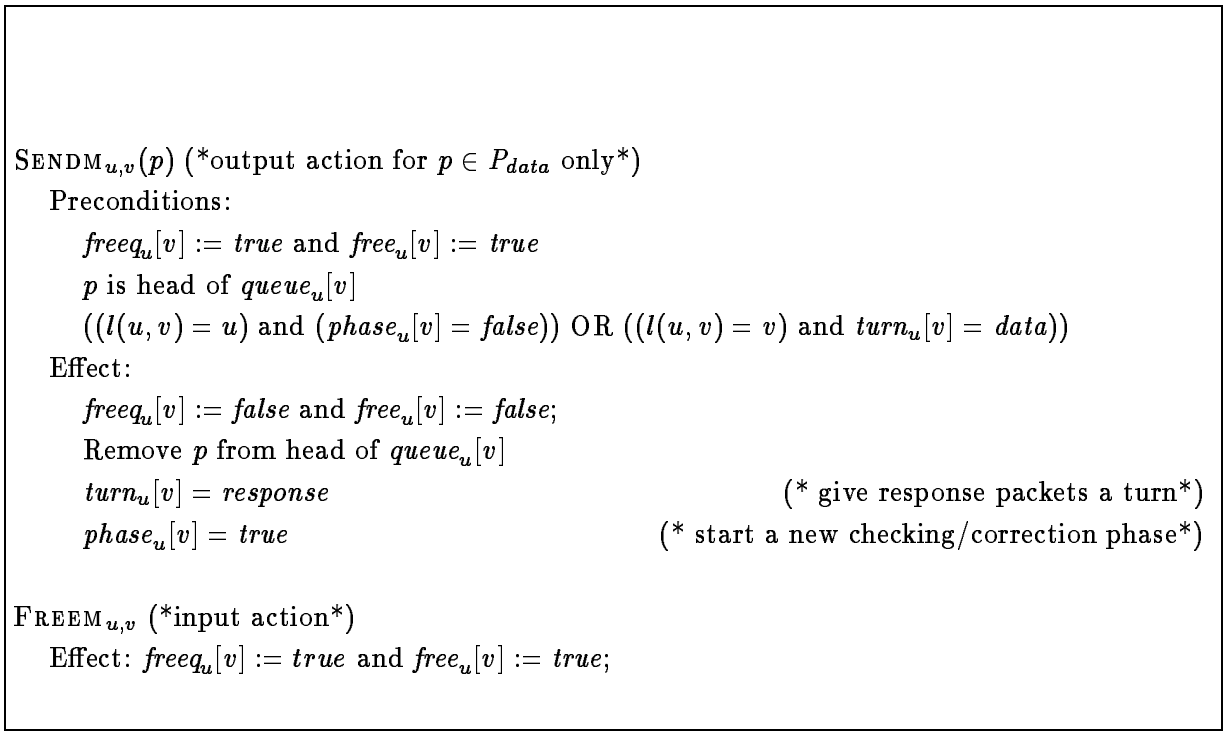



Figure 8.1: Code for the modified SEND$_{u,v}(p)$ actions at a modified node $N_u^+$ in order to do Global Correction.

SENDM$_{u,v}(p'_{req})$ (*output action: $u$ repeatedly sends a request till it gets a response*)

Precondition:

$l(u,v) = u$ (*$u$ is the leader of link subsystem*)

($phase_u[v] = true$) or ($queue_u[v]$ is empty) (*phase in progress or no data waiting*)

$freeq_u[v] = true$ (* no message in transit on link to $v$ *)

$p'_{req}.count = count_u[v]$; (* counter correct*)

Effect:

$freeq_u[v] = false$ (* set to false until link says it is free*)

$phase_u[v] := true$; (*remains true until matching response returns*)

RECEIVEM$_{v,u}(p'_{req})$ (*input action, receive request at $u$ from $v$*)

Effect:

If $p'_{req}.count \neq count_u[v]$ and $l(u,v) = v$ then (*not a duplicate or invalid message*)

$count_u[v] := p'_{req}.count$; (*remember count*)

Figure 8.2: Code to send and receive request messages at node $u$ in order to do Global Correction.

SENDM$_{u,v}(p'_{resp})$ (*output action: $u$ repeatedly sends a response to last request*)
  Preconditions:
    $l(u,v) = v$ (*$u$ is not the leader of link subsystem*)
    ($turn_u[v] = response$) or ($queue_u[v]$ is empty) (*response's turn or no data messages waiting*)
    $freeq_u[v] = true$ (* no message in transit on link to $v$ *)
    $p'_{resp}.count = count_u[v]$;
    $p'_{resp}.node_state = s|u$
  Effect:
    $turn_u[v] := data$ (*give data messages a turn*)
    $freeq_u[v] := false$ (* set to false until link says its free*)

RECEIVEM$_{v,u}(p'_{resp})$ (*input action to receive response at $u$ from $v$*)
  Effect:
    If ($count_u[v] = p'_{resp}.count$) and ($phase_u[v] := true$) and ($l(u,v) = u$) then
      If $(s|u, nil, nil, p'_{resp}.node_state) \notin L_{u,v}$ then $requestbit_u := true$
      $phase_u[v] := false$; (*end of phase*)
      $count_u[v] := (count_u[v] + 1) \bmod 4$;

REQUEST$_u$ (*input action to make a reset request*)
  Preconditions:
    $requestbit_u := true$;
  Effect:
    $requestbit_u := false$;

SIGNAL$_u$ (*input action to receive a Signal at $u$*)
  Effect:
    For all neighbors $v$ (*make links to all neighbors clean*)
      $phase_u[v] := false$;
      If $l(u,v) = u$ then $count_u[v] := 1$ else $count_u[v] := 0$
      Empty $queue_u[v]$ and set $freeq_u[v]$ and $free_u[v]$ to $false$
    $s|u = I_u$ (*initialize node state*)

Figure 8.3: Code to send and receive response messages and to make reset requests and respond to signals at node $u$.

causality, timeliness, and the mating relation. Refer to Chapter 5 for definitions of a phase and a clean link.

The proof consists of showing that all reset requests will eventually stop and that in linear time after this $L$ holds. Define a $(u,v)$ *reset trigger* to be the receipt of a snapshot response from $v$ at $u$ that causes $u$ to set $requestbit_u = true$. The main part of the proof consists of showing that all reset triggers disappear in linear time (which by causality implies that all signals disappear). We start with some preliminary definitions.

Consider an execution $\alpha$ of $\mathcal{N}^+$. Define a *quiescent state* in $\alpha$ to be a state $s_i$ in $\alpha$ such that:

- All messages received after state $s_i$ in $\alpha$ are normal.
- All signal events that occur after state $s_i$ are preceded by a request event that occurs in linear time before the signal event.
- For any pair of neighbors $u$ and $v$, a signal event $a_j$ at $u$ that occurs after state $s_i$ is accompanied by a signal event at node $v$ that occurs within linear time before or after $a_j$.

We have the following lemma:

**Lemma 8.2.1 Quiescent Lemma:** *A quiescent state occurs within linear time of any execution of* $\mathcal{N}^+$.

**Proof:** $\mathcal{N}^+$ is the composition of the augmented node automata with $\mathcal{R}|L$. We know the behaviors of $\mathcal{R}|L$ are timely and causal. The lemma follows from the first timeliness property, the first causality property, and the fourth timeliness property. ∎

Suppose that $u$ is the leader of some $(u,v)$ subsystem. We show that in any execution $\alpha$ of $\mathcal{N}^+$, there exists some linear time $t$ such that all $(u,v)$ reset triggers disappear in time $t$ after a quiescent state in $\alpha$. Since a quiescent state occurs in linear time after the start of $\alpha$, it follows that all reset triggers disappear in linear time after the start of $\alpha$.

The proof is in two parts. First, we show that one of two cases must occur in linear time after a quiescent state. Next, we show that no $(u,v)$ reset triggers can occur after the occurrence of *either* of the two cases.

### One of Two Cases must Occur in Linear Time after a Quiescent State

To define the two cases, we need two simple preliminary definitions. Recall from Chapter 7 the definition of a signal interval and the definition of a send corresponding to a normal message. Intuitively, an *initialized signal interval* at a node is a signal interval that begins with a signal event at the node; we give it this name because each node *initializes* its local state immediately after a signal event. Next, a *regular* message receipt is a message that was received in an initialized signal interval and was sent in an initialized signal interval. Formally:

**Definition 8.2.2** *An* initialized signal interval *at node $u$ is a signal interval at node $u$ that begins with a* $\text{SIGNAL}_u$ *event.*

**Definition 8.2.3** *A message receive event is* regular *if*

- *The message receive event is normal*
- *The message is received in an initialized signal interval at the receiving node.*
- *The send corresponding to the receive event occurs in an initialized signal interval at the sender.*

Next, we state the main lemma of this subsection.

**Lemma 8.2.4 Quiescent Cases:** *Consider any execution $\alpha$ of $\mathcal{N}^+$ and any leader edge $(u,v)$. Within linear time after any quiescent state $s_i$ of $\alpha$ there is a state $s_j$ such that one of the following two cases is true:*

- **Case 1:** *Any message received by $u$ from $v$ (and vice versa) after $s_j$ is* regular *OR*
- **Case 2:** *In state $s_j$, $L_{u,v}$ holds and $(u,v)$ is clean. Also, the interval $[s_i, s_j]$ is contained in some signal interval at $u$ as well as some signal interval at $v$, and these two signal intervals are mates.*

**Proof:** (Sketch) Recall the definition of a $(u,v)$ phase from Chapter 5. Roughly speaking, a $(u,v)$ phase is an interval that begins with the leader $u$ sending a request and ends with $u$ receiving matching response. Recall also the definition of a clean phase from Chapter 5. The proof also makes considerable use of the timeliness, consistency, and causality properties (see Definitions 7.3.3, 7.3.5, and 7.3.6) of the reset automaton $\mathcal{R}|L$.

We choose state $s_j$ as some state that occurs after state $s_i$ and satisfies either one of two properties:

a) **Property 1**: Six $(u,v)$ phases occur in the interval $[s_i, s_j]$ AND there are no signal events at $u$ or $v$ in $[s_i, s_j]$ AND $L_{u,v}$ holds in state $s_j$.

OR

b) **Property 2**: There is some state $s_k$ in the interval $[s_i, s_j]$ such that at least one signal event occurs at *both* $u$ and $v$ before $s_k$. Also, for any message receive event that occurs after $s_j$, the corresponding send event occurs after $s_k$ (i.e., all messages in transit in state $s_k$ are delivered before state $s_j$).

We now show that we can find such a state $s_j$ that occurs within linear time after $s_i$.

From the second and third timeliness conditions, we see that message delivery between $u$ and $v$ behaves exactly like a UDL in the absence of signal events at $u$ or $v$. In other words, the reset subsystem delivers messages and free notifications in constant time in the absence of signal events. Thus from the proof of the Phase Rate lemma in Chapter 5 (Lemma 5.6.5), we see that in the absence of signal events at either $u$ or $v$, a $(u,v)$ phase will complete in constant time. Similarly, in the absence of signal events at either $u$ or $v$, six $(u,v)$ phases will complete in constant time. By Lemma 5.6.12, the sixth $(u,v)$ phase is a *clean* phase – i.e., a phase in which the matching response is sent by $v$ after the receipt of the request. Thus the sixth phase will produce a correct snapshot of the $(u,v)$ subsystem. Thus by the end of the sixth phase, either $L_{u,v}$ holds or node $u$ will make a reset request. But if node $u$ makes a a reset request, then by the second causality property, a signal event will occur at $u$ in linear time after $s_i$.

We conclude from the last paragraph that in linear time after $s_i$ either we reach a state $s_j$ satisfying Property 1 or a signal event occurs at either $u$ or $v$. But if a signal event occurs at *either* $u$ or $v$ in linear time after $s_i$, then we know (from our choice of $s_i$ and the fourth timeliness property), that a signal event occurs at *both* $u$ and $v$ by

some state $s_k$ that occurs within linear time after $s_i$. But in the latter case, the first timeliness property tells us that there is some state $s_j$ that occurs within linear time of $s_k$ and such that all messages "in transit" in state $s_k$ have been delivered before state $s_j$. Thus in linear time after $s_i$ we reach a state $s_j$ satisfying either Property 1 or Property 2.

We are now ready to prove the lemma. If $s_j$ satisfies Property 2 we show that Case 1 must be true; and if $s_j$ satisfies Property 1 we show that Case 2 must be true.

If $s_j$ satisfies Property 2 then we know that all messages received at $u$ from $v$ (and vice versa) after state $s_j$ was sent after some state $s_k$. We also know that at least one signal event has occurred at both $u$ and $v$ before state $s_k$. Thus all such message receive events are regular, and we have Case 1 in the lemma statement.

If $s_j$ satisfies Property 1 then we know by Lemma 5.6.12 that $(u,v)$ is clean in state $s_j$. We also know that $L_{u,v}$ holds in $s_j$ and that no signals occur at either $u$ or $v$ in $[s_i, s_j]$. Thus there must be some signal interval (say $S_u$) at $u$ that contains the interval $[s_i, s_j]$. Similarly, there must be some signal interval (say $S_v$) at $v$ that contains the interval $[s_i, s_j]$. Now the sixth $(u,v)$ phase in $[s_i, s_j]$ is a clean phase. But in a clean phase, $v$ sends a response to $u$ after $v$ receives a request from $u$. Thus there is at least one message sent after $s_i$ by $u$ that was received before $s_j$ at $v$. Hence, from the consistency property, the intervals $S_u$ and $S_v$ must be mates. Thus we have Case 2 in the lemma statement. ∎

### All $(u,v)$ triggers disappear after either Case 1 or Case 2

Next we show that for any leader edge $(u,v)$, all $(u,v)$ reset triggers stop after one of these two cases listed above occur. Since we have just shown that that one of these two cases must occur in linear time, we conclude that all reset triggers disappear in linear time.

We first show all $(u,v)$ triggers disappear after Case 1 occurs:

**Lemma 8.2.5 Case 1 Trigger Termination:** *Consider any execution $\alpha$ of $\mathcal{N}^+$ and any leader edge $(u,v)$. Suppose that after any quiescent state $s_i$ of $\alpha$ there is a state $s_j$ such that any message received by $u$ from $v$ (and vice versa) after $s_j$ is* regular. *Then no $(u,v)$ triggers occur after state $s_j$ in $\alpha$.*

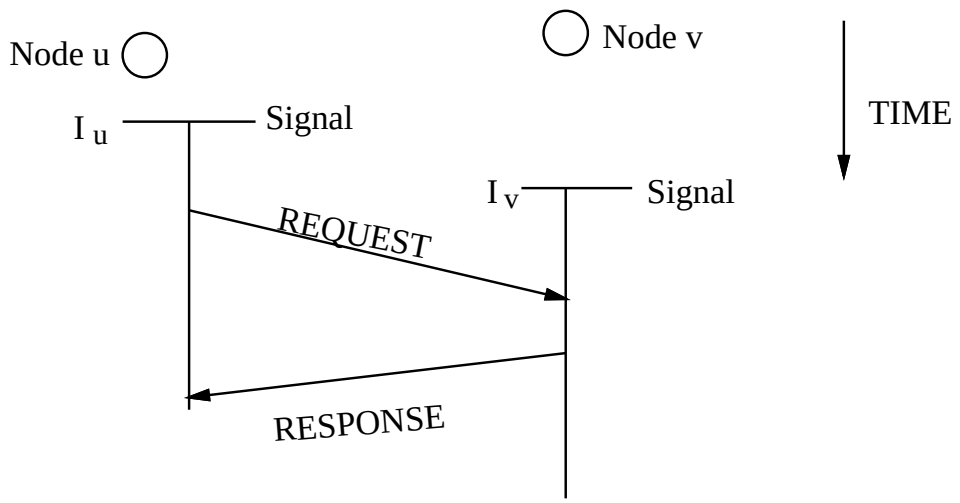



Figure 8.4: Correct Snapshots after nodes $u$ and $v$ have each performed a signal event.

**Proof:** (Sketch) Recall that a $(u,v)$ trigger is the receipt of a response at $u$ which causes $u$ to set $requestbit_u = true$. Consider the receipt of any response at $u$ after $s_j$. By construction, this response must be received in an initialized signal interval (say $S_u$ at $u$) and must correspond to a response sent in an initialized signal interval (say $S_v$ at $v$). This is shown in Figure 8.4.

We claim that the sequence of messages received by $u$ in $S_u$ is a prefix of the sequence of messages sent by $v$ in $S_v$, and vice versa. We will refer to this as the *prefix* property. The prefix property follows from the consistency property because:

- $S_u$ and $S_v$ are mates,
- All messages received after $s_i$ are normal and
- All messages sent in an initialized signal interval are safe.

We now claim that that the matching response cannot be a $(u,v)$ trigger because of the following argument. First, at the start of $S_u$, $u$ sets its basic state to $I_u$ and at the start of $S_v$, node $v$ sets its basic state to $I_v$. But $(I_u, nil, nil, I_v) \in L_{u,v}$. Next, at the start of $S_u$, $count_u[v]$ is initialized to 1; also at the start of $S_v$, $count_v[u]$ is initialized to 0. Thus by the prefix property, the sequence of states and messages sent and received in $S_u$ and $S_v$ *could have occurred in some execution* $\gamma$ of the $(u,v)$ subsystem, such

that $(u,v)$ is clean and $L_{u,v}$ holds in the first state of $\gamma$. Thus, as in Chapter 5, any matching responses will carry accurate snapshot information, and will not result in a $(u,v)$ reset trigger. ∎

Note that what makes this and the result in Chapter 5 work is that $L_{u,v}$ *is a closed local predicate* — once $L_{u,v}$ holds, it continues to hold, regardless of the behavior of other subsystems. Next, we show all $(u,v)$ triggers disappear after Case 2 occurs:

**Lemma 8.2.6 Case 2 Trigger Termination:** *Consider any execution $\alpha$ of $\mathcal{N}^+$ and any leader edge $(u,v)$. Suppose that after any quiescent state $s_i$ of $\alpha$ there is a state $s_j$ such that:*

- *$L_{u,v}$ holds and $(u,v)$ is clean in $s_j$.*
- *The interval $[s_i, s_j]$ is contained in some signal interval at $u$ as well as in some signal interval at $v$, and these two signal intervals are mates.*

*Then no $(u,v)$ triggers occur after state $s_j$ in $\alpha$.*

**Proof:** (Sketch) The argument is extremely similar to that for Case 1 except in the signal intervals at both $u$ and $v$ that contain the interval $[s_i, s_j]$. This is shown in Figure 8.5. Let the signal interval at $u$ that includes state $s_j$ be called $S_u$. Let the signal interval at $v$ that includes state $s_j$ at be called $S_v$. We know that $S_u$ and $S_v$ are mates by assumption, and that in $s_j$, $L_{u,v}$ holds and $(u,v)$ is clean. This is shown in Figure 8.5.

Thus the sequence of states and messages sent and received in $S_u$ and $S_v$ after state $s_j$ *could* have occurred in some execution $\gamma$ of the $(u,v)$ subsystem, such that $(u,v)$ is clean and $L_{u,v}$ holds in the first state of $\gamma$. But then, any matching responses will carry accurate snapshot information, and will not result in a $(u,v)$ reset trigger. Thus in the first signal intervals for Case 2 we rely on an explicit state $s_j$ (in which the local predicate holds and the link is clean). By contrast, in Case 1 we relied on the first signal intervals being initialized signal intervals.

The argument for signal intervals after $S_u$ and $S_v$ in Case 2 is identical to the argument for Case 1 because such signal intervals are initialized signal intervals. ∎

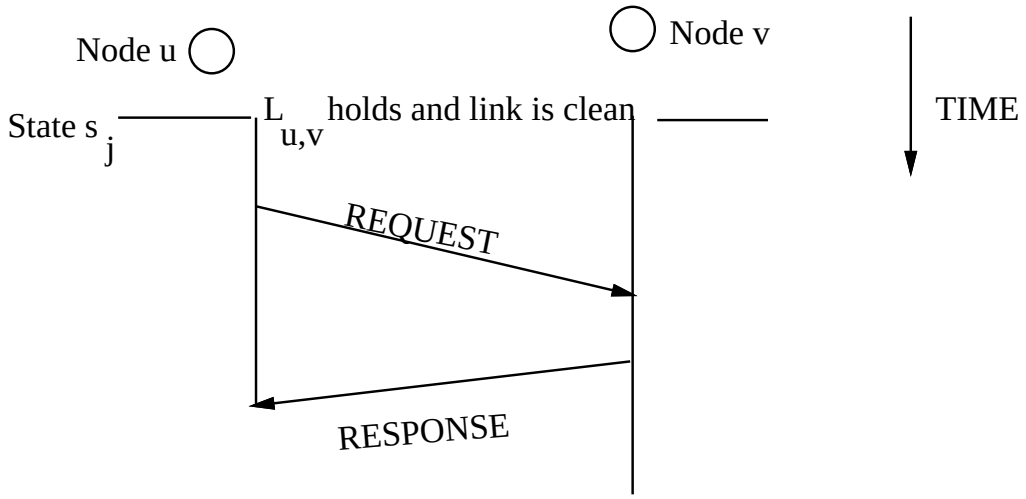



Figure 8.5: Correct Snapshots after $L_{u,v}$ holds and link $(u, v)$ is clean.

### 8.2.3 All Local Predicates Hold and All Signals Stop in Linear Time

From the Quiescent Lemma, the Quiescent Cases Lemma, and the Case 1 and Case 2 Trigger Termination Lemmas, it follows that all reset triggers disappear in linear time after the start of any execution of $\mathcal{N}^+$. Hence, by causality, all signal events also disappear in linear time. More precisely, any execution $\alpha$ of $\mathcal{N}^+$ has some $O(n)$-suffix $\gamma$ in which there are no signal events.

But now it is easy to see that in constant time in $\gamma$, all local predicates hold. We know from the second and third timeliness conditions (and the fact that there are no signals in $\gamma$) that six $(u, v)$ phases will complete in constant time after the start of $\gamma'$. But after these six phases are complete, $L_{u,v}$ must hold. If not, $u$ would detect a violation in the sixth phase (which is guaranteed to be a clean phase) and $u$ would have made a reset request. But this will result in a signal event in $\gamma$, a contradiction.

Lastly any execution of $\mathcal{N}^+$ which contains no signal events and in which $L_{u,v}$ holds for all links can be shown to be an execution of $\mathcal{N}(c)|L$ for some constant $c$. As in Chapter 5, the $c$ represents a constant slowdown due to the overhead of periodic checking.

### 8.2.4 Proof of Global Correction Theorem is Modular

We note that in the transformation used to prove the Global Correction Theorem we used the reset protocol $\mathcal{R}|L$ from Chapter 7. However, the proof only uses properties of the behaviors of $\mathcal{R}|L$ and thus $\mathcal{R}|L$ could have been replaced by any automaton that has the same interface as $\mathcal{R}|L$ and has the same behaviors. Further, note that the modified node automata $N_u^+$ (that we create by the transformation) are UIOA and $\mathcal{R}|L$ is a CIOA (since every reachable state is also a start state). Thus, the modularity theorem assures us that we can replace $\mathcal{R}|L$ by its stabilizing implementation (the automaton $\mathcal{R}^+$ as described in Chapter 7) without affecting the Global Correction result.

## 8.3 A Locally Checkable Spanning Tree protocol

We begin by describing earlier stabilizing spanning tree protocols and their disadvantages. Then we describe a locally checkable spanning tree protocol. This protocol will be used in Section 8.5 to construct a stabilizing spanning tree protocol

### 8.3.1 Previous Work on Spanning Tree Protocols

The basic idea in virtually all spanning tree algorithms is that nodes report the smallest node ID seen so far (and the shortest distance to this smallest ID node) to their neighbors. Each node then picks as its parent the neighbor that knows of the smallest ID. If more than one neighbor reports the smallest ID, the node picks from among these the neighbor that reports the smallest distance. If two neighbors report the same distance and root ID, then an arbitrary tie-breaker is used to select the parent. A node sets its estimate of the root ID to equal its parent's root ID, and updates its distance from the root to be one plus the parent's distance. Because of the way the distance from the root is calculated, this approach is basically an adaptation of the Bellman-Ford Algorithm.

However, in a dynamic network (and in a stabilizing setting), this approach encounters an obstacle known as "ghost roots". This phenomena occurs whenever the root crashes: its ID, which was the smallest in the system, is still the smallest from

the point of view of nodes that do not know of its crash. Even in a static network, the same effect can be caused by initial errors that introduce a root ID lower than the ID of any network node. This ID can potentially remain forever in the system! Even if the nodes maintain a counter to reflect their distance from the alleged root, this counter will simply grow unboundedly. Since the ghost root phenomenon is not a rare event, several ways to overcome this difficulty have been suggested.

One solution, known as the "hop counter" approach, is to have some pre-determined bound on the diameter of the network at each node, and to discard the old root whenever the associated counter reaches some limit (see, e.g.,[AG90]). Unfortunately, this pre-determined bound must be quite high, and hence, the stabilization time of such counting up schemes is poor in practice [CRKG89a, CRKG89b, Gar89, RF89, Awe90].

Another widely used stabilizing Spanning tree protocol is the IEEE 802.1 bridge routing protocol which is based on the design in [Per85]. This solution uses an approach that we call timer flushing. The basic idea [Per85] is that each node "times out" information received from its neighbors unless the information is refreshed periodically. Any node that thinks it is the root is responsible for periodically sending updates to all its neighbors in order to refresh their state information. Any other node $X$ sends estimates to its neighbors only after receiving a message from the node $X$ thinks is its parent. The upshot is that if there is an old root in the system, the information about this old root will eventually "time out", after which the system will stabilize.

However, timer flushing suffers from several drawbacks. First, the node clocks (by which packet lifetimes are enforced) can have different rates. Second, the message delivery time over links typically has large variance; since the topology of the network is not known in advance, the variance of the message delivery time across the whole network is even higher. A conservative timeout bound must take into account high message latencies and the worst-case topology, even though the latencies of an actual execution may be considerably (e.g., an order of magnitude) smaller. This leads to an order of magnitude slowdown of the stabilization process. Lastly, the parameters of such "global" timeout protocols often have complicated dependencies. Tuning these parameters is typically quite difficult.

Th stabilizing spanning tree protocol we will describe in Section 8.5 is extremely similar to the two schemes we have described above. However, it uses a different mechanism for detecting and recovering from states with ghost roots that speeds up the stabilization time of the resulting protocol.

The detection mechanism is based on the following observation. Consider an execution of the simple spanning tree protocol that starts with a state in which all nodes are correctly initialized and there are no messages in transit on the links. Now focus on some node. Throughout the execution the node maintains a current estimate of the root ID, and another estimate for its distance from this alleged root. It can be shown that in the course of legal executions, the node's estimate of the root ID never goes up; also while such a root estimate is fixed, the distance estimate never goes up. This property can be cast in the form of a local predicate for each link. If the predicate holds, then the algorithm will produce a spanning tree. This immediately suggests the stabilizing algorithm: whenever the predicate is violated, the node that detects the violation makes a reset request. In the execution that follows the last reset signal, all the information will be correct.

Rather than directly describe the final spanning tree protocol of Section 8.5, we will derive it in the following way. In the next subsection, we will describe a locally checkable (but non-stabilizing) spanning tree protocol on which the final spanning tree protocol is based. Of course, the Global Correction Theorem is applicable to this protocol. However, we will show in the following section that the spanning tree protocol is also *one-way checkable*. We then combine these observations to produce the final stabilizing spanning tree protocol in Section 8.5. The resulting protocol is simpler and more efficient than a spanning tree protocol based only on Global Correction.

### 8.3.2 Code for Locally Checkable Spanning Tree Protocol

We now describe the code for our locally checkable (but non-stabilizing!) spanning tree protocol. After all local predicates hold, this protocol computes a spanning tree in time proportional to the diameter of the network.

Before we delve into the code of the protocol, we define what it means for a spanning tree protocol to be stabilizing. We assume that the network topology is described by some topology graph $G$ and that there is an output action $\text{REPORT}_u(p)$ at every node $u$. This action is used to report the parent $p$ of node $u$ in the spanning tree, where $p$ is some neighbor of $u$ in the network graph. We say that a spanning tree protocol is stabilizing if it stabilizes to the *stable tree behaviors* for graph $G$. The stable tree behaviors for graph $G$ are the behavior $\beta$ in which for every node $u$:

- For every $t$-suffix $\gamma$ of $\beta$ and for every node $u$, a $\text{REPORT}_u$ event occurs in constant time. (i.e., nodes report their parent values at constant intervals of time.)
- For every node $u$, if there is $\text{REPORT}_u(p_1)$ event and a later $\text{REPORT}_u(p_2)$ event in $\beta$, then $p_1 = p_2$. (i.e., the parent values reported by nodes never change).
- Consider any set of $\text{REPORT}_*(*)$ actions in $\beta$ containing exactly one $\text{REPORT}_u(p_u)$ action for every node $u$. Then the graph induced by the values of the $p_u$ variables is a spanning tree of $G$. (i.e., the parent values reported by nodes forms a spanning tree of $G$)

Our locally checkable spanning tree protocol is described by the automaton $T_u$ shown in in Figure 8.6. The protocol is based on the simple idea alluded to earlier. Each node keeps track of the smallest node ID seen so far and the shortest distance to this smallest ID node. Each node uses this information to compute its parent in the tree.

Thus each node $u$ maintains a parent pointer $parent_u$, current estimate of the root's identity $r_u$, and a current distance estimate $d_u$. We denote by $(r_u, d_u)$ the ordered pair at node $u$. We will use lexicographic ordering for 2-tuples and 3-tuples. For example, $(r_v, d_v) < (r_u, d_u)$ means that either $r_v < r_u$, or that $r_v = r_u$ and $d_v < d_u$. Similarly, $(r_v, d_v, parent_v) < (r_u, d_u, parent_u)$ means that that either $(r_v, d_v) < (r_u, d_u)$, or that $(r_v, d_v) = (r_u, d_u)$ and $parent_v < parent_u$. Each node also maintains a copy (possibly outdated) of the root and distance estimates of each its neighbors. Thus the estimates of neighbor $v$ are stored at $u$ in the variables $r_u[v]$ and $d_u[v]$.

For compactness, we let both arrays have an entry for the node itself, in which the default values are "hardwired": thus $r_u[u] = u$, and $d_u[u] = -1$. Now a node always chooses its own estimate based on the minimum of the estimates of all its neighbors and its own default estimate. Node $u$ chooses its own estimate of the root from this minimum estimate; $u$ also chooses its own estimate of the distance as one plus the distance from the minimum estimate. Thus if $u$ itself is the smallest root, the distance $u$ chooses for itself will be $-1 + 1 = 0$, which is as it should be. Thus $d_u[u]$ is set to $-1$ simply as a sentinel value to avoid an extra case.

Every node $u$ periodically sends its own estimate of root and distance to all its neighbors using an "Announce" packet. To make the $T_u$ a node automaton, we have followed the convention (see Chapter 5) of first enqueuing the "Announce" packet on an

outbound queue for the link called $queue_u[v]$; then the packet is sent out from the queue when the link is free. The Announce packet is encoded as a tuple $(Announce, r, d)$. When a node $u$ receives an Announce packet from $v$, $u$ first checks whether the estimate in the packet is greater than the previous estimate stored from $v$ and the distance in the estimate is not already at the maximum possible value. If this is not the case, $u$ stores the latest estimate it has received from $v$; $u$ also updates its own estimate as the minimum of all neighbor estimates.

Let $T_u$ be the automaton shown in in Figure 8.6. We assume that for any $u, v$, all actions of the form $\text{SEND}_{u,v}(*)$ are in a separate class, and any $\text{ENQUEUE}_{u,v}$ action is in a separate class. Similarly for any $u$, all actions of the form $\text{REPORT}_u(*)$ are in a separate class.

Let $\mathcal{T}$ be the composition of the automata node $T_u$ for every node in $G$ with a UDL for every edge $(u, v)$ in $G$. In the next subsection we show that $\mathcal{T}$ is locally checkable for a predicate $L$ that we define. In the following subsection, we show that $\mathcal{T}|L$ stabilizes to the stable tree behaviors in time proportional to the diameter of graph $G$. This sets the stage for applying the global correction theorem to $\mathcal{T}$.

### 8.3.3 Local Predicates for $\mathcal{T}$

We state a set of local predicates for the spanning tree protocol. Let us denote by $O_{u,v}$ the intersection of the local predicates shown in Figure 8.7 for any edge $(u, v)$. Note that $O_{u,v}$ consists of two predicates, $O1(u, v)$ and $O2(u, v)$. $O1(u, v)$ says that the sentinel values that node $u$ uses (as a neighbor estimate from itself) are correct. It also says that node $u$'s estimate is the minimum of all its neighbors.

$O2(u, v)$ shows that the sequence of estimates sent from $u$ to a neighbor $v$ are non-increasing. Suppose there is an $(Announce, r, d)$ packet in transit from $u$ to $v$, then the estimates that $v$ already has stored from $u$ can be no less than the estimates in the packet (i.e., $(r_v[u], d_v[u]) \geq (r, d)$). Also the estimates in the packet can be no less than the estimates at $u$ (i.e., $(r, d) \geq (r_u, d_u)$). If there is no $(Announce, *, *)$ packet in transit from $u$ to $v$, then the estimates that $v$ already has stored from $u$ can be no less than than the current estimates at $u$ (i.e., $(r_v[u], d_v[u]) \geq (r_u, d_u)$). Recall that a Unit Capacity Link only allows one outstanding packet on each link, just as in a UDL.

State

| | |
|---|---|
| $neighborSet_u$ | set containing all neighbors of $u$ and $u$ itself |
| $parent_u$ | parent pointer, in $neighborSet_u$ |
| $d_u$: | distance estimate, integer in the range $0..n'$, $n'$ is an upper bound on number of nodes. |
| $r_u$: | root estimate, all root estimates are in the range of node identifiers |
| $r_u[v]$: | estimate of $r_v$ for each $v$ in $neighborSet_u$ except $r_u[u] = u$ |
| $d_u[v]$: | estimate of $d_v$ for each $v$ in $neighborSet_u$ except $d_u[u] = -1$ |
| $free_u[v]$ | boolean, true if link to $v$ is free for each $v$ in $neighborSet_u$ except $u$ itself. |
| $queue_u[v]$ | a queue containing at most one $(Announce, *)$ packet to be sent to $v$. |

Actions

FREE$_{u,v}$ (*input action to receive free notification from link to neighbor $v$*)
Effects: $free_u[v] = true$

SEND$_{u,v}(p)$ (*output action to send estimate to neighbor $v$*)
Preconditions: $p$ is the head of $queue_u[v]$ and $free_u[v] = true$
Effects: $free_u[v] = false$; Remove $p$ from $queue_u[v]$

ENQUEUE$_{u,v}$ (*output action to enqueue estimate to neighbor $v$*)
Preconditions: $queue_u[v]$ is empty
Effects: Add $(Announce, r_u, d_u)$ to $queue_u[v]$

RECEIVE$_{v,u}(Announce, r, d)$ (*input action to receive estimate from neighbor $v$*)
Effects:
If $(r, d) \leq (r_u[v], d_u[v])$ then (*received estimate is no greater than previous estimate*)
If $(d < n')$ (*received estimate not at max value *)
$(r_u[v], d_u[v]) := (r, d)$ (*update estimate from $v$*)
$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

REPORT$_u(q)$ (*output action to report parent estimate*)
Preconditions: $q = parent_u$

Figure 8.6: Spanning tree protocol. Code for a node $u$.

**Definition 8.3.1** *For any edge $(u,v)$, we let $L_{u,v}$ be the intersection of $O_{u,v}$ and $O_{v,u}$. Let $\mathcal{L}$ be the link predicate set containing $L_{u,v}$ for every $(u,v)$ in $G$ and let $L = Conj(\mathcal{L})$.*

It is easy to show that each $L_{u,v}$ is a closed local predicate.

**Lemma 8.3.2** *For a leader edge $(u,v)$ and transition $(s,a,s')$ of $\mathcal{R}$, if $s$ satisfies $L_{u,v}$, then $s'$ satisfies $L_{u,v}$.*

**Proof:** First, $O1(u,v)$ is clearly closed because the code never changes the value of $r_u[u]$ and $d_u[u]$. Also, if the code changes $(r_u, d_u, parent_u)$, it always sets this 3-tuple equal to the minimum of $(r_u[v], d_u[v], parent_u[v])$ for all $v$ in $neighborSet_u$ as required by $O1(u,v)$.

Next, $O2(u,v)$ is closed because of a simple observation: *the $(r,d)$ estimates sent in $(Announce, r, d)$ packets by node $u$ are always non-increasing.* This follows because node $u$ changes its own estimate after receiving an *Announce* packet from some neighbor $w$. But (see code), $u$ will not accept the packet from $w$ unless the new estimate from $w$ is no greater than the previous value that $u$ has stored from $w$ and if the distance from $w$ is not already at the maximum value $n'$. But if the new estimate from $w$ is no greater than the previous value that $u$ has stored from $w$, then the new estimate that $u$ calculates will be no greater than before. But if $O2(u,v)$ is true initially and all estimates sent by $u$ are non-increasing, then it is easy to see that $O2(u,v)$ remains true. Note that if $u$ does not accept a packet from neighbor $v$ (e.g., because the distance estimate in the packet is equal to $n'$) this does not affect $O2(v,u)$

Finally, since the above argument can be applied symmetrically for the link $(v,u)$, it follows that $L_{u,v}$ is closed. ∎

Note that the check in the code that prevents $v$ from accepting an estimate larger than a previously stored estimate is really an application of the heuristic of removing unexpected packet transitions (see Chapter 6). Clearly if $u$ receives an estimate from $w$ larger than the previous estimate stored from $w$, then $O2(w,u)$ is not true in the state before the estimate was received. Thus this is an unexpected packet transition; by removing such a transition, we ensure that $L_{u,v}$ is a closed predicate.

The following theorem is immediate from the last lemma.

**Theorem 8.3.3** *The network automaton $\mathcal{T}$ can be locally checked for $L$ using $\mathcal{L}$.*

O1(u,v):

$r_u[u] = u$

$d_u[u] = -1$

$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

O2(u,v):

If there is an $(Announce, r, d)$ packet in transit from $u$ to $v$ then

$(r_v[u], d_v[u]) \geq (r, d) \geq (r_u, d_u)$

Else (*there is no Announce packet in transit*)

$(r_v[u], d_v[u]) \geq (r_u, d_u)$

Figure 8.7: Spanning Tree Protocol: Local Predicates for edge $(u, v)$.

### 8.3.4 Fast Computation of Spanning Tree after all Link Predicates Hold

We now show that $\mathcal{T}|L$ stabilizes to the stable tree behaviors in time proportional to the diameter of $G$. Recall that $\mathcal{T}|L$ is the spanning tree automaton described in Figure 8.6 such that all local predicates hold in the initial state.

The basic idea is to show first that if all local predicates hold then there can be no ghost roots; next, we show that if the spanning tree protocol begins in a state that does not contain a ghost root, then the protocol quickly converges to a stable spanning tree.

We assume that each node has a unique ID that is modelled by the name of the node in the topology graph. Thus the unique ID of node $u$ is $u$. Our spanning tree protocol depends heavily on the fact that each node has a unique ID that cannot be corrupted. Thus we are assuming that the node ID is part of the code at a node.

We first define what it means to have a ghost root in a state of $\mathcal{T}$. Informally, consider the set of all the IDs present (in all the root estimates at nodes and in all the *Announce* packets in the links) in state $s$. If the minimal ID of all nodes in the network is not the the minimal ID in this set, then we have a ghost root. Formally:

**Definition 8.3.4** *Consider a state $s$ of $\mathcal{T}$. We say that $s$ has a ghost root $r$ if $r < u$ for all nodes $u$ in $G$ and either:*

- *There is an $(Announce, r, *)$ packet in transit from $u$ to $v$ in $s$ for some $u, v$, OR*
- *$r_u[v] = r$ in $s$ for some $u, v$, OR*
- *$r_u = r$ in $s$ for some $u$.*

The only way we can have ghost roots in any execution is because of bad data in links and nodes when the protocol starts up.

**Lemma 8.3.5** *Consider any execution $\alpha$ of $\mathcal{T}$. Suppose there is a state $s_i$ in $\alpha$ such that there is no ghost root in $s_i$. Then then there is no ghost root in the state following $s_i$ in $\alpha$.*

**Proof:** From the code it is easy to see that the only way a new value is added to set of existing roots is if a node $u$ adds its own ID to the set (e.g., by changing its root estimate to the default). Thus ghost roots cannot be created after state $s_i$. ∎

Next, we show that any state in which all local predicates hold cannot contain a ghost root.

**Lemma 8.3.6** *There is no ghost root in any state of $\mathcal{T}|L$.*

**Proof:** Suppose we did have a ghost root in some state $s$ of $\mathcal{T}|L$. Then there must be a ghost root $r$ with the lowest ID among ghost roots in $s$. If it is in an *Announce* packet in transit from say $v$ to $u$, then we know from $O2(v, u)$ that $r_v \leq r$ and hence $r_v = r$. On the other hand, if there is some $w$ such that $r_w[x] = r$ or $r_w = r$, then we know from $O1(w, *)$ that $r_w \leq r$ and hence $r_w = r$. Thus, we have some node, say $w$, such that $r_w = r$.

By $O1(u, v)$ if $parent_w \neq w$ we can continue inductively following parent pointers. Each time we move from say $w$ to $x = parent_w$, we know from $O1(w, *)$ that $(r_w, d_w) = (r_w[x], d_w[x] + 1)$. But since $r_w = r$, $(r_w[x], d_w[x]) < (r, d_w)$. But we know from $O2(x, w)$ that that $(r_x, d_x) \leq (r_w[x], d_w[x])$. Thus we conclude that $(r_x, d_x) < (r, d_w)$.

But the distance estimates are non-negative and the root estimates in this chain are all equal to $r$ since $r$ is the lowest ID ghost root. Hence the process of following *parent* pointers must terminate at some node say $z$ such that $parent_z = z$ and $r_z = r$. Thus by $O1$ applied to $z$, we know that $r = z$. But that contradicts the fact that $r$ is a ghost root. ∎

The last important fact to observe is that if there are no ghost roots and all local predicates hold, then the protocol converges in time proportional to the diameter of the network. As usual, we will say that $t = O(d)$ to mean $t \leq cd$ where $c$ is a constant. We denote the diameter of $G$ by $D$.

**Lemma 8.3.7** *Consider an execution $\alpha$ of $\mathcal{T}|L$. Then there is some $t$-suffix of execution $\alpha$ (where $t = O(D)$) whose behavior is a stable tree behavior for graph $G$.*

**Proof:** We know from from Lemma 8.3.6 that there is no ghost root in the initial state of $\alpha$ and hence by Lemma 8.3.5 there is no ghost root in any state of $\alpha$. From the fact that $O2(u,v)$ is closed, we know that $O2(u,v)$ holds for all links $(u,v)$ in all states of $\alpha$. Let $q$ be the node with the smallest ID in graph $G$. We prove the lemma using induction on the distance $d$ of a node $u$ from $q$.

**Inductive Hypothesis:** There exists some constant $c$ such that within $c \cdot d$ time after the start of $\alpha$, there is some state in which $(r_u, d_u, parent_u) = (q, d, v)$, where $v$ is some neighbor of $u$ at distance $d-1$ from $q$. Also $u$ will not change $r_u$,$d_u$ or $parent_u$ from this state onwards in $\alpha$.

The inductive hypothesis implies the lemma because it implies that within time proportional to the diameter, every node has a parent pointer that points to a node that is one hop closer to the root $q$. It also implies that the parent pointers do not change after this point. Also, since for each $u$, all $\text{REPORT}_u(*)$ actions are in a separate class, such a $\text{REPORT}_u(*)$ action must occur in constant time in any suffix of an execution.

**Basis**, $d = 0$: Node $q$ is the only node at distance 0 from itself. In all states $s_i$ of $\alpha$, $(r_q, d_q, p_q) = (q, 0, q)$, or there would be a ghost root in state $s_i$ which we have already ruled out.

**Inductive Step:** Suppose it is true for all nodes at a distance $d$ from $s$ and we wish to show that it is true for a node $v$ at a distance $d+1$ from $s$. Thus there is some state $s_i$ such that all nodes $u$ within distance $d$ from $q$ have $(r_u, d_u, parent_u) = (q, d, *)$.

Also, $s_i$ occurs within $c \cdot d$ time after the start of $\alpha$. We first claim that in all states of $\alpha$, $(r_v, d_v) \geq (q, d+1)$. Suppose this were not true in some state $s_k$ of $\alpha$. Then by following parent pointers as we did in the proof of Lemma 8.3.6 and by using predicates O1 and O2 iteratively, we can show that we must end in a ghost root. Since we have ruled out ghost roots, $(r_v, d_v) \geq (q, d+1)$ in all states of $\alpha$.

Next, $v$ must have some neighbor at a distance $d$ from $q$. Let $u$ be the neighbor with the lowest ID among the neighbors of $v$ at distance $d$ from $q$. Thus by the properties of a UDL, within constant time after state $s_i$, an $(Announce, q, d)$ packet from $u$ will arrive at $v$ which will cause $(r_v, d_v) = (q, d+1)$. (Note that this packet will be accepted at $v$ because in the previous state $(r_v, d_v) \geq (q, d+1)$ as we have just shown and $d < n'$.) Also since $u$ is the lowest ID neighbor at distance $d$ from $q$, $v$ will set $parent_u = u$ and this will remain unchanged for the rest of the execution. ∎

**Theorem 8.3.8** *$\mathcal{T}|L$ stabilizes to the stable tree behaviors for graph $G$ in $O(D)$ time, where $D$ is the diameter of $G$.*

**Proof:** From Lemma 8.3.7 ∎

At this stage, we could apply the Global Correction Theorem to obtain a stabilizing version of $\mathcal{T}$. However, there is an even simpler transformation because $\mathcal{T}$ is *one-way checkable*. Before we present the final version of the stabilizing spanning tree protocol, we first define what we mean by a one-way checkable protocol.

## 8.4 One Way Predicates and Local Checking

In Section 8.1, we claimed that every locally checkable protocol could be stabilized using the global reset protocol of Chapter 7. The proof sketch suggested that this could be achieved by periodically doing a local snapshot of each local subsystem and making a reset request if a violation is detected.

However, in this section we will show that if a locally checkable protocol $P$ is also what we call *one-way checkable*, then we can locally check $P$ using a simpler and faster method than doing a local snapshot. In this method, each node periodically sends its entire state to each of its neighbors. We can call this *local checking by periodic sending of state* or simply periodic sending. The question that remains is: when is periodic

sending adequate to detect local violations in lieu of a local snapshot? The answer is that periodic checking is sufficient if each local predicate is what we call a *separable* local predicate.

Intuitively, a separable local predicate can be separated into two one-way predicates, one for each direction of a link subsystem. Intuitively a *one-way predicate* for link $(u,v)$ is a predicate that only involves the state of $u$, the state of the link $(u,v)$, and the state of node $v$. *Note that it does not depend on the the state of the link* $(v,u)$. Formally:

**Definition 8.4.1** *Consider any network automaton $\mathcal{N}$ with graph $G$. Let $(u,v)$ be any edge in $G$. We say that $O_{u,v}$ is a one-way predicate for edge $(u,v)$ of $\mathcal{N}$ if:*

- *$O_{u,v}$ is a local predicate of $\mathcal{N}$ for edge $(u,v)$.*
- *For any two states $s$ and $s'$, suppose $s$ satisfies $O_{u,v}$ and $s|u = s'|u$ and $s|v = s'|v$ and $s|(u,v) = s'|(u,v)$. Then $s'$ satisfies $O_{u,v}$.*

For example, consider the second predicate of the spanning tree protocol listed in 8.7. Recall that it essentially says that the estimate values in transit from $v$ to $u$ are always non-increasing. Clearly this is a one-way predicate. It is not hard to see that the first predicate (which involves only the state of one node) is also a one-way predicate.

**Definition 8.4.2** *Consider any network automaton $\mathcal{N}$ with graph $G$. Let $(u,v)$ be any edge in $G$. We say that $L_{u,v}$ is a separable local predicate for edge $(u,v)$ of of $\mathcal{N}$ if there exist two one-way predicates $O_{u,v}$ and $O_{v,u}$ (one for edge $(u,v)$ and edge $(v,u)$ respectively) such that:*

*Any state $s$ of $\mathcal{N}$ satisfies $L_{u,v}$ iff $s$ satisfies both $O_{u,v}$ and $O_{v,u}$*

Once again it is easy to see that the spanning tree protocol has a link predicate set that consists of separable predicates. In this case, we say that the spanning tree protocol is *one-way checkable.*

**Definition 8.4.3** *A network automaton $\mathcal{N}$ is one-way checkable for predicate $L$ using link predicate set $\mathcal{L}$ if:*

- *$\mathcal{N}$ is locally checkable for predicate $L$ using link predicate set $\mathcal{L}$.*
- *Each $L_{u,v} \in \mathcal{L}$ is a separable local predicate.*

We now claim the following fact informally. *Any one-way checkable automaton can be locally checked by periodic sending.* We simply add actions to send the state of a node periodically to each of its neighbors and actions to receive such packets.

Suppose that in state $s$, node $u$ receives such a periodic packet from node $v$ containing the value $x$. Then $u$ checks whether $(x, nil, nil, s|u) \in O_{v,u}$; if this is not true $u$ detects a local violation.

Notice that local checking by periodic sending is not just simpler than doing a local snapshot but is also *faster*. The proof of Lemma 5.6.12 in Chapter 5 tells us that it takes 6 checking/correction phases for the local snapshot protocol to stabilize. Roughly, this means that the local snapshot protocol takes 12 link delays (where a link delay is the time to send a control packet from a node to its neighbor) to stabilize. By contrast, the periodic sending protocol takes 1 link delay to stabilize. Thus *for one-way checkable protocols it always pays to use periodic sending for local checking.*

## 8.5 Simple Stabilizing Spanning Tree Protocol

We have shown that the spanning tree protocol described in Figure 8.6 is locally checkable. Thus we can apply the Global Correction theorem to this protocol. This requires checking link predicates using local snapshots. However, since the spanning tree protocol described in Figure 8.6 is also one-way checkable, we can can replace the local snapshot by periodic sending of state information. The resulting protocol is simpler and faster than a spanning tree protocol that uses local snapshots.

If we look at the protocol in Figure 8.6, we see that each node sends $(Announce, r, d)$ packets periodically containing the root and distance estimates at the node. But if we look at the predicates in Figure 8.7, $O1(u, v)$ involves only variables at $u$, and $O2(u, v)$ only involves root and distance estimates at $v$. *Thus we can rely on the* $(Announce, r, d)$ *message for periodic checking.* This is shown in the code of Figure 8.8 which only describes the changes to the code of Figure 8.6 to convert it into a stabilizing protocol.

The first change is that we have applied message renaming to the original protocol and replaced all packet events with message events.

The rule for one-way checking is as follows. If in state $s$, node $u$ receives a periodic message from node $v$ containing the value $x$, then $u$ checks whether $(x, nil, nil, s|u) \in O_{v,u}$; if this is not true $u$ detects a local violation. In our case, the $(Announce, r, d)$ message does not contain the complete state $x$ of node $v$, but only a projection of the state of node $v$ that is sufficient to do checking. But $(x, nil, nil, s|u) \in O2(v, u)$, iff $(r, d) \leq (r_u[v], d_u[v])$. Thus to check whether $O2(v, u)$ holds it is sufficient to check whether $(r, d) \leq (r_u[v], d_u[v])$ and make a reset request if a violation is detected. Also notice that we do not need to check for the $O1(v, u)$ predicate because we have added a few lines of code that ensure that $O1(v, u)$ will hold after the first *Announce* message received by $v$.

Let $T'_u$ be the modified node automaton shown in Figure 8.8. Let $\mathcal{R}|L$ be the reset automaton for graph $G$ as described in Chapter 7. Let $\mathcal{T}'$ be the automaton formed by composing $T'_u$ for each node $u$ with $\mathcal{R}|L$.

Then we have the following theorem:

**Theorem 8.5.1** *$\mathcal{T}'$ stabilizes to the stable tree behaviors for graph $G$ in $O(n)$ time.*

**Proof:** (Sketch) We first show that $\mathcal{T}'$ stabilizes to the behaviors of $R(\mathcal{T}(c)|L)$ in $O(n)$ time. The operator $R$ denotes message-renaming, as in the Global Correction Theorem. This part of the proof uses arguments similar to the ones used in the proof of the Global Correction Theorem except that the arguments are simpler because of the use of one-way predicates and one-way checking.

Next, we know from Theorem 8.3.8 that $\mathcal{T}|L$ stabilizes to the stable tree behaviors of graph $G$ in $O(D)$ time. $D$ is the diameter of $G$ and is no greater than $n$, the number of nodes. But if $\mathcal{T}|L$ stabilizes to the behaviors in $P$ in $O(D)$ time, then so does $R(\mathcal{T}(c)|L)$. This is because the set of stable tree behaviors is invariant under the operations of message-renaming and scaling by constant factors. Finally, since behavior stabilization is transitive, we conclude that $R(\mathcal{T}(c)|L)$ stabilizes to the behaviors in $P$ in $O(D + n) = O(n)$ time. ∎

Finally, we note that because $T'_u$ is a UIOA and $\mathcal{R}|L$ is a CIOA the modularity theorem assures us that we can replace $\mathcal{R}|L$ by its stabilizing implementation ($\mathcal{R}^+$ as

described in Chapter 7) without affecting the result. This is an elegant application of the modularity theorem because it means that we can begin with a stabilizing version of a UDL; next, as shown in Chapter 7 we can use the stabilizing UDL implementations to construct a stabilizing version of a reset protocol; and finally we can use the stabilizing version of the reset protocol to construct a stabilizing spanning tree protocol. In the process, we have two applications of the modularity theorem. We can go even further and change $\mathcal{T}'$ so that it offers a suitable interface to the token passing example of Chapter 6. If we do so, we could construct a stabilizing token passing protocol using the stabilizing spanning tree, once again relying on the modularity theorem.

In any of these modular constructions, we can replace major pieces of the construction by other modules that offer the same external behavior.

## 8.6 Stabilizing topology update protocol

The topology update task is to broadcast the list of incident operating edges at a node to all other network nodes. Thus the goal of the topology update task is to produce at each node $u$ a database listing the operating edges of each node that is reachable from $u$.

The following simple strategy is often used for topology maintenance. Each node maintains a sequence number. Whenever a change occurs, the incident nodes increment their sequence number and broadcast the new status of all their incident links in a link state packet ([Per83]). Any link state packet sent by node $u$ contains node $u$'s sequence number. Whenever a node $v$ receives a message purporting to be from $u$, $v$ checks whether the sequence number on the message is larger (i.e., newer) than the sequence number of the last update $v$ has stored from $u$. If so, $v$ stores this new update from $u$ (after discarding any previous update it has stored from $u$) and broadcasts the update to all neighbors. Otherwise, the message is simply discarded as an outdated update.

Now the sequence number field is finite. Even if we allocate 64 bits for sequence numbers, it is always possible (due to errors) for a link state packet with a maximum sequence number value to be present in the initial state of the network.

In the ARPANET [MRR80] and DECNET [Per83] protocols, erroneous updates with the maximum number are removed by what we had earlier called "timer flushing".

State

$requestbit_u[v]$ boolean, true if a request is to be done later.
All other variables remain unchanged.

Modified Actions

$\text{FREEM}_{u,v}$ (*message-renamed free action, replaces $\text{FREE}_{u,v}$ action*)
Effects: $free_u[v] = true$

$\text{SENDM}_{u,v}(p)$ (*message-renamed output action to send estimate to neighbor $v$*)
Preconditions: $p$ is head of $queue_u[v]$ and $free_u[v] = true$
Effects: $free_u[v] = false$; Remove $p$ from $queue_u[v]$

$\text{RECEIVEM}_{v,u}(Announce, r, d)$ (*input action to receive estimate from neighbor $v$*)
Effects:
If $(r, d) \leq (r_u[v], d_u[v])$ then (*received estimate is no greater than previous estimate*)
If $d < n'$ (*distance estimate not already at max value*)
$(r_u[v], d_u[v]) := (r, d)$ (*update estimate from $v$*)
Else $requestbit_u = true$ (*if received estimate is greater make reset request later*)
$(r_u[u], d_u[u]) := (u, -1)$; (*next two lines make $O1(u, v)$ hold*)
$(r_u, d_u, parent_u) := min\{(r_u[v], d_u[v] + 1, v), v \in neighborSet_u\}$

$\text{REQUEST}_u$ (*output action to request a reset*)
Preconditions: $requestbit_u = true$
Effects: $requestbit_u = false$

$\text{SIGNAL}_u$ (*input action to receive a reset signal*)
Effects:
For all $v \neq u \in neighborSet_u$ do
$(r_u[v], d_u[v]) := (\infty, \infty)$ (*reset all estimates*)
$free_u[v] := false$
$(r_u[u], d_u[u]) := (u, -1)$;
$(r_u, d_u, parent_u) := (u, 0, u)$

Figure 8.8: Spanning tree protocol. Modifications to the Code for a node $u$.

Each update (and hence the associated counter value) is assumed to remain in the network only for some limited "lifetime," after which it is discarded. This prevents the problem because after its lifetime expires, an erroneous counter value is no longer present in the network. Once again, the disadvantage of timer flushing is that the timeout periods have to be high, which results in high stabilization times.

Instead of using timer flushing, we can use global correction to create a faster stabilizing topology update protocol. This is because the protocol is easily seen to be locally checkable for the property $L$ which states that the maximum value of any sequence number is not present in the network. Thus we can apply the Global Correction theorem to ensure that maximum size sequence numbers are removed from the network. On receiving a signal, each node deletes the link state packet of all nodes other than itself, and resets its own sequence number to 0. A node that does not have a link state packet from $u$ will accept any update sent by $u$ as being newer.

Thus, the number of bits allocated for the sequence number affects only the performance of the algorithm: errors that cause the sequence numbers to reach the maximum value incur a performance penalty — a network latency of $O(n)$.

A complete design of a stabilizing topology update protocol would also have to add a number of other actions as suggested in [Per83]. For instance, each node must periodically increment its sequence number and send its latest Link State Packets to all its neighbors. Basically, we suggest keeping the essential simplicity of [Per83] and only replace the need for global timers in [Per83] with global correction.

## 8.7 Summary

The major point of this chapter is to show that any locally checkable protocol can be stabilized using the global reset protocol developed in the last chapter. In general, this is done by having the leader of every every link subsystem do a periodic snapshot; if the leader detects a local violation, it makes a reset request. However, in the special case that the local predicates can be separated into one-way predicates, we can dispense with a snapshot and use periodic sending of state. Local Checking by periodic sending is simpler and faster than using a local snapshot.

An important example of both techniques is furnished by the spanning tree protocol described in Section 8.3. Both the spanning tree and topology update protocols

described are quite practical and could be used in real networks.

The topology update protocol demonstrates a useful paradigm for designing stabilizing protocols. The essence of the idea is to design a stabilizing protocol assuming the use of unbounded counters. Next, we use global correction to convert this protocol into a more practical protocol that uses bounded counters.

This chapter also demonstrates why the specification of a reset protocol must specify the behavior in non-final signal intervals and not just in the final signal intervals. The proof of the Global Correction relies strongly on the mating relation between non-final intervals. The intuition is that the augmented automaton is doing local checking of the system during non-final intervals; if local checking does not eventually observe consistent behavior, the augmented automaton may keep making reset requests and the protocol may never stabilize.

# Chapter 9

# Compiling Synchronous Protocols

In the previous chapters we have shown how we can stabilize certain protocols by local checking and/or correction of the state of a node and its neighbors. We applied this technique to obtain stabilizing solutions to some important tasks such as mutual exclusion, network resets, and computing spanning trees. By contrast, this chapter provides a general technique for a special class of problems (known as *non-interactive* tasks), for many of which (e.g., Minimal Spanning Tree, Min Cost Flows, etc.) *no locally-checkable implementation is known.* In fact, for many of the problems solved in this chapter, no efficient stabilizing solution was known previously.

In this chapter, we describe two compilers (Sections 9.5 and 9.6) that convert a deterministic synchronous protocol $\pi$ into a stabilizing, asynchronous version of $\pi$. In essence, we efficiently transform a solution for the most restrictive model (synchronous, fault-free networks) to a solution that works in a very permissive model (asynchronous networks with catastrophic faults that stop).

Let $T_\pi$ be the time complexity of $\pi$ and $S_\pi$ be the space complexity of $\pi$. Let $n$ be the number of nodes in the network. The first compiler produces a version of $\pi$ that stabilizes in time $O(n + T_\pi)$ and has the same space complexity as $\pi$. Thus the first compiler is extremely efficient if the time complexity of the original protocol $\pi$ is at least $O(n)$. The second compiler produces a version of $\pi$ that stabilizes in time $O(T_\pi)$ and has a space complexity of $T_\pi \cdot S_\pi$. Thus the second compiler is extremely efficient if the time complexity of the original protocol $\pi$ is very small – i.e., $T_\pi = O(log(n))$.

These two compilers allow us to provide efficient stabilizing solutions for many

problems including minimum spanning trees, colorings, maximum flows, and maximal independent sets.

Despite the apparent change of direction, the ideas in this chapter are closely related to the ideas in the previous chapters. First, as we have said earlier, the ideas in this chapter extend the range of our general techniques. Our previous general techniques (Local, Tree and Global Correction) only apply to protocols that are locally checkable. Second, the compilers in this chapter are implemented using the the techniques developed earlier. The first compiler is based on a simplified form of local checking and correction that we call one-way checking and correction (see Chapter 8). The second compiler is based on local checking and global correction.

When we first presented these results in [AV91], the second compiler, the *Resynchronizer*, was based on a complicated construction. In this chapter, we provide a simplified construction using the reset protocol described in Chapter 7. We do not have a complete proof of the simplified construction, but we will outline why we believe our simplified construction is correct. Thus while our confidence in the *Resynchronizer* result is based on the original result in [AV91], we believe that the construction in this chapter offers the potential for a much simpler solution.

This chapter is organized as follows. First, we describe how we model interactive protocols and synchronous protocols. Next, we summarize the major results of the chapter. Then we contrast the notion of *distributed checking* (that we use in this Chapter) with the independent local checking that we have used in previous chapter. Next we briefly describe the first *Rollback* compiler and then describe the simplified version of the *Resynchronizer* compiler. Finally, we outline extensions for randomized protocols.

## 9.1 Non-interactive Protocols

Non-interactive protocols form an important subclass of distributed algorithms. These are protocols whose correctness can specified by a relation (the I/O relation) between its input and output. For example, if the protocol has to compute a spanning tree then the output (the tree) should be a spanning tree of the input (the graph $G$). By contrast, mutual exclusion is an interactive task because (see Chapter 6) its correctness must be expressed in terms of sets of valid behaviors or sets of valid executions.

We define the notion of a non-interactive protocol more formally using a slight variant of the network model introduced in Chapter 5. Recall that in Chapter 5, we modelled a network using a topology graph $G = (V, E, l)$, where $V$ is a set of nodes, $E$ is a set of directed edges and $l$ is a leader function. For this chapter we will augment the notion of a topology graph to have an *extra component* $I$ that represents the input. We will define an *augmented topology graph* $G = (V, E, l, I)$ where $V$,$E$ and $l$ are as before. $I$, however, is a vector of inputs such that for any node $u$, $I_u$ represents the *local input* at node $u$. (In previous chapters the only input to a node automaton for node $u$ was the list of neighbors of $u$ and, in the case of a tree topology, the parent of $u$.)

We start by fixing an input domain $\mathcal{I}$ and an output domain $\mathcal{O}$.

**Definition 9.1.1** *An augmented topology graph $G = (V, E, l, I)$ is a topology graph $(V, E, l)$ together with an input specifier $I$, where $I$ is vector of local inputs $(I_u)$ and $I_u \in \mathcal{I}, \forall u \in V$.*

Informally, this amounts to modelling the input to the non-interactive protocol as part of the local code of each node (which cannot be corrupted) and not as part of the state of each node (which can be corrupted). Clearly no non-interactive protocol can hope to produce correct output from corrupted input! However, more generally, the input to the protocol could be the output of another stabilizing protocol. Thus in Chapter 5, we assumed the list of neighbors of each node $u$ was part of the code at node $u$. But in many real networks, the list of neighbors of a node would be provided as the output of a stabilizing Data Link protocol. In this chapter, we will often refer to an augmented topology graph as an augmented graph.

Once again, we will model the protocol using node automata $N_u$ (one for each node $u$ in $G$), and unit capacity data links $C_{u,v}$ (one for each edge $(u, v)$ in $G$).

**Definition 9.1.2** *An automaton for augmented graph $G = (V, E, l, I)$ is an automaton consisting of an IOA for each $u \in V$ together with a UDL $C_{u,v}$ for each $(u, v) \in E$.*

Refer to Chapter 5 for a definition of the UDL $C_{u,v}$. Recall that for any state $s$ of the automaton we let $s|u$ denote the state of the IOA for node $u$. For an augmented

graph, we also let $(X_u)$ denote a vector consisting of an element $X_u$ for each node $u$ in the graph.

For non-interactive protocols we will also assume that there is some *local output function* $O_u$ of the state of each $N_u$, such that $O_u(s)$ provides the *local output* of $N_u$ in state $s$. Let $\mathcal{N}$ be the network automaton formed by the composition of the node and channel automata as described in Chapter 5. Using the $O_u$ we can define the *output function* $O$ to be a function of the state $s$ of the network automaton $\mathcal{N}$ such that $O(s)$ produces a vector of local outputs that is identical to the outputs produced by each $O_u$ when applied to $s|u$. Thus:

**Definition 9.1.3** *Let $\mathcal{N}$ be an automaton for augmented graph $G = (V, E, l, I)$. We say that $O$ is an output function for $\mathcal{N}$ if $O$ is a vector of functions $(O_u)$, such that for every state $s$ of $\mathcal{N}$ and for every node $u \in V$, $O_u(s|u) \in \mathcal{O}$. We will also abuse notation by defining $O(s) = (O_u(s|u)), \forall u \in V$.*

Traditionally, the correctness of a non-interactive protocol $\mathcal{N}$ is described using an input-output relation $R$. In the traditional definitions, $\mathcal{N}$ is allowed to be an ordinary IOA (i.e., we are allowed to specify initial states). Next, the correct executions of N are those executions $\alpha$ of $\mathcal{N}$ in which there is some $t$-suffix $\gamma$ of $\alpha$ such that for all states $s$ in $\gamma$, $(I, O(s)) \in R$. In other words, in all executions of the protocol the output must eventually "settle" to a value that satisfies the I/O relation. Thus:

**Definition 9.1.4** *Let $\mathcal{N}$ be an automaton for augmented graph $G = (V, E, l, I)$. An I/O relation for $\mathcal{N}$ is a set of tuples $(I', O')$ where each $I'$ is a vector $(I'_u), I'_u \in \mathcal{I}$ and each $O'$ is a vector $(O'_u), O'_u \in \mathcal{O}$.*

Finally we define a non-interactive protocol to consist of two components: an automaton and an output function.

**Definition 9.1.5** *A non-interactive protocol for augmented graph $G$ is a tuple $(\mathcal{N}, O)$ where $\mathcal{N}$ is an automaton for augmented graph $G$ and $O$ is an output function for $\mathcal{N}$.*

For stabilization, we will keep the the correctness definition exactly as in the traditional definitions *except that $\mathcal{N}$ will typically be a UIOA.*

**Definition 9.1.6** *Let $\mathcal{P} = (\mathcal{N}, O)$ be a non-interactive protocol for augmented graph $G = (V, E, l, I)$ and let $R$ be an I/O relation for $\mathcal{N}$. Let $C$ be the set of executions $\alpha$ of $\mathcal{N}$ such that for all states $s$ in $\alpha$, $(I, O(s)) \in R$. We say that non-interactive protocol $\mathcal{P}$ stabilizes to I/O relation $R$ in time $t$ if $\mathcal{N}$ stabilizes to the executions in $C$ in time $t$.*

Thus this definition is a special case of the general definition of "stabilizes to the executions of" given in Chapter 3.

We now formally define two complexity measures that we use to evaluate stabilizing non-interactive protocols: the first measures the worst-case time for the protocol to stabilize, and the second measures the worst-case amount of space used by any node.

We define the *stabilization time* of a non-interactive protocol $\mathcal{P}$ with respect to I/O relation $R$ to be the infimum of all $t$ such that $\mathcal{P}$ stabilizes to $R$ in time $t$.

The *space complexity* of $\mathcal{P} = (\mathcal{N}, O)$ is the worst case across all nodes $u$ of the size of the set $\{s|u : s \in states(\mathcal{N})\}$

## 9.2 Synchronous Protocols

We model a deterministic synchronous protocol $\pi$ as follows. The network topology is again specified by an augmented graph $G = (V, E, l, I)$, where $I_u$ once again represents the input to node $u$. The protocol is synchronous because it works in rounds numbered from 0 to $T_\pi$, where $T_\pi$ is some finite, known bound on the running time. The channels are no longer UDL's but simply obey the property that any packet sent on a channel at the start of a round is delivered by the end of the round. Similarly, the node protocol at each node $u$ is no longer modelled by a node automaton but instead by a 4-tuple $(s_u^0, F_u, M_u, O_u)$, where $s_u^0$ is an *initial state*, $F_u$ is a *state transition function*, $M_u$ is a *message generation function*, and $O_u$ is the *output function*. All these correspond to the standard notions for such protocols but we explain them briefly below.

The state $s$ of the synchronous protocol consists of the state $s|u$ of each node in $G$. An *execution* of the synchronous protocol is generated as follows. At the start of the first round, each node $u$ is placed in the initial state $s_u^0$ and all channels are empty. At the start of each round, a node sends a packet to each neighbor. The contents of

the packet are determined by the message generation function $M_u$ that takes as input the state of $u$ at the start of the round. At the end of each round, a node changes its state using the state transition function $F_u$ that takes as input the messages received by $u$ in the round as well as the state at the start of the round.

The output of $\pi$ in any execution is determined by the value of $O_u(s|u)$ for every node $u$, where $s|u$ is the state of node $u$ at the end of round $T_\pi$. Exactly as for non-interactive protocols, we define an output function $O$ to be a function of the state $s$ of $\pi$ such that $O(s)$ produces a vector of local outputs that is identical to the outputs produced by each $O_u$ when applied to $s|u$.

Exactly as for non-interactive protocols, we define the correctness of synchronous protocols using an I/O relation $R$. The synchronous protocol $\pi$ is said to be correct if all runs of $\pi$ end in a state $s$ such that $(I, O(s)) \in R$.

The *time complexity* of a synchronous protocol $\pi$ is the number of rounds, $T_\pi$. The space complexity of $\pi$ is the worst case across all nodes $u$ of the number of states of $u$.

Let $\pi$ be a synchronous protocol with time complexity $T_\pi$ and Input-Output relation $R$. We also introduce the notion of a *checker* for $\pi$. A synchronous protocol $\chi$ is said to be a checker for $\pi$ if it satisfies the following property.

Suppose each node $u$ in $\chi$ is given as input both the input of $\pi$ (i.e., $I$) as well as a value $O'$ that purports to be the output of $\pi$ on input $I$. Informally, $\chi$ must detect (at some node) if the purported output $O'$ of $\pi$ could have been produced in some run of $\pi$ on input $I$. Thus $\chi$ has a boolean output at every node. If any node outputs the value *false*, then it must be that $O'$ could never have been produced by $\pi$; if all all nodes output the value *true*, then it must be that $O'$ could have been produced by $\pi$.

Note that our definition only allows deterministic synchronous protocols whose output is a function of its input. Thus for such protocols $\pi$, $\pi$ has a trivial checker that simply runs $\pi$ again and compares the output $O_u(s)$ at each node with the purported output $O'_u$. The checker outputs *true* at node $u$ iff $O_u(s) = O'_u$.

## 9.3 Results

Recall the definitions of stabilization time and space complexity for a non-interactive protocol $\mathcal{P}$ and the definitions of time and space complexity for a synchronous protocol

| Method | Stab. Time | Space |
|---|---|---|
| *Rollback* | $T_\pi$ | $T_\pi \cdot S_\pi$ |
| *Resynchronizer* | $T_\pi + n$ | $S_\pi$ |

Figure 9.1: Comparison of the complexity of our compilers

$\pi$. Recall that $n$ is the number of nodes in the network.

Our solutions are in the form of compilers that can compile synchronous protocols into stabilizing versions that have the same I/O relation when run on an asynchronous network. The performance of our compilers is summarized in Figure 9.1

Our simplest compiler is the *Rollback* compiler that takes a synchronous protocol $\pi$ as input and produces a stabilizing, non-interactive version of $\pi$. This is expressed as the following theorem:

**Theorem 9.3.1 Rollback Compiler**: *Consider any synchronous protocol $\pi$ for augmented graph $G$ with I/O relation $R$, time complexity $T_\pi$ and space complexity $S_\pi$. Then there is a corresponding non-interactive protocol $\mathcal{P} = (\mathcal{N}, O)$ such that:*

- *$\mathcal{N}$ is a UIOA.*
- *The space complexity of $\mathcal{P}$ is $O(T_\pi \cdot S_\pi)$.*
- *The stabilization time of $\mathcal{P}$ with respect to $R$ is $O(T_\pi)$.*

The main contribution of this chapter is a second compiler called a *Resynchronizer*. In its simplest form, the *Resynchronizer* takes a *deterministic* synchronous protocol $\pi$ as input and produces a stabilizing non-interactive version of $\pi$.

**Theorem 9.3.2 Resynchronizer Compiler**: *Consider any synchronous protocol $\pi$ for augmented graph $G$ with with I/O relation $R$, time complexity $T_\pi$ and space complexity $S_\pi$. Then there is a corresponding non-interactive protocol $\mathcal{P} = (\mathcal{N}, O)$ such that:*

- *$\mathcal{N}$ is a UIOA.*
- *The space complexity of $\mathcal{P}$ is $O(S_\pi)$.*
- *The stabilization time of $\mathcal{P}$ with respect to $R$ is $O(T_\pi + n)$.*

Our original proof of the Resynchronizer Compiler Theorem was based on a complicated construction given in [AV91]. We conjecture that the simplified construction in Section 9.6 also provides a proof of the same theorem.

We will also see in Section 9.7.2 how to use the *Resynchronizer* to compile randomized, synchronous protocols (of course, this would entail generalizing our model of a synchronous protocol to allow coin-tosses at each node.)

Note that, when compared to the *Resynchronizer*, the *Rollback* construction removes the additive factor of $n$ in the stabilization time but increases the space by a factor of $T_\pi$. Thus *Rollback* is useful only for "fast" protocols that have $T_\pi \ll n$. The two compilers can be used to efficiently stabilize several non-interactive tasks.

Some sample results obtained by applying the *Resynchronizer* are as follows. For the problems of computing a spanning tree and single source shortest paths we achieve $O(n)$ stabilization time and $O(\log n)$ space. This is comparable to the results achieved in Chapter 7 and the best previous results. For the problem of computing a minimal spanning tree [GHS83] we achieve $O(n)$ stabilization time (which is as good as the time of the best non-stabilizing synchronous protocol) using $O(\log n)$ space. For the problem of computing a maximum flow [Gol85, GT88] we achieve $O(n^2)$ stabilization time, which is as good as the time of the best non-stabilizing synchronous protocol.

The *Rollback* compiler gives good results when applied to symmetry breaking problems such as the problems of computing a Maximum Independent Set [AGLP89], $\Delta+1$ Coloring in sparse networks [GPS87], and $\Delta^2$ Coloring in general networks [Lin87]. For instance, for $\Delta + 1$ Coloring in sparse networks we achieve $\log^* n$ for both measures. This is much better than any previous results.

## 9.4 Distributed Program Checking

A deterministic sequential algorithm can make itself stabilizing by periodically saving its output and running itself again; when it finishes it can check its output. For

sequential algorithms, this is ugly and unnatural – after all, we want the computer to move on and run other programs!

However, as we have said before, periodic checking is quite natural for distributed computing. Once we have accepted the inevitable periodic cost of stabilization we can ask: *why not run a checking process to check the algorithm periodically?*. If the check reveals a problem, we restart the main algorithm. In the previous chapters we have already seen how we can do this in some cases if each link subsystem independently does local checking. However, that method (which we can call independent checking of link subsystems) is limited to locally checkable protocols.

One approach to check a distributed program introduced by Katz and Perry [KP90] is to collect all information at a single "leader" node, thus reducing distributed checking to centralized checking. However, the space and time complexities of this method are quite large because of the bottleneck at the "leader" node.

In this chapter, we will introduce a form of distributed checking that is neither the independent local checking of the previous chapters nor the centralized checking introduced by Katz and Perry. In many cases, we can improve performance by doing distributed program checking.

Such distributed program checking clearly requires coordination of all the network nodes. For example, we need to ensure that a node not move to a new phase (whether it be checking or executing) before every other node in the network has completed this phase. The main difficulty is to implement this coordination in a stabilizing fashion. We will describe two such implementations – the *Rollback* protocol in Section 9.5 and the *Resynchronizer* in Section 9.6.

## 9.5 Rollback Compiler

There is a naive implementation of distributed checking that requires a large amount of storage and bandwidth and works only for deterministic protocols. In the naive method, every node keeps a log of every state transition it has taken to reach its current state. If each node constantly sends its current log to all neighbors, every node can check and correct every transition it has made in the past. Since the inputs are always correct, eventually all transitions are corrected. This method works only because it is possible to check transitions in a stabilizing fashion.

Clearly, for an arbitrary asynchronous protocol the logs can grow very large. However, if the asynchronous protocol is simulating an underlying synchronous protocol $\pi$ then the size of the log can be reduced to the time complexity $T_\pi$ of $\pi$. This idea is implemented in the *dynamic synchronizer* of [AS88]. However, in [AS88] the logs are only used to avoid unnecessary recomputation after an input change, and hence are not periodically checked. By adding the periodic checking of logs, we obtain a *Modified* Dynamic Synchronizer that we call *Rollback*.

The disadvantage of the *Rollback* method is that it blows up the space utilization and periodic bandwidth by a factor of $T_\pi$. This is not a problem for protocols with small time complexity – i.e., those for which $T_\pi \ll n$. Thus the naive method leads to efficient solutions for such problems as coloring and MIS. However, for protocols in which $T_\pi \geq n$, *Rollback* is a poor choice.

Notice that the *Rollback* compiler combines the use of logs with a simplified form of local checking and correction. The *Rollback* compiler does local checking by periodic sending of state (as opposed to doing a local snapshot). This is because the *Rollback* compiler is one-way checkable (see Chapter 8 for a definition of a one-way checkable protocol). Also, the *Rollback* compiler does local correction by simply correcting the local state of a node when periodic sending detects a problem (as opposed to doing a local reset). If local correction can be performed in this simple manner, we will call the protocol one-way correctable. Chapter 10 discusses one-way correctability in a little more detail.

## 9.6 Simplified Resynchronizer Compiler

In the previous section, we saw how *Rollback* did distributed checking by maintaining a log of its computation. Consider a deterministic protocol $\pi$. Clearly, we can avoid the need for a log if we could simply re-execute $\pi$. However, this constant re-execution requires coordination among the network nodes which must be implemented in a stabilizing fashion. In general, the *Resynchronizer* avoids a log by constantly re-executing a checker $\chi$ for $\pi$. We introduce the basic idea by assuming that $\pi$ is deterministic and that $\pi$ is its own checker. We return to separate checking later.

The *Resynchronizer* can be thought of as a stabilizing version of a *synchronizer* [Awe85]. Any synchronous protocol can be simulated asynchronously by using a pulse

number at each network node. Let us call a node *synchronized* if its pulse number differs by at most 1 from any of its neighbors. In ordinary synchronizers, every node is initially synchronized by setting the pulse numbers of all nodes to 0. Thereafter, a node increments its pulse number from $p$ to $p+1$ only after all its neighbors have reached pulse $p$, thereby maintaining synchronization. If each node executes the corresponding step of the synchronous protocol at pulse $p$ just before incrementing to $p+1$, then the asynchronous protocol has the same I/O relation as the underlying synchronous protocol.

Since a stabilizing synchronizer cannot rely on correct initialization, we introduce a *Termination_Detection* phase. This phase will two do things. First, this phase ensures that each node has finished executing the synchronous protocol. Second, once each node has finished executing the synchronous protocol, a reset (see Chapter 7) is initiated. Once all nodes get a signal (see Chapter 7) all nodes reset their pulse numbers to 0 and the cycle continues.

Thus the *Resynchronizer* can be considered to be an application of global correction to the simplest synchronizer protocol described in [Awe85]. Our original construction and proof [AV91] relied on a special-purpose reset protocol that was specially crafted to work with the synchronizer protocol. In this chapter, we will describe a simplified version of the construction that uses the general purpose network reset protocol of Chapter 7. While we do not have a complete proof of the simplified construction, we believe the construction in this chapter provides the basis for a simpler and more elegant solution. *In what follows, we only describe the simplified construction.*

When $pulse_u \in [0, T_\pi]$ we say that node $u$ is in the *Execute* phase. In the *Execute* phase node $u$ simply executes the normal synchronizer algorithm described earlier. It also implements the underlying synchronous protocol, starting from initialization at pulse 0 followed by writing its output at pulse $T_\pi$. Let $D$ be an upper bound on the diameter of the network.[1] We assume that $D \leq n$.

We denote by $Max = T_\pi + D$ the maximum value of $pulse_u$. When $pulse_u \in [T_\pi + 1, Max]$ we say that node $u$ is in the *Termination_Detection* phase. Suppose that all nodes are synchronized when some node exits the *Execute* phase. Then the *Termination_Detection* phase ensures that *every* node has had a chance to correct its

[1] Unfortunately our protocol needs an upper bound on the diameter of the network. This is a liability of the protocol.

output before the pulse number of some node wraps around to 0. If the nodes are synchronized by the start of this phase, a node need only wait $D$ dummy pulses to make sure that all other nodes have reached the *Termination_Detection* phase.

If $pulse_u$ reaches $Max$, node $u$ makes a reset request and waits to get a signal. This potentially destroys synchronization. However, we can rely on the properties of the reset protocol to deliver a signal at all nodes in a consistent way. When node $u$ receives a signal it sets its pulse number to 0 and the cycle continues. Notice that all nodes constantly reexecute the underlying synchronous protocol.

The *Resynch* compiler must deal with arbitrary pulse numbers and arbitrary messages on links in the initial state. Second, it must cope with the fact that the pulse numbers are finite and hence have to wrap around. Recall that one of our motivations for studying stabilization was to make real network protocols more fault-tolerant. Any real counter implementation is bounded.

In our description and in the code below we only describe the *Resynchronizer* as a tool that can be used to compile a deterministic protocol by re-executing it. It is easy to extend these ideas slightly to add a separate checking phase to the *Resynchronizer*.

### 9.6.1 Resynchronizer Code

We described how to reduce the problem of stabilizing the output of a synchronous protocol $\pi$ to the problem of building a stabilizing synchronizer that constantly re-executes pulse numbers in the execute region. Thus when presenting the code, for simplicity we ignore the details of executing $\pi$; instead we concentrate on the major task of synchronizing pulse numbers. To actually execute $\pi$, we need to supplement the code as follows:

- Additional state variables are added at every node $u$ to keep track of the state of $\pi$. Also, we assume that every pulse message with pulse $p$ carries the state of $\pi$ (at the sending node) on pulse number $p$ and $p-1$.
- Whenever $pulse_u$ reaches $p$ and $p$ is in the *Execute* phase, the synchronous protocol $\pi$ is executed at node $u$. (Sometimes the code will cause the pulse number at node $u$ to jump from say $p$ to $p+2$. In that case, the synchronous protocol must be executed at pulse $p+1$ and pulse $p+2$.)

- Whenever $pulse_u$ reaches $T_\pi$, node $u$ corrects its output to be the local output of $\pi$.

The protocol is formally presented below in Figures 9.2 and 9.3. Every node $u$ keeps the variables described in Figure 9.2.

We assume that each node is a user of the reset protocol $\mathcal{R}|L$ described in Chapter 7. As in Chapter 7, we assume that $\mathcal{R}|L$ has an interface to make reset requests and accept reset signals as shown in Figure 7.2 in Chapter 7. Recall also that the reset subsystem also has an output action FREEM that tells the user (in this case the spanning tree protocol) when it can send a new user message. Of course, the similarity to the FREE action of a UDL is no coincidence. Just as the user of a UDL needs to keep a *free* variable to record whether the UDL is free, each node (exactly as in the spanning tree protocol of Chapter 8) will keep a variable *free* for each neighbor. In fact, the entire protocol bears a strong resemblance to the spanning tree protocol of Chapter 8 although it performs an entirely different function.

Every node $u$ periodically sends its own pulse number to all neighbors. using a "Pulse" message. The Pulse message is encoded as a tuple $(Pulse, p)$. When a node $v$ receives an Pulse message from $u$, $v$ compares the pulse in the message (say $p$) to the previous pulse estimate stored from $u$ ($pulse_v[u]$). During normal synchronizer operation the following local predicate always holds: $pulse_v[u] \leq p$ and $p \leq pulse_v + 1$. (Intuitively, the pulse estimates sent by $u$ to $v$ are non-decreasing and are never more than one higher than the current pulse number of $v$) If $v$ realizes that the local predicate has been violated, $v$ makes a reset request. If this is not the case, $v$ stores the latest estimate it has received from $u$; $v$ also updates its own estimate as the minimum of all neighbor estimates.

Once node $u$ finds that $Pulse_u = Max$, it makes a reset request. Once node $u$ gets a signal, node $u$ resumes normal synchronizer operation.

We assume that the topology of the network is specified by an augmented graph $G$. Let $\mathcal{R}|L$ be the reset protocol described in Chapter 7. Let $SS_u$ be the automaton shown in in Figure 9.3. Let $\mathcal{S}$ be the composition of the automata node $SS_u$ for every node in $G$ with $\mathcal{R}|L$.

**Variables**

- $pulse_u$: Highest pulse which was performed correctly until now. Its domain is restricted to lie in $\{0..Max\}$. Recall that $Max = T_\pi + D$, where $D$ is an upper bound on the network diameter.
- $pulse_u[v]$: Node $u$'s estimate of the current pulse number at node $v$. In normal synchronizer, the estimate is always a lower bound on the true value. Its domain is also restricted to lie in $\{0..Max\}$.
- $free_u[v]$: Boolean, which if true indicates that link to $v$ is free to accept new messages.
- $requestbit_u[v]$: Boolean, which is set to true to remember that node $u$ should make a reset request in the future.

Figure 9.2: Declarations of variables at node $u$ used by Resynchronizer code.

### 9.6.2 Proof Sketch for the Simplified *Resynchronizer* Construction.

We outline an argument that explains why we believe the simplified construction is correct. We emphasize that this is an intuitive argument that needs further polishing.

Just as in the spanning tree protocol of Chapter 8, the normal operation of the synchronizer can be defined by two one-way predicates $S1(u,v)$ and $S2(u,v)$ shown in Figure 9.4. The first states that a node's pulse number is one higher than the smallest estimate it has from any neighbor). The next states that the pulse estimates sent by node $u$ to neighbor $v$ are strictly non-decreasing. However, any estimate sent by $u$ can never be more than one higher than the current pulse number of $v$. If we compare the local predicates of Figure 8.7 with the local predicates of our spanning tree protocol (Figure 9.4) we notice a remarkable similarity between two different protocols.

If these two local predicates hold for all edges $(u,v)$ we can show that all nodes are *synchronized* - i.e., the pulse number of each node differs by at most 1 from any of its neighbors. This is sufficient to ensure that the pulse numbers will eventually grow until some node reaches the maximum value.

Unlike the spanning tree protocol, however, the *Resynchronizer* protocol keeps making reset requests and there will never be final signal intervals in any execution

RECEIVEM$_{v,u}(Pulse, p)$ (*receive pulse message from neighbor $v$*)

If $(pulse_u[v] \leq p)$ and $(p \leq pulse_u + 1)$ then

$pulse_u[v] := p$

If $min\{pulse_u[v], v \in neighborSet_u\} + 1 \leq Max$ then

$pulse_u := min\{pulse_u[v], v \in neighborSet_u\} + 1$

Else $pulse_u := Max$

Else $requestbit_u = true$

SENDM$_{u,v}(Pulse, p)$ (*output action to send pulse to neighbor $v$*)

Preconditions: $p = pulse_u$ and $free_u[v] = true$

Effects: $free_u[v] = false$

REQUEST$_u$ (*output action to request a reset*)

Preconditions: $requestbit_u = true$ or $pulse_u = Max$

Effects: None

SIGNAL$_u$ (*input action to receive a reset signal*)

Effects:

$pulse_u := 0$

For all neighbors $v$ of $u$ do

$pulse_u[v] := 0$

$free_u[v] := false$

FREEM$_{u,v}$ (*receive free signal from link to neighbor $v$*)

$free_u[v] := true$

Figure 9.3: Resynchronizer Code for any node $u$

of the protocol. Thus we need to make heavy use of the causality property of reset behaviors. For example, we can show that within $O(n)$ time of any suffix of an execution, some node must make a reset request. This is proved by contradiction. If not, in linear time all signals must stop and in constant time after this all local predicates must hold. (The second part follows because the code makes a reset request if it detects a violation of $S2(u,v)$. Also, the receipt of the first pulse message at $u$ will make $S1(u,*)$ hold.) Thus by normal synchronizer operation, the node with the lowest pulse number will increase its pulse number by 1 in constant time; thus in $O(\mathit{Max}) = O(n)$ time, some node will have reached the maximum pulse number and hence will make a reset request.

The overall argument consists of showing that there is some $t$-suffix suffix $\gamma$ of an execution of the Resynchronizer protocol, where $t = O(n)$ and such that:

1. All nodes receive a signal event in linear time after the start of $\gamma$.
2. In linear time after all nodes reach pulse number 0, some node reaches pulse number $\mathit{Max}$ and all nodes reach pulse number $T_\pi = \mathit{Max} - D$

As in the proof of the Global Correction Theorem, we can argue that in linear time, all reset requests that are caused by local predicate violations will disappear. Thus we choose $\gamma$ such that reset requests in $\gamma$ are only caused by nodes reaching the maximum pulse number $\mathit{Max}$.

The intuition behind the first part is that since some node makes a reset request within $O(n)$ time of $\gamma$, within $O(n)$ time in $\gamma$ all nodes will receive a signal. Notice that when a node gets a signal, the node resets its pulse number to 0.

The intuition behind the second part is as follows. After all nodes have reached pulse number 0, we can again argue that some node makes a reset request in linear time. By our choice of $\gamma$, a reset request is only caused by some node, say $u$, reaching pulse number $\mathit{Max}$. But since $u$ was previously at pulse number 0 in $\gamma$, $u$'s pulse number must have grown from 0 to $\mathit{Max}$ in $\gamma$. We then argue that all neighbors of $u$ have grown from 0 to $\mathit{Max} - 1$ in $\gamma$, and finally that all nodes have grown from 0 to $\mathit{Max} - D$ in $\gamma$.

Finally, if all nodes grow from 0 to $\mathit{Max} - D$ in $\gamma$, we argue that by the end of $\gamma$ all nodes have correct output. Intuitively, this is because each node begins executing the synchronous protocol at pulse 0 and corrects its output at pulse $T_\pi = \mathit{Max} - D$.

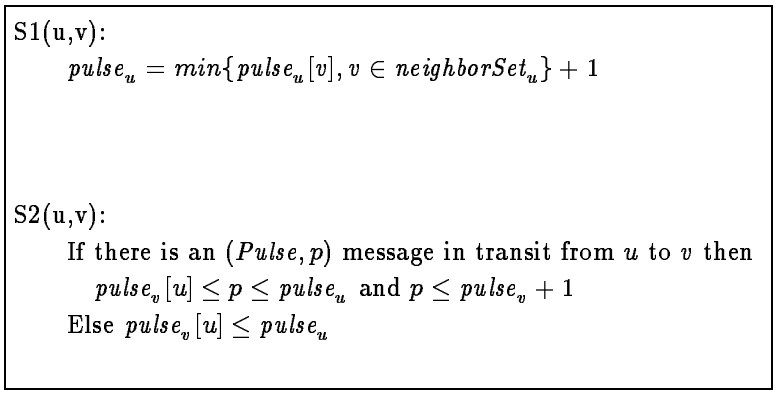



Figure 9.4: Normal Synchronizer: Local Predicates for edge $(u, v)$.

## 9.7 Extensions

### 9.7.1 Better Synchronous Checkers for Deterministic Protocols

Suppose we limit ourselves to stabilizing protocols $\pi$ that execute a distinct checking protocol $\chi$ even after reaching a legal state. Let us informally define the stabilization bandwidth of $\pi$ to be the worst case message complexity per link of the checking protocol. Clearly the checking protocol must be executed at least once every $T$ time units – where $T$ is the stabilization time – even after the protocol has stabilized. Hence this is really a bandwidth cost. For example, the stabilization bandwidth of the protocol in [KP90] is the worst case message complexity per link to do a snapshot, which is $\Omega(m)$, where $m$ is the number of links.

If we check a deterministic protocol $\pi$ by re-executing $\pi$, we have to pay a high price ($T_\pi$) in stabilization bandwidth. Suppose instead, we can find a checker $\chi$ for $\pi$ that has $T_\chi = 1$ – i.e., can check $\pi$ in a single pulse. Then after the execution phase we can add a single check pulse number $T_{cp}$. When a node reaches pulse $T_{cp}$ it stays at $T_{cp}$ executing the checking protocol until it detects a problem; if it does detect a problem it drops back to pulse 0. By avoiding multiple pulses for checking, we remove the need for costly a termination detection and reset operation in the checking phase. The stabilization bandwidth drops to $O(S_\pi)$.

There are a number of tasks that we can check in a single pulse. These include the classical problems of shortest paths, topology update, leader election, computing a spanning tree, and computing a maximum flow. Because the first four tasks are commonly used in real networks, it is important to improve their stabilization bandwidth.

We do so by what by essentially introducing a new form of local checking which we will call *synchronous local checking.* The idea of doing local checking was already introduced in previous chapters. In the previous chapters, we described asynchronous protocols which continuously checked the state of every link subsystem. Note that state of of every link subsystem includes the state of the channels as well as that of the nodes. However, to build an efficient synchronous checker, all we have to do is to locally check a *synchronous* protocol that has *terminated.* Hence the checking procedures are much simpler. Note that for a synchronous protocol, the state of every link subsystem consists only of the state of two nodes.

Consider the problem of checking the shortest cost paths from a given source to every other node. The key is the ability to check, in a single pulse, the shortest cost distances from a given source to every other node. Let $s$ denote the source and $D_i$ denote the distance node $i$ has recorded to $s$; let $B_{i,j}$ denote the cost of a link from Node $i$ to Node $j$, and $N(i)$ the set of neighbors of Node $i$. Then in a checking pulse a) $s$ checks that $D_s = 0$ b) Nodes other than $s$ check that $D_i > 0$ and $D_i = Min_{j \in N(i)}(D_j + B_{i,j})$. It is quite simple to prove that if any of the distances are wrong some node will detect this after checking.

By adding an extra distance label to the other tasks, the other tasks can also be checked in one pulse, using the same trick. We can do this, for instance, in leader election by adding the distance to the leader, and in maximum flow by adding the distance to the source in the residual graph.

In general, given our application, the design of faster checkers for synchronous protocols becomes an interesting and practical problem. For instance, we can we check a minimum spanning tree in fewer pulses than it would take to compute the tree from scratch while using only small storage? There are a number of similar open problems that arise from our work.

### 9.7.2 Randomized Protocols

In all our models, both of asynchronous and synchronous protocols we have disallowed the possibility of nodes tossing coins. Thus our formal models preclude the description of *randomized protocols.* In a randomized protocol, nodes are given the ability to toss coins. The definitions of correctness now become probabilistic and it is much harder to give careful definitions of stabilization. However, we will assume that the reader has an intuitive understanding of randomized protocols to understand the following informal discussion.

First, for randomized protocols, we will assume that any supposedly random bits in the initial state of a node can be arbitrarily corrupted and hence non-random. However, subsequent coin-tosses will produce truly random bits.

Next, note that we need a separate checker to compile a randomized protocol. This is because re-executing the original protocol can lead to a different output, and cause the checker to detect an error when there was none. Saving the original random bits in the state does not help either as these bits could be corrupted (see previous assumption). Further, this checker must be *oblivious*: it must not depend on the correctness of the supposedly random bits currently in the state. It appears that a stabilizing algorithm that uses a randomized checker needs an infinite supply of random bits since it cannot rely on the old random bits at any stage.

A simple example of compiling a randomized protocol is furnished by the problem of electing a leader in an anonymous network - i.e., a network in which nodes do not have any unique IDs.[2] Clearly we need randomization to break symmetry. To construct a stabilizing protocol for this task, we demonstrate a synchronous protocol for execution together with an oblivious synchronous checker.

Recall that we use $D$ to denote an upper bound on the diameter of the network. In the execution protocol, each node picks a random ID uniformly and independently in a space of $1..X$. In the next $D$ pulses, a node considers itself as the leader if it finds out that its ID is the largest in the network. In the checking protocol, each node $i$ that considers itself a leader picks a new random value $t_i$ in the space $1..X$ and broadcasts $t_i$ during the next $D$ pulses. At the end of $D$ pulses, a node detects an error if it has received either no values or it has received more than one value. While both checking

[2] If nodes have unique IDs we must assume that the IDs are protected; dropping this assumption makes the system more fault tolerant.

and execution can fail, by picking $X$ to be a polynomial (of sufficiently high degree) in the number of nodes, we can ensure that a correct output will be produced in constant expected number of phases. Since each phase takes $O(D)$ the *expected* stabilization time is $O(D)$.

A more efficient protocol for this purpose (that works in time proportional to the actual diameter as opposed to a bound on the diameter) is given in [DIM91b]. However, our solution seems to be simpler. As in the case of deterministic protocols, checking randomized synchronous protocols seems an interesting research area.

## 9.8 Summary and open problems

The main result of this chapter the *Resynchronizer*, is a compiler that transforms any synchronous protocol into a stabilizing version for dynamic asynchronous networks. The transformation adds $O(n + D)$ overhead to the time complexity of the protocol, where $D$ is a bound on the diameter of the final network. Clearly $D$ and $n$ can be much larger than the actual diameter of the final network. A natural open problem is to obtain a compiler whose time overhead only depends on the actual diameter of the final network.

When the results in this chapter were first presented [AV91], the *Resynchronizer* compiler used a much more complicated construction (which could be regarded as a special reset protocol optimized for the case of synchronizer operation). The transformation in [AV91] added only $O(D)$ overhead to the time complexity of the protocol, removing the factor of $n$ which arises from the use of the reset protocol described in Chapter 7. However, for many real networks the only meaningful bound on the diameter of the network is the number of nodes. Thus this does not seem to be a meaningful distinction in practice. Also, the construction in this chapter is much simpler than the original. However, a careful proof of the simplified construction is needed.

# Chapter 10

# Conclusions and Open Questions

This thesis investigates the power and applicability of local checking and correction for the design of stabilizing network protocols. The thesis provides a rigorous theoretical foundation for these concepts. However, the emphasis is on using these concepts to devise novel fault-tolerant protocols for real networks.

We have confined the notion of "local" to link subsystems consisting of a pair of neighboring nodes and the two unidirectional links between them.[1] We have defined a protocol to be *locally checkable* for a good property $L$ if two conditions hold. First, there is a local property for each link subsystem (formalized by a predicate of the link subsystem) such that property $L$ holds if the local properties of all link subsystems hold. Second, each local property is *closed* - i.e, once the local property is true, it remains true.

We have also defined a protocol to be *locally correctable* to property $L$ if the protocol is locally checkable for $L$ and there is a way to locally correct each node with respect to a link subsystem (formalized by the existence of a *local reset function*) such that the global property $L$ becomes true. The index contains pointers to the formal definitions of these concepts.

This chapter summarizes and evaluates the major contributions of the thesis. The chapter ends with a list of open questions.

[1] However, in the list of open questions we suggest that this notion of *locality* can be usefully generalized.

## 10.1 Contributions

The major contribution of this thesis is the construction of general techniques for designing stabilizing protocols. All our techniques revolve around the concepts of local checking and correction. We use the general techniques to design new stabilizing protocols (many of which are practical) and to understand existing stabilizing protocols. In the process of formalizing these techniques, we were also led to invent new definitions of stabilization, a simple modularity theorem, and a new Data Link model. We believe this modelling effort is useful in its own right. Thus we divide the contributions of this thesis into five categories:

- General techniques for constructing stabilizing protocols.
- New or improved stabilizing solutions to specific problems.
- The Modularity Theorem
- Modelling of self-stabilization.
- Better understanding of existing work in self-stabilization.

We describe these contributions in more detail in the following subsections.

### 10.1.1 General Techniques

The thesis contains four general techniques and a powerful heuristic. The four general methods are all organized around the theme of local checking and form a natural progression of ideas. The methods are:

- **Local Correction**: The Local Correction Theorem (Theorem 5.4.3) of Chapter 5 shows that every locally checkable and correctable protocol can be stabilized in time proportional to the height of the underlying partial order. This is achieved by adding actions to do *independent* checking and resetting of each link subsystem

  The Local Correction Theorem formalizes a useful design technique for building stabilizing protocols. First, we add some (hopefully small) state to the protocol

to make it locally checkable. Next, we look for a local reset function that can be used for local correction. In some cases, it is possible to construct a local reset function by combining several actions of the original protocol. In Chapter 7 we conjectured that this (i.e., the construction of a local reset function) is possible for many existing protocols that work in dynamic networks. The stabilizing reset protocol of Chapter 7 is an example of this design technique.

- **Tree Correction**: The Tree Correction Theorem (Theorem 6.5.1) of Chapter 5 shows that every locally checkable protocol that works on a tree topology can be stabilized in time proportional to the height of the tree. Thus we can dispense with the need for local correctability if the underlying topology is a tree. This is achieved by adding actions to do *independent* checking and resetting of each link subsystem in such a way that the correction of a link $(u, v)$ does not invalidate the correctness of links above link $(u, v)$ in the tree.

  This theorem also formalizes a useful design technique. First, we construct a spanning tree of the network using a stabilizing tree protocol. Next, we design a locally checkable protocol to solve the desired problem that works over a tree. Finally, we apply the transformation underlying the Tree Correction theorem. The stabilizing token passing protocol of Chapter 6 is an example on which this design technique could be applied.[2]

- **Global Correction**: The Global Correction Theorem (Theorem 8.1.2) of Chapter 8 shows that every locally checkable protocol can be stabilized in time proportional to the number of network nodes. That is we can dispense with the need for local correctability or the restriction to a tree topology if we are willing to incur a stabilization time that is proportional to the number of network nodes. This is achieved by adding actions to do *independent* checking of each link subsystem as well as actions to do a *coordinated global reset* of the entire network if a local violation is detected.

  This theorem also formalizes a useful design technique. We construct a locally checkable protocol and then apply Global Correction. However, since the stabilization time of this method is comparatively high, it pays to use the Local Correction Theorem whenever a locally checkable protocol can also shown to be

[2] However, in Chapter 6 this example is derived using Local Correction.

locally correctable. The stabilizing spanning tree protocol of Chapter 6 is an example of the Global Correction technique.

- **Stabilizing Compiler for Synchronous Protocols**: The Resynchronizer and Rollback Compiler ideas in Chapter 9 show that any synchronous protocol $\pi$ with time complexity $T_\pi$ can be converted to an asynchronous, stabilizing version of $\pi$, with either an additive cost of $O(n)$ in stabilization time or a multiplicative factor of $T_\pi$ in storage. Thus for such protocols *we can dispense with even the need for local checkability.* This is achieved by applying global correction to a simple synchronizer protocol. While Chapter 9 sketches the main ideas for both compilers, the method still requires further development, especially to complete and simplify the proofs.

  The synchronous compilers also suggest a useful design technique. Suppose the correctness of a task can be specified by an Input/Output relation. Then we can construct a synchronous protocol for a given task and then apply one of the two compilers. This technique provides stabilizing solutions for many tasks for which locally checkable solutions are not known to exists. For instance, this can be used to produce efficient and stabilizing minimal spanning tree, min-cost flow and coloring protocols.

Besides the four general methods, we have a useful heuristic.

**Heuristic of Removing Unexpected Packet Transitions**: In many cases we can make a protocol weakly locally checkable for a good property $L$ (i.e., $L$ holds if the local properties of all link subsystems hold) by adding a small amount of state. To make the protocol locally checkable for $L$ we also have to ensure that each local property is closed - i.e., if the local property is ever true, it remains true. This can often be done by the method of removing *unexpected packet transitions* described in Chapter 6. The basic idea is that when a node $u$ receives a packet $p$ from a neighbor $v$, $u$ only accepts the packet if this transition could have occurred in some good state of the $(u,v)$ subsystem. If the $(u,v)$ subsystem is in a bad state but the $(u,w)$ subsystem is not, the only way the $(u,v)$ subsystem can affect the $(u,w)$ subsystem is if $v$ sends "unexpected" packets to $u$. Weeding out such unexpected packet transitions is often sufficient to ensure that each local predicate remains closed.

The heuristic of removing enexpected packet transitions is used throughout the thesis. It is used in the Token Passing Protocol of Chapter 6, the Reset Protocol of

Chapter 7, the Spanning Tree Protocol of Chapter 8, and the Resynchronizer Protocol of Chapter 9.

### Comparison with Only Previously Known General Technique

The only previously known general method for stabilization is the elegant result of Katz and Perry [KP90]. How do our methods compare with the general method of [KP90]? Recall that in [KP90], checking is done by a single leader node periodically taking a snapshot of the entire network.

- *Message Congestion*: From a practical standpoint, the most important difference is that *all* our methods have considerably smaller stabilization bandwidth than that of [KP90]. Recall from Chapter 9 that stabilization bandwidth is the periodic overhead of checking that must be paid *even when the protocol is in a good state and behaving correctly.* Suppose the protocol is to stabilize in time $T$. Roughly, [KP90] has to pay the price of a snapshot of the entire network every $T$ units of time. Now a snapshot requires at least $O(m)$ state, where $m$ is the number of network links. Thus links *leading to the leader node* must carry $O(m)$ message bits every $T$ units of time. Thus the worst case bandwidth per link is very high. By contrast, in our local correction, tree correction, and global correction methods, each link only carries a *constant* number of message bits every $T$ units of time. Even the naive synchronous compilers of Chapter 9 have better stablization bandwidth than the method of [KP90] because the communication overhead of checking is *spread out among all the links* rather than being concentrated on links leading to the leader. In real networks, each link has a limited bandwidth and the worst case bandwidth requirement is an important measure of link congestion.

- *Speed*: If all links are UDL's, the method of [KP90] requires $O(n)$ time to stabilize, where $n$ is the number of network nodes. The local correction method, tree correction method, and Rollback compiler can all provide much faster stabilization times.

- *Storage*: The method of [KP90] requires $O(m)$ storage at the leader to store the snapshot information, where $m$ is the number of links. For fault-tolerance,

every node must be prepared to be the leader of the network and be able to store $O(m)$ information bits. By contrast, except for the Rollback compiler, the storage required by our methods is negligible.

- *Generality*: The method of [KP90] is clearly more general than our methods. Our methods require protocols to be either locally checkable or to have a correctness specification that can be expressed (see Chapter 9) in terms of an I/O relation. However, there is at least one case where the method of local checking and correction is applicable where the method of [KP90] is not. This is the stabilizing end-to-end protocol that we describe in [APV91b]. The problem here is that some unknown set of network links may have infinite delay. Thus the global snapshot of [KP90] may never terminate. However, it is sufficient ([APV91b]) to do local checking and correction on the so-called *viable* links that have bounded delay.

To summarize, our methods are less general but more efficient than the method of [KP90]. Despite the loss of generality, our methods can be used to stabilize many useful tasks, as summarized below.

### 10.1.2 New or Improved Stabilizing Solutions for Specific Problems

Our general techniques provide new or improved solutions for Mutual Exclusion, Network Resets, Computing Spanning Trees, Topology Update, Minimal Spanning Trees and other tasks. We have also applied local correction to an important theoretical problem – the problem of end-to-end message delivery in unreliable networks. In this problem links can fail *continuously* – the only guarantee is that there is no cut of permanently failed links that separate the sender and receiver. We have not described our solution to this problem in this thesis but more details can be found in [APV91b].

There are a number of other protocols to which we believe local checking and correction can be applied. These include the efficient resource allocation algorithm of Awerbuch and Saks ([AS90]), and the elegant virtual circuit protocol due to Spinelli ([Spi88b]). We hope to produce stabilizing versions of these protocols. We believe our general methods provide solutions to many important networking tasks.

We emphasize that *many of our solutions are practical and can be applied in real networks.* Messages required for local checking can easily be piggybacked on the "keep-alive" traffic sent between neighbors in real networks. Thus solutions based on the first three of our general methods (all of which are based on local checking) can be added to real networks without appreciable loss of efficiency. Solutions based on the synchronous compilers of Chapter 8 can be practical if either the time complexity of the underlying synchronous protocol is low (for the Rollback compiler) or if the underlying synchronous protocol has an efficient synchronous checker (for the Resynchronizer compiler). A disadvantage of the synchronizer methodology is that the method tends to slow down the network to the speed of the slowest link. Thus the synchronous compiler methodology is best suited for use in networks where all links have roughly the same speed.

Among the most useful practical protocols described in this thesis are the stabilizing reset protocol of Chapter 7 (which was briefly tested in a trial implementation on the AUTONET) and the spanning tree and topology update protocols of Chapter 8. The topology update protocol also illustrates another general paradigm that may be useful in practice. We start with a simple protocol $P$ that uses unbounded sequence numbers and use global correction to convert $P$ into a stabilizing version $P'$ that uses bounded sequence numbers. In the absence of catastrophic errors, $P'$ is as efficient (except for the small overhead of local checking) and simple as $P$.

### 10.1.3 Modularity Theorem

The Modularity Theorem (Theorem 3.5.7) is simple but extremely useful. It helps us to prove facts about the stabilization of a big system by proving facts about the stabilization of each of its parts, as long as each part is suffix-closed. The modularity theorem gives us a formal basis for a building-block approach. For example, we can start with a stabilizing implementation of a UDL; use this to build a stabilizing reset protocol as shown in Chapter 7; and then use the reset protocol as a building block to construct a spanning tree protocol (Chapter 8) and a compiler for synchronous protocols (Chapter 9). As we have argued at the end of Chapter 4, the requirement that each of the parts be suffix-closed is not very restrictive. Essentially, this is because our methods tend to produce CIOAs (i.e., automata in which every reachable state is a start state) that are suffix-closed.

As we build up a complex stabilizing protocol in several layers, the stabilization time of the system can be calculated by applying the Modularity Theorem and the Transitivity Lemma for behaviors (Lemma 3.2.6). For example, suppose the overall system is described by an automaton $P$. Suppose also that $P$ is identical to the stabilizing reset protocol $\mathcal{R}^+$ of Chapter 7 *except* that every UDL $C_{u,v}$ in $\mathcal{R}^+$ is replaced by a stabilizing implementation of a UDL called $C'_{u,v}$. Suppose that $C'_{u,v}$ stabilizes to the behaviors of $C_{u,v}$ in time $t_1$. Then by applying the Modularity Theorem (which is possible because all the constituent automata in $\mathcal{R}^+$ are UIOA) we conclude that $P$ stabilizes to the behaviors $\mathcal{R}^+$ in time $t_1$. But we know from Chapter 7, that $\mathcal{R}^+$ stabilizes to the behaviors of $\mathcal{R}|L$ in some constant time, say $t_2$. (Recall that $\mathcal{R}|L$ is a "correctly working" reset protocol in which all local predicates are true in the initial state.) Thus we conclude from the Transitivity Lemma that $P$ stabilizes to $\mathcal{R}|L$ in time $t_1 + t_2$. Notice that the stabilization times of each "layer" add up due to the Transitivity Lemma. Proceeding similarly, we can calculate the stabilization time of a version of the stabilizing tree protocol of Chapter 8 in which the reset protocol $\mathcal{R}|L$ is replaced by $P$.[3]

Our theorem can be viewed as a generalization of previous modularity results [DIM90]. Previous results only applied to the case when a lower layer protocol computed values of a shared variable that were read by a higher layer protocol. By contrast, our theorem applies to more dynamic interaction between the various components of a system.

### 10.1.4 Modelling

The models and proof techniques we use in this thesis are based on the large body of existing work that has been done in the area of protocol verification. Our model of computation is based on the timed I/O automaton [MMT91] which is in turn based on the I/O automaton of [LT89]. We have even described the shared memory network model in Chapter 4 as an I/O automaton that meets certain restrictions.

We also use standard proof techniques. We use mapping techniques (Refinement Mapping Theorem, Theorem 3.4.3) to show that one automaton has the same behaviors as another automaton. We prove that a local predicate is closed using the standard

[3]This time, however, we can apply the Modularity Theorem because $\mathcal{R}|L$ is a CIOA and the other nodes implementing the spanning tree protocol are UIOA.

inductive techniques used to prove program invariants. Perhaps the only unusual technique is the Execution Convergence Theorem (Theorem 3.4.5) which is a key tool for proving stabilization results. To apply this theorem we have to prove a stability condition and a "liveness" condition. We prove the stability condition using standard inductive arguments. We prove the "liveness" condition by proving time bounds on the occurence of events. We have proved time bounds in a fairly ad hoc, operational way. However, there is no reason why the time bounds could not be established by more rigorous inductive arguments (as in [LA90]). Thus while the Execution Convergence Theorem may appear slightly unusual, it is really a combination of existing verification techniques.

We believe this continuity (with the body of existing work in verification) is an advantage of our work. By contrast, previous papers on stabilization have sometimes tended to invent new models and proof techniques. Despite our strong linkage with existing work, we do have two interesting modelling contributions in this thesis. These are the use of a behavior specification for stabilization and the use of unit storage Data Links.

### Behavior Specification

The definitions of stabilization in terms of external behaviors are different from 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). This can be done using a definition of stabilization in terms of executions if we are prepared to introduce an abstraction function into the definitions as in Lamport's work [Lam83]. However, we prefer the behavior definitions as they seem to be more natural; we prefer to use the equivalent of an abstraction function to *prove* behavior stabilization results using the Refinement Mapping Theorem.

### Unit Storage Data Links

In a stabilizing setting it is necessary to define Data Links that have bounded storage. First, as shown in [DIM91a], almost any non-trivial network task is impossible in

a stabilizing setting in which the links have unbounded storage and the nodes are restricted to be finite state machines. Second, bounded storage models correspond to physical reality.

Thus we use the standard asynchronous message passing model of a computer network except that each link is what we call a Unit Storage Data Link (UDL) that can store at most one packet. We have chosen unit storage links (UDLs) because they are practical (see Section 5.2) and they can be modelled elegantly. We have also defined a stabilizing interface to a UDL. This is done by having the link periodically deliver a free signal (to avoid deadlock) and by having the sender keep a variable that indicates whether the link is free. We hope the UDL model will be used by others. The UDL model can easily be generalized to Bounded Storage Data Links by changing the free signal to report the number of packets currently stored in the link.

Earlier papers in stabilization (e.g., [DIM90]) seem to have used shared memory models for communication in order to avoid the problems caused by unbounded storage links. It does seem very likely (see Open Problems) that protocols that work in the low atomicity shared memory model of [DIM90] can be transformed to work correctly in our network model. However, we believe that our protocol descriptions are more accessible to designers of "real" network protocols because most "real protocol" specifications assume the use of message passing primitives.

### 10.1.5 Understanding Existing Work

The concept of local checking helps in crisply understanding many existing stabilizing protocols. Chapter 4 shows that some existing work in the shared memory model can be understood crisply in terms of local checking and correction. We believe that many existing stabilizing protocols can be understood using three general ideas – one way checking and correction, counter flushing, and timer flushing.

#### One Way Checking and Correction

In Chapter 8, we said that a protocol $P$ was one-way checkable if it was locally checkable using what we called one-way predicates. Intuitively, a one-way predicate is a predicate that involves the state of a node $u$, the state of any neighbor $v$ of $u$, and the state of the link between $u$ and $v$. Unlike a general local predicate, a one-way

predicate only depends on the state of one of the links between the two nodes. We saw that one-way checkable protocols could be checked without the need for a local snapshot – it is sufficient for each node to periodically send its state to its neighbors. The protocols in [Per83] and [Per85] do checking in this way.

In some cases, the protocol is also one-way correctable – i.e., we can apply local correction to a one-way checkable protocol without the need for a local reset of the link subsystem. For example, when $v$ receives a copy of $u$'s state and detects a violation of the one-way predicate from $u$ to $v$, it may be possible for $v$ to apply a local reset function to its own state so as to make the one-way predicate true. Of course, this could falsify other "adjacent" one-way predicates. For the protocol to be one-way correctable, the dependency relation among the one-way predicates must be acyclic, as in the definition of local correctability. For example, the *Rollback* protocol of Chapter 9 is one-way correctable. We speculate that the stabilizing topology update protocol of Spinelli-Gallager [SG89] is also one-way correctable

### Counter Flushing

Suppose a sender wishes to periodically send a REQUEST packet to a set of network nodes. The responders must each send back a RESPONSE packet before the sender sends its next request. In Chapter 5, for example, we implemented local snapshots and resets using such a request-response protocol initiated by the leader of each link subsystem. In order to properly match responses to requests, the sender numbers each request with a counter. Let $m$ be the number of packets that can be in transit between the sender and responder and let $n$ be the number of responders. Then the sender uses a counter that has $Max = m + n + 1$ distinct values. For example, in Chapter 5 we used a counter in the range 0...3 because there can be at most two packets in transit in a link subsystem and there is only one responder.

Responders only accept REQUEST packets with a number different from the last REQUEST accepted. After accepting a REQUEST the responder sends back a RESPONSE with the same number as the REQUEST. The sender retransmits the current REQUEST till it receives each matching RESPONSE with the same number. After all matching RESPONSE packets arrive, the sender increments its counter.

The size of *Max* ensures that within *Max* increments of the counter, the sender will reach what we call a "fresh" counter value – i.e. a counter value that is not

currently stored in either the links or the responders. We call the method counter flushing because the request-response protocol must guarantee the following "flushing" property. *Suppose the sender sends a request numbered c, where c is a fresh value. Then after all matching responses to this request arrive, there must no counter values other than c that are stored in the links or at the responders.* In other words, the sending of a freshly numbered request and the receipt of all matching responses, should "flush" the links and responders of "old" counter values.

In Chapter 5, the flushing condition is guaranteed because the sender and receiver are connected by two FIFO links in either direction. Similar forms of counter flushing can be used to implement Data Links ([AB89]) and token passing [DIM91a]) in link subsystems with bounded storage. Counter flushing is, however, not limited to link subsystems. The first example in [Dij74] can be simply understood as counter flushing in a unidirectional ring (see Appendix E for more details). Katz and Perry [KP90] extend the use of counter flushing to arbitrary networks in an ingenious way. Our stabilizing end-to-end protocol ([APV91b]) is obtained by first applying local correction to the Slide protocol [AGR92] and then applying a variant of counter flushing to the Majority protocol of [AGR92].

### Timer Flushing

The main idea is to bound the lifetime of "old" state information in the network. This is done by using node clocks that run at approximately the same rate and by enforcing a maximum packet lifetime over every link. State information that is not periodically refreshed is "timed out" by the nodes. In Perlman's [Per83] topology update protocol, timer flushing is used to get rid of erroneous updates that are numbered with the maximum possible sequence number. In Perlman's [Per85] spanning tree protocol, timer flushing is used to get rid of "ghost" roots (see Chapter 8 for details.) Spinelli [Spi88b] uses timer flushing to build stabilizing Data Link and virtual circuit protocols.

## 10.2 Open Questions and Further Problems

The following is a list of further problems. They are arranged under four categories: modelling, increased understanding of local checking and correction, new algorithms, and new directions.

### 10.2.1 Modelling

The following is a list of what we believe are the further problems that are motivated by our work in modelling stabilizing protocols.

- **Extend model and theorems of Chapter 3 to allow the use of randomization and lower bounds on the time between actions**: The model of Chapter 3 makes no provision for randomization or for lower bounds on the time between actions. While the model in Chapter 3 is sufficient for modelling asynchronous protocols it cannot be used to model many interesting randomized and timing based algorithms. This seems to be an important problem. The hard part is extending the model to obtain corresponding versions of the modularity and transitivity results.

- **Find a Proof technique for Generalized Stabilization Definitions**: The definitions of a stabilizing reset protocol and other protocols can be made more elegant if we modify the stabilization definitions as follows. We only require that the $t$-suffix of a behavior (execution) be a $t$-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). While a general proof technique for this purpose may be infeasible, it may be possible to find a proof technique that works for a large number of cases.

- **Discover stronger variants of the modularity theorem.** The modularity theorem allows us to infer the stabilization properties of a large system from the stabilization properties of its pieces. However, we required that each piece be suffix-closed. It is natural to look for weaker or alternate conditions. This may be somewhat technical as the suffix-closed requirement does not appear to be very restrictive.

- **Find a General Definition of Stabilization Bandwidth**: In Chapter 9, we intuitively described an important measure for a stabilizing protocol – the amount of periodic bandwidth the protocol consumes. The definition we gave in Chapter 9 only applies to protocols that have a certain structure. A precise, general definition would be very useful.

- **Find a way of Compiling Stabilizing Protocols in the Shared Memory Model into our Network Model.** A number of interesting stabilizing protocols have been described using the low atomicity, shared memory model introduced by [DIM90]. It would be nice to have a compiler that that could convert protocols from their model to our network model and vice versa. This seems to be feasible.

### 10.2.2 Increased Understanding of Local Checking and Correction

Our understanding of local checking and correction is far from complete. This motivates the following problems:

- **Obtain a better understanding of local checkability.** Recall that a protocol is locally checkable for some good predicate $L$, if $L$ is a conjunction of local predicates, and each local predicate is closed. Thus we can study this problem by asking two separate questions.
  - **How much state do we need to add to a protocol so that it's legal states are a conjunction of local predicates?** In the token passing protocol of Chapter 6 and the reset problem of Chapter 7, we made protocols locally checkable by adding a constant amount of state to each node. It is natural to ask what the minimum amount of storage is that must be added in order to make a protocol locally checkable.
  - **Can any weakly locally checkable protocol be transformed into an (equivalent) locally checkable protocol?** Recall that a weakly locally checkable protocol is a protocol whose legal states are a conjunction of local predicates; however, the local predicates are not necessarily closed. In Chapter 6 we described a simple protocol transformation that consisted of removing unexpected packet transitions. In many cases, this heuristic is sufficient to ensure that the local predicates are closed. We have used this heuristic successfully but we still don't completely understand when this heuristic is guaranteed to work. In Chapter 6, we described a sufficient condition called local extensibility. Are there weaker conditions than local extensibility?

- **Obtain a better understanding of local correctability.** In Chapter 5, we used a fairly operational definition of local correctability in terms of a local reset function. That definition is adequate for the thesis but it gives little insight into the structure of locally correctable protocols. Why are some protocols locally correctable but not others? Are there simpler sufficient conditions? It is also important to formally understand the connection between local correctability and locally checkable protocols that work in dynamic networks.

- **Generalize the Definitions of Locality and Local Checkability in Chapter 5**: In Chapter 5, we defined locality in terms of link subsystems. More generally, a subsystem is the composition of the nodes and channels corresponding to some subgraph of the network graph. For example if we define locality using the subsystems corresponding to the entire network graph, then our method reduces to the method of Katz and Perry [KP90]. Another interesting subsystem would consist of a single node and all its incoming and outgoing channels. Another interesting possibility is to consider subgraphs defined by the sparse network partitions [AP90] defined by Awerbuch and Peleg.

  Another simple generalization is to allow more than one local predicate per local subsystem. For example, consider an example consisting of two link subsystems and four link predicates, $L_1, L_2, L_3$ and $L_4$. Suppose the system is locally correctable using a partial order $<$ such that $L_1 < L_2 < L_3 < L_4$. Then as we apply local checking and correction to this system $L_1$ will become true first, followed by $L_2$ followed by $L_3$ and then $L_4$. Having multiple local predicates per local subsystem is only useful if these local predicates are independently ordered in the partial order. (If this were not the case we could simply work with a new local predicate that is the conjunction of all the predicates.) An example of the need for this generality is the *Rollback* protocol in Chapter 9.

- **Obtain a better understanding of Synchronous Checking** In Chapter 9, we introduced the notion of synchronous checking, which is a synchronous protocol that can check the output of another synchronous protocol. Synchronous checking is a little easier than asynchronous checking because all the nodes run in lockstep and we do not have to worry about messages in channels. Also, the checker need only check the *final output* of the protocol and not the intermediate states. Are there synchronous checkers for minimum spanning tree and max flow

protocols that are faster than the protocols being checked?

### 10.2.3 New Algorithms

We suggest the following algorithmic problems:

- **Invent a practical stabilizing reset protocol that stabilizes in $O(d)$ time, where $d$ is the actual diameter of the network.** By way of comparison, the reset protocol of Chapter 7 stabilizes in $O(n)$ where $n$ is the number of nodes in the network. This seems to be a major open problem. A solution to this problem would yield $O(d)$ stabilizing solutions for the spanning tree, topology update, and other problems described in Chapters 8 and 9. To be practical, the constants hidden in the $O(d)$ notation must be small (between 1 and 3).

- **Invent a practical stabilizing token passing protocol for rings and arbitrary graphs**: We have already described how to construct a tree from an arbitrary graph, and shown how to execute a stabilizing token passing protocol on a tree. However, the latency of token traversal on a tree can be quite high, for example if the original graph is a ring. The first example in [Dij74] appears extensible to token passing in a ring. An efficient stabilizing token passing protocol for rings would be useful and practical. Existing token ring protocols such as the IEEE 802.3 and FDDI protocols have a number of *ad hoc* mechanisms to deal with failures.

- **Invent a Stabilizing Clock Synchronization Scheme**: Fault-tolerant clock synchronization schemes typically have to defend against Byzantine faults. It would be desirable to invent a practical, stabilizing version of such a protocol. This would be an interesting example of a protocol that is robust against both Byzantine faults that continue as well as catastrophic faults that stop.

- **Simplify Existing Flow Control Schemes for Transport Protocols and Data Links**: A flow control scheme is a scheme by which a receiver regulates the rate at which a sender sends data in order to prevent buffer overflows at the receiver. In [CSV89] we propose an extremely simple stabilizing flow control scheme for physical links. It can be considered to be a trivial application of local checking and correction to the sender-receiver flow control protocol. The

resulting protocol is robust and simple enough to be implemented in hardware. Now, robust flow control schemes are a major component of Transport and Data link layer protocols. Perhaps existing flow control schemes can be simplified using local checking and correction.

- **Invent a stabilizing, distributed protocol to compute Sparse Partitions**: Awerbuch and Peleg [AP90] have shown how to decompose a network into what they call sparse partitions. They use sparse partitions to build efficient solutions for online tracking of mobile users, network synchronization, and network routing with low memory. The most efficient *distributed algorithm* for constructing sparse partitions is based on the work of Linial and Saks [LS91]. A stabilizing distributed algorithm for sparse partitions would be an extremely useful tool.

- **Make the Synchronizer Methodology practical for networks with links of different speeds**: The synchronizer methodology was introduced in [Awe85] and is extended to a stabilizing setting in Chapter 9. However, the method suffers from a severe drawback in networks in which links have varying delays. Essentially, it slows down all links to the speed of the slowest link. It may be possible to modify the methodology to avoid this drawback.

- **Invent a stabilizing version of the Bootstrap Protocol for End-to-End Communication**: Our previous work on stabilizing end-to-end communication [APV91b] has concentrated on producing a stabilizing version of the simple and elegant Slide protocol [AGR92]. However, the Bootstrap Protocol [AG91] is more efficient in some cases, and so a stabilizing version would be of theoretical interest. The current solutions are still too inefficient (for example in storage) for the end-to-end problem to have any practical application.

### 10.2.4 New Directions

We suggest the following new directions for research in self-stabilization.

- **Local Checking for Randomized Protocols**: Local checking of randomized protocols is an important research area. There are several randomized protocols

(e.g., the Rabin-Lehman dining philosophers protocol of [RL81]) that use randomization to break symmetry and to guarantee termination. Such protocols can often be locally checked in a deterministic fashion. For example, in graph coloring, it is easy to check whether a neighboring node has the same color. If, however, the check reveals a problem, a *randomized local reset action must often be applied.* Thus in graph coloring, a node may randomly choose a new color once it discovers that it has the same color as one of its neighbors. Now the randomized local reset functions can be applied at a node after all other nodes have already chosen their colors. Thus the dynamics (and the analysis) of the stabilizing protocol may be quite different from that of the original randomized protocol. Preliminary work in this area has been done by Baruch Awerbuch, Leonore Cowen, and Mark Smith at MIT.

Protocols whose behavior is correct only with high probability are even harder to check locally. Suppose that a (non-stabilizing) randomized protocol computes sparse partitions such that with high probability the partitions have low diameter. Then even if local checking can detect "poor" partitions, how can we tell whether the partitions are bad a) because of errors in the initial random bits at the nodes or b) because of a low probability outcome of the protocol?

- **Stabilizing Data Structures**: We have suggested earlier that programs that run on a single shared memory can be made stabilizing by using domain restriction – the states of the program are restricted to legal states. Consider the problem of building a stabilizing data structure (e.g., a dictionary or a queue) within the memory of a single processor. The data structure provides a certain set of operations (for example to insert and delete elements). The correctness of the data structure can be defined in terms of allowable sequences of operations. This notion of correctness is similar to the behavior specifications of I/O automata. Thus we can define a stabilizing data structure to be one in which the data structure can begin in an arbitrary state; however, any sequence of operations on the structure will eventually have a suffix that is a correct sequence (or a suffix of a correct sequence) of the data structure. For example, in a stabilizing dictionary, we would require that any elements inserted after the structure stabilizes would be found when searched for.

  Now any such data structure can be trivially made stabilizing in the following way. Before any operation on the data structure, we first check the invariants of

the data structure, and reinitialize the data structure if an error is found. Unfortunately, this slows down regular operations. Thus if we charge for processing (which we have not done in the distributed model in this thesis), there appears to be a tradeoff between stabilization time and the time to complete normal operations. For some data structures, the tradeoff can be extremely good. For example, we can easily implement a stabilizing, bounded size queue using a fixed size array. The head and tail pointers can be restricted to stay within the array bounds. However, it appears to be much harder to obtain good tradeoffs for say a tree-based implementation of a dictionary. We have done some preliminary thinking in this area [MPV90].

- **Applying Self-Stabilization to Other Areas**: Most of the stabilizing protocols described in this thesis are used for routing, scheduling, and resource allocation – tasks typically found in the network and Data Link layers of the communication hierarchy. However, in principle there is no reason why self-stabilization cannot be added as an "extra envelope" of fault-tolerance for higher layer protocols – e.g., file transfer and database protocols. Such protocols should be designed to avoid errors in the case of common faults such as node and link crashes. However, it is also desirable that these protocols recover by themselves after catastrophic errors. Self-stabilization is also applicable at the lowest layer of the protocol hierarchy. For example, it would be desirable to have stabilizing clock recovery and framing protocols at the physical layer.

- **Generalized Local Checking**: The method of Katz and Perry [KP90] consists of a single leader that checks the entire network. In our method of local checking, nodes independently (and in parallel) check each link subsystem. The Katz and Perry method is more general but less efficient. Local checking of link subsystems is efficient but only applicable to locally checkable protocols. Thus there may be intermediate approaches in which the notion of locality is generalized to an arbitrary set of subnetworks of the original network.

## 10.3 Summing Up

Self-stabilization abstracts the ability of a protocol to tolerate catastrophic faults that stop. On the other hand, the cost of self-stabilization is often low. Thus self-

stabilization can provide a cheap way to improve the fault-tolerance of network protocols.

Local checking can be used to *design* efficient, stabilizing protocols. The resulting protocols can be *proved* correct in a systematic way. The overhead of local checking can be *piggybacked* on existing keep-alive traffic between network nodes. Local checking may prove to be a useful *debugging* tool because it provides a continuous check of system predicates, and violations can be logged.

Our research into stabilization and local checking has helped us better understand existing protocols, both stabilizing and non-stabilizing. For instance, in the process of understanding Global Correction, we realized that a Global Reset protocol provides a mating relation that is a generalization of the guarantees of Data Link protocols. We also saw that many existing protocols use a special form of local checking that we called one-way checking.

Whitehead once said that the interest of a generalization is the interest of a road for those who know what travel is; and the pleasure of the road has its roots in the labor of the journey. After struggling to understand many examples of stabilizing protocols, we have begun to appreciate the simplicity of understanding provided by such concepts as local checking and counter flushing. They have helped us to see things a little more clearly and to travel a little distance. We hope that our readers will go much further.

# Appendix A

# Notation

We summarize notation that is commonly used in this thesis. Other definitions can be found using the index. For example, if the notation for $C_{u,v}$ refers to a UDL, the definition of a UDL can be found by using the index.

- $\{u, v\}$: For an pair of neighboring nodes $u$ and $v$, this denotes the unordered pair corresponding to the directed edge $(u, v)$. This is useful in defining a partial order on unordered pairs of nodes.
- $A|L$: For any automaton $A$ and any subset $L$ of the states of $A$, $A|L$ is the automaton that is identical to $A$ except that the set of start states of $A|L$ is $L$.
- $C_{u,v}$: The Unit Capacity Data Link (UDL) corresponding to directed edge $(u, v)$ in a topology graph.
- $Conj(\mathcal{L})$: For any link predicate set $\mathcal{L}$, $Conj(\mathcal{L})$ denotes the conjunction of the predicates in $\mathcal{L}$.
- $\mathcal{N}(t)$: For any network automaton $\mathcal{N}$, $\mathcal{N}(t)$ is the automaton that is identical to $\mathcal{N}$ except that the link and node delays in $\mathcal{N}(t)$ are equal to $t$.
- $Q_{u,v}$: The packet stored on the UDL corresponding to directed edge $(u, v)$ in a topology graph. If the value of the variable is *nil*, then there is no packet stored.
- $queue_u[v]$: A queue of packets for outgoing edge $(u, v)$ in the node automaton corresponding to node $u$. See the definition of a node automaton.

- $xqueue_u[v]$: The concatenation of $Q_{u,v}$ and $queue_u[v]$. Only used in some proofs. Can be the read as the extended queue at node $u$ corresponding to neighbor $v$,
- $U(A)$: For any automaton $A$, $U(A)$ is the automaton that is identical to $A$ except that the set of start states of $U(A)$ is equal to the set of states of $A$. Thus $U(A)$ creates a UIOA from an IOA.

The following symbols are used frequently to denote the following quantities

- $<$: A partial order on a set of local predicates.
- $\alpha$: An execution of an automaton
- $a_j$: The $j$-th action in a behavior or execution.
- $\beta$: A behavior of an automaton.
- $f$: A reset function.
- $G$: A graph or a topology graph.
- $L$: A predicate (set of states) of an automaton.
- $L_{u,v}$ A local predicate for edge $(u, v)$
- $\mathcal{L}$: A set of local predicates.
- $\mathcal{N}$: A network automaton.
- $\mathcal{N}^+$: An augmented network automaton.
- $P$: A packet alphabet.
- $s$: A state of an automaton. A state with a subscript like $s_i$ often denotes the $i$-th state in an execution.
- $t$: A time. Times with subscripts like $t_n$, $t_p$ denote special constants; many of these are listed in the index.

# Appendix B

# Proofs for Chapter 5

Let $\mathcal{N}^+ = Augment(\mathcal{N}, \mathcal{L}, f)$. We will use $s$ and $\tilde{s}$ to denote states of $\mathcal{N}^+$. In deriving time bounds recall that all locally controlled actions at nodes take $t_n$ time; also for any link all actions have upper bound $t_l$.

## B.1 Any Execution of $\mathcal{N}^+$ is infinite

Our first lemma states that all executions of $\mathcal{N}^+$ are infinite and hence in all executions of $\mathcal{N}^+$, time grows without bound. This follows because we have ruled out the possibility of so-called Zeno executions in our model (see Chapter 2). We will assume this implicitly in what follows without making explicit reference to this lemma.

**Lemma B.1.1** *Any execution $\alpha$ of $\mathcal{N}^+$ is infinite.*

**Proof:** Suppose not. Then there is some final state $s$ of $\alpha$ after which no action takes place. Consider any channel $C_{u,v}$. If $s.Q_{u,v} = nil$, then a $\text{FREE}_{u,v}$ action is enabled; but $s.Q_{u,v} = p \neq nil$, then a $\text{RECEIVE}_{u,v}(p)$ action is enabled. Both cases contradict the assumption that $s$ is a final state. ∎

Notice also that since $\mathcal{N}^+$ is a UIOA, any suffix of an execution of $\mathcal{N}^+$ that begins with a timed state is also an execution of $\mathcal{N}$. We will assume this implicitly when we apply some of the lemmas and claims described below.

## B.2 Basic Properties of links

The first lemma states that once a link is drop-free (see Definition 5.6.7), it remains drop-free.

**Lemma B.2.1** *For any $(u,v)$ and any transition $(s,\pi,\tilde{s})$ of $\mathcal{N}^+$, if $s \in F_{u,v}$, then $\tilde{s} \in F_{u,v}$.*

**Proof:** If $s.freeq_u[v] = true$ then since $s \in F_{u,v}$, $s.Q_{u,v} = nil$. Thus the only action that can cause $\tilde{s}.Q_{u,v} \neq nil$ is a $\text{SEND}_{u,v}(p)$ event that also sets $\tilde{s}.freeq_u[v] = false$. If $s.freeq_u[v] = false$ then the the only action that can cause $\tilde{s}.freeq_u[v] = true$ is a $\text{FREE}_{u,v}$ event which in turn can only occur if $s.Q_{u,v} = \tilde{s}.Q_{u,v} = nil$. ∎

The next lemma says that $(u,v)$ becomes drop-free after the first action that sends a packet to $C_{u,v}$, or the first signal from $C_{u,v}$ that it is free.

**Lemma B.2.2** *For any edge $(u,v)$ and any execution $\alpha$, $(u,v)$ is drop-free in all states of $\alpha$ that follow a $\text{FREE}_{u,v}$ or a $\text{SEND}_{u,v}(p)$ action.*

**Proof:** After a $\text{FREE}_{u,v}$ action, $Q_{u,v} = nil$, and after a $\text{SEND}_{u,v}(p)$ action, $freeq_u[v] = false$. Hence if $\tilde{s}$ follows either of these actions in $\alpha$, then $s_i \in F_{u,v}$. The lemma follows from the stability of $F_{u,v}$ (Lemma B.2.1). ∎

Once a link is drop-free we can be sure that all packets sent on the link will be delivered.

## B.3 Time Bounds for Correction Phases

Recall the definition of a correction phase on a link from 5. This section builds up to a few important time bounds that are useful in the main body of the proof.

Only the last two lemmas are important. The remaining claims are only useful in proving the last two lemmas. In the following claims, when we say that an event occurs within time $t$ in execution $\alpha$, we mean that the event occurs within time $t$ of the first state in $\alpha$.

The first claim says that a packet on a link will be delivered in $t_l$ time.

**Claim B.3.1** *For all executions $\alpha$ and any $(u,v)$, if in the initial state, $Q_{u,v} = p \neq nil$, then a* $\text{RECEIVE}_{u,v}(p)$ *event occurs within $t_l$ time units.*

The next claim says that a sender will know that a link is free in $2t_l$ time. Also after this period, the link will be drop-free. (Intuitively it takes $t_l$ time units to deliver any packet on the link, and $t_l$ time units to deliver the FREE signal to the sender.)

**Claim B.3.2** *For all executions $\alpha$ and any $(u,v)$, within $2t_l$ time units there is a state $s$ such that $s.freeq_u[v] = true$ and $(u,v)$ is drop-free in $s$.*

**Proof:** By Claim B.3.1 in $t_l$ time units after the first state of $\alpha$, there must be a state $s$, such that $s.Q_{u,v} = nil$. Now we have two cases. First, suppose within $t_l$ time units after $s$ a $\text{SEND}_{u,v}(p)$ event occurs. Then in the state preceding this event, $freeq_u[v] = true$ and $Q_{u,v} = nil$ and we are done. On the other hand if the first case does not occur, then $Q_{u,v} = nil$ for $t_l$ time units after $s$. Hence in $t_l$ time units after $s$ a $\text{FREE}_{u,v}$ action must occur, resulting in a state $\tilde{s}$ such that $\tilde{s}.freeq_u[v] = true$. By Claim B.2.2, $(u,v)$ is drop-free in $\tilde{s}$. ∎

The next claim bounds the time before a response will be sent. Note that in the code, responses are sent continuously even without waiting for a request. This helps avoid deadlock in arbitrary initial states. We rely on the matching process to weed out meaningless responses.

**Claim B.3.3 Response Time:** *For all executions $\alpha$ and any leader edge $(u,v)$, there must be some* $\text{SEND}_{v,u}(p_{resp})$ *action in $\alpha$ that occurs in $3t_n + 4t_l$ time units.*

**Proof:** From Claim B.3.2, within $2t_l$ time, there is a state $s$ such that $s.freeq_v[u] = true$. If $queue_v[u]$ is empty for $t_n$ time units after $s$, then a $\text{SEND}_{v,u}(p_{resp})$ will occur in this period. If not, $queue_v[u]$ becomes non-empty within time $t_n$ after $s$, which enables the sending of a data packet. In this case, within $2t_n$ time units after $s$, a $\text{SEND}_{v,u}(p_{data})$ will occur, resulting in a state $\tilde{s}$ in which $turn_v[u] = response$. From Claim B.3.2, in 2 $t_l$ time units after $\tilde{s}$, there is a state $s'$ such that $s'.freeq_v[u] = true$ and $turn_v[u] = response$. Finally, in $t_n$ time units after $s'$, a $\text{SEND}_{v,u}(p_{resp})$ will occur. ∎

The next claim bounds the time before a new phase will start on a link.

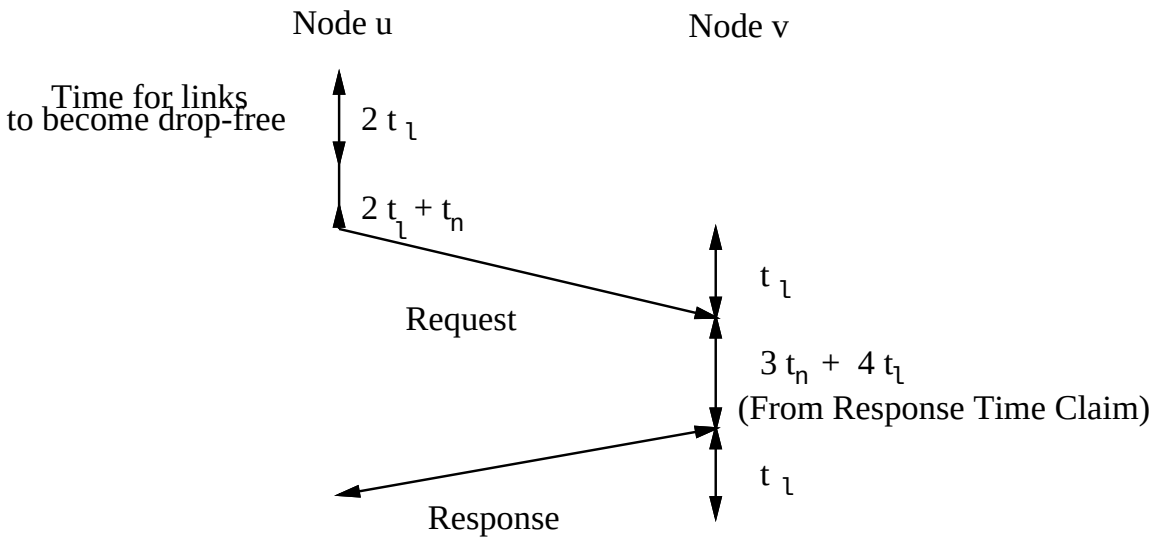



Figure B.1: Summary of the proof of Claim B.3.5.

**Claim B.3.4** *For all executions $\alpha$ and any leader edge $(u, v)$, within $2t_n + 2t_l$ time units there is a state in which $phase_u[v] = true$.*

**Proof:** Assume $phase_u[v] = false$ in the initial state of $\alpha$ or we are done trivially. From Claim B.3.2, within $2t_l$ time, there is a state $s$ such that $s.freeq_u[v] = true$. If $queue_u[v]$ is empty for $t_n$ time units after $s$, then a $\text{SEND}_{u,v}(p_{req})$ action occurs within this period, resulting in a state in which $phase_u[v] = true$. On the other hand, suppose that $queue_u[v]$ becomes non-empty within time $t_n$ after $s$. Then a $\text{SEND}_{u,v}(p_{data})$ event will occur within $2t_n$ time units after $s$, resulting in a state in which $phase_u[v] = true$. ∎

The next claim bounds how long it can take for a phase on a link to complete once it has started.

**Claim B.3.5** *For all $\alpha$ and any leader edge $(u, v)$, within $4t_n + 10t_l$ time units there is a state in which $phase_u[v] = false$.*

**Proof:** The proof is summarized in Figure B.1. We assume the claim is false, so $phase_u[v] = true$ for $4t_n + 10t_l$ time units after the start of $\alpha$. First, from Claim B.3.2 in $2t_l$ time, there is a state $s$ such that both $(u, v)$ and $(v, u)$ are drop-free in $s$. Thus

any any requests and responses that are sent in states after $s$ will be delivered to the corresponding receiver.

From Claim B.3.2, within $2t_l$ time after $s$, there is a state $s'$ such that $s'.freeq_u[v] = true$. Thus a a $\text{SEND}_{u,v}(p_{req})$ action is enabled in $s'$. Thus within $t_n$ time units after $s'$ a $\text{SEND}_{u,v}(p_{req})$ action occurs. In $t_l$ time units after this $\text{SEND}_{u,v}(p_{req})$ action, a $\text{RECEIVE}_{u,v}(p_{req})$ action occurs resulting in a state say $s''$. Also in all states after $s''$, $count_v[u] = count_u[v]$. Then by Claim B.3.3 in $3t_n + 4t_l$ after $s''$ a $\text{SEND}_{v,u}(p_{resp})$ event occurs with $p_{resp}.count = count_u[v]$. Finally, another $t_l$ time units after this event, a $\text{RECEIVE}_{v,u}(p_{resp})$ event occurs with $p_{resp}.count = count_u[v]$. In the state immediately after this event, $phase_u[v] = false$. This contradicts the assumption that $phase_u[v] = false$ for $4t_n + 10t_l$ time units after the start of $\alpha$. ∎

All the previous claims are only useful in proving Lemma 5.6.5 and Lemma 5.6.6. We now prove Lemma 5.6.6.

**Proof:** First assume that $l(u,v) = u$. Now we consider two cases. If $s_0.phase_u[v] = false$ then by Lemma B.3.4, within $2t_n + 2t_l$ time after $s_0$, there is a state $s_i$ in which $s_i.check_u[v] = true$. However, the action before $s_i$ must be a $\text{SEND}_{u,v}(p_{data})$ event. (The only other action that sets $check_u[v] = true$ is a $\text{SEND}_{u,v}(p_{req})$ action, but such an action is not enabled if $queue_u[v]$ is non-empty and $phase_u[v] = true$.) If $s_0.phase_u[v] = true$, then by Lemma B.3.5 within $4t_n + 10t_l$ after $s_0$ we reach a state in which $phase_u[v] = false$ and we are back to the previous case. Thus in both cases, a $\text{SEND}_{u,v}(p_{data})$ occurs within $t_p$ after $s_0$.

Now suppose that $l(u,v) = v$. Once again we have two cases. Suppose $s_0.turn_u[v] = data$. From Claim B.3.2, within $2t_l$ time after $s_0$, there is a state $s_i$ such that $s_i.freeq_v[u] = true$ and $s_i.turn_u[v] = data$. Thus within $t_n$ time after $s_i$ a $\text{SEND}_{u,v}(p_{data})$ event occurs. If $s_0.turn_u[v] \neq data$, then by Lemma B.3.3, we see that in at most $3t_n + 4t_l$ after $s_i$, a $\text{SEND}_{v,u}(p_{resp})$ event occurs that will result in state in which $turn_u[v] = data$. Then we are back to the first case. In both cases, a $\text{SEND}_{u,v}(p_{data})$ occurs within $t_p$ after $s_0$. ∎

## B.4 Proof that Clean Edges remain Clean

The following claim is useful in determining how and when the counter values stored in the links and the receiver can change. Recall that we used *undefined* when there is

no counter value stored on a link.

**Claim B.4.1** *In any execution, and for any leader edge $(u,v)$, the only action that can add a new element that is not undefined to countset$(u,v)$ is a* $\text{SEND}_{u,v}(p_{req})$ *action.*

**Proof:** The only other actions that affect *countset*$(u,v)$ are a $\text{RECEIVE}_{u,v}(p_{req})$ , $\text{SEND}_{v,u}(p_{resp})$ and $\text{RECEIVE}_{v,u}(p_{resp})$ , none of which can add any new values other than *undefined*. ∎

A nice property is that once an edge becomes clean, it remains clean. This is stated in Claim 5.6.11. We now prove Claim 5.6.11.

**Proof:** We assume that leader edge $(u,v)$ is clean in $s$ and we show that it remains clean in $\tilde{s}$. We will refer to the five predicates in the definition of a clean link by their numbers. The stability of the first predicate follows directly from Lemma B.2.1.

If $s.phase_u[v] = false$ and $\tilde{s}.check_u[v] = false$, then no $\text{SEND}_{u,v}(p_{req})$ action can occur which by Lemma B.4.1 is the only action that can change *countset*$(u,v)$. Thus the second predicate holds and the remaining three hold trivially. If $s.phase_u[v] = false$ and $\tilde{s}.check_u[v] = true$, then $\pi$ must be a $\text{SEND}_{u,v}(p_{req})$ or a $\text{SEND}_{u,v}(p_{data})$ event which will cause Predicate 3 to hold. Also by Predicate 2 in state $s$, $s.count_u[v] \neq s.respcount(u,v)$ and $s.count_u[v] \neq s.count_v[u]$. Since a $\text{SEND}_{u,v}(p_{req})$ or a $\text{SEND}_{u,v}(p_{data})$ event does not change $count_u[v]$ or *respcount*$(u,v)$ or $count_v[u]$, Predicates 4 and 5 hold in $\tilde{s}$; also Predicate 2 holds trivially.

If $s.phase_u[v] = true$ and $\tilde{s}.check_u[v] = false$, then $\pi$ must be a $\text{RECEIVE}_{v,u}(p_{resp})$ action and $s.count_u[v] = respcount(u,v)$ and $\tilde{s}.count_u[v] = (s.count_u[v] + 1) \bmod 4$. Then we can use the fact that Predicates 3 and 4 hold in $s$ to infer that Predicate 2 holds in $\tilde{s}$. Also Predicates 3, 4, and 5 hold trivially in $\tilde{s}$. If $s.phase_u[v] = true$ and $\tilde{s}.check_u[v] = true$, then the only actions of interest are $\text{SEND}_{u,v}(p_{req})$ (which makes Predicate 3 true and leaves the others true), $\text{SEND}_{v,u}(p_{resp})$ (which makes Predicate 4 true and leaves the others true) and a $\text{RECEIVE}_{u,v}(p_{req})$ . For a $\text{RECEIVE}_{u,v}(p_{req})$ , we can use the fact that Predicate 3 holds in $s$ to infer that Predicate 4 holds in $\tilde{s}$. All other Predicates hold trivially. Note that $\pi$ cannot be a $\text{SEND}_{u,v}(p_{data})$ event because $s.phase_u[v] = true$. ∎

## B.5 How Links Become Quiet: Detailed Proofs

Recall the definition of a quietlink from Chapter 5. For $(u,v)$ to be quiet, we want to show not only that $L_{u,v}$ holds by the end of the second $(u,v)$ phase but also that no more "reset" actions can occur after this point so that $L_{u,v}$ will remain true. This motivates the following definitions of a reset transition.

**Definition B.5.1** *Consider any transition $(s,\pi,\tilde{s})$ of $\mathcal{N}^+$. We say that this transition is a reset transition at node $u$ with respect to node $x$ if:*

- $s.mode_u[x] = reset$ *AND*
- *Either $\pi$ is a* $\text{SEND}_{u,x}(p_{resp})$ *or $\pi$ is a* $\text{RECEIVE}_{x,u}(p_{resp})$ *and* $l(u,x) = x$.

Intuitively, a reset transition at node $u$ causes the local reset function $f$ to be applied to the state of $u$.

We start by proving 5.6.17, which is the stability condition for a quiet link. **Proof:** We will show that each of the five predicates used to define a quiet link (Definition 5.6.15) holds in $\tilde{s}$. We will refer to the four predicates using the numbers given in Definition 5.6.15. The first predicate holds in $\tilde{s}$ because of Claim 5.6.11.

Consider the second predicate. We wish to show that $(\tilde{s}|u, \tilde{s}|(u,v), \tilde{s}|(v,u), \tilde{s}|v) \in L_{u,v}$. We know that the second predicate holds in $s$. We also know that if $\pi$ is any action of $\mathcal{N}$, then the second predicate holds in $\tilde{s}$ as well because $L_{u,v}$ is a closed predicate by definition. The only other transitions of $\mathcal{N}^+$ that can affect $L_{u,v}$ are reset transitions at $u$ or $v$.

Suppose $(s,\pi,\tilde{s})$ is a reset transition at $u$. From the fact that the third predicate holds in $s$, it is easy to see that $(s,\pi,\tilde{s})$ cannot be a reset transition at node $u$ with respect to $v$. Suppose $(s,\pi,\tilde{s})$ is a reset transition at node $u$ with respect to some neighbor $x \neq v$. Let $e$ denote the leader edge corresponding to edge $(u,x)$. Then $e$ is not quiet in state $s$. From the hypothesis, we infer that $\{u,x\} \not< \{u,v\}$. Thus by the stability condition for local correctability, and since $\tilde{s}|u = f(s|u,x)$, we infer that the second predicate holds in $\tilde{s}$ even if $(s,\pi,\tilde{s})$ is a reset transition at $u$. A similar argument works if $(s,\pi,\tilde{s})$ is a reset transition at $v$.

We can infer that the third predicate holds in $\tilde{s}$ from the fact that the third, fourth and fifth predicates hold in $s$. The only actions to consider are $\text{RECEIVE}_{v,u}(p_{resp})$ and a $\text{RECEIVE}_{u,v}(p_{req})$ . We can infer that the fourth predicate holds in $\tilde{s}$ from the fact that the third and fourth predicates hold in $s$. The only action to consider is a $\text{SEND}_{u,v}(p_{req})$ action.

Consider the fifth predicate. Let $X$ be the predicate $Q_{u,v} = p_{resp}$ and $p_{resp}.count = count_u[v]$. The only actions that can change the truth or falsity of predicate $X$ are $\text{SEND}_{v,u}(p_{resp})$ or $\text{RECEIVE}_{v,u}(p_{resp})$ . Suppose that either $\pi$ is a $\text{SEND}_{v,u}(p_{resp})$ action and $s.count_u[v] \neq s.count_v[u]$ OR $\pi$ is a $\text{RECEIVE}_{v,u}(p_{resp})$ action. Then $X$ will become false in $\tilde{s}$ and the fifth predicate holds trivially. On the other hand, suppose $\pi$ is a $\text{SEND}_{v,u}(p_{resp})$ action and $s.count_u[v] = s.count_v[u]$ Thus $X$ will become true in $\tilde{s}$. However, in this case, from the definition of a clean link, $s.Q_{u,v} \notin P_{data}$. Thus $\pi$ will make the fifth predicate hold in $\tilde{s}$.

Thus, suppose that $X$ is true in $s$ and also in $\tilde{s}$. The only other actions to consider are actions that change the state of $s|u$. We know that if $\pi$ is any action of $\mathcal{N}$, then the fifth predicate holds in $\tilde{s}$ as well because $L_{u,v}$ is a closed predicate by definition. The only other transition of $\mathcal{N}^+$ that can affect the fifth predicate is a reset transition at $u$. Suppose $(s, \pi, \tilde{s})$ is a reset transition at node $u$ with respect to some neighbor $x \neq v$. Let $e$ denote the leader edge corresponding to edge $(u, x)$. Then $e$ is not quiet in state $s$. From the hypothesis, we infer that $\{u, x\} \not< \{u, v\}$. Thus by the stability condition for local correctability, and since $\tilde{s}|u = f(s|u, x)$, we infer that the fifth predicate holds in $\tilde{s}$ even if $(s, \pi, \tilde{s})$ is a reset transition at $u$. ∎

Next we show the required liveness condition: that a leader edge $(u, v)$ will become quiet in bounded time if all leader edges less than $(u, v)$ are already quiet.

In the rest of this section, we prove Lemma 5.6.18, which describes how and when links become quiet. A quick overview of this section can be obtained by skimming Claim B.5.4, Figure B.2, Definition B.5.7 and the last three lemmas in the section.

We start by proving some more detailed properties of the local snapshot and reset protocols during a clean phase. For all the claims and lemmas in this section, we fix an execution $\alpha$ of $\mathcal{N}^+$ and a leader edge $(u, v)$. When we refer to a $(u, v)$ phase we mean a $(u, v)$ phase in $\alpha$. When we refer to the code, we mean the code in Figures 5.6 and 5.7.

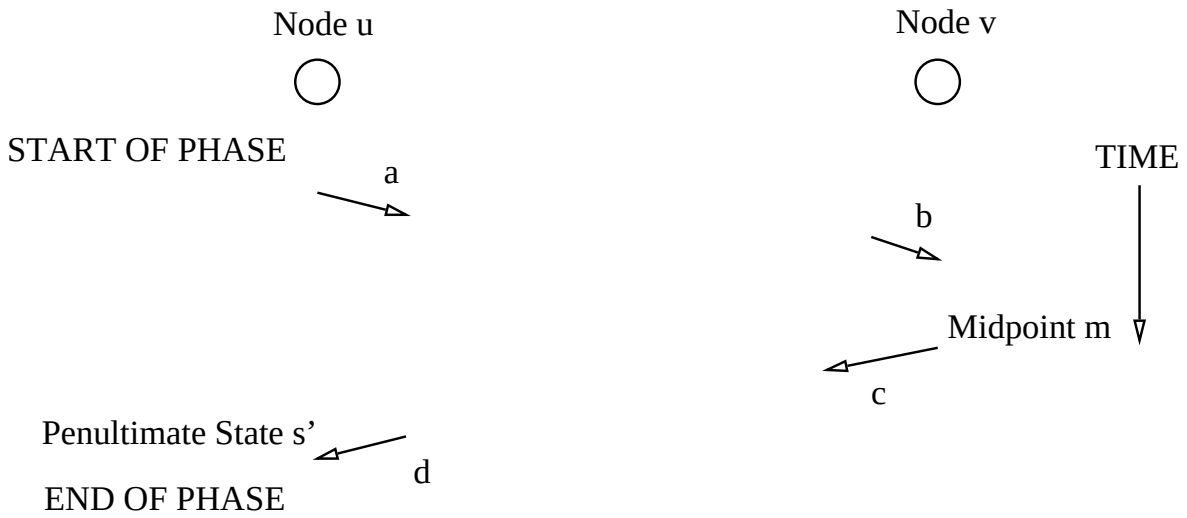



Figure B.2: More detailed structure of a clean phase

Since neither the mode or the counter at the leader can change except at the end of a phase it makes sense to talk of the mode and counter value of a phase.

**Definition B.5.2** *Consider any clean $(u,v)$ phase $\mathcal{P}$ with first state $r$. Then we use $mode(\mathcal{P})$ to denote $r.mode_u[v]$ and $count(\mathcal{P})$ to denote $r.count_u[v]$.*

**Claim B.5.3** *For all states $s$ in a clean $(u,v)$ phase except the last state, $s.count_u[v] = count(\mathcal{P})$ and $s.mode_u[v] = mode(\mathcal{P})$.*

**Proof:** From the code, the values of $count_u[v]$ and $mode_u[v]$ can only change at the end of the phase. ∎

Figure B.2 shows the structure of a clean phase in more detail for some leader edge $(u,v)$. The next claim formalizes the intuition behind Figure B.2 and defines two important states within a phase: the *midpoint* and the *penultimate state*.

**Claim B.5.4** *Any clean $(u,v)$ phase $\mathcal{P}$ contains the following four actions in the order shown:*

- *A* $\text{SEND}_{u,v}(p_{req})$ *action with $p_{req}.count = count(\mathcal{P})$ and $p_{req}.mode = mode(\mathcal{P})$.*

- *A* $\text{RECEIVE}_{u,v}(p_{req})$ *action with* $p_{req}.count = count(\mathcal{P})$ *and* $p_{req}.mode = mode(\mathcal{P})$.
- *Exactly one* $\text{SEND}_{v,u}(p_{resp})$ *action with* $p_{resp}.count = count(\mathcal{P})$ *and* $p_{resp}.node_state = m|v$, *where* $m$ *is the state immediately following this action. We will call* $m$ *the midpointphase!midpoint of the phase.*
- *Exactly one* $\text{RECEIVE}_{v,u}(p_{resp})$ *action with* $p_{resp}.count = count(\mathcal{P})$ *and* $p_{resp}.node_state = m|v$, *where* $m$ *is the midpoint of the phase. We will call the state immediately before this action, the penultimate statephase!penultimate state of the phase.*

**Proof:** The four actions are shown in Figure B.2 as $a$, $b$, $c$, and $d$ respectively. The midpoint and penultimate states are also marked.

Informally, the claim follows from two facts. First since both $(u,v)$ and $(v,u)$ are drop-free in all states of $\mathcal{P}$ any $\text{SEND}_{u,v}(p)$ action in $\mathcal{P}$ must be followed by a state in which $Q_{u,v} = p$. (i.e., any packet sent on channel $C_{u,v}$ will be stored in the link.). A similar statement holds for link $(v,u)$. The second fact is that, by definition, at the start of a clean phase, $count_u[v] \notin countset(u,v)$. Thus at the start of the phase, there are no potentially confusing requests or responses numbered with the value of the counter at the sender. This ensures, for instance, that when a response $p_{resp}$ is received at $u$ with $p_{resp}.count = count_u[v]$, then $p_{resp}$ was sent in the phase. Similarly it ensures that when a request $p_{req}$ is received with $p_{req}.count = count_u[v]$, then the request was sent in the phase.

The formal argument is quite tedious. We start by proving the last statement in the claim and then working backwards to prove the other three. Let's sketch the first part of the argument.

We know that $\mathcal{P}$ can only end with a $\text{RECEIVE}_{v,u}(p_{resp})$ with $p_{resp}.count = count_u[v]$. Thus in the state $s'$ before this action, $respcount(u,v) = count_u[v]$. But in the first state of $\mathcal{P}$, we know that since $\mathcal{P}$ is clean, $respcount(u,v) \neq count_u[v]$. Thus there must have been a $\text{SEND}_{v,u}(p_{resp})$ action in $\mathcal{P}$. Also, there can be only one such action in $\mathcal{P}$: any earlier $\text{SEND}_{v,u}(p_{resp})$ action would have caused the phase to have ended earlier; no later $\text{SEND}_{v,u}(p_{resp})$ action can occur because $Q_{v,u} \neq nil$ for all remaining states in the phase except the last state. If the state after the only $\text{SEND}_{v,u}(p_{resp})$ action is $m$, then from the code $p_{resp}.node_state = m|v$. Similar arguments can be used to show the remainder of the claim. ∎

In Figure B.2, in all states following $b$ and before action $c$, $mode_v[u] = mode_u[v]$. Informally, this is because $mode_v[u]$ is set to the mode carried in the request packet and cannot change until action $c$ occurs at which point $mode_v[u]$ is changed to *snapshot*. Then $mode_v[u]$ remains at this value till the end of the phase. Formally:

**Claim B.5.5** *Consider a clean $(u,v)$ phase $\mathcal{P}$ with midpoint $m$. Then in the state before $m$, $mode_v[u] = mode_u[v]$ and in all remaining states in $\mathcal{P}$, $mode_v[u] = snapshot$.*

**Proof:** After the first $\text{RECEIVE}_{u,v}(p_{req})$ action in $\mathcal{P}$, $mode_v[u]$ becomes equal to $mode_u[v]$. The value of $mode_v[u]$ can only change due to $\text{RECEIVE}_{u,v}(p_{req})$ actions and $\text{SEND}_{v,u}(p_{resp})$ actions. However, future $\text{RECEIVE}_{u,v}(p_{req})$ actions in $\mathcal{P}$ cannot change $mode_v[u]$ because from the definition of a clean link, $p_{req}.count = count_v[u]$. After the first (and only, see Claim B.5.4) $\text{RECEIVE}_{v,u}(p_{resp})$ action, $mode_v[u] = snapshot$ and remains at that value till the end of the phase. ∎

The following definition is convenient:

**Definition B.5.6** *The leader edge corresponding to an edge $(u,v)$ is $(u,v)$ if $l(u,v) = u$ and $(v,u)$ if $l(v,u) = v$.*

In order to guarantee correction or checking at end of a $(u,v)$ phase, we need restrictions on the states of links adjacent to either $u$ or $v$. Otherwise, concurrent checking/correction on these adjacent links may invalidate the checking/correction done in the $(u,v)$ phase.

**Definition B.5.7** *We will call a $(u,v)$ phase $\mathcal{P}$* well-behaved *if $\mathcal{P}$ is clean and for all states $s$ in the phase and for all $\{w,x\} < \{u,v\}$, the leader edge corresponding to $\{w,x\}$ is quiet in $s$.*

Now in a well-behaved phase, the response contains the state of $v$ at the midpoint. However, the response is received in the penultimate state. Despite the fact that the state of $v$ recorded in the response is "old", the recorded state is stil useful in the following precise sense.

**Claim B.5.8 Phase Invariant:** *Consider any clean $(u,v)$ phase $\mathcal{P}$ with midpoint $m$ and penultimate state $s'$. Then for all states $r$ in the interval $[m,s']$, the following predicates hold in $r$:*

- $r.Q_{v,u} = p_{resp}$ *and* $p_{resp}.count = count(\mathcal{P})$.
- *For any* $y$*: If* $(y, nil, nil, p_{resp}.node_state) \in L_{u,v}$ *then* $(y, nil, nil, r|v) \in L_{u,v}$

**Proof:** The first part of the claim follows directly from Claim B.5.4. We establish the second predicate by induction on the length of the interval $[m, s']$. First, this predicate is true in $m$ because $m.Q_{v,u} = p_{resp}$ and $p_{resp}.node_state = m|v$ by Claim B.5.4.

Next, assume the second predicate holds in some state $r'$ before $r$ and consider the transition $(r', \pi, r)$. The only actions of interest are the actions that change the state of $v$. If $\pi$ is any action of $\mathcal{N}$, then the second predicate holds in $r$ as well because $L_{u,v}$ is a closed predicate. The only other transitions of $\mathcal{N}^+$ that can affect the state of $v$ are reset transitions at $v$. We first see that $(r', \pi, r)$ cannot be a reset transition at node $v$ with respect to $u$. Suppose $(r', \pi, r)$ is a reset transition at node $v$ with respect to some neighbor $x \neq u$. Let $e$ denote the leader edge corresponding to $\{v, x\}$. Then $e$ is not quiet in state $r'$. But since the phase is well-behaved, we infer that $\{v, x\} \not\leq \{u, v\}$. Thus by the stability condition for local correctability, and since $r|v = f(r'|v, x)$, we infer that the second predicate holds in $r$. ∎

We can now state two lemmas that explain why the local snapshot and reset procedures work correctly. Let us call a snapshot phase a well-behaved phase $\mathcal{P}$ such that $mode(\mathcal{P}) = snapshot$. Similarly a reset phase is a well-behaved phase $\mathcal{P}$ such that $mode(\mathcal{P}) = reset$. The first lemma states that if $L_{u,v}$ does not hold at the end of a snapshot $(u, v)$ phase, then at the end of the phase $mode_u[v] = reset$. This ensures that the next $(u, v)$ phase will be a reset phase.

**Lemma B.5.9** *For any snapshot* $(u, v)$ *phase* $\mathcal{P}$ *with last state* $s$ *the following is true. If* $(s|u, s|(u, v), s|(v, u), s|v) \notin L_{u,v}$*, then* $s.mode_u[v] = reset$.

**Proof:** Let $s'$ be the penultimate state immediately before $s$ in $\mathcal{P}$. The action just before $s$ is a $\text{RECEIVE}_{v,u}(p_{resp})$ action. Since $s'$ is the penultimate state of a snapshot phase, $mode_u[v] = snapshot$ and $Q_{v,u} = p_{resp}$ and $p_{resp}.count = count_u[v]$ in $s'$.

Thus, from the code, $s|u = s'|u$ and $s|v = s'|v$ (i.e., the basic states of nodes $u$ and $v$ remain unchanged after the $\text{RECEIVE}_{v,u}(p_{resp})$ action.). Next, from the fact that $(u, v)$ is clean in $s'$, we deduce that $s|(u, v) = s'|(u, v) = nil$. Also, after a $\text{RECEIVE}_{v,u}(p_{resp})$ action $s|(v, u) = nil$.

Thus if $(s|u, s|(u,v), s|(v,u), s|v) \notin L_{u,v}$ then $(s'|u, nil, nil, s'|v) \notin L_{u,v}$. But by the phase invariant (Claim B.5.8) if $(s'|u, nil, nil, s'|v) \notin L_{u,v}$ then we can be sure that $(s'|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$. Thus if $(s|u, s|(u,v), s|(v,u), s|v) \notin L_{u,v}$ then $(s'|u, nil, nil, p_{resp}.node_state) \notin L_{u,v}$. Hence, from the code of the RECEIVE$_{v,u}(p_{resp})$ action it is easy to see that $s.mode_u[v] = reset$. ∎

The second lemma states that at the end of a $(u,v)$ reset phase, $(u,v)$ is quiet.

**Lemma B.5.10** *For any reset $(u,v)$ phase $\mathcal{P}$ with last state $s$, $(u,v)$ is quiet in $s$.*

**Proof:** Let $s'$ be the state immediately before $s$ in $\mathcal{P}$. The action just before $s$ must be a RECEIVE$_{v,u}(p_{resp})$ action. Since $s'$ is the penultimate state of a reset phase, $mode_u[v] = reset$ and $Q_{v,u} = p_{resp}$ and $p_{resp}.count = count_u[v]$ in $s'$. Let $m$ be the midpoint of $\mathcal{P}$ and $m'$ be the state immediately before $m$ in $\mathcal{P}$. By Claim B.5.5, $m'.mode_v[u] = m'.mode_u[v] = reset$, and hence $p_{resp}.node_state = f(m'|v,u)$.

Now from the correction property of a local reset function, $(f(s'|u,v), nil, nil, f(m'|v,u) \in L_{u,v}$. Since $p_{resp}.node_state = f(m'|v,u)$, $(f(s'|u,v), nil, nil, p_{resp}.node_state) \in L_{u,v}$. But by the phase invariant, Claim B.5.8, this implies that $(f(s'|u,v), nil, nil, s'|v) \in L_{u,v}$. But by similar arguments as in the proof of the previous lemma: $s|(u,v) = nil$, $s|(v,u) = nil$, and $s|v = s'|v$. However, since $s'.mode_u[v] = reset$, $s|u = f(s'|u)$. Together, these equations imply that $(s|u, s|(u,v), s|(v,u), s|v) \in L_{u,v}$.

Next, $s.mode_u[v] = s.mode_v[u] = snapshot$ by Claim B.5.3 and Claim B.5.5. Also, since $(u,v)$ is clean in state $s'$, if $s.Q_{u,v} = p_{req}$, then $p_{req}.count = s.count_v[u]$. Now it is easy to verify now that all five predicates used in the definition of a quiet link (Definition 5.6.15) hold in state $s$. ∎

We are now ready to prove Lemma 5.6.18. The proof is almost immediate from the last two lemmas. We recall the statement of Lemma 5.6.18. If every leader edge $(w,x) < (u,v)$ is quiet in some state $s_i$ of some execution $\alpha$ of $\mathcal{N}^+|C$, then $(u,v)$ is quiet in some state that occurs within $3 \cdot t_p$ of $s_i$ in $\alpha$.

**Proof:** First from the hypothesis and Lemma 5.6.17, every $(u,v)$ phase in $\alpha$ is well-behaved. Consider the first $(u,v)$ phase $\mathcal{P}$ in $\alpha$ that starts after state $s_i$. If $mode(\mathcal{P}) = reset$, then we know from Lemma B.5.10 that at the end of $\mathcal{P}$, $(u,v)$ is quiet. If $mode(\mathcal{P}) = snapshot$, then we know from Lemma B.5.9 that at the end of $\mathcal{P}$, $mode_u[v] = reset$. Consider the next $(u,v)$ phase in $\alpha$, say $\mathcal{P}'$. Since $mode_u[v]$ only changes at the

end of a phase, $mode(\mathcal{P}') = reset$. Thus by Lemma B.5.10, $(u, v)$ is quiet at the end of $\mathcal{P}'$.

Hence there is some state $s_j$ that occurs before the end of the second $(u, v)$ phase that follows $s_i$ in $\alpha$ and such that $(u, v)$ is quiet in $s_j$. Thus by the phase rate lemma, Lemma 5.6.5, $s_j$ must occur within $3t_p$ time units after $s_i$ in $\alpha$. ∎

# Appendix C

# The AAG reset protocol

In this chapter, we describe why three phases were used in the original AAG protocol [AAG87] and also describe the changes that were required to convert the AAG protocol into the reset protocol described in Chapter 7. We also show that the mating relation provided by the AAG protocol is not transitive.

## C.1 Why three phases are used in the AAG protocol

The AAG reset protocol [AAG87] is much more conservative than the Simple Reset Protocol about allowing a node to to return to *Ready* mode.

The point of all the conservatism in the AAG protocol is as follows. The use of three phases ensures that if reset requests stop being made, all nodes will eventually return to *Ready* mode. The AAG protocol makes this guarantee even in dynamic networks. The additional rules the AAG protocol uses to work in dynamic networks are remarkably simple. Suppose a link from node $u$ to node $v$ fails. If node $v$ is node $u$'s parent (in the abort tree), node $u$ takes over as the root of the abort tree; if node $u$ is expecting an ack from node $v$, node $u$ assumes it has got an ack from $v$. Finally, when a link from $u$ to $v$ comes up, nothing special is done!

Here is an intuitive explanation of why three phases are used in the AAG protocol.

Each execution $\alpha$ of the reset protocol can be used to induce *work intervals* at nodes. Let us call a work interval at a node $u$, a maximal subsequence of $\alpha$ during which node $u$ is not in *Ready* mode. Thus from the point of view of node $u$, each execution $\alpha$ can be divided into work intervals followed by *Ready* intervals (intervals during which $u$ is in *Ready* mode.) Now each work interval at node $u$ can be considered to be "caused" by an ABORT packet from a neighbor $v$ (in which case we say that the work interval at $u$ has as its parent a work interval at $v$) or by a reset request (in which case, we say that the work interval at $u$ has no parent). Thus, starting from an execution $\alpha$, we can assign each work interval at each node to a tree of work intervals.

We claim that each tree of work intervals has a height of at most $n$, where $n$ is the number of nodes in the graph. This follows if we can show that any work interval tree can contain at most one work interval from any given node. This brings us to the crucial observation. Consider any work interval tree $T$ in $\alpha$. Let $I_r$ be the work interval corresponding to the root. *The use of three phases guarantees us that there is some state in $I_r$ that is contained in all the work intervals contained in $T$.* This is the state in which $r$ sends READY packets to all its children. But that implies that a given node $u$ cannot have two distinct work intervals in $T$ because these two disjoint work intervals do not share a common state. Hence the height of $T$ is at most $n$.

Thus, each root interval can "cause" at most one work interval at any node $u$. But each root interval corresponds to either a reset request (or to a link failure in the case of dynamic networks). Thus the number of work intervals at a node is at most the number of reset requests (plus the number of topology changes in the case of dynamic networks). Thus if the number of reset requests (and topology changes) is finite, there will only be a finite set of work intervals at each node. Next, it is possible to show that each work interval terminates in $O(n)$ time by using induction on the height of the abort tree. We combine the last two observations to prove the causality property – in finite time after all reset requests (and topology changes) stop, all nodes return to *Ready* mode.

It is interesting to return to the simple reset protocol (SRP). Consider an execution $\alpha$ of SRP that begins in a "bad state" as shown in Figure 7.5. Suppose we construct work interval trees from execution $\alpha$. The result is that we get a single work interval tree of infinite height. Each work interval ends in finite time but there are an infinite number of work intervals at each node!

## C.2 Overview of the changes required for stabilization

In a non-stabilizing setting (e.g. [AAG87], buffers like $buffer_u[v]$ are modelled by unbounded queues. However, just as in the case of links, stabilizing reset protocols must use bounded size queues if they are to stabilize in bounded time. Our solution is for $v$ to use a single buffer to store messages from $u$, and to require that $v$ sends a special $\Sigma - Ack$ packet whenever it removes a message from the buffer. This special packet is not needed in the original AAG protocol that uses unbounded queues.

The first step in making this protocol stabilizing is to make it locally checkable. A clear problem with the AAG protocol is that it will deadlock if in the initial state some *parent* edges form a cycle. As in stabilizing spanning tree algorithms [AKY90, AG90], we mend this flaw by maintaining a distance variable at each node, such that a node's distance is one greater than that of its parent. Specifically, distance is initialized to 0 upon reset request, and its accumulated value is appended to the abort packets. Thus we encode an abort packet as a tuple (ABORT, $d$), where $d$ is a distance.

Next, we list all the local predicates that are necessary to ensure correct operation of the Reset protocol. Note that a significant advantage of our approach is that all we have to do is to prove that all local predicates eventually hold. Once we do that we can rely on the correctness of the original protocol. However, since a rigorous correctness argument for the original protocol did not exist (as far as we knew), we produced a correctness argument anyway.

To ensure that the local predicates are closed predicates, we use the heuristic of removing unexpected packet transitions. Recall from Chapter 6, that to do this we add checks before processing any packet arriving at the node. We check whether this packet could have possibly been sent when the link subsystem (on which the packet arrived) is in a good state. Two checks we have added (which were not needed in the AAG protocol) are: when an ACK arrives, a node checks whether the ACK expected before processing it; when a READY packet arrives, a node accepts it only if it is in *Converge* mode and the packet has come from the node's parent.

The use of the distance variable introduces a new problem. Since the distance field has a maximum value, say $n'$, we have to consider the case of a node $u$ that receives an (ABORT, $n'$) packet from neighbor $v$. If $u$ is in *Ready* mode, $u$ cannot simply accept $v$

as its parent and set its own distance to be one greater than $n'$ since $n'$ is the maximum value. Instead, in our code, $u$ pretends that it has received a reset request before the (ABORT, $n'$) packet. Thus $u$ will first become a root and send ABORT packets to all its neighbors; then it will send back an ack to $v$. We call this a spurious reset request action.

Since the new action we have added is just a combination of two existing actions in the original protocol, the new action preserves all local predicates and consistency conditions of the original protocol. However, it does slightly complicate the proof of termination (and hence of the causality condition). Clearly, if the reset protocol can keep producing such "spurious reset requests" the protocol may never terminate even if all real reset requests stop. Luckily, it is easy to show that within linear time after all local predicates of the original protocol hold, no (ABORT, $n'$) packets can be received. Thus spurious reset requests stop in linear time, and the termination proof is only slightly more complicated.

Next we have to design a local correction action for links, that is taken when a violation of the predicates is detected. The main difficulty about designing a correcting strategy is making it local, i.,e., to ensure that when we correct a link we do not affect the correctness of any other link. An interesting heuristic for this purpose is to notice that protocols designed for dynamic networks (like [AAG87]) had to deal with link failures and recovery. Now when a link fails and then immediately recovers, the original protocol must have established the local predicates for the link that failed without affecting the correctness of the other links. Thus the local correction procedure we use is essentially identical to the combined code in [AAG87] that is invoked when a link fails and recovers. This seems to be a powerful heuristic in general.

## C.3 Mating Relation is not Transitive

Consider the reset protocol described in Chapter 7 (or the protocol described in [AAG87]) when all local predicates hold. We use the scenario shown in Figure C.1 to show that the mating relation between signal intervals at neighboring nodes is not transitive. Thus the reset protocol may cause inconsistent states of the user protocol during initial signal intervals. However, the mating relation between final intervals is indeed transitive. The original paper [AAG87] showed an example in which the

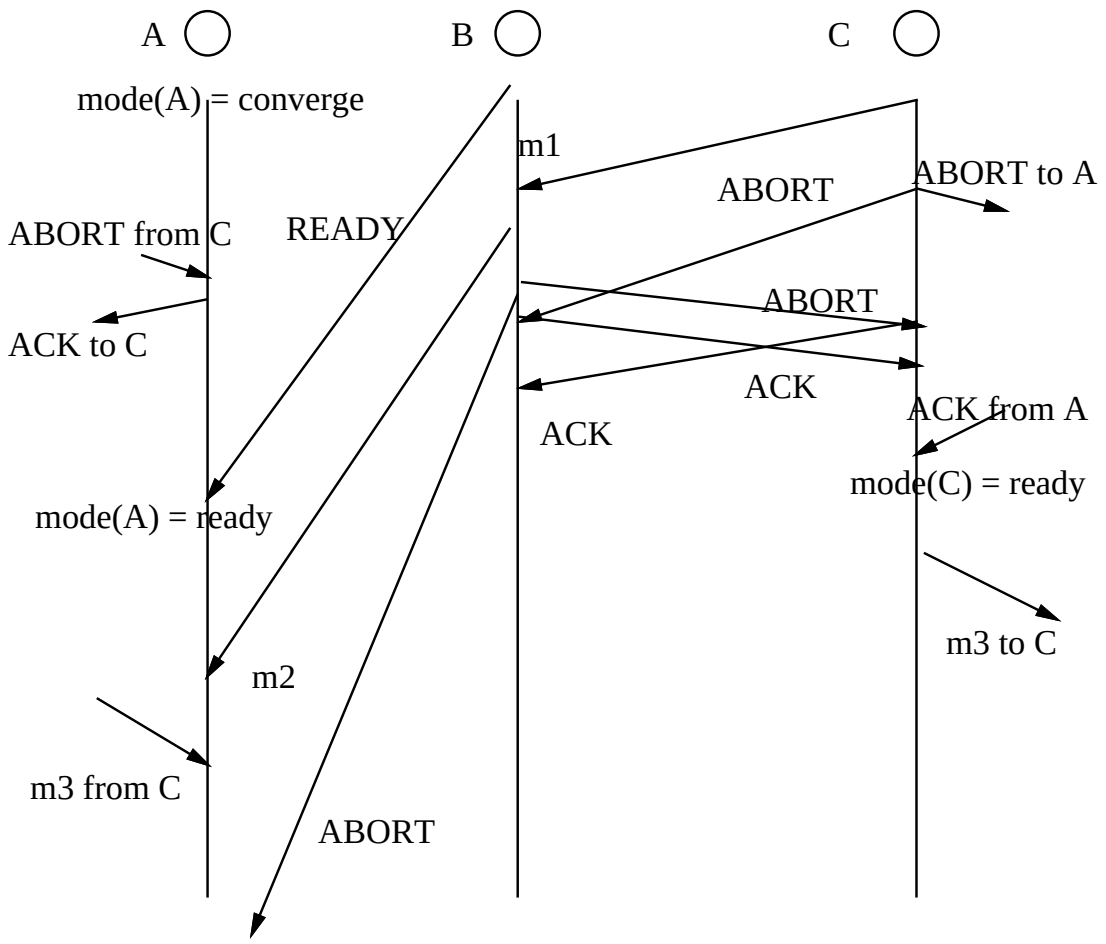



Figure C.1: Counterexample to Show that Mating Relation is not Transitive

mating relation was not transitive; however, the original paper used an optimization (in which ABORT packets were only sent on edges on which messages had either been sent or received in the last signal interval). Thus it was not clear whether the problem was due to the optimization. In fact, the simple counterexample in [AAG87] does not work as soon as we remove the optimization.

However, we show in Figure C.1 that there is a (more complicated) counterexample.

In Figure C.1 there are three nodes $A$, $B$ and $C$ that are connected in a cycle (not shown). The vertical axis represents time; time increases as we go downwards. Because this is a cycle, there is a link between $A$ and $C$. When $A$ sends a packet to $C$ (or vice versa), we show this by showing an arrow leaving $A$ and going off the left

of the page. We then depict the packet receipt at $C$ by an arrow coming in to $C$ from the right end of the page. Thus in the second event at $C$, $C$ sends an ABORT packet to $A$ which is received as the second event at $A$.

In the initial state, $mode(A) = Converge$ and $parent_A = B$. Thus $A$ is waiting for a READY message from $B$ that is sent out at the start of the execution. Assume that after $B$ has sends out the READY packet, $B$ does a SIGNAL event immediately and so $mode(B) = Ready$ at the start of the execution. We assume that $mode(C) = Ready$ and that there is no other packet in transit in the initial state. Clearly this is a valid initial state.

The first event at $C$ is that $C$ sends a user message (say $m1$) which is received by $B$ and then delivered. Immediately after this $B$ sends a user message $m2$ to $A$. Shortly after this, a reset request occurs at both $B$ and $C$. This causes $B$ to send an ABORT packet to $A$ and $C$, and $C$ to send an ABORT packet to $A$ and $B$. The ABORT packet sent from $C$ to $A$ arrives before the READY packet arrives at $A$. Thus $A$ will send an ACK back. Similarly $B$ sends an ACK immediately back to $C$. The net result is that by the time $A$ receives its READY packet, $C$ is already in READY mode. We also assume $C$ has done a signal event immediately after going to READY mode. Next the message $m2$ sent by $B$ arrives at $A$ and $C$ sends a third user message $m3$ which is received by $A$. All this happens before the ABORT packet in transit from $B$ arrives at $A$.

The net result is as follows. $C$ sends two messages $m1$ and $m3$ in two different signal intervals at $C$, call them $S_C$ and $S'_C$. Message $m1$ is received in a signal interval say $S_B$ at $B$. Node $B$ then sends a message $m2$ in signal interval $S_B$ that is received in a signal interval (say $S_A$ at $A$.) Finally the message sent by $C$ in $S'_C$ is also received in $S_A$.

By the definition of the mating relation, messages can only be received from a unique mate at a neighbor. Thus $S_C$ mates to $S_B$ and $S_B$ mates to $S_A$. Also $S'_C$ mates to $S_A$. If the mating relation were transitive, $S_C$ would mate to $S'_C$ which is impossible.

The significance of this counterexample is that arguments like "making a reset request guarantees a fresh start of the application protocol" do not work. If the application is doing any general form of checking, it may detect an inconsistent state and keep making reset requests, leading to non-termination. For example, if we were to use the global snapshot protocol due to [KP90] to check the application and then

use our stabilizing reset, the protocol may never terminate! Our stabilizing reset is still useful in a number of cases. For instance, (Chapter 8) when used in conjunction with a local snapshot protocol that checks for closed local predicates. In any case, termination of the resulting protocol requires careful argument.

Another significant thing about the counterexample is that it shows that our pairwise definition of a mating relation between neighbors is probably the only statement that one can make about the non-final intervals of a reset protocol. The original specification of [AAG87] used a state-based specification. It seems hard to specify this weak safety property of non-final intervals in terms of states as opposed to using an external behavior specification. In either, case [AAG87] only needed to specify the behavior in the final interval. For stabilizing applications, we *must* specify the behaviors in non-final intervals.

It is interesting to note that the reset protocol of Arora and Gouda [AG90] (after adaptation to a message passing model) is likely to ensure a transitive mating relation. This is because it does a reset protocol on a tree and only the root (effectively) sends out the quivalent of ABORT packets. Of course, our stabilizing protocol can be first used to construct a spanning tree, after which we do another reset which is guaranteed to work from a single root outwards.

# Appendix D

# Proofs for Reset Protocol in Chapter 7

## D.1 Proving that the Local Predicates of the Reset Protocol are Closed

Recall the definition of $L_{u,v}$ in Definition 7.6.3.

Our strategy for showing that $L_{u,v}$ is closed is as follows. If we had to consider every action we would have a large number of cases to consider. However, each transition only affects a small number of variables. Thus we first isolate the transitions that can affect each variable. This reduces the number of cases we have to consider.

First, we need to define the key transitions. Recall that the code in Figure 7.7 has certain code paths marked as $VR$, $VA$, $DA$, $IA$, $FA$, $RA$, and $RR$. We now define these more precisely. It is helpful to refer to the code of Figure 7.7 in understanding the following definitions.

A informal description of these transitions is as follows. A $VR$ (for *Valid Request*) transition is a reset request that causes a node to change its *mode* to *Abort*. A $VA$ (for *Valid Abort*) transition is the receipt of an (ABORT, $d$) packet with $d < n'$ that causes a node to change its *mode* to *Abort*. A $DA$ (for *Distance Invalid Abort*) transition is the receipt of an (ABORT, $n'$) packet that causes a node to change its *mode* to *Abort*.

An *IA* (for *Invalid Abort*) transition is the receipt of an (ABORT, $*$) packet that *does not* cause a node to change its *mode* to *Abort*. A *FA* (for *Final Ack*) transition is the receipt of an ACK packet that causes say node $u$ to send an ACK packet to its parent. It is not hard to see that the ack that was received must have been the last ack that node $u$ was waiting for. A *RA* (for *Root Ack*) transition is the receipt of an ACK packet at a root node that causes the root node to change its *mode* to *Ready*. A *RR* (for *Regular Ready*) transition is the receipt of an (READY) packet at a node that causes the node to change its *mode* to *Ready*. More carefully:

**Definition D.1.1** *We call a transition* $(s, a, s')$ *of* $\mathcal{R}$*:*

- *A VR transition at* $u$ *if* $a = \text{REQUEST}_u$ *and* $s.mode(u) = Ready$.
- *A VA transition at* $u$ *if* $a = \text{RECEIVE}_{*,u}(\text{ABORT}, d)$ *and* $d < n'$ *and* $s.mode(u) = Ready$.
- *A VA transition at* $u$ *with respect to* $v$ *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) = Ready$ *and* $d < n'$.
- *A DA transition at* $u$ *with respect to* $v$ *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) = Ready$ *and* $d = n'$.
- *An IA transition at* $u$ *with respect to* $v$ *if* $a = \text{RECEIVE}_{v,u}(\text{ABORT}, d)$ *and* $s.mode(u) \neq Ready$.
- *A FA transition at* $u$ *with respect to* $v$ *if* $a = \text{RECEIVE}_{*,u}(\text{ACK})$ *and* $s.mode(u) = Abort$ *and* $s.parent_u = v$ *and* $s'.mode(u) = Converge$.
- *A RA transition at* $u$ *if* $a = \text{RECEIVE}_{*,u}(\text{ACK})$ *and* $s.mode(u) = Abort$ *and* $s'.mode(u) = Ready$.
- *A RR transition at* $u$ *if* $a = \text{RECEIVE}_{*,u}(\text{READY})$ *and* $s.mode(u) \neq Ready$ *and* $s'.mode(u) = Ready$.

In the following lemma we will say that a Boolean condition $b$ is *established* by some transition $(s, a, s')$ if $b$ is *false* in $s$ but *true* in $s'$. Recall that we wish to prove that each $L_{u,v}$ is closed. The next lemma makes this job easier, by isolating the transitions that can establish various Boolean conditions used in the definition of $L_{u,v}$.

*We start with the observation that we do not need to consider the transition DA explicitly because a DA transition at u with respect to v can be simulated by two other transitions*: first a *VR* transition at $u$ followed immediately by an *IA* transition at $u$ with respect to $v$. Thus in the following lemmas and proofs we assume that the *DA* transition does not exist,

**Lemma D.1.2**

1. $ack_u[v]$ = *true can only be established by a VR or a VA transition at* $u$.
2. $A1(u,v)$ = *true can only be established by a VR or a VA transition at* $u$.
3. $A2(u,v)$ = *true can only be established by a VA transition at* $v$ *with respect to* $u$.
4. $A3(u,v)$ = *true can only be established by a IA or a FA transition at* $v$ *with respect to* $u$.
5. $A2(u,v)$ = *false can only be established by a FA transition at* $v$ *with respect to* $u$.
6. $ack_u[v]$ = *false can only be established by a transition* $(s,a,s')$ *such that* $s.ack_u[v]$ = *true and* $a = \text{RECEIVE}_{v,u}(\text{ACK})$.
7. $parent_u = v$ *can only be established by a VA transition at* $u$ *with respect to* $v$.
8. $mode(u)$ = *Converge can only be established by a FA transition at* $u$ *with respect to some neighbor* $x$.
9. $mode(u) \neq$ *Converge can only be established by a RR transition at* $u$.
10. $C1(u,v)$ = *true can only be established by a IA or a FA transition at* $u$ *with respect to* $v$.
11. $C2(u,v)$ = *true can only be established by a transition* $(s,a,s')$ *such that* $s.ack_v[u]$ = *true and* $a = \text{RECEIVE}_{u,v}(\text{ACK})$.
12. $C3(u,v)$ = *true can only be established by a RR or RA transition at* $v$.
13. $C2(u,v)$ = *false can only be established by a RR or RA transition at* $v$.

**Proof:** By inspection of the code. ∎

Notice that in the code we often enqueue packets to $queue_u[v]$. But since $queue_u[v]$ is finite (it only has room for 5 packets), this allows the possibility of a transition $(s, \pi, s')$ causing a packet to be dropped if $queue_u[v]$ is full in the previous state. The next lemma shows that packets will not be dropped if $L_{u,v}$ holds in $s$. We will tacitly assume this lemma in what follows without making explicit reference to it.

**Lemma D.1.3** *For any leader edge $(u, v)$ and any transition $(s, a, s')$ of $\mathcal{R}$, if $s \in L_{u,v}$. then in $s'$, $\mathcal{Q}$ holds. Also if as part of the code for $a$, a packet $p$ is enqueued on $queue_u[v]$ then $p$ will be added to the tail of $queue_u[v]$ in $s'$.*

**Proof:** To show $\mathcal{Q}$, we show that when a packet of a certain type is in $xqueue_u[v]$, then no action will enqueue a packet of the same type. The five types of packets to consider are (ABORT, $*$) packets, ACK packets, READY packets, $\Sigma$ – ACK packets and $\Sigma$-messages. The second part also follows from this claim because $xqueue_u[v]$ has room for five packets.

Suppose there is an (ABORT, $*$) packet in $xqueue_u[v]$ in $s$. Then (by $\mathcal{A}$) $s.mode(u) =$ *Abort* and hence this cannot be a *VA* or *VR* transitions at $u$. But these are the only transitions that can enqueue another (ABORT, $*$) packet to $xqueue_u[v]$ (see Lemma D.1.2, item 2).

Suppose there is an ACK packet in $xqueue_u[v]$ in $s$. Then, (by $\mathcal{B}$) $A1(v, u) =$ *false* and $A2(v, u) =$ *false* in $s$. Thus (by Lemma D.1.2, item 4), this cannot be a transition that can enqueue another ACK packet to $xqueue_u[v]$.

Suppose there is a READY packet in $xqueue_u[v]$ in $s$. Then (by $\mathcal{G}$) either $s.mode(u) =$ *Ready* or there is an ABORT in $xqueue_u[v]$ after the READY packet. In the former case, (by Lemma D.1.2, item 12) this this cannot be a transition that can enqueue another READY packet to $xqueue_u[v]$. In the latter case, (by $\mathcal{A}$) $s.ack_u[v] =$ *true*, and (by $\mathcal{B}$) $A3(u, v)$ is *false* in $s'$. Thus again (by Lemma D.1.2, item 12) this this cannot be a transition that can enqueue another READY packet to $xqueue_u[v]$.

Suppose there is a $\Sigma$ message in $xqueue_u[v]$ in $s$. Then (by $\mathcal{H}$) in $s$, $freem_u[v] =$ *false*. Thus even if $a = \text{SENDM}_{u,v}(m)$, the code will not enqueue $m$ on $xqueue_u[v]$.

Suppose there is a $\Sigma$ – ACK message in $xqueue_u[v]$ in $s$. Then (by $\mathcal{H}$), $buffer_u[v]$ must be empty and so the code cannot queue a $\Sigma$ – ACK on $xqueue_u[v]$. ∎

We now proceed to prove that each of the predicates from $\mathcal{A}$ to $\mathcal{H}$ are closed in a series of four lemmas: Lemma D.1.4, Lemma D.1.5, Lemma D.1.6, Lemma D.1.7, and Lemma D.1.8.

**Lemma D.1.4** *For any leader edge $(u,v)$ and any transition $(s,a,s')$ of $\mathcal{R}$, if $s \in L_{u,v}$. then in $s'$, $\mathcal{A}$ and $\mathcal{B}$ hold.*

**Proof:** We consider four cases:

- Suppose $ack_u[v] = \mathit{false}$ in $s$ but $ack_u[v] = \mathit{true}$ in $s'$. Then (by $\mathcal{A}$) $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in $s$. Also by Lemma D.1.2, the transition must be a *VA* or a *VR* transition at $u$ which causes $A1(u,v)$ to become *true* and leaves $A2(u,v)$ and $A3(u,v)$ as *false* in $s'$.

- Suppose $ack_u[v] = \mathit{false}$ in $s$ and $s'$. Then (by $\mathcal{A}$) $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in $s$. Also by Lemma D.1.2, items 1 and 2, $A1(u,v)$ cannot hold in $s'$ without making $ack_u[v] = \mathit{true}$ hold in $s'$. Also by Lemma D.1.2, item 3, $A2(u,v)$ cannot become *true* in $s'$ if $A1(u,v)$ is *false* in $s$. Also by Lemma D.1.2, item 4, $A3(u,v)$ cannot become *true* in $s'$ if $A1(u,v)$ and $A2(u,v)$ are *false* in $s$. Thus $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in $s'$.

- Suppose $ack_u[v] = \mathit{true}$ in $s$ but $ack_u[v] = \mathit{false}$ in $s'$. By Lemma D.1.2, item 6, $a = \text{RECEIVE}_{v,u}(\text{ACK})$ and so $A3(u,v)$ is true in $s$. By $\mathcal{B}$, $A1(u,v)$ and $A2(u,v)$ are *false* in $s$. Also, (by $\mathcal{Q}$) there is exactly one ACK packet in $s.xqueue_v[u]$ and hence $A1(u,v)$, $A2(u,v)$, and $A3(u,v)$ are *false* in $s'$.

- Suppose $ack_u[v] = \mathit{true}$ in $s$ and $s'$. Then (by $\mathcal{A}$) exactly one of $A1(u,v)$,$A2(u,v)$, or $A3(u,v)$ is true in $s$. If $A1(u,v)$ is *true* in $s$, then $A1(u,v)$ becomes *false* in $s'$ (by Lemma D.1.2, items 3 and 4) iff exactly one of $A2(u,v)$ or $A3(u,v)$ becomes *true* in $s'$. If $A2(u,v)$ is *true* in $s$, then $A2(u,v)$ becomes *false* in $s'$ (by Lemma D.1.2, item 5) iff $A3(u,v)$ becomes *true* in $s'$. Finally if $A3(u,v)$ is *true* in $s$, then $A3(u,v)$ cannot become *false* in $s'$ (by Lemma D.1.2, item 6) without causing $ack_u[v] = \mathit{false}$ in $s'$, a contradiction.

∎

**Lemma D.1.5** *For any leader edge $(u,v)$ and any transition $(s,a,s')$ of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{C}$ and $\mathcal{D}$ are true for $(u,v)$ in $s'$.*

**Proof:** We consider cases:

- Suppose $parent_u \neq v$ in $s'$. Then $\mathcal{C}$ and $\mathcal{D}$ hold trivially in $s'$. Suppose $parent_u \neq v$ in $s$ and $parent_u = v$ in $s'$. Then by Lemma D.1.2, item 7, this must be a $VA$ transition at $u$ with respect to $v$. Thus $s'.mode(u) = Abort$, Thus $A1(v,u)$ is *true* in $s$ and hence (by $\mathcal{A}$) $A3(v,u) =$ *false* in $s$ and hence $C1(u,v)$ is *false* in $s$ and hence in $s'$. Also, (by $\mathcal{A}$) $ack_v[u] =$ *true* in $s$ and $s'$ and hence $C2(u,v)$ is *false* in $s'$. Next (by $\mathcal{G}$ and $\mathcal{Q}$) we can infer that $C3(u,v)$ is *false* in $s$ and hence in $s'$. (This is because if $C3(u,v)$ were *true* in $s$ then (by $\mathcal{G}$) there must be a second (ABORT, $*$) packet in $xqueue_v[u]$ which would violate $\mathcal{Q}$.)

  Thus in the remaining cases we assume that $parent_u = v$ in $s$ and $s'$.

- Suppose $mode(u) \neq Converge$ in $s$ and $s'$. Then $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in $s$. Also by Lemma D.1.2, item 10, $C1(u,v)$ can become *true* in $s'$ only if $mode(u) = Converge$ in $s'$. [1] By Lemma D.1.2, item 11, $C2(u,v)$ can become *true* in $s'$ only if $C1(u,v)$ is true in $s$. Finally by Lemma D.1.2, item 12, $C3(u,v)$ can become *true* in $s'$ only if $C1(u,v)$ or $C2(u,v)$ is true in $s$. Thus $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in $s'$.

- Suppose $mode(u) \neq Converge$ in $s$ and $mode(u) = Converge$ in $s'$. Then (by $\mathcal{C}$) $C1(u,v)$, $C2(u,v)$, and $C3(u,v)$ are *false* in $s$. Also by Lemma D.1.2, item 8, $a$ is a $FA$ transition at $u$ with respect to $v$. Thus after this transition, $C1(u,v)$ becomes *true* in $s'$, and $C2(u,v)$ and $C3(u,v)$ remain *false* in $s'$.

- Suppose $mode(u) = Converge$ in $s$ and $mode(u) \neq Converge$ in $s'$. Then by Lemma D.1.2, item 9, $a$ is a $RR$ transition at $u$. Thus $C3(u,v)$ is *true* in $s$; hence (by $\mathcal{D}$) $C1(u,v)$ and $C2(u,v)$ are *false* in $s$ and $s'$. Also, (by $\mathcal{Q}$) there is exactly one READY packet in $xqueue_v[u]$ in $s$ and this is removed after the transition, and so $C3(u,v)$ is *false* in $s'$.

[1] Note that this cannot be an $IA$ transition at $u$ with respect to $v$ because otherwise $s.mode(u) = Abort$ which together with $s.parent_u = v$ would violate $\mathcal{A}$.

- Suppose $mode(u) = Converge$ in $s$ and $s'$. Then (by $\mathcal{C}$ and $\mathcal{D}$) exactly one of $C1(u,v)$,$C2(u,v)$, or $C3(u,v)$ is true in $s$. First suppose $C1(u,v)$ is *true* in $s$. Then (by $\mathcal{A}$) $ack_v[u] = true$ in $s$. Thus $C1(u,v)$ becomes *false* in $s'$ iff exactly one of $C2(u,v)$ or $C3(u,v)$ becomes *true* in $s'$. Also if $C1(u,v)$ remains *true* in $s'$, then $ack_v[u] = true$ in $s'$ by Lemma D.1.2, item 6. Thus by Lemma D.1.2, items 11 and 12, $C2(u,v)$ and $C3(u,v)$ cannot become *true* in $s'$.

  Next, notice that since we have assumed that $mode(u) = Converge$ in $s$, $C1(u,v)$ remains *false* in $s'$ (by Lemma D.1.2, item 10) if it is *false* in $s$. Suppose $C2(u,v)$ is *true* in $s$. Then $C2(u,v)$ becomes *false* in $s'$ (by Lemma D.1.2, item 13) iff $C3(u,v)$ becomes *true* in $s'$.

  Finally if $C3(u,v)$ is *true* in $s$, then $C3(u,v)$ cannot become *false* in $s'$ without causing $mode(u) = Ready$ in $s'$, a contradiction. Also by Lemma D.1.2, item 11 $C2(u,v)$ cannot become *true* in $s'$ since $C1(u,v)$ is *false* in $s$.

∎

**Lemma D.1.6** *For any leader edge $(u,v)$ and any transition $(s,a,s')$ of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{E}$ and $\mathcal{F}$ are true for $(u,v)$ in $s'$.*

**Proof:** First consider $\mathcal{F}$. Suppose there is no (ABORT, $*$) packet in $xqueue_u[v]$ in $s$. Then if an (ABORT, $*$) packet is in $xqueue_u[v]$ in $s'$, then this must be a (by Lemma D.1.2, item 2) *VR* or *VA* transition at $u$. However, after such a transition there is an (ABORT, $d$) packet in $xqueue_u[v]$, where $d = dist_u + 1$. Suppose there is a (ABORT, $d$) packet in $xqueue_u[v]$ in $s$ with $d = s.dist_u + 1$. Then (by $\mathcal{A}$) $s.mode(u) = Abort$. Now the only transitions that can change $dist_u$ are *VR* or *VA* transitions at $u$, but such transitions are not enabled in if $s.mode(u) \neq Ready$.

Now consider $\mathcal{E}$ and consider three cases.

- $parent_u \neq v$ in $s'$: then $\mathcal{E}$ holds trivially.
- $parent_u \neq v$ in $s$ but $parent_u = v$ in $s'$: Then by Lemma D.1.2, item 7, $a$ must be a RECEIVE$_{v,u}$(ABORT, $d$) event with $s.mode(u) = Ready$. But by $\mathcal{F}$ in $s$, $d = s.dist_v + 1$. Thus in $s'$, $dist_u = dist_v + 1$. and $mode(u) \neq Ready$.

- $parent_u = v$ in $s$ and $s'$. If in $s$, $C3(u,v)$ is *true*, then $C3(u,v)$ can become *false* in $s'$ if $a = \text{RECEIVE}_{v,u}(\text{READY})$, but in that case $s'.parent_u = nil$. If in $s$, $dist_u = dist_v + 1$ and $mode(v) \neq Ready$, then this transition cannot be a $VR$ or $VA$ transition at $v$ or $u$. But only such transitions can change either $dist_v$ or $dist_u$. Also if $s'.mode(v) = Ready$, then this must be a $RR$ or $RA$ transition at $v$ which results in $C3(u,v)$ becoming true in $s'$.

∎

**Lemma D.1.7** *For any leader edge $(u,v)$ and any transition $(s,a,s')$ of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{G}$ is true for $(u,v)$ in $s'$.*

**Proof:** Suppose in $s$ there is no $p$ in $xqueue_u[v]$ such that $p$ is either a READY packet or $p$ is a $\Sigma$-message or $p$ is a $\Sigma - \text{ACK}$. Then if there is such a packet in $s'$, then, by the code, $s.mode(v) = Ready$ and we are done. Suppose in $s$ there is a $p$ in $xqueue_u[v]$ such that $p$ is either a READY packet or $p$ is a $\Sigma$-message or $p$ is a $\Sigma - \text{ACK}$. Then by $\mathcal{G}$ either there is an $(\text{ABORT}, *)$ packet in $xqueue_u[v]$ or $s.mode(u) = Ready$. Consider the first case. Since the channel from $u$ to $v$ is FIFO, the $(\text{ABORT}, *)$ packet cannot be removed from $xqueue_u[v]$ in $s'$ without also removing packet $p$. Consider the second case. Then if $s'.mode(v) \neq Ready$ then this transition must be a $VA$ or $VR$ transition at $u$ which would result in adding an $(\text{ABORT}, *)$ packet to the end of $xqueue_u[v]$ in $s'$.
∎

**Lemma D.1.8** *For any leader edge $(u,v)$ and any transition $(s,a,s')$ of $\mathcal{R}$, if $s \in L_{u,v}$, then $\mathcal{H}$ is true for $(u,v)$ in $s'$.*

**Proof:** We consider all the actions $a$ that can affect this predicate. For each action considered, a symmetrical argument holds for the action with $u$ and $v$ interchanged.

If $a = \text{SENDM}_{u,v}(m)$ and $s.freem_u[v] = false$, then message $m$ is dropped and there is no change to the concerned variables. If $a = \text{SENDM}_{u,v}(m)$ and $s.freem_u[v] = true$, then there is no $\Sigma$-message in $M_{u,v}$ in $s$ and no $\Sigma - \text{ACK}$ in $s.xqueue_v[u]$. Also, (by $\mathcal{Q}$and $\mathcal{H}$) there is at most one $(\text{ABORT}, *)$, ACK, or READY packet in $queue_u[v]$ in $s$. Since $queue_u[v]$ can store five packets, after this event $m$ is placed in $queue_u[v]$ and $s'.freem_u[v] = false$ and and there is no $\Sigma - \text{ACK}$ in $s.xqueue_v[u]$.

If $a = \text{RECEIVE}_{u,v}(m)$, $m \in \Sigma$, then $m$ is enqueued in $buffer_v[u]$ and none of the concerned variables change. Note that by $\mathcal{H}$, $buffer_v[u]$ is empty in $s$. If $a = \text{RECEIVE}_{u,v}(\text{ABORT}, *)$ and $s.buffer_v[u]$ is empty in $s$, then there is no change to the concerned variables. Suppose $a = \text{RECEIVE}_{u,v}(\text{ABORT}, *)$ and $s.buffer_v[u]$ contains a message $m$ in $s$ or $a = \text{RECEIVEM}_{u,v}(m)$. Then (by $\mathcal{H}$) in $s$, $freem_u[v] = false$, there is no $\Sigma - \text{ACK}$ in $xqueue_v[u]$, and no other message in $M_{u,v}$ besides $m$ in $buffer_v[u]$. Then in $s'$, all variables remain unchanged except that $buffer_v[u]$ becomes empty and a $\Sigma - \text{ACK}$ is added to $s.xqueue_v[u]$.

Similarly, if $a = \text{RECEIVE}_{v,u}(\Sigma - \text{ACK})$, then by $\mathcal{H}$in $s$, $freem_u[v] = false$, there is no $\Sigma$ message in $M_{u,v}$ and exactly one $\Sigma - \text{ACK}$ in $xqueue_v[u]$. The result is that the $\Sigma - \text{ACK}$ is removed from $xqueue_v[u]$ and $freem_u[v]$ becomes $true$ in $s'$. ∎

## D.2 Proving that the Reset Protocol Behaviors are Timely, Causal, and Consistent

We will show that every behavior of $\mathcal{R}|L$ is timely and causal and satisfies the consistency property. We will do so in the next five subsections. First, we prove a series of useful preliminary lemmas. In the second subsection, we prove that every behavior is timely. In the third subsection, we prove that every behavior satisfies the consistency property. In the fourth subsection, we prove that every behavior is causal. Finally, we tie everything together and show that every behavior $\beta$ of $\mathcal{R}|L$ is in $RP$.

We will assume this claim implicitly in what follows.

### D.2.1 Useful Claims and Lemmas for Reset Protocol

The first lemma is the important Termination Lemma that states that the *mode* will become *Ready* in $O(n)$ time.

**Lemma D.2.1 Termination Lemma** *Consider an execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. For any state $s_i$ there is another state $s_j$ that occurs within $O(n)$ time after $s_i$ and such that $s_j.mode(u) = Ready$.*

**Proof:** A formal argument can be made based on the intuitive "proof" given in Section 7.7.1. We omit it here. ∎

The next lemma is the Signal Lemma. The lemma states that once the *status* of a node $u$ is *off* (recall that this means that either $signalbit_u = true$ or $mode_u \neq Ready$) then a $\text{SIGNAL}_u$ event is guaranteed to occur in linear time.

**Lemma D.2.2 Signal Lemma:** *For any execution $\alpha$ of $\mathcal{R}|L$, if $s_i.status(u) = off$ then a $\text{SIGNAL}_u$ event occurs within $O(n)$ time after $s_i$.*

**Proof:** Suppose $s_i.mode(u) = Ready$ and $s_i.signalbit_u = true$. Then it is easy to see from the code that a $\text{SIGNAL}_u$ action is enabled and will remain enabled until it occurs in constant time after $s_i$.

The only other possibility is that $s_i.mode(u) \neq Ready$. Let $s_k$ be the first state after $s_i$ such that $s_k.mode(u) = Ready$. We know from Lemma D.2.1 that such a state exists and $s_k$ occurs within $O(n)$ time after $s_i$. Then $s_{k-1}.mode(u) \neq Ready$ and $s_k.mode(u) = Ready$. Thus from the code it is easy to see that $s_k.signalbit = true$. Now we are back to the first case and hence a $\text{SIGNAL}_u$ event must occur in constant time after $s_k$. ∎

The next claim is a variation of the Signal Lemma. It states that the *status* of a node $u$ cannot change from *off* to *on* until the next $\text{SIGNAL}_u$ event.

**Claim D.2.3** *For any execution $\alpha$ of $\mathcal{R}|L$, if $s_i.status(u) = off$ and $s_k.status = on$ and $k > i$, then there is a $\text{SIGNAL}_u$ event between $s_i$ and $s_k$ in $\alpha$.*

**Proof:** From the code, the only way $status(u)$ can change from *off* to *on* is by a $\text{SIGNAL}_u$ event. Note that any action that changes $mode(u)$ to *Ready* also sets $signalbit_u$ to *true*, which leaves $status(u)$ unchanged. ∎

Next we show that any packet queued on the outbound queue for a link will be delivered in constant time. This follows from the fact that the outbound queue has a size of at most 4.

**Claim D.2.4** *Consider any pair of neighbors $u,v$ and any execution $\alpha$. If there is a packet $p$ in $xqueue_u[v]$ in some state $s_i$ of $\alpha$, then in constant time after $s_i$ there is a $\text{RECEIVE}_{u,v}(p)$ event. (i.e., any packet in either the outbound queue for a link or on the link itself is delivered within constant time.)*

**Proof:** We know from Lemma 5.6.6 that the packet at the head of $queue_u[v]$ is placed in $Q_{u,v}$ (i.e., is placed in the channel) in $t_p$ time and that any packet in $Q_{u,v}$ is delivered in $t_l$ time. The lemma follows since $queue_u[v]$ has a size of at most 4, and $t_p$ is a constant. ∎

Next, we show that within constant time the *signalbit* variable at a node becomes *false*.

**Claim D.2.5** *Consider any node $u,v$ and any execution $\alpha$. If $signalbit_u$ = true in some state $s_i$ of $\alpha$, then in constant time after $s_i$ there is a state in which $signalbit_u$ = false.*

**Proof:** This follows because if $signalbit_u$ = *true* in $s_i$, the $\text{SIGNAL}_u$ event is enabled and $signalbit_u$ will remain *true* unless the $\text{SIGNAL}_u$ event occurs (see code). Thus since each $\text{SIGNAL}_u$ action is in a separate class, a $\text{SIGNAL}_u$ action will occur in constant time after $s_i$, resulting in a state (see code) in which $signalbit_u$ = *false*. ∎

The next claim (which is used to show the consistency property) states the following. Suppose there is some interval in which the *mode* of a node $u$ is *Ready* at the start and end of the interval but is not *Ready* somewhere within the interval. Consider any neighbor $v$ of $u$. Then there must be some point within the interval during which an ABORT packet arrives at $v$; also at this point any messages in transit from $u$ to $v$ have been "flushed" out.

**Claim D.2.6** *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any three states $s_i$, $s_j$ and $s_k$ in $\alpha$ such that $i < j < k$ and $s_i.mode(u)$ = Ready, $s_j.mode(u) \neq$ Ready and $s_k.mode(u)$ = Ready. Then there is some $j'$ such that $i < j' < k$ and $a_{j'}$ is a $\text{RECEIVE}_{u,v}(\text{ABORT},*)$ action and $s_{j'}.M_{u,v}$ does not contain any $\Sigma$-message.*

**Proof:** Let $s_l$ be the first state after $s_i$ in which $s_i.mode(u) \neq$ *Ready*. Such a state must exist by hypothesis and it must be that $l \leq j$. Thus by the code, $A3(u,v)$ is true in $s_l$ (i.e., there is an abort packet in $xqueue_u[v]$ in $s_l$). Let $s_{j'}$ be the first state after $s_l$ in which $A3(u,v)$ is *false* (i.e., the first state after $s_l$ in which the abort packet is delivered). We know from Claim D.2.4 that such a state exists. Also, we know from $\mathcal{A}$, that in the interval $[s_l, s_{j'}]$, $mode(u) \neq$ *Ready*. Thus $j' < k$. Also $a_{j'}$ must

be a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ event and following such an event it is easy to see from the code that $mode(v) \neq Ready$ and that $buffer_v[u]$ is empty. Also we know that in the interval $[s_l, s_{j'}]$, since $mode(u) \neq Ready$ no $\Sigma$-message was added to $M_{u,v}$. Also any $\Sigma$-message in $xqueue_u[v]$ in $s_l$ must have been removed from $xqueue_u[v]$ before $s_{j'}$ because $xqueue_u[v]$ is a FIFO queue. Thus $s_{j'}.M_{u,v}$ does not contain any $\Sigma$-message. ∎

The next claim (which is also used to show the consistency property) is a mild corollary of the previous claim. Suppose there is some interval in which $mode(u) = Ready$ at the start of the interval, and $u$'s signal bit is false in the interval, and $u$ does a signal at the end of the interval. Then the *mode* of $u$ must be *Ready* at the end of the interval and must have been not *Ready* somewhere within the interval. Thus the previous claim applies, along with its consequences.

**Claim D.2.7** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any state $s_i$ such that $s_i.mode(u) = Ready$ and another state $s_{i'}$, $i' \geq i$, such that $s_{i'}.signalbit_u = false$. Suppose that after $s_{i'}$ there is a $\text{SIGNAL}_u$ action $a_k$. Then there is some $j$ where $i < j < k$ such that:*

- *$a_j$ is a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action.*
- *$s_j.status(v) \neq on$*
- *$s_j.M_{u,v}$ does not contain any $\Sigma$-message.*

**Proof:** In the state just before $a_k$, $signalbit_u = true$ but in $s_k$, $signalbit_u = false$. Let $s_l$ be the first state before $s_{k-1}$ in which $signalbit_u = false$. Also $l \geq i'$ because $s_{i'}.signalbit_u = false$. Thus from the code it must be that $s_l.mode(u) \neq Ready$ and $s_{l+1}.mode(u) = Ready$. The lemma follows by using Claim D.2.6 to the three states $s_i$, $s_l$ and $s_{l+1}$ and by observing that in any state that follows a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action, $mode(v) \neq Ready$ and hence $status(v) \neq on$. ∎

Notice that the requirements of the previous claim are satisfied if $status(u) = on$ at the start of the interval and there is a signal at $u$ at the end of the interval. We state this corollary to the previous claim as a separate claim as it is used often.

**Claim D.2.8** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any state $s_i$ such that $s_i.status(u) = on$. Suppose that after $s_i$ there is a* $\text{SIGNAL}_u$ *action $a_k$. Then there is some $j$ where $i < j < k$ such that:*

- *$a_j$ is a* $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ *action.*
- *$s_j.status(v) \neq on$*
- *$s_j.M_{u,v}$ does not contain any $\Sigma$-message.*

**Proof:** Follows from Claim D.2.7. ∎

The next three claims are all used to prove the timeliness property.

Consider any two neighboring nodes $u$ and $v$. The next claim states that if the *status* of both $u$ and $v$ is *on* for a sufficiently large constant, then $u$ must deliver a free event (indicating that $u$ is willing to accept a new message to be sent to $v$) within this time.

**Claim D.2.9** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any state $s_i$. Then in constant time after $s_i$ either a* $\text{FREEM}_{u,v}$ *occurs or there is a state $s_j$ such that either $s_j.status(u) = off$ or $s_j.status(v) = off$.*

**Proof:** Suppose that $status(u) = on$ and $s_j.status(v) = on$ for $c$ time after $s_i$, where $c$ is a large enough constant to make the following argument work. By $\mathcal{H}$, in state $s_i$ either:

- $freem_u[v]$ is *true*.
- $xqueue_v[u]$ contains a $\Sigma - \text{ACK}$.
- $M_{u,v}$ contains a $\Sigma$-message.

In the first case, assuming $c$ is large enough, $status(u) = on$ for a constant time after $s_i$ which causes a $\text{FREEM}_{u,v}$ to occur in constant time after $s_i$.

In the second case, if there is a $\Sigma - \text{ACK}$ in $xqueue_v[u]$ then by Claim D.2.4 within constant time a $\text{RECEIVE}_{v,u}(\Sigma - \text{ACK})$ event occurs which causes $freem_u[v]$ to become *true*, leaving us in the first case.

For the third case, we can use an argument similar to the second case to show that in constant time after $s_i$, a RECEIVE$_{u,v}(m)$ event occurs which causes $m$ to be placed in $buffer_v[u]$. Assuming again that $c$ is large enough this implies that in constant time after $s_i$ either a RECEIVEM$_{u,v}(m)$ event or a RECEIVE$_{u,v}$(ABORT) event occurs. Either of these events will cause a $\Sigma$ – ACK to be placed in $xqueue_v[u]$, which brings us back to Case 2. ∎

Consider any two neighboring nodes $u$ and $v$. The next claim states that if the *status* of both $u$ and $v$ is *on* for a sufficiently large constant after a message is sent from $u$ to $v$, then the message will be delivered to $v$ within this time.

**Claim D.2.10** *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$.. Consider any safe* SENDM$_{u,v}(m)$ *action in $\alpha$. Then in constant time after this action either a* RECEIVEM$_{u,v}(m)$ *occurs or there is a state $s_j$ such that either $s_j.status(u) =$ off or $s_j.status(v) =$ off.*

**Proof:** Similar to proof of Claim D.2.9. Let us denote the safe SENDM$_{u,v}(m)$ event by $a_j$. Suppose that $status(u) = on$ and $s_j.status(v) = on$ for $c$ time after $a_j$, where $c$ is a large enough constant to make the following argument work. Then since $a_j$ is a safe send it is easy to see from the code that $freem_u[v]$ is *true* in the state before $a_j$. Also by our assumption, $status(u) = on$ in the state after $a_j$. So $m$ is placed in $queue_u[v]$ in the state after $a_j$. Thus by Claim D.2.4, in constant time after $a_j$, $m$ is placed in $buffer_v[u]$ and if $status(v) = on$ for constant time after this, a RECEIVEM$_{u,v}(m)$ event occurs. ∎

## D.2.2 Every Behavior of $\mathcal{R}|L$ is timely

We prove that every behavior $\beta$ of $\mathcal{R}|L$ is timely by showing each of the four properties in Definition 7.3.3. Corresponding to the four properties, we have four lemmas.

We will leave the first property of a timely behavior (i.e., that all messages received after $O(n)$ time are normal) to the end of this section. We start by showing the second property.

**Lemma D.2.11 Periodic Free Events:** *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$ and any state $s_j$ in $\alpha$. Then either a* FREEM$_{u,v}(m)$ *occurs*

*in constant time after $s_j$ or a* $\text{SIGNAL}_u$ *action occurs in $O(n)$ time after $s_j$ or or a* $\text{SIGNAL}_v$ *action occurs in $O(n)$ time after $s_j$.*

**Proof:** We know from Claim D.2.9 that for some constant $c$, in $c$ time after $s_j$ a $\text{FREEM}_{u,v}(m)$ occurs or there is a state $s_k$ such that either $s_k.status(u) = off$ or $s_k.status(v) = off$. In the first case, we are done. In the second case, we know from the Signal Lemma (Lemma D.2.2), that within $O(n)$ time after $s_k$ either a $\text{SIGNAL}_u$ or a $\text{SIGNAL}_v$ event occurs, ∎

Next, we prove that $\mathcal{R}|L$ satisfies the third property of a timely behavior.

**Lemma D.2.12 Timely Message Delivery:** *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any $a_j$ that is a safe* $\text{SENDM}_{u,v}(m)$ *action in $\alpha$. Then either a* $\text{RECEIVEM}_{u,v}(m)$ *occurs in constant time after $a_j$ or a* $\text{SIGNAL}_u$ *action occurs in $O(n)$ time after $a_j$ or or a* $\text{SIGNAL}_v$ *action occurs in $O(n)$ time after $a_j$.*

**Proof:** We know from Claim D.2.10 that for some constant $c$, in $c$ time after $a_j$ a $\text{RECEIVEM}_{u,v}(m)$ occurs or there is a state $s_j$ such that either $s_j.status(u) = off$ or $s_j.status(v) = off$. In the first case, we are done. In the second case, we know from the Signal Lemma (Lemma D.2.2) that within $O(n)$ time after $s_j$ either a $\text{SIGNAL}_u$ or a $\text{SIGNAL}_v$ event occurs, ∎

Next, we prove that $\mathcal{R}|L$ satisfies the fourth property of a timely behavior.

**Lemma D.2.13 Signals at a Node induce Signals at Neighbors***: Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$. There is some constant $c$ such that for every* $\text{SIGNAL}_u$ *event $a_j$ that occurs at time greater than $\alpha.start + c \cdot n$ there is a* $\text{SIGNAL}_v$ *event that occurs in linear time before or after $a_j$.*

**Proof:** First, within linear time of the start of $\beta$, there must be some state $s_h$ in which $mode(u) = Ready$ (by the Termination Lemma). In constant time after $s_h$, there must be some state $s_i$ in which $signalbit_u = false$ by Claim D.2.5. Consider any $\text{SIGNAL}_u$ action $a_j$ that occurs after state $s_i$. In the state before $a_j$, $signalbit_u = true$ but in state $s_i$, $signalbit_u = false$. Consider the first state $s_k$ before $s_j$ in which $signalbit_u = false$. By Claim D.2.5, $s_j$ occurs in constant time after $s_k$. Also since $s_{k+1}.signalbit_u = true$,

the code tells us that that $s_k.mode(u) \neq ready$. But we know that $s_h.mode(u) = Ready$. Consider the first state $s_l$ before $s_k$ in which $mode(u) = Ready$. By the Termination Lemma, $s_l$ occurs in linear time before $s_k$.

Thus we have identified two states $s_l$ and $s_k$, with $k \geq l$ such that both states occur before the $\text{SIGNAL}_u$ event $a_j$ and such that $s_l mode(u) = Ready$ and $s_k.signalbit_u = false$. By applying Claim D.2.7 to $s_l$, $s_k$ and $a_j$ we know that there is some state $s_m$ in the interval $[s_l, s_j]$ in which $status(v) \neq on$. Thus by the Signal Lemma, a $\text{SIGNAL}_v$ event occurs within linear time after $s_m$. But since $s_l$ occurs within linear time before $s_j$, the $\text{SIGNAL}_v$ event occurs in linear time before or after $a_j$. ∎

Finally, we prove that $\mathcal{R}|L$ satisfies the first property of a timely behavior. This requires more work and so we start with two claims. The claims are quite intuitive.

The first claim states that in any suffix of an execution, all except possibly the first packet received on a link are normal packets that have been sent in this execution.

**Claim D.2.14** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Let $s_i$ be the first state in $\alpha$ such that there is no $\Sigma$-message in $s_i.M_{u,v}$. Consider any $a_k$ that is a $\text{RECEIVE}_{u,v}(m)$ event that occurs after $s_i$ in $\alpha$. Then:*

- *There is a $\text{SEND}_{u,v}(m)$ action $a_j$ before $a_k$ and such that there are no $\text{RECEIVE}_{u,v}(*)$ events in between $a_j$ and $a_k$. We will call the earliest such $a_j$ the send corresponding to $a_k$ in $\alpha$.*
- *In state $s_j$ (i.e., the state immediately after $a_j$ in $\alpha$), $status(u) = on$ and in state $s_k$, $status(v) = on$*
- *In all states in the interval $[s_j, s_{k-1}]$, $m$ is in $M_{u,v}$.*

**Proof:** It is clear from the code that in $s_{k-1}$, $m$ must be in $buffer_u[v]$ and hence $m$ is in $M_{u,v}$. But since in $s_i$, $M_{u,v}$ is empty, there must be a $\text{SEND}_{u,v}(m)$ action between $s_i$ and $s_{k-1}$ which added $m$ to $M_{u,v}$. Let $a_j$ be the first such action that occurs before $a_k$. In the state immediately after $a_j$ by the code $status(u) = on$. Clearly, there cannot be a $\text{RECEIVE}_{u,v}(*)$ action between $a_j$ and $a_k$ because (from $\mathcal{H}$), $M_{u,v}$ contains at one most one message in any state. But a $\text{RECEIVE}_{u,v}(m)$ action is the only action that can remove $m$ from $M_{u,v}$ and hence $m$ is in $M_{u,v}$ in the interval $[s_j, s_{k-1}]$. Also from the code, in the state immediately after a $\text{RECEIVE}_{u,v}(m)$ action, $status(v) = on$. ∎

**Claim D.2.15** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$ and any state $s_i$ in $\alpha$. Then there is some state $s_m$ that occurs in $O(n)$ time after $s_i$ such that there is no $\Sigma$-message in $s_m.M_{u,v}$.*

**Proof:** From Lemma D.2.1, there is a state $s_j$ that occurs in $O(n)$ time after $s_i$ such that $s_j.mode(v) = Ready$. From Claim D.2.5, in constant time after $s_j$ there is a state $s_{j'}$ in which $signalbit_v = false$.

From Lemma D.2.1, there is a state $s_k$ that occurs in $O(n)$ time after $s_{j'}$ such that $s_k.mode(u) = Ready$. From Claim D.2.5, in constant time after $s_k$ there is a state $s_{k'}$ in which $signalbit_u = false$.

From Lemma D.2.11, some event $a_l$ occurs within $O(n)$ time after $s_{k'}$, where $a_l$ is either a $\text{FREEM}_{u,v}$ event or $\text{SIGNAL}_u$ or a $\text{SIGNAL}_v$ event. In the first case, we are done by predicate $\mathcal{H}$ which shows that $\text{FREEM}_{u,v}$ cannot occur unless $M_{u,v}$ is empty. In the second case $a_l$ is a $\text{SIGNAL}_u$ event, then by Claim D.2.7, there is some state $s_{l'}$ in the interval $[s_k, s_l]$ such that there is no $\Sigma$-message in $s_k.M_{u,v}$. If $a_l$ is a $\text{SIGNAL}_v$ event, then by Claim D.2.7 there is a state say $s_{l'}$ in the interval $[s_j, s_l]$ such that $s_{l'}.status(u) = off$. Thus by the Signal Lemma (Lemma D.2.2), a $\text{SIGNAL}_u$ event occurs within $O(n)$ time after $s_{l'}$, which brings us back to the second case. ∎

We can now show the first part of the consistency property (see Definition 7.3.5), that every message received after $O(n)$ time is normal; it is almost immediate from the last two claims.

**Lemma D.2.16 Normal Receipt of Messages**: *Consider any any execution $\alpha$ of $\mathcal{R}|L$ and any suffix $\gamma$ of $\alpha$ with first state $s_0$. There is some constant $c$ such that every every receive event that occurs at time greater than $s_0.time + c \cdot n$ in $\alpha$ is normal. Also if $a_j$ is any normal receive event and $a_i$ is the send corresponding to $a_j$, then $a_j$ occurs within $O(n)$ time after $a_i$.*

**Proof:** Consider any pair of neighbors $u, v$. By Claim D.2.15, there is some $s_j$ that occurs in $O(n)$ time after $s_0$ and such that there is no $\Sigma$-message in $s_j.M_{u,v}$. Let us call $j$ the quiescent index for link $(u, v)$. Further, let $k$ be the largest quiescent index over all possible links $(u, v)$. Clearly $s_k$ occurs in $O(n)$ time after $s_0$. Also by Claim D.2.14, all receive events that occur after $s_k$ in $\gamma$ are normal.

Also let $a_j$ be any normal receive event and $a_i$ be the send corresponding to $a_j$. By Claim D.2.14, in all states in the interval $[s_i, s_{j-1}]$, $m$ is in $M_{u,v}$. But by Claim D.2.15, there is some state $s_k$ that occurs in $O(n)$ time after $s_i$ such that there is no $\Sigma$-message in $s_k.M_{u,v}$. Thus $a_j$ occurs within $O(n)$ time after $a_i$. ∎

And now we come to the main result of this subsection:

**Lemma D.2.17** *Every behavior of $\mathcal{R}|L$ is timely.*

**Proof:** Immediate from Definition 7.3.3 and Lemmas D.2.16, D.2.13, D.2.11, and D.2.12. ∎

### D.2.3 Every Behavior of $\mathcal{R}|L$ satisfies the consistency property

We prove that every behavior $\beta$ of $\mathcal{R}|L$ satisfies the consistency property by showing each of the five properties in Definition 7.3.5. We have the following preliminary claim that is essential for proving the consistency property.

The claim states the following. Consider some interval and some node $u$ and suppose that the *status* of $u$ is *on* at the start of the interval and the interval ends with $u$ receiving a normal message $m$ from $v$. Suppose also that there is a $\text{SIGNAL}_u$ event in the interval. Then from Claim D.2.8 we know that there is some point within the interval during which an ABORT packet arrives at $v$ and such that any messages in transit from $u$ to $v$ have been "flushed" out. However. this claim goes further and states that the point at which the ABORT packet arrives at $v$ occurs *before* $v$ sends message $m$.

**Claim D.2.18** *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Consider any $i, j, k$ such that $i < j < k$. Suppose $s_i.status(u) = on$, $a_j$ is a $\text{SIGNAL}_u$ event, and $a_k$ is a normal receive event at $u$ from $v$. Let $a_{k'}$ be the send corresponding to $a_k$. Then there is some $i < j' < k'$ such that $status(v) \neq on$ and $s_{j'}.M_{u,v}$ is empty.*

**Proof:** By Claim D.2.8 applied to $s_i$ and $a_j$ we know there is some $j'$ where $i < j' < j$ such that:

- $a_{j'}$ is a $\text{RECEIVE}_{u,v}(\text{ABORT}, *)$ action.
- $s_{j'}.mode(v) \neq Ready$
- $s_{j'}.M_{u,v}$ does not contain any $\Sigma$-message.

As long as we can prove that $j' < k'$ we are done. Suppose not. By $\mathcal{A}$ we know that $s_{j'}.ack_u[v] = true$. But since $a_k$ is a receive action, $s_k.status(u) = on$, and so we know that $s_k.ack_u[v] = false$. Let $s_l$ be the first state after $s_{j'}$ such that $s_l.ack_u[v] = false$. Thus $j' < l < k$. But we know from the code that $ack_u[v]$ is only set to *false* after a $\text{RECEIVE}_{v,u}(\text{ACK})$ event and so $a_l$ must be a $\text{RECEIVE}_{v,u}(\text{ACK})$ event. But by $\mathcal{B}$ we know that there is no ACK packet in $M_{v,u}$ in state $s_{j'}$. Thus there must be some $n$, $j' \leq n < k$ such that $a_n = \text{SEND}_{v,u}(\text{ACK})$. Intuitively, what we have shown is that in the interval $[s_{j'}, s_k]$ an ack packet must have been sent by $v$ and received by $u$.

Now we can obtain the required contradiction. We will only sketch the rest of the argument informally. Suppose for contradiction, that $j' \geq k'$. Then it must be that the ack sent (in action $a_n$) was sent *after* the user message sent (in action $a_{k'}$). But then (essentially because the channels and queues are FIFO), the corresponding user message receipt (i,e., action $a_k$) must have occurred before the corresponding ack packet receipt (call this action $a_l$). Thus $k < l$. Also, since $j' < k$, we know that $k$ lies in the interval $[s_{j'}, s_{l-1}]$. But we know from $\mathcal{A}$ that in the interval $[s_{j'}, s_{l-1}]$, $mode(u) \neq Ready$ since in this interval $u$ is still waiting for an ack from $v$. But this contradicts the fact that in a state $s_k$ immediately following a receive event such as $a_k$, $mode(u)$ must be *Ready*. ∎

Next, we show a lemma (see Figure D.1) which states (in essence) that messages sent in a signal interval at $v$ can be received in at most one signal interval at $u$; conversely messages received in a signal interval at $u$ could have been sent in at most one signal interval at $v$. This will help establish that each signal interval at $u$ can have at most one mated signal interval at $v$.

**Lemma D.2.19 Send Consistency**: *Consider any pair of neighbors $u$,$v$ and any execution $\alpha$ of $\mathcal{R}|L$. Let $a_j$ and $a_k$ be any two normal receive events at $u$ from $v$ in $\alpha$. Let $a_l$ and $a_m$ be the send events corresponding to $a_j$ and $a_k$ respectively. Then there is a $\text{SIGNAL}_v$ event between $a_l$ and $a_m$ iff there is a $\text{SIGNAL}_u$ event between $a_j$ and $a_k$.*

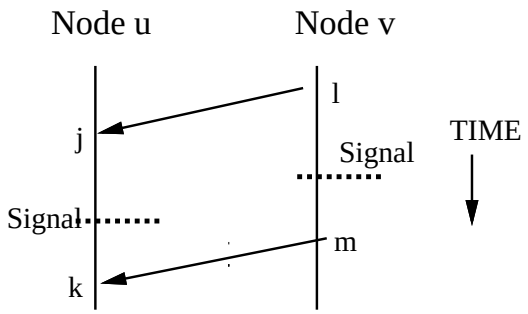



Figure D.1: Send consistency: there is a signal between the two receives at $u$ iff there is a signal between the two corresponding sends at $v$.

**Proof:** Assume without loss of generality that $j < k$. Suppose there is a SIGNAL$_v$ event, say $a_{m'}$ between $a_l$ and $a_m$. By Claim D.2.14, $s_l.status(v) = on$. Then by Claim D.2.8 there must be some $p$, $l < p < m'$, such that $s_p.status(u) = off$ and $M_{v,u}$ is empty. But by Claim D.2.14, $M_{v,u}$ is non-empty in the interval $[s_l, s_j]$. Thus since $p > l$, it must be that $p > j$. Also $p < m'$ and $m' < m$ and by Claim D.2.14, $m < k$. So $p < k$. Thus there is a state $s_p$ that occurs in the interval $[s_j, s_k]$ in which $status(u) = off$. Also we know by Claim D.2.14 that $s_k.status(u) = on$. Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_j, s_k]$.

The reverse argument is slightly different. Suppose there is a SIGNAL$_u$ event, say $a_{k'}$ between $a_j$ and $a_k$. By the code, $s_j.status(v) = on$. Thus by Claim D.2.18, applied to $s_j$, $a_{k'}$ and $a_k$ we know that there is some $j'$ such that $j < j' < m$ and $s_{j'}.status(v) \neq on$. But since $l < j$, $s_{j'}$ occurs in the interval $[s_l, s_m]$ and $status(v) = off$. Also we know by Claim D.2.14 that $s_m.status(v) = on$. Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_l, s_m]$. ∎

Next, we show a second lemma (see Figure D.2) which states (in essence) that a signal interval at $u$ cannot send messages to and receive messages from different signal intervals at $v$. This will help show that the mating relation is symmetric

**Lemma D.2.20 Send-Receive Consistency**: *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Let $a_j$ be a normal receive event at $u$ from $v$ and let $a_m$ be a normal receive event at $v$ from $u$. Let $a_l$ and $a_k$ be the send events corresponding to $a_j$ and $a_m$ respectively. Then there is a* SIGNAL$_v$ *event between $a_l$ and $a_m$ iff there is a* SIGNAL$_u$ *event between $a_j$ and $a_k$.*

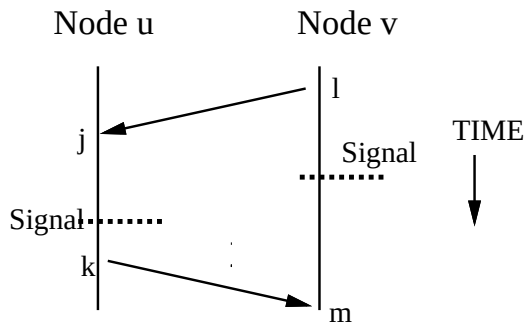



Figure D.2: Send-Receive consistency: there is a signal between the receive and the send at $u$ iff there is a signal between the corresponding send and receive at $v$.

**Proof:** Assume that $k > j$ as shown in the Figure D.2. The other cases are similar.

Suppose there is a SIGNAL$_u$ event, say $a_{k'}$ between $a_j$ and $a_k$. By Claim D.2.14, $s_j.status(u) = on$. Then by Claim D.2.8 there must be some $p$, $j < p < k'$, such that $s_p.status(v) = off$. Thus since $p > j$, it must be that $p > l$. Also $p < k$ and $k < m$. So $p < m$. Thus there is a state $s_p$ that occurs in the interval $[s_l, s_m]$ in which $status(v) = off$. Also we know by Claim D.2.14 that $s_m.status(v) = on$. Thus by Claim D.2.3, a SIGNAL$_v$ event must occur in the interval $[s_l, s_m]$.

The reverse argument is slightly different. Suppose there is a SIGNAL$_v$ event, say $a_{m'}$ between $a_l$ and $a_m$. By the code, $s_l.status(v) = on$. Thus by Claim D.2.18, applied to $s_l$, $a_{m'}$ and $a_m$ we know that there is some $p$ such that $p < k$ and $s_p.status(u) \neq on$ and such that $s_p.M_{v,u}$ does not contain a $\Sigma$-message. We claim that $j < p$. If not not $s_p$ must lie in the interval $[s_l, s_j]$ and in this interval we know that there is always a $\Sigma$-message in $M_{v,u}$ by Claim D.2.14. But this contradicts the fact that $s_p.M_{v,u}$ does not contain a $\Sigma$-message.

Thus $s_p$ occurs in the interval $[s_j, s_k]$ and $status(u) \neq on$. Also we know by Claim D.2.14 that $s_k.status(v) = on$ Thus by Claim D.2.3, a SIGNAL$_u$ event must occur in the interval $[s_j, s_k]$. ∎

We now show the third part of the consistency property (see Definition 7.3.5).

**Lemma D.2.21 Successful Sending of Messages**: *Consider any pair of neighbors $u, v$ and any execution $\alpha$ of $\mathcal{R}|L$. Between any safe* SENDM$_{v,u}(m)$ *event and a later* FREEM$_{v,u}$ *event, there is either a* RECEIVEM$_{v,u}(m)$ *event or a* SIGNAL$_v$ *event.*

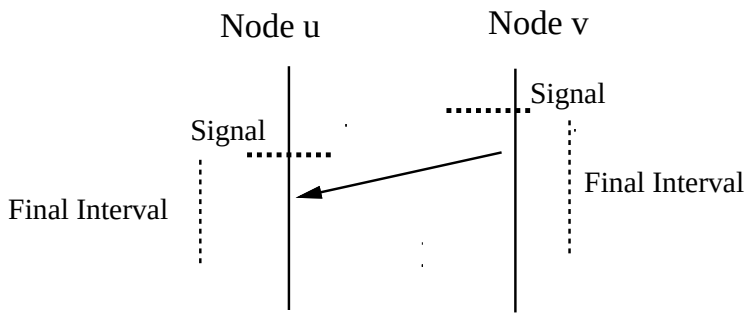



Figure D.3: Mating of Final Signal Intervals: Messages sent in a final interval can only be received in another final interval.

**Proof:** Let us denote the $\text{SENDM}_{v,u}(m)$ event by $a_i$ and the $\text{FREEM}_{v,u}$ event by $a_k$. Clearly $l < k$.

Thus, by the code $status(v) = on$ in $s_k$. Also by the code, $freem_v[u] = true$ in $s_k$ and hence by $\mathcal{H}$, $M_{v,u}$ does not contain any $\Sigma$-message in $s_k$.

Next, consider $a_i$. Since $a_i$ is safe, by definition there must an action $a_h = \text{FREEM}_{v,u}(m)$ such that $h < i$ and such that there is no other $\text{SENDM}_{v,u}(*)$ action between $a_h$ and $a_i$. Thus, by the code $freem_v[u] = true$ in $s_h$ and $s_{i-1}$. Thus, by the code, we see that either $s_i.status(v) = off$ or $m$ belongs to $M_{v,u}$ in $s_i$ (i.e., the message $m$ is placed on the queue at $v$ to send to $u$). But in the first case, we are done by Claim D.2.3, which tells us there must be a $\text{SIGNAL}_v$ event between $s_i$ and $s_k$.

So consider the second case where $m$ belongs to $M_{v,u}$ in $s_i$ and $s_i.status(v) = on$. But we know that in the later state $s_k$, $m$ does not belong to $M_{v,u}$. Now, from the code, the only two actions that could remove $m$ from $M_{v,u}$ in the interval $[s_i, s_k]$ are a $\text{RECEIVEM}_{v,u}(m)$ event or a $\text{RECEIVE}_{v,u}(\text{ABORT}, *)$ event. In the former case, we are done; so consider the latter case. Now, because $s_i.mode(v) = on$, we know from $\mathcal{A}$ that there is no $(\text{ABORT}, *)$ packet in $s_i.xqueue_v[u]$. Thus there must have been an action $a_j = \text{SEND}_{v,u}(\text{ABORT}, *)$ in the interval $[s_i, s_k]$. From the code, $s_j.status(v) = off$. The lemma now follows from Claim D.2.3, which tells us there must be a $\text{SIGNAL}_v$ event between $s_j$ and $s_k$. ∎

We now show the fourth part of the consistency property (see Definition 7.3.5). The lemma on which it is based is sketched in Figure D.3.

**Lemma D.2.22 Mating of Final Signal Intervals**: *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$. Let $a_j$ be a normal receive event at $u$ from $v$ in $\alpha$ and $a_l$ be the send corresponding to $a_j$. Then there is no* $\text{SIGNAL}_u$ *event after $a_j$ iff there there is no* $\text{SIGNAL}_v$ *event after $a_l$.*

**Proof:** Suppose there is a $\text{SIGNAL}_v$ event $a_n$ after $a_l$. Then by Claim D.2.8, there is a state $s_m$ between $a_l$ and $a_n$ such that $s_m.mode(u) \neq Ready$ and $s_m.M_{v,u}$ does not contain any $\Sigma$ messages. It follows from Claim D.2.14 that $s_m$ must occur after $a_j$. Thus there is a state after $a_j$ in which $mode(u) \neq Ready$. Thus by the Signal Lemma (Lemma D.2.2), a $\text{SIGNAL}_u$ event will occur after $a_j$.

The reverse argument is similar but slightly simpler. ∎

**Lemma D.2.23 Mating Relation Preserves Temporal Ordering**: *Consider any pair of neighbors $u,v$ and any execution $\alpha$ of $\mathcal{R}|L$. Suppose a signal interval $S_u$ at $u$ is mated to a signal interval $S_v$ at $v$ and a signal interval $S'_u$ at $u$ is mated to a signal interval $S'_v$ at $v$. Then if $S'_u$ occurs later than $S_u$ then $S'_v$ occurs later than $S_v$.*

**Proof:** Omitted. Follows, in essence, from the FIFO properties of the underlying UDLs and the fact that ABORT packets sent between signal intervals flush the links and buffers of previously sent messages. ∎

And now we come to the main result of this subsection:

**Lemma D.2.24** *Every behavior of $\mathcal{R}|L$ satisfies the consistency property.*

**Proof:** We define a signal interval at a node $u$ in an execution $\alpha$ by analogy with the definitions for behaviors. We define two signal intervals $I_u$ at $u$ and $I_v$ at $v$ to be mates if any normal message received in $I_u$ was sent in $I_v$ or vice versa. Lemma D.2.19 and Lemma D.2.20 show that the mating relation is well-defined and that any signal interval at $u$ can have at most one mate and that the relation is symmetric. The last three conditions in Definition 7.3.5 follow from Lemma D.2.21, Lemma D.2.22 and Lemma D.2.23. ∎

### D.2.4 Every Behavior of $\mathcal{R}|L$ is causal

We can now prove that the Reset protocol is causal. We first prove the first property in Definition 7.3.6.

**Theorem D.2.25** *Consider any execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. There is some constant $c$ such that every signal event $a_k$ that occurs at time greater than $s_0.time + cn$ is preceded by a request event $a_j$ such that $a_k.time - a_j.time \leq cn$.*

**Proof:** A formal proof can be patterned after the intuitive argument given in Section 7.7. ∎

Next, we prove the second property in Definition 7.3.6.

**Lemma D.2.26** *Consider an execution $\alpha = s_0, a_1, s_1, \ldots$ of $\mathcal{R}|L$. There is some constant $c$ such that a* SIGNAL$_u$ *event occurs within $cn$ time of any* REQUEST$_u$ *event.*

**Proof:** We know from the code that in the state immediately following a REQUEST$_u$ event, $status(u) = off$. The lemma follows immediately from the Signal Lemma (Lemma D.2.2). ∎

And now we come to the main result of this subsection:

**Lemma D.2.27** *Every behavior of $\mathcal{R}|L$ is causal.*

**Proof:** Immediate from Definition 7.3.6 and Lemmas D.2.25 and D.2.26. ∎

### D.2.5 The Main Theorem

Finally, we can state our main theorem of this section, which follows from the main lemmas of the last three subsections:

**Theorem D.2.28** *Every behavior of $\mathcal{R}|L$ is in RP.*

**Proof:** Immediate from Definition 7.3.7 and Lemmas D.2.17, D.2.24, and D.2.27. ∎

# Appendix E

# Dijkstra's Token Protocol as an Example of Counter Flushing

In Chapter 10, we described a paradigm called counter flushing. We now show that Dijkstra's first example protocol in [Dij74] can be simply understood using this paradigm.

Dijkstra's first example is modelled by the automaton $D2$ shown in Figure E.1. As in the previous example, the nodes (once again numbered from 0 to $n-1$) are arranged such that node 1 has node 0 and node 2 as its neighbors and so on. However, in this case we also assume that Process 0 and $n-1$ are neighbors. In other words, by making 0 and $n-1$ adjacent we have made the line into a ring. For process $i$, let us call Process $i-1$ (we assume that all arithmetic on indices and counters is mod $n$) the anticlockwise neighbor of $i$ and $i+1$ the clockwise neighbor of $i$.

Each node has a counter $count_i$ in the range $0, \ldots n$ that is incremented mod $n+1$. Once again the easiest way to understand this protocol is to understand what happens when it is properly initialized. Thus assume that initially Process 0 has its counter set to 1 while all other processes have their counter set to 0. Processes other than 0 are only allowed to "move" (see Figure E.1) when their counter differs in value from that of their anticlockwise neighbor; in this case, the process is allowed to make a move by setting its counter to equal that of its anticlockwise neighbor. Thus initially, only Process 1 can make a move after which Process 1 has its counter equal to 1; next, only Process 2 can move, after which Process 2 sets its counter equal to 1; and so on, until the value 1 moves clockwise around the ring until all processes have their counter equal to 1.

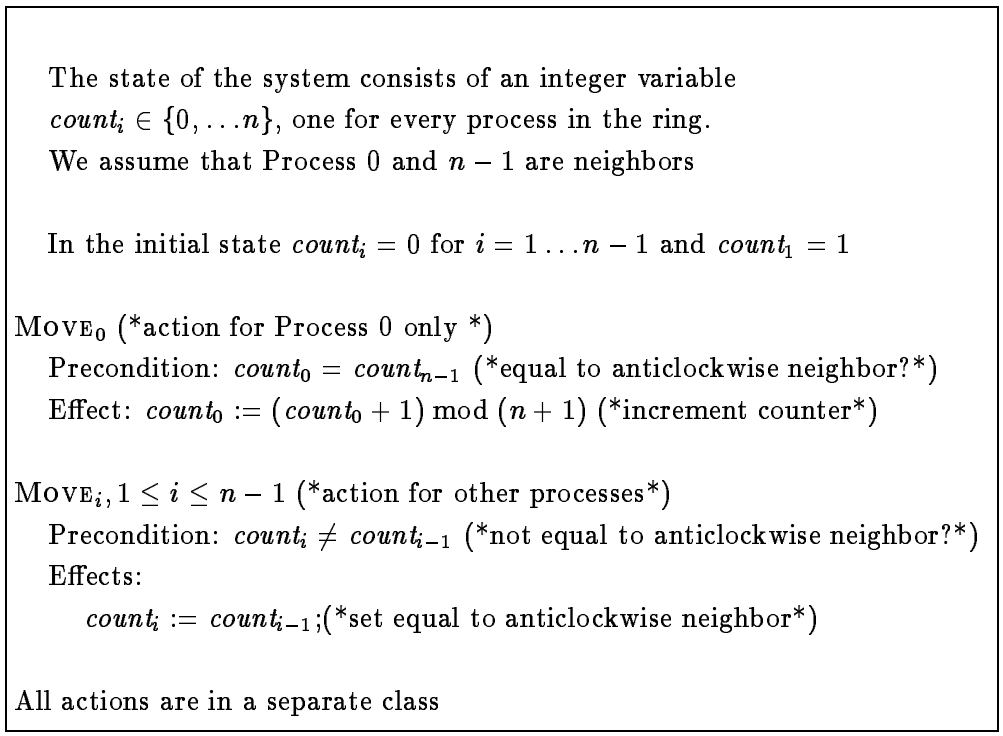



Figure E.1: Automaton $D1$: a version of Dijkstra's first example with initial states. The protocol does token passing on a ring using nodes with $n$ states.

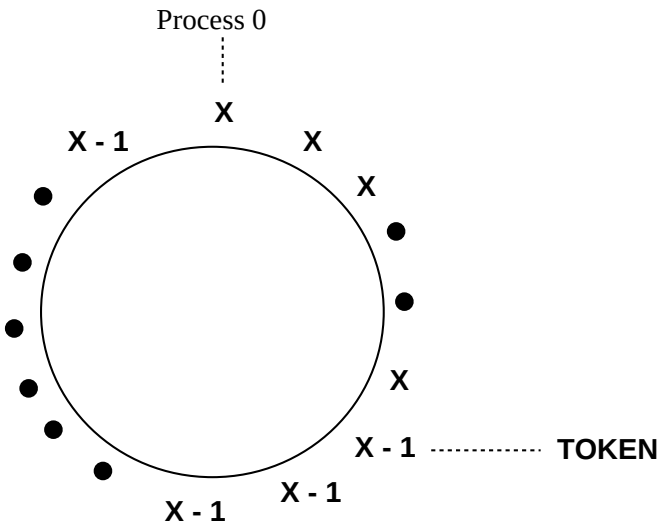



Figure E.2: In the good states for Dijktra's first example, the ring can be partitioned into 2 bands with the token at the boundary.

Process 0 on the other hand cannot make a move until Process $n-1$ has the same counter value as Process 0. Thus until Process 1 sets its counter to 1, Process 0 cannot make a move. However, when this happens, Process 1 increments its counter mod $n+1$. Then the cycle repeats as now the value 2 begins to move across the ring (assuming $n > 2$) and so on. Thus after proper initialization, this system does perform a form of token passing on a ring; each node is again considered to have the token, when the system is in a state in which the node can take a move.

The good global states of the protocol can be sketched in Figure E.2. Notice that the ring can be partitioned into 2 bands. All counter values within a band are equal and the band that includes the top node has a counter value one higher than the lower band. The token is at the boundary between the two bands; after that node makes a move, the top band becomes larger and the lower band becomes smaller.

It is easy to see that the system is in a good state iff the following local predicates are true.

- For $i = 1 \ldots n-1$, either $count_{i-1} = count_i$ or $count_{i-1} = count_i + 1$.
- Either $count_0 = count_{n-1}$ or $count_0 = count_{n-1} + 1$.

The system is locally checkable but it does not appear to be locally correctable. However, it does stabilize using a paradigm that we can call *counter flushing*. Even if the counter values are arbitrarily initialized (in the range $0, \ldots, n$) the system will eventually begin executing as some suffix of a properly initialized execution. We will prove this informally using three claims:

- **In any execution, Process 0 will eventually increment its counter.** Suppose not. Then since Process 0 is the only process that can "produce" new counter values, the number of distinct counter values cannot increase. If there are two or more distinct counter values, then moves by Processes other than 0 will reduce the number of distinct counter values to 1, after which Process 0 will increment its counter.

- **In any execution, Process 0 will eventually reach a "fresh" counter value that is not equal to the counter values of any other process.** To see this, note that that in the initial state there are at most $n$ distinct counter values. Thus there is some counter value say $m$ that is not present in the initial state. Since, process 0 keeps incrementing its counter, Process 0 will eventually reach $m$ and in the interim no other process can set their counter value to $m$.

- **Any state in which Process 0 has a fresh counter value $m$ is eventually followed by a state in which all processes have counter value $m$.** It is easy to see that the value $m$ moves clockwise around the ring "flushing" any other counter values, while Process 0 remains at $m$. This is why we call this paradigm counter flushing.

The net effect is that any execution of $D1$ eventually reaches a good state in which it remains. The reader should compare our proof of this protocol with the general description of counter flushing found in Chapter 10.

# Bibliography


[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[AB89] Yehuda Afek and Geoffrey Brown. Self-stabilization of the alternating bit protocol. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[AE86] Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AG91] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, 1991.

[AG92] Anish Arora and Mohamed G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. Unpublished manuscript, February 1992.

[AGLP89] Baruch Awerbuch, Andrew Goldberg, Michael Luby, and Serge Plotkin. Network decomposition and locality in distributed computation. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, May 1989.

[AGR92] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks. In *Proceedings of the 11th PODC*, Vancouver, British Columbia, August 1992.

[AKY90] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[APV91a] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Creating self-stabilizing protocols by using a reset protocol. Draft, 1991.

[APV91b] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pages 206–220, October 1988.

[AS90] Baruch Awerbuch and Mike Saks. A dining philosophers algorithm with polynomial response time. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[Awe90] Baruch Awerbuch. Shortest paths and loop-free routing in dynamic networks. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Philadelphia, PA*, September 1990.

[BGW87] G.M. Brown, M.G. Gouda, and C.L. Wu. A self-stabilizing token system. In *Proceedings of 20th Hawai International Conference on System Sciences*, pages 218–223, 1987.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11(2):330–344, 1989.

[BSW69] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12:260–261, 1969.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comput. Syst.*, 3(1):63–75, February 1985.

[CRKG89a] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 224–236, 1989.

[CRKG89b] Chunhsiang Cheng, Ralph Riley, Srikanta P.R. Kumar, and Jose J. Garcia-Luna-Aceves. A loop-free extended Bellman-Ford routing protocol without bouncing effect. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 224–236. ACM SIGCOMM, ACM, September 1989.

[CSV89] Jeff Cooper, Robert Simcoe, and George Varghese. Stabilizing, hardware-based implementation of a flow control scheme for high speed links: Presented at ATM Forum, Aug 1993. Unpublished manuscript, January 1989.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[DIM90] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[DIM91a] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message driven protocols. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, Montreal, Canada, August 1991.

[DIM91b] Shlomo Dolev, Amos Israeli, and Shlomo Moran. Uniform self-stabilizing leader election. Unpublished manuscript, 1991.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Process. Letters*, 11(1):1–4, August 1980.

[Fin79] Steven G. Finn. Resynch procedures and a fail-safe network protocol. *IEEE Trans. on Commun.*, COM-27(6):840–845, June 1979.

[Gal76] Robert G. Gallager. A shortest path routing algorithm with automatic resynch. Technical report, MIT, Lab. for Information and Decision Systems, March 1976.

[Gar89] Jose J. Garcia-Luna-Aceves. A unified approach to loop-free routing using distance vectors or link states. In *Proceedings of the Annual ACM SIGCOMM Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 212–223. ACM SIGCOMM, ACM, September 1989.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[GM90] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. Technical Report TR-90-20, Dept. of Computer Science, University of Texas at Austin, June 1990.

[Gol85] Andrew V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, MIT, Lab. for Computer Science, November 1985.

[GPS87] A. V. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry breaking in sparse graphs. In *Proc. 19th ACM Symp. on Theory of Computing*. ACM SIGACT, ACM, May 1987.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *J. of the ACM*, 35(4):921–940, October 1988. Also appeared in 18th stoc (1986).

[IJ90] Amos Israel and Marc Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[KP90] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, Canada, August 1990.

[LA90] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5(2):190–222, April 1983.

[Lam84] L. Lamport. Solved problems, unsolved problems, and non-problems in concurrancy. In *Proc. of the 3rd ACM Symp. on Principles of Distributed Computing, Vancouver, BC, Canada*, pages 1–11, Aug 1984. Invited address at the 2nd ACM Symp. on Principles of Distributed Computing, Montreal, Canada, August 1983.

[Lau90] Tony Lauck. personal communication. unpublished, 1990.

[Lin87] Nathan Linial. Locality as an obstacle to distributed computing. In $27^{th}$ *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[LS91] N. Linial and M. Saks. Decomposing graphs into regions of small diameter. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 320–330. ACM/SIAM, January 1991.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MAM+90] M.Schroeder, A.Birrell, M.Burrows, H.Murray, R.Needham, T.Rodeheffer, E.Sattenthwaite, and C.Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Technical Report 59, Digital Systems Research Center, April 1990.

[MMT91] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. In *CONCUR 91*, pages 408–423, 1991.

[MP91] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83, 1991.

[MPV90] Yishay Mansour, Boaz Patt, and George Varghese. Self-stabilizing data structures. unpublished manuscript, December 1990.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. on Programming Lang. and Syst.*, 4(3):455–495, 1982.

[Per83] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[Per85] Radia Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proceedings of the the 9th Data Communication Symposium*, pages 44–53, September 1985.

[Per88] Radia Perlman. *Network Layer Protocols With Byzantine Robustness*. PhD thesis, MIT, Laboratory for Computer Science, August 1988.

[RF89] Balasubramanian Rajagopalan and Michael Faiman. A new responsive distributed shortest path routing algorihtm. In *Proceedings of the Annual ACM SIGCOMM*

*Symposium on Communication Architectures and Protocols, Austin, Texas*, pages 237–246. ACM SIGCOMM, ACM, September 1989.

[RL81] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th POPL*, pages 133–138, 1981.

[Ros81] E. C. Rosen. Vulnerabilities of network control protocols: An example. *Computer Communications Review*, July 1981.

[SG89] John M. Spinelli and Robert G. Gallager. Broadcasting topology information in computer networks. *IEEE Trans. on Commun.*, May 1989.

[Spi88a] John M. Spinelli. Reliable communication. Ph.d. thesis, MIT, Lab. for Information and Decision Systems, December 1988.

[Spi88b] John M. Spinelli. Reliable communication data links. Technical Report CICS-P-102, MIT, Lab. for Information and Decision Systems, December 1988.

# Index

# Index


actions, 42
    disabled, 43
    enabled, 43
    external, 42
    input, 42
    internal, 42
    locally controlled, 42
    output, 42
augmented topology graph, 240
automaton, 42
    behavior, 44
    classes, 42
      upper bounds, 42
    execution, 43
    untimed behavior, 44
    untimed execution, 44
automaton for augmented graph, 240
automaton for graph
    network model, 105

behavior sequence, 44

causality, 167
checker
    for synchronous protocols, 243
CIOA , 62
$C$, all edges clean, 127
clean edge, 125
closed local predicate, 100
closed predicate, 78
compatible automata, 45
composition, of automata, 45
compositional model, 46
conjunction, 101
consistency, 165
*countset*, 124
counter flushing, 115, 268

drop-free, 124

event, 43

ghost root, 228
Global Correction Theorem, 208

height, of partial order, 104
hiding, 47

I/O relation, 241
initialized signal interval, 214
interactive protocols, 239
interval of a behavior, 47
interval of an execution, 47

leader edge, 92
    corresponding to an edge, 290
link delay, 93
link predicate set
    network model, 101
    shared memory model, 78

link subsystem
    network model, 100
    shared memory model, 78
Local Correction Theorem, 106
local predicate
    separable, 231
    shared memory model, 78
local reset function
    network model, 104
locally checkable
    network model, 101
    shared memory model, 78
locally correctable
    network model, 104
locally extensible
    network model, 147
    shared memory model, 81

mating relation, 165
    transitivity, 168
message-renaming, 207
mode
    of a node in reset protocol, 173

network automaton
    network model, 97
    shared memory model, 77
node automaton, 96
node delay, 96
non-interactive protocol, 241
non-interactive protocols, 239
normal receive, 163

one-way checkable, 231
one-way correctable, 268
one-way predicate, 231
output function, 241

$P_{control}$, 93
$P_{data}$, 93
phase, 121
    mode, 288
    well-behaved, 290
predicate, 47
    holds, 47
    is true, 47
    remains true, 47
$Proj$, 120

quiescent state, 213
$Q$, all edges quiet, 129
quiet edge, 128
$\mathcal{Q}$, quiet edge predicate set, 129

refinement mapping
    theorem, 57
regular message receipt, 214
*reqcount*, 124
reset transition, 286
reset trigger, 213
*respcount*, 124

safe, 162
satisfies, 100
self-stabilizes, 51
send
    corresponding to normal receive event, 163
signal interval, 165
solve, 52
space complexity, 242
stabilization time, 54

with respect to I/O relation, 242
stabilized using predicate set, 60
stabilizes
to behaviors in set, 53
to behaviors of another IOA, 54
to executions in set, 50
to executions of another IOA, 51
to I/O relation, 242
stable tree behaviors
used to define spanning tree protocol correctness, 222
status
of a node in reset protocol, 177
suffix-closed, 62
synchronous protocol, 242
space complexity, 243
time complexity, 243

t-suffix
of behavior, 53
of execution, 50
$t_l$, default link delay, 97
$t_p$, time for a phase, 122
$t_n$, default node delay, 97
time sequence, 43
*start*, 43
$t_q$, time for a link to become quiet, 131
timed element, 43
timed action, 43
timed state, 43
timeliness, 164
timer flushing, 221, 234, 269
topology graph, 92
tree automaton
network model, 137
shared memory model, 77
Tree Correction Theorem, 148
tree graph, 137

$U(A)$, 51
UIOA , 61
unexpected packet transition, 148
Unit Storage Data Link (UDL), 93

weakly checkable, 147