Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms

by

Jennifer Lundelius Welch

B. A., The University of Texas at Austin
(1979)

S. M., Massachusetts Institute of Technology
(1984)

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

June 1988

© 1988 by Massachusetts Institute of Technology

Signature of Author [illegible] 3/22/88
Department of Electrical Engineering and Computer Science

Certified by [illegible] 3/22/88
Nancy A. Lynch, Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith, Chair
Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 26 1988
ARCHIVES
LIBRARIES

Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms

by

Jennifer Lundelius Welch

Submitted to the Department of
Electrical Engineering and Computer Science on March 22, 1988,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract: This thesis solves problems in two distinct areas of theoretical distributed computing. The first problem area concerns the impact of the degree of synchrony in a distributed system on what problems can be solved. Results in this area include studies of the transaction commit problem in a partially synchronous model, and a simulation showing that one timing model is equivalent to a seemingly more powerful model. The second problem area is the modular decomposition and verification of algorithms, especially their liveness properties. Two existing resource allocation algorithms are studied, and by using modularity new algorithms are obtained that are more efficient. In addition, a lattice-style decomposition is used to prove correct a network minimum spanning tree algorithm, whose correctness had not previously been formally given. Several techniques are presented for verifying liveness proofs in a modular way.

Keywords: Distributed systems, time bounds, fault tolerance, lower bounds, randomized protocols, simulation, consensus problem, modularity, liveness proofs, drinking philosophers, minimum spanning tree.

Thesis supervisor: Nancy A. Lynch
Title: Professor of Computer Science and Electrical Engineering

Contents

Acknowledgments

Graduate school was a successful and worthwhile experience for me because of my thesis advisor, Nancy Lynch. Her ideas pervade this thesis: she is a coauthor of Chapters 4 and 5, and suggested the topics of Chapters 2 and 3. She read the entire document with great care, and provided copious comments for corrections and improvements. She was always accessible, and interested in my professional development.

The other members of my thesis committee, Baruch Awerbuch and Silvio Micali, helped me view the results in my thesis from new and interesting perspectives.

I had the pleasure of working together on Chapter 2 with Brian Coan, who took time out from finishing his own thesis to help me overcome thesis block. I am grateful to Leslie Lamport for his generous collaboration on Chapter 5.

Barbara Liskov and Bill Weihl, together with Nancy, suggested the problem studied in Chapter 2; the presentation of the solution was improved by insightful remarks from Yoram Moses. I received helpful comments from Gil Neiger, Larry Stockmeyer and an anonymous referee pertaining to Chapter 3. Discussions with Yehuda Afek, Steve Garland, Michael Merritt, and Liuba Shrira helped me clarify in my mind the key concepts in Chapter 5.

I enjoyed many stimulating conversations relating to research done for this thesis with Brian Coan, Alan Fekete, Ken Goldman and Mark Tuttle. During the four years that we shared an office, I often took advantage of Mark's willingness to share his time and knowledge. I also benefited from the Theory of Distributed Systems group meetings, which were a lively forum for discussing research.

Marilyn Pierce in the EECS Graduate Office and Betty Pothier in the Laboratory for Computer Science saved me many bureaucratic headaches.

The inevitable frustrations and tensions of graduate school were eased by having good friends. I thank Debbie Thurston for her enduring friendship; my fellow graduate students in Tech Square, especially Kathy Yelick and Brian Oki for their unlimited moral support. and the Aloof Bicyclists Club for all that exercise.

I couldn't have done it without the love and encouragement I received from my husband George and my parents.

Financial support for this work was provided by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014–83–K–0125, by the National Science Foundation under Grants DCR–83–02391 and CCR–86–11442, by the Office of Army Research under Contract DAAG29–84–K–0058, and by the Office of Naval Research under Contract N00014–85–K–0168.

Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms

1 Introduction

This thesis solves problems in two distinct areas of theoretical distributed computing. The first problem area concerns the impact of the degree of synchrony in a distributed system on what problems can be solved. The second problem area is the modular decomposition and verification of algorithms. In this chapter, we present an overview of these areas, and indicate how the results of this thesis fit in.

1. The Impact of Partial Synchrony

The solvability of many problems in a failure-prone distributed computer system depends strongly on the assumptions made about the degree of synchrony exhibited by the system. For instance, the problem of agreeing on a bit (*Byzantine agreement* [PSL] [LSP]) can be solved even if almost one third of the processors are subject to arbitrary faults, as long as there are known upper bounds on relative processor speeds and message delivery times. Yet, if there are no such bounds, then no protocol can guarantee agreement if there is the possibility of even one processor crashing undetectably [FLP]. Both these models represent extremes — real distributed systems are seldom so completely synchronous or completely asynchronous. Consequently, researchers have begun to study some intermediate, or *partially synchronous* models. Work in [DDS] and [DLS] has proposed a variety of partially synchronous models and classified the solvability (or unsolvability) and maximum failure-resiliency of the agreement problem in these models.

Chapter 2 of this thesis studies the transaction commit problem (a specialization of the very general agreement problem defined in [FLP] and [DDS]) in a new partially synchronous model. The definition of the model is tailored to the particular problem, but we believe it is realistic. The fault-tolerant transaction commit problem has no solution, either deterministic or randomized, in the completely asynchronous model. We present a positive result about partial synchrony, that there is a randomized protocol for the transaction commit problem in this model. Our protocol works as long as fewer than half the processors are subject to (undetectable) crash failures. We prove a lower bound showing that the number of faults tolerated is optimal. Our protocol terminates in a small constant expected number of asynchronous rounds. Our definition of asynchronous rounds is tailored to our partially synchronous model. Additional justification for the definition is provided by our result that no protocol in this model can terminate in a bounded expected number of steps, even if processors are synchronous.

Chapter 3 of this thesis demonstrates the limitations of one form of partial synchrony; it states that a distributed system in which both relative processor speeds and message delays are asynchronous can simulate a system in which processors operate in lockstep synchrony and message delays are asynchronous, in the presence of various types of processor faults. The notion of simulation used is that each processor undergoes the same sequence of state transitions in an execution in one system as it does in a corresponding execution of the other system. One application of this result is that the result in [FLP], that no consensus protocol for asynchronous processors and communication can tolerate one failstop fault, implies a result in [DDS] that no consensus protocol for synchronous processors and asynchronous communication can tolerate one failstop fault.

2. Modular Decomposition of Algorithms

The second problem area investigated in this thesis is using modularity to describe and analyze distributed algorithms, including their liveness properties. One kind of modularity we investigate is composing separate "subroutines" to produc e an algorithm, and showing how properties of the subroutines combine to give properties of the composition. The other kind of modularity we study is describing an algorithm at different levels of abstraction, and showing how properties at one level imply properties at another level.

Many resource allocation algorithms presented in the literature [PF] [CM] are described as using a solution to another problem as a "subroutine," yet the description employs a particular solution, with the details of its implementation inextricably embedded in the algorithm. The use of the word "subroutine" suggests attempting to describe such algorithms modularly, so that any solution at all to the sub-problem can be used. We use the I/O automaton model of [LT] to describe a resource allocation algorithm for the drinking philosophers problem [CM] in Chapter 4. The description uses a solution to another problem as a true subroutine; nothing needs to be known about the implementation, only that its "interface" is correct. The advantages of this approach are that the algorithm is easier to describe and prove correct. As an added bonus, one can sometimes obtain more efficient (in some complexity measure) solutions by using a more efficient subroutine. In fact, we obtain a faster drinking philosophers algorithm with this method.

A different notion of modular decomposition is applied to the problem of proving correct the minimum spanning tree algorithm in [GHS]. In that paper, a complex and delicate algorithm is presented, but no proof of correctness. The approach taken in Chapter 5 to proving correctness of this algorithm is to describe it at different levels of abstraction. These different descriptions do not form the usual chain hierarchy, but instead are organized into a lattice. Thus the original algorithm, or implementation, is described by several more abstract algorithms, each expanding on a different aspect of the implementation, and incomparable to the others. We present a framework of definitions and theorems to show the relationships between algorithms in the lattice and to justify why these relationships imply the correctness of the implementation.

2 Transaction Commit in a Realistic Fault Model

An important problem in the construction of fault-tolerant distributed database systems is the design of nonblocking transaction commit protocols. This problem has been extensively studied for synchronous systems (i.e., systems where no messages ever arrive late). In this chapter, the synchrony assumption is relaxed. A new partially synchronous timing model is given. In this model, a new nonblocking randomized transaction commit protocol is given, based on a Byzantine agreement protocol of Ben-Or. The new protocol works as long as fewer than half the processors fail. A lower bound is proved, showing that the number of processor faults tolerated is optimal. The protocol exhibits a graceful degradation property: when more than half the processors fail, the protocol blocks, but no processor produces a wrong answer. A notion of asynchronous round is defined and the protocol is shown to terminate in a small constant expected number of asynchronous rounds. The final result is that no protocol in this model can terminate in a bounded expected number of steps, even if processors are synchronous.

1. Introduction

In a distributed database system a transaction may be processed concurrently at several different processors. To maintain the integrity of the database these processors must take consistent action regarding the transaction — either the results

---

This chapter is joint work with Brian Coan; a preliminary version appeared in the *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Systems* [CL].

of the transaction are installed in the database at all processors (the transaction is *committed*), or the results are installed at no processor (the transaction is *aborted*). The decision whether to abort or commit a transaction is made by a *transaction commit protocol.* The objective for such a protocol is to commit as many transactions as possible subject to the constraint that each processor must be able to abort a transaction unilaterally.

A transaction commit protocol must never produce inconsistent decisions, and it must allow unilateral aborts. It has some leeway, though. Some protocols can produce more aborts than others, and some protocols fail to terminate in some situations. If failures can cause some nonfaulty processors to remain undecided about the fate of a transaction (at least as long as the failure persists), a processor is said to *block*, and the protocol is called *blocking.* Otherwise, the protocol is *nonblocking.* The most common transaction commit protocol in practice, two phase commit, is a blocking protocol. A blocking protocol is preferable in real systems to one that allows inconsistent decisions to be made, since it allows consistent decisions to be reached after the failures are repaired. A nonblocking protocol would be more preferable still.

Many elegant nonblocking transaction commit protocols [Sk] [DS] have been developed for completely synchronous systems. An obstacle to using these protocols in real systems is that a single violation of the timing assumptions (i.e., a late message) can cause the protocol to produce the wrong answer. The most common alternative timing model, the completely asynchronous model, unfortunately does not allow any solution to the transaction commit problem, either randomized or deterministic.[1] We give a new timing model that is intermediate between the synchronous and asynchronous models previously studied. In this model, we give a new nonblocking transaction commit protocol.

We model real systems in which messages are usually delivered within some known time bound but sometimes come late. We do this by assuming a completely

[1]**The intuition behind this impossibility result is the following. Suppose there is a protocol that works in an asynchronous system, and guarantees that nonfaulty processors eventually decide (with probability 1); if the processors all begin with commit and there are no failures, then they all decide commit; and if any processor begins with abort, then the nonfaulty processors decide abort. Consider a run in which all processors but *p* begin with commit and are nonfaulty, while *p* fails initially. Eventually, the rest of the processors must decide. Since *p* could have started with abort, the processors must decide abort. But there is another run that looks identical up to the decision point to all the processors except *p*, in which *p* begins with commit, and all its messages are delayed until after the decision is made. But in this run, the decision should have been commit.**

asynchronous system, in which relative processor speeds are unbounded and messages can take arbitrarily long to arrive, and letting the timing behavior affect the correctness conditions for the transaction commit problem, as follows. If every processor initially wants to commit the transaction, then the common decision must be to commit, provided no processors fail and all messages arrive within some known fixed time bound. If any processor initially wants to abort the transaction, then the common decision must be to abort, no matter what the timing and fault behavior of the system is. This problem definition takes advantage of the leeway allowed in specifying when processors must commit. Assuming that failures and late messages are relatively rare, the overall progress of the transaction processing system will not be impeded very much. A similar division is made in [DLS], in which properties that must always hold are separated from properties that only need hold when the system is well-behaved. In most other respects our model differs from theirs.

We prove that in our model no transaction commit protocol can terminate in a bounded expected number of steps. Consequently a new measure is needed to analyze the time performance of our protocol. One of the contributions of this chapter is such a measure, which we call an asynchronous round. Our definition of asynchronous round is strong enough to allow us to show that our protocol terminates in a small constant expected number of asynchronous rounds. In Section 2 we argue that this notion of asynchronous round is not unrealistically strong.

Randomization is needed in the protocol because a result of [DDS] implies that no deterministic protocol is possible. In order to analyze a randomized protocol, we must define the adversaries against which the protocol will work. Our notion of the adversary is drawn from [CMS]. The adversary in our model chooses the order in which processors take steps, when each message will be delivered, and which processors fail and when (as long as fewer than half fail). It makes these decisions dynamically, during the execution of the protocol, using unlimited computational power. The adversary has available at any point in the execution all information about the hardware and software of the processors, and the pattern of communication up to that time, but it does not know the contents of the messages sent, nor the local states of processors, nor the processors' local random choices, unless that information is deducible from the pattern of communication. We will be careful to design our protocol so that it is not deducible.

Our protocol uses a modified version of a solution to the agreement problem. In the *agreement problem* each processor begins with an initial value, 0 or 1, and decides on a final value. All nonfaulty processors' final values must be equal, and if all processors have the same initial value, then that value must be the final value. Thus if one processor begins with 0 and the rest with 1, either 0 or 1 is a correct answer to the agreement problem, whereas in the transaction commit problem, the answer must be 0 (if 0 is identified with abort).

An important difference between the transaction commit problem and the agreement problem is that in the former, all processors that decide are required to agree, including processors that decide and subsequently fail. This strict agreement condition is imposed because we assume that failed processors will eventually recover. The hope is that processors that fail and subsequently recover can be reintegrated using a separate recovery protocol. Skeen's thesis has an excellent discussion of recovery protocols [Sk]. We do not discuss these protocols further in this chapter.

We assume that the faulty processors fail by crashing (i.e., stopping without warning). This is a realistic assumption that is commonly made in the database literature [Sk]. The number of faults tolerated by our protocol is optimal, since we prove a matching lower bound. Our protocol works as long as more than half the processors are nonfaulty. An important property of our protocol is that it degrades gracefully if the bound on the number of faulty processors is exceeded — instead of producing a wrong answer, the protocol simply fails to terminate.

At the beginning of our protocol, processors exchange some messages, and then execute a modification of Ben-Or's asynchronous agreement protocol [Be] to decide the fate of the transaction. The preliminary message exchanges serve two purposes: first, the differences between the input-output relations for the transaction commit and agreement problems are resolved, and second, a number of identical random bits are distributed.[1] These identical random bits are used in the agreement protocol to lower the expected running time from exponential to constant. There is a body of work dealing with attaining constant expected running time for the agreement problem [R] [CMS]; our technique does not solve this problem, for the following reason. In our protocol, if the identical random bits are not distributed in a timely fashion, processors can unilaterally decide 0 (abort), because we are solving the transaction commit problem. Such action is not an option for processors trying to solve the agreement problem, because it could violate the condition that all processors decide 1 if they all start with 1.

[1] **We have not solved the global coin toss problem, however, because our protocol does not guarantee that the identical random bits are successfully distributed; the nature of the transaction commit problem, as discussed above, is such that our protocol can tolerate this failure.**

The transaction commit protocols of Skeen [Sk] and Dwork and Skeen [DS] tolerate any number of processor faults, while our protocol only handles fewer than half of the processors failing. However, if half or more of the processors fail, our protocol does not produce a wrong answer but merely fails to terminate, leaving open the opportunity for processors to recover. Late messages are not a problem for our protocol because of our model, but as we noted earlier they can cause the protocols in [Sk] and [DS] to produce a wrong answer.

In summary, the principal contributions of this chapter are a realistic timing model, a method for analyzing the time performance of protocols in this model, an efficient fault-tolerant protocol for the transaction commit problem, and lower bounds showing that the protocol has optimal fault-tolerance, and that no protocol can terminate in a constant expected number of steps for each processor.

Following an exposition of our formal model in Section 2, we present our randomized transaction commit protocol in Section 3. Section 4 contains the lower bound proof showing that our protocol tolerates the maximal number of faulty processors. Finally, in Section 5 we show that no transaction commit protocol can guarantee that each processor terminate in a bounded expected number of its own steps, even if processors are synchronous.

2. Model

Processors are modeled as state machines that communicate by sending messages. Messages can take arbitrarily long to arrive. Our protocol works even in a very weak model in which there is no bound on the relative frequency with which processors take steps, and in which there is no atomic broadcast of messages. Our lower bound results are shown for the stronger case in which processors run in lock-step synchrony and possess atomic broadcast. In this section we present the weaker model. In Sections 4 and 5 we indicate the necessary changes for the stronger model. Our model is similar to those in [FLP] and [DDS].

Throughout this chapter, 1 is identified with "commit" and 0 with "abort."

2.1 Basic Model

A *raw message* consists of some text, and the names of the sending and receiving processors. A *message* is a (raw message, integer) ordered pair; the integer denotes the sending time, as will be explained later. The reason for distinguishing between messages and raw messages is that we do not wish to require timestamps on all (raw) messages sent by processors, yet this information is useful in the exposition of the model for distinguishing multiple instances of the same raw message and determining message delays.

A ***processor*** is an infinite state machine, together with a message buffer, and a random number generator. The message buffer holds messages that have been sent to the processor but not yet received, and is modeled as a set of messages. The random number generator supplies an infinite sequence of n-bit strings. The state machine's transition function uses the current state, current random bit string and set of raw messages received to compute the new state and raw messages to be sent. Certain states are initial states, designated $(id, inital)$, where id is a nonnegative integer and $inital$ is either 0 or 1. The id element of the initial state is the processor's name, or identification number. The $inital$ element is the processor's initial value. Each processor can send zero or one message to every processor in one step. There is an integer in each processor's state, called its *clock*, which is 0 in an initial state, and is always incremented by 1 by the transition function. Thus, the clock counts how many steps the processor has taken so far. A ***protocol*** is a set of n processors.

A *configuration* C consists of n states, one for each processor, and n sets of messages, one for each processor's buffer. An *initial configuration* has all processors in initial states and all buffers equal to the empty set.

An *event* is denoted (p, M, b), in which processor p receives the set of messages M (which can be empty), and the random bit string b.

An event $e = (p, M, b)$ is ***applicable*** to configuration C if every message in M is an element of p's buffer in C. Let s and M' be the state and set of raw messages resulting from applying p's transition function to p's state in C, b, and the raw messages extracted from M. The configuration resulting from applying e to C, denoted $e(C)$, is obtained from C by removing all messages in M from p's buffer, changing p's state to s, and adding the message (m, i), for each $m \in M'$, to the appropriate buffer, where i is the value of p's clock in s.

A ***schedule*** is a finite or infinite sequence of events. A finite schedule $\sigma = e_1 e_2 \ldots e_k$ is ***applicable*** to configuration C if e_1 is applicable to C, e_2 is applicable to $e_1(C)$, etc. The resulting configuration is denoted $\sigma(C)$. An infinite schedule is applicable to C if every finite prefix of the schedule is applicable to C.

Given configuration C_1 and schedule σ applicable to C_1, we define the *run* $R = run(C_1, \sigma)$ obtained from C_1 and σ, as follows. If $\sigma = e_1 e_2 \ldots e_k$ is finite, then R is the sequence $C_1 e_1 C_2 e_2 \ldots e_k C_{k+1}$, where $C_{i+1} = e_i(C_i)$, $1 \leq i \leq k$. If $\sigma = e_1 e_2 \ldots$ is infinite, then R is the sequence $C_1 e_1 C_2 e_2 \ldots$, where, for all i, $C_1 e_1 C_2 e_2 \ldots e_i C_{i+1} = run(C_1, e_1 e_2 \ldots e_i)$. We also denote σ by $sched(R)$. Informally, a run is a schedule together with its associated configurations.

Processor p is *nonfaulty* in an infinite run or schedule if it takes an infinite number of steps; otherwise it is *faulty.* An infinite run or schedule is *failure-free* if no processor is faulty in it. Since the interleaving of processors' steps in a run or schedule may be arbitrary, no particular degree of synchronization is necessarily achieved.

A message sent by processor p at event e in infinite run R is *guaranteed* if e is not the last step of p in R. An infinite run R is *t-admissible*, for $0 \leq t \leq n$, if

- the first configuration is an initial configuration,
- at most t processors are faulty, and
- all guaranteed messages sent to nonfaulty processors are eventually received.

The notion of guaranteed messages is used to model the lack of atomic broadcast. Since messages sent at a processor's last step do not have to be received, we effectively model a processor failing in the middle of a broadcast.

There are two disjoint sets of *decision states,* Y_0 and Y_1, such that if a processor enters a state in Y_0 or Y_1 it stays in that set forever. A processor *decides* v when it is in a state in Y_v. A run is *deciding* if every nonfaulty processor decides. A configuration C has *decision value* v if there is some processor whose state in C is an element of Y_v.

2.2 Timing Constraints

We fix a positive constant $K \geq 1$, which is used to define late messages. A message m from p to q is *late* in run $R = C_1 e_1 C_2 e_2 \ldots$ if event e_s adds m to q's message buffer, and one of the following is true. (1) There is no event in R that removes m from q's message buffer, and some processor takes more than K steps in R after e_s. (2) There is an event e_r that removes m from q's message buffer, and some processor takes more than K steps in the schedule $e_{s+1} \ldots e_r$. A run is *on-time* if it contains no late messages.

Ideally we would like a processor to decide in a constant expected number of its own steps. Unfortunately, as we prove in Section 5, we cannot do this, even if processors run in lockstep synchrony. Instead, we characterize the time performance of our protocol using the following definition. Given an infinite run, a processor is defined inductively to be in a particular *asynchronous round* (or *round*) as follows. Asynchronous round 1 begins for processor p when p first takes a step and ends after p's K^{th} step. Asynchronous round r, $r > 1$, begins for p at the end of p's round $r-1$ and ends either K of p's steps after the end of p's round $r-1$, or as soon as p receives every received message sent by a processor q in q's round $r-1$, whichever happens later. (We say "every *received* message" in order to make sure that no round lasts infinitely long due to p's waiting for a non-guaranteed message from q that never arrives.)

This definition uses two criteria for ending a round, the number of processor steps taken and the collection of messages received. These criteria seem natural in our timing model, in which processors can take actions depending on the receipt of messages, as well as on timeouts.

A processor cannot compute its current asynchronous round; the definition is for our use as omniscient observers as we analyze protocols. The reason we require a round to last at least K steps is to prevent a round from collapsing to nothing if no messages are sent in the previous round. If processors take steps in round-robin order, and receive and send messages only at the beginning of a round, and if each message sent at the sender's i^{th} step is received at the recipient's $(i+K)^{th}$ step (for all i), then this definition is essentially the same as the synchronous round definition in [DS]. Thus this definition is not unreasonably strong.

2.3 Safety Conditions

The following definition restricts what must happen if a processor decides, but does not require any processor to decide. A protocol is a *transaction commit protocol* if for every t-admissible run R:

- *Agreement Condition:* Every configuration has at most one decision value.
- *Abort Validity Condition:* If the initial value of any processor is 0, then no configuration has decision value 1.
- *Commit Validity Condition:* If the initial value of all processors is 1 and R is failure-free and on-time, then no configuration has decision value 0.

To exclude uninteresting protocols, we require that each processor must be able to receive at least n messages at each step. Otherwise, processors could swamp the message system, causing messages to become late not because the message system misbehaves, but because the ability of the processors to handle all the incoming message traffic is inadequate.[1] For instance, the protocol "cause the run to be not on-time by flooding the message system and then abort" is not of much practical interest.

2.4 Adversary

The adversary can be considered a scheduler — it decides which processor takes a step next and what messages are received. In the introduction we gave an informal description of the adversary. This subsection formalizes the notion.

The *message pattern* of finite run $R = C_1 e_1 \ldots e_k C_{k+1}$, where $e_i = (p_i, M_i, f_i)$ for all $1 \leq i \leq k$, is the sequence of triples $(p_1, E_1, P_1) \ldots (p_k, E_k, P_k)$, where P_i is the set of processors to which messages were sent by event e_i, and E_i is a set of integers indexing the events in the run that sent the messages, M_i, received in e_i. The point of making this definition is to isolate the pattern of message sending and receiving while hiding the contents of the messages.

An *adversary* is a function that takes a message pattern, and returns a processor p and a set E of integers (which may be empty) satisfying the following condition. If i is in E, then in the i^{th} element of the message pattern, (p_i, E_i, P_i), p is in P_i (i.e., there actually was a message sent to p at the i^{th} event), and in no element of the message pattern does p receive this message (i.e., the message in question has not yet been received). Thus, the adversary decides on the next processor to take a step, plus a collection of messages to be received.

Let $\mathcal{F}$ be the collection of all n-tuples of infinite sequences of n-bit strings. Each element of $\mathcal{F}$ is a possible set of choices returned by the n processors' random number generators in an infinite run. A run is uniquely determined by an adversary A, an initial configuration I, and an element F of $\mathcal{F}$. Denote this run by $run(A, I, F)$. The construction of $run(A, I, F) = C_1 e_1 C_2 e_2 \ldots$ is inductive. Let $C_1 = I$. Suppose the run up to configuration C_i has been constructed. Let p and E be the result of A acting on the message pattern of run $C_1 e_1 \ldots C_i$. Then e_i consists of the processor p, the messages sent to p in all the events indexed by E, and the next unused bit

[1] **Suppose each processor can send n messages per step but only receive $n-1$. Consider the protocol: At each step, broadcast a message; at step 1, decide 0. We now show that no infinite run is on-time. Let R be an infinite run. After $Kn(n-1)+n$ events, $(Kn(n-1)+n)n$ messages have been sent, and at most $(Kn(n-1)+n)(n-1)$ have been received. So there are at least $Kn(n-1)+n$ outstanding messages. By the pigeonhole principle, some processor p has at least $K(n-1)+1$ outstanding messages (to be received). It will take p at least $K+1$ steps to receive all those messages, by which time the run will no longer be on-time.**

string in the sequence for p in F. Finally, $C_{i+1} = e_i(C_i)$. Since the adversary is a total function, $run(A, I, F)$ is an infinite run, and thus at least one processor is nonfaulty.

If the adversary were not restricted in any way, it could cause all processors (but one) to fail or no messages to be delivered, and no protocol would be possible. We limit the power of the adversary in the following reasonable way. We define a *t-admissible* adversary to be an adversary such that for all initial configurations I and all F in $\mathcal{F}$, $run(A, I, F)$ is t-admissible.

For predicate P defined on runs, let $\Pr[P]$ be the probability of the event $\{F \in \mathcal{F} : run(A, I, F) \text{ satisfies } P\}$, for a fixed adversary A and initial configuration I.

The expected value of any complexity measure for a fixed randomized protocol is defined as follows. Let T be a random variable that given a run returns the complexity measure of interest for that run. For fixed t-admissible adversary A and initial configuration I, let the expected value of T, taken over the random numbers F, be denoted $E(T_{A,I})$. Define the expected value for the protocol, ET, to be $\max_{A,I}\{E(T_{A,I})\}$.

2.5 Liveness Condition

Given infinite run R and integer r, let $\textsc{done}(R, r)$ be the predicate that every nonfaulty processor decides by its asynchronous round r in R. A protocol is *t-nonblocking* if for any t-admissible adversary A and any initial configuration I,

$$\lim_{r \to \infty} \Pr[\textsc{done}(run(A, I, F), r)] = 1.$$

2.6 Problem Statement

Our goal is to design a t-nonblocking transaction commit protocol.

3. Randomized Commit Protocol

Our protocol to solve the transaction commit problem is based on the asynchronous agreement protocol in [Be]. Similar protocols have been widely used [Br] [CC] [CMS] [R]. For the rest of this section, we assume a fixed t with $n > 2t$.

3.1 The Protocol

In this subsection, we present the randomized transaction commit protocol by describing, for each processor p, the states and transition function of p. First, we give an informal description.

Throughout the protocol each processor keeps a *vote* telling what it currently wants to do with the transaction. The processor with *id* 0 is the *coordinator*; at its first step, it chooses n random bits and distributes them to the other processors, the *participants*, by broadcasting a coins message containing the bits. If a participant receives no message at its first step, it sends a request message to the coordinator (to try to jog it awake); if no reply is received within 2K steps, the participant sets its vote to 0 and decides 0. If a participant receives a message at its first step, it extracts the n bits and broadcasts them in a coins message, to indicate "I am participating in the protocol." If a processor does not receive a coins message from everyone within 2K steps after broadcasting one, it sets its vote to 0 and decides 0. Then each processor broadcasts its vote. If a processor does not receive n votes for 1 within a short time, it sets its vote to 0, but remains undecided.

The rest of the protocol proceeds in stages (as in [Be]), numbered from 1 up without bound. In stage s, each processor p broadcasts its vote in a stage $(s,1)$ message and waits to receive $n-t$ stage $(s,1)$ messages. If p receives more than $n/2$ stage $(s,1)$ messages with vote $v \in \{0,1\}$, then p broadcasts v in a stage $(s,2)$ message; otherwise p broadcasts "?" in a stage $(s,2)$ message. Then p waits to receive $n-t$ stage $(s,2)$ messages. If p receives a stage $(s,2)$ message with value $v \in \{0,1\}$, then p sets its vote to v; otherwise, p sets its vote to a random bit, either the s^{th} random bit from the coins message if $s \leq n$, or else a locally-determined random bit. If p receives $n-t$ stage $(s,2)$ messages for value $v \in \{0,1\}$, then p decides v.

Processor p uses the following constants, variables and subroutines. Constants are p, n and K. Variables are:

- $clock_p$: nonnegative integer; initially 0.
- $stage_p$: values are "asleep," "request," "coins," "vote," $(s,1)$ and $(s,2)$ for all $s \geq 1$; initially "asleep."
- $timer_p$: nonnegative integer or *nil*; initially *nil*.
- $coins_p$: n-bit string or *nil*; initially *nil*.
- $vote_p$: boolean; initially p's initial value.
- $decide_p$: boolean or *nil*; initially *nil*.

The text of each raw message consists of the sending processor's current stage, and optionally a *value* (0, 1 or "?"), and an n-bit string.

Below we describe p's transition function, acting on state q of p, set M of raw messages, and n-bit string b. The description consists of several clusters of pseudocode. Each cluster is preceded by a predicate on q and M. The predicate of at most one cluster is true for any q and M. The state of p returned by the transition function is obtained from q by incrementing $clock_p$ by 1, remembering the set M, and then executing the cluster (if any) whose predicate is true of q and M. The set of raw messages returned by the transition function is that indicated by the send and broadcast statements of the appropriate cluster. If no cluster is true, then no raw messages are sent, the only changes to the state are that $clock_p$ is incremented and the received messages are remembered.

```
/* coordinator initiates protocol by distributing $n$ random bits */

    $stage_p$ = "asleep" for $p$ = coordinator:
      $coins_p := b$
      $stage_p$ := "coins"
      $timer_p := clock_p + 2K$
      broadcast ($stage_p$,"?",$coins_p$)

/* non-coordinator wakes up and requests that coordinator initiate */

    $stage_p$ = "asleep" for $p \neq$ coordinator and $M = \emptyset$:
      $stage_p$ := "request"
      $timer_p := clock_p + 2K$
      send "request" to coordinator

/* non-coordinator receives coins */

    $stage_p$ = "asleep" or "request" for $p \neq$ coordinator and
    there is a message in $M$ with text $(s, v, coins)$:
      $coins_p := coins$
      $stage_p$ := "coins"
      $timer_p := clock_p + 2K$
      broadcast ($stage_p$,"?",$coins_p$)

/* non-coordinator times out while waiting to receive coins */

    $stage_p$ = "request" and $clock_p = timer_p$:
        vote$_p$ := 0
        decide$_p$ := 0

/* distributing votes */

    $stage_p$ = "coins" and either $clock_p = timer_p$ or $n$ coins messages have been
    received:
        $stage_p$ := "vote"
        $timer_p := clock_p + 2K$
        if less than $n$ coins messages have been received then [
            $vote_p$ := 0
            $decide_p$ := 0 ]
        broadcast ($stage_p$,$vote_p$,$coins_p$)

/* completing stage 0 */

    $stage_p$ = "vote" and either $clock_p = timer_p$ or $n$ vote messages have been
    received:
        $stage_p$ := (1,1)
        if $n$ votes for 1 have been received
            then $vote_p$ := 1
            else $vote_p$ := 0
        broadcast ($stage_p$, $vote_p$, $coins_p$)

/* finishing first part of stage $s$ */

    $stage_p = (s,1)$ and at least $n-t$ stage $(s,1)$ messages have been received:
        $stage_p := (s,2)$
        if more than $n/2$ stage $(s,1)$ messages received have value $v$, for some $v$,
            then broadcast ($stage_p$, $v$, $coins_p$)
            else broadcast ($stage_p$,"?", $coins_p$)

/* finishing second part of stage $s$ */

    $stage_p = (s,2)$ and at least $n-t$ stage $(s,2)$ messages have been received:
        $stage_p := (s+1,1)$
        if a stage $(s,2)$ message received has value $v$, for some $v$, then [
            $vote_p := v$
            if at least $n-t$ stage $(s,2)$ messages received have value $v$ then $decide_p := v$
]
        else if $s \leq n$ then $vote_p := coins_p[s]$ else $vote_p$ := first bit of $b$
```

```
broadcast $(stage_p, vote_p, coins_p)$
```

Transaction Commit Protocol: p's transition function on input M, b, and arbitrary state

---

3.2 Proof of Correctness

The proof is organized as follows. Section 3.2.1 shows the safety properties, i.e., that the protocol is a transaction commit protocol. Section 3.2.2 contains the probabilistic analysis, which is applied to show the t-nonblocking property in Section 3.2.3.

3.2.1 Safety Conditions

Section 3.2.1 culminates in Theorem 8, which shows that the protocol is a transaction commit protocol.

All the lemmas in Section 3.2.1 hold for any (infinite) run from an initial configuration. In particular, they hold for runs in which more than t processors fail. Stating these results in this way allows us to show the graceful degradation property of the protocol.

In run R, processor p is said to be *in stage s*, for $s \geq 1$, if $stage_p = (s, 1)$ or $(s, 2)$. We say p *completes* stage $s \geq 0$ if p ever sets $stage_p$ to $(s+1, 1)$ in R. Let p's decision states Y_0 and Y_1 be states with $decide_p = 0$ and $decide_p = 1$ respectively; Lemma 7 below shows that once p enters a state in Y_v, it stays in that set forever. Note that if no nonfaulty processor ever receives a coins message, then no processor completes stage 0.

Lemma 1: *In any run from an initial configuration, if some processor p has $vote_p =$ 0 initially, then every stage (1,1) message has value 0.*

Proof: No processor ever receives a vote message with value 1 from p. Thus no processor sets its vote to 1 at the end of its vote stage, and no processor broadcasts a stage (1,1) message with value 1. □

Lemma 2: *In any infinite run from an initial configuration, if every processor p has $vote_p = 1$ initially, and the run is failure-free and on-time, then every processor broadcasts a stage (1,1) message with value 1.*

Proof: First we show that each processor p broadcasts a vote message with value 1. Suppose either p is the coordinator, or p receives a message at its first step.

Then p broadcasts a coins message at its first step. By time K on p's clock, each processor receives p's coins message and broadcasts its own coins message (if it has not already done so). By time $2K$ on p's clock, p receives n coins messages. Thus p broadcasts a vote message with value 1.

Now suppose p is not the coordinator and does not receive any messages at its first step. It sends a request message to the coordinator, which is received by time K on p's clock. The coordinator then broadcasts a coins message, if it has not already done so, and p receives the coins message by time $2K$ on p's clock. Then p broadcasts a coins message; by time $3K$ on p's clock, each processor receives p's coins message and broadcasts its own coins message (if it has not already done so). By time $4K$ on p's clock p receives n coins messages. Thus p broadcasts a vote message with value 1.

Now we show that every processor p receives n vote messages within $2K$ of its clock ticks after it broadcasts its vote. Processor p broadcasts its vote as soon as it receives its n^{th} coins message. Suppose its clock reads T then. Since the run is on-time, every other processor receives its n^{th} coins message, and broadcasts its vote, by the time p's clock reads $T + K$. Thus p receives all n vote messages by the time its clock reads $T + 2K$. Then p broadcasts its stage (1,1) message with value 1. □

Lemma 3: *In any run from an initial configuration, if every stage (s,1) message has value $v \in \{0,1\}$, then every processor that completes stage s decides v at stage s, for any $s \geq 1$.*

Proof: Let p be any processor that broadcasts a stage $(s,2)$ message. Then p receives at least $n-t$ stage $(s,1)$ messages, all with value $v \in \{0,1\}$ by assumption. Since $n > 2t$, $n-t > n/2$. Thus p broadcasts a stage $(s,2)$ message with value v.

Now let p be any processor that completes stage s. Then p receives at least $n-t$ stage $(s,2)$ messages, all with value v. Thus p decides v. □

For any $s \geq 1$, we call a stage $(s,2)$ message with value $v \in \{0,1\}$ an *S-message* ("S" for "set"), because the receipt of such a message can cause a processor to set its vote to v (if this message is among the first $n-t$ stage $(s,2)$ messages received by the processor).

Lemma 4: *In any run from an initial configuration, there is at most one value sent in S-messages during any stage $s \geq 1$.*

Proof: In order to send an S-message for some value v at stage s, a processor must receive more than $n/2$ stage $(s, 1)$ messages with value v. Since processors do not broadcast conflicting messages, fewer than $n/2$ processors can broadcast a stage $(s, 1)$ message with value $w \neq v$. Thus, no processor receives more than $n/2$ stage $(s, 1)$ messages with value w, and no processor sends an S-message for w at stage s. □

Lemma 5: *In any run from an initial configuration, if any processor decides v before stage 1, then*
(1) $v = 0$, and
(2) every processor that completes stage 1 decides v by the end of stage 1.

Proof: Suppose p decides before stage 1.

(1) By inspecting the code, we see that p decides 0, and sets its vote to 0 before broadcasting its vote message.

(2) As in the proof of Lemma 1, every stage (1,1) message has value 0, and by Lemma 3, every processor that completes stage 1 decides 0. □

Lemma 6: *In any run from an initial configuration, if some processor decides v at stage $s \geq 1$, then*
(1) no processor decides $w \neq v$ at stage s, and
(2) every processor that completes stage $s + 1$ decides v at stage $s + 1$.

Proof: Suppose processor p decides v at stage $s \geq 1$. Let q be any processor that completes stage s. Since p decides v at stage s, it receives at least $n - t$ stage $(s, 2)$ messages with value v before completing stage s. Thus, since $n > 2t$ and q receives at least $n - t$ stage $(s, 2)$ messages before completing stage s, at least one of these messages is from a processor from which p receives an S-message for v in stage s. Since processors do not broadcast conflicting messages, q receives at least one S-message for v at stage s. By Lemma 4, q sets its vote to v, and thus q broadcasts a stage $(s + 1, 1)$ message with value v.

(1) If q decides in stage s, then q decides v.

(2) By Lemma 3, every processor that completes stage $s + 1$ decides v at stage $s + 1$. □

Lemma 7: *In any run from an initial configuration, $decide_p$ changes value at most once, for every processor p.*

Proof: Pick any processor p. If $decide_p$ is set before stage 1, then by Lemma 5, every processor that completes stage 1 decides v at stage 1. If $decide_p$ is set for the first time in stage $s \geq 1$, then by Lemma 6, every processor that completes stage $s + 1$ decides v by the end of stage $s + 1$. Lemma 3 shows that for any $r \geq 1$, if every processor that completes stage r decides v at stage r, then any processor that completes stage $r + 1$ decides v at stage $r + 1$.

Theorem 8: *Protocol 1 is a transaction commit protocol.*

Proof: Let R be a t-admissible run. First we show the agreement condition, that there is at most one decision value in every configuration of R. If some processor decides before stage 1, then Lemmas 5 and 7 give the result. If no processor decides until stage $s \geq 1$, then Lemmas 6 and 7 give the result.

Next we show the abort validity condition. Suppose some processor begins with initial value 0. If no processor completes stage 0, then Lemma 5 shows that no processor decides 1. If some processor completes stage 0, then all nonfaulty processors complete stage s, for all $s \geq 0$. Lemmas 1 and 3 (with $u = t$) give the result.

Finally, we show the commit validity condition. Suppose R is failure-free and on-time, and all processors begin with 1. Then Lemmas 2 and 3 give the result. □

Since Lemmas 1 through 7 are true for any (infinite) run from an initial configuration, the agreement, abort validity, and commit validity conditions are true even for runs in which more than t processors fail. This is the graceful degradation property exhibited by our protocol.

3.2.2 Probabilistic Properties

The analysis in this subsection is directed toward showing that the probability that all processors that complete stage s, decide by stage s, apporaches 1 as s increases. Recall that probabilities are taken over the random information, holding the adversary and initial configuration fixed.

For the following definitions, fix adversary A, initial configuration I, and F and F' in $\mathcal{F}$. Let $R = run(A, I, F)$ and $R' = run(A, I, F')$.

Define $F(p, k)$ to be the k^{th} element in the sequence for p in F.

Define $coins(F)$ to be $F(0,1)$ (i.e., the coordinator's first n-bit string). It is easy to see that if $coins_p$ is ever nonnil in R, then it equals $coins(F)$, for all p. We denote the s^{th} element of $coins(F)$ by $coins(F)[s]$.

For processor p and $s \geq 1$, define $index(R,p,s)$ to be the number of steps taken by p to complete stage s in R. If p does not complete stage s, then $index(R,p,s)$ is undefined. Thus $index(R,p,s)$ is also the index into the sequence for p in F of the bit string used to determine the value of $vote_p$ in stage s, in case $s > n$ and p receives no S-message in stage s.

The next definition maps a bit to each processor and each stage $s > n$ in a run, such that each stage gets "new" bits. This mapping is consistent with the mapping implemented in the protocol for those cases where a processor uses a random bit. Let $random(R,p,s)$, for processor p and $s > n$, be defined as follows. (1) If p completes stage s in R, then $random(R,p,s)$ is the first bit of $F(p,k)$, where $k = index(R,p,s)$. (2) If p does not complete stage s in R, then $random(R,p,s)$ is the second bit of $F(p,s+1)$ (i.e., a safe default).

For $0 \leq s \leq n$, define F and F' to be (A,I,s)-*equal* if $coins(F)[i] = coins(F')[i]$ for all i, $1 \leq i \leq s$. For $s > n$, define F and F' to be (A,I,s)-*equal* if F and F' are (A,I,n)-equal, and for every i, $n+1 \leq i \leq s$, and every processor p, $random(R,p,s) = random(R',p,s)$.

For $s \geq 1$, define $v(R,s)$ to be the value of an S-message sent in run R at stage s. If no S-message is sent in R at stage s, then let $v(R,s) = 0$. By Lemma 4, $v(R,s)$ is uniquely defined.

Define MATCH(R,s) to be the predicate that if $s \leq n$, then $coins(F)[s] = v(R,s)$, and if $s > n$, then $random(R,p,s) = v(R,s)$ for all p.

Define SAME(R,s) to be the predicate that all processors that complete stage s in R set their votes to the same value in stage s.

Define DECIDE(R,s) to be the predicate that each processor that completes stage s has decided by the end of stage s in R.

The next lemma characterizes two aspects of runs that are unchanged, once an adversary and initial configuration are fixed.

Lemma 9: *Let A be an adversary, I an initial configuration, and F and $F' \in \mathcal{F}$. Let $R = run(A,I,F) = C_1 e_1 C_2 \ldots$ and $R' = run(A,I,F') = C'_1 e'_1 C'_2 \ldots$.*

(1) For all $i \geq 1$, the message pattern of $C_1 e_1 \ldots C_i$ is the same as the message pattern of $C'_1 e'_1 \ldots C'_i$.
(2) For all processors p and all $s \geq 1$, $index(R, p, s) = index(R', p, s)$.

Proof: (1) The structure of the protocol is such that the random information does not affect which processors send messages to which other processes — it only affects the values of the local variables and the message contents. But this is the very information not available to the adversaries under consideration. Thus, for a fixed adversary and initial configuration, the sequence of processor steps and the message delays are the same, regardless of the random information.

(2) Follows from (1). □

The next lemma states that the value of an S-message sent in stage $s+1$ only depends on the random information available through stage s, once an adversary and initial configuration are fixed.

Lemma 10: *Let $R = run(A, I, F)$ and $R' = run(A, I, F')$ for adversary A, initial configuration I, and F and F' in $\mathcal{F}$. If F and F' are (A, I, s)-equal, then $v(R, s+1) = v(R', s+1)$, for any $s \geq 0$.*

Proof: By Lemma 9, the message patterns for R and R' are the same. Since F and F' are (A, I, s)-equal, the random information that affects the local variables and message contents in R and R' up through stage s is the same in F and F'. Thus, the values of corresponding processors' variables, and the contents of corresponding messages sent up through stage s are the same in R and R'. The random information used in stage $s+1$ is not used until the end of stage $s+1$, so the same messages are sent in stage $s+1$ in R and R', even though the stage $s+1$ random information might be different in F and F'. □

The next lemma states some simple relationships between MATCH, SAME, and DECIDE.

Lemma 11: *Let $R = run(A, I, F)$ for adversary A, initial configuration I and $F \in \mathcal{F}$. For all $s \geq 1$,*
(1) MATCH(R, s) implies SAME(R, s), and
(2) SAME(R, s) implies DECIDE$(R, s+1)$.

Proof: Fix $s \geq 1$.

(1) If $s \leq n$, then $\textsc{match}(R, s)$ means that $coins(F)[s] = v(R, s)$. Thus $coins_p$ has the same value as any S-message sent in stage s of R, for all p. Thus, each processor that completes stage s sets its vote to $v(R, s)$, and $\textsc{same}(R, s)$ is true.

If $s > n$, then $\textsc{match}(R, s)$ means that the first bit of $F(p, k)$, where $k = index(R, p, s)$, is equal to the value of any S-message sent in stage s of R, for all p. Thus, each processor that completes stage s sets its vote to $v(R, s)$, and $\textsc{same}(R, s)$ is true.

(2) If $\textsc{same}(R, s)$ is true, then all stage $(s+1, 1)$ messages have the same value $v \in \{0, 1\}$. Thus all stage $(s+1, 2)$ messages have value v. Thus, every processor that completes stage $s+1$, decides v, and $\textsc{decide}(R, s+1)$ is true. □

The following technical lemma concerns any equivalence class of $\mathcal{F}$, where the equivalence is defined by (A, I, s)-equality.

Lemma 12: *Fix adversary A, initial configuration I, and $s \geq 0$. Partition $\mathcal{F}$ into the maximal equivalence classes, within each of which all elements are (A, I, s)-equal. Pick any class C.*
(1) $\textsc{match}(run(A, I, F), i) = \textsc{match}(run(A, I, F'), i)$ for all i, $1 \leq i \leq s$, and any F and F' in C.
(2) If $s < n$, then $\textsc{match}(run(A, I, F), s+1)$ is true for half the elements F of C; if $s \geq n$, then $\textsc{match}(run(A, I, F), s+1)$ is true for a $1/2^n$ fraction of the elements F of C.

Proof: (1) Choose any i, $1 \leq i \leq s$, and any F and F' in C. Let $R = run(A, I, F)$ and $R' = run(A, I, F')$. Since F and F' are $(A, I, i-1)$-equal, $v(R, i) = v(R', i)$, by Lemma 10. Since F and F' are (A, I, i)-equal, $coins(F)[i] = coins(F')[i]$ if $i \leq n$, and $random(R, p, i) = random(R', p, i)$ for all p if $i > n$; thus $\textsc{match}(R, i) = \textsc{match}(R', i)$.

(2) By Lemma 10, $v(run(A, I, F), s+1)$ is the same for all $F \in C$.

Suppose $s < n$. In half the elements F of C, $coins(F)[s+1] = 0$, and in half $coins(F)[s+1] = 1$, since all the elements of C are (A, I, s)-equal. Thus $\textsc{match}(run(A, I, F), s+1)$is true for half the elements F of C.

Suppose $s \geq n$. Let $R = run(A, I, F)$ for F in C. $\textsc{match}(R, s+1)$ means $random(R, p, s+1) = v(R, s+1)$ for all p. The position of $random(R, p, s+1)$ in F depends on whether p completes stage $s+1$ in R or not. By Lemma 9, either p completes stage $s+1$ in R for all F in C, or p fails to complete stage $s+1$ in R

for all F in C. If p does not complete stage $s+1$, then $random(R, p, s+1)$ is the second bit of $F(p, s+2)$, obviously a fixed position for all F in C. If p does complete stage s, then $random(R, p, s)$ is the first bit of $F(p, k)$, where $k = index(R, p, s)$. By Lemma 9, k is the same for all F in C, so this is also a fixed position for all F in C. The positions of $random(R, p, s)$ for all p are all distinct. Thus a $1/2^n$ fraction of the elements F of C have $random(R, p, s) = v(R, s)$ for all p. □

The next lemma is the key to the termination of the protocol, as well as the good time performance. It says that there is a high probability that the random information used to set votes matches the value in S-messages for the first n stages, and there is a smaller, but still positive probability for subsequent stages.

Lemma 13: *Fix adversary A and initial configuration I. Then*

$$Pr[\text{MATCH}(run(A, I, F), s)] = 1/2 \text{ if } s \leq n, \text{ and } 1/2^n \text{ if } s > n.$$

Proof: By part (2) of Lemma 12, since the lemma is true for every equivalence class of $\mathcal{F}$. □

The next lemma shows that the events of not matching in different stages are independent.

Lemma 14: *Fix adversary A and initial configuration I. Let $R = run(A, I, F)$ for $F \in \mathcal{F}$. Then for any $s \geq 1$,*

$$\Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,s)] = \Pr[\neg\text{MATCH}(R,1)] \cdots \Pr[\neg\text{MATCH}(R,s)].$$

Proof: Pick any i, $1 \leq i \leq s$. We will show that

$$\Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i)]$$

$$= \Pr[\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i-1)] \cdot \Pr[\neg\text{MATCH}(R,i)].$$

Let X be the set of all $F \in \mathcal{F}$ such that $\neg\text{MATCH}(R,1) \wedge \ldots \wedge \neg\text{MATCH}(R,i-1)$ is true, where $R = run(A, I, F)$. Partition $\mathcal{F}$ into equivalence classes based on $(A, I, i-1)$-equality. If F is in X, and F and F' are $(A, I, i-1)$-equal, then F' is also in X, by part (1) of Lemma 12. Pick any equivalence class C that is a subset of X. Part (2) of Lemma 12 gives the result. □

The next lemma shows that the probability that all processors that complete stage s, decide by stage s, approaches 1 as s increases.

Lemma 15: *For any adversary A and initial configuration I,*

$$\lim_{s\to\infty} \Pr[\textsc{decide}(run(A,I,F),s)] = 1.$$

Proof: Let $R = run(A,I,F)$. First note that

$$\Pr[\textsc{decide}(R,s)] \geq \Pr[\textsc{match}(R,1) \vee \ldots \vee \textsc{match}(R,s-1)].$$

The reason is that if $\textsc{match}(R,s')$ is true for some s', $1 \leq s' \leq s-1$, then by Lemma 11, $\textsc{same}(R,s')$ is true, and thus $\textsc{decide}(R,s'+1)$ is true. Since $s'+1 \leq s$, $\textsc{decide}(R,s)$ is true.

$$\begin{aligned}
&\Pr[\textsc{match}(R,1) \vee \ldots \vee \textsc{match}(R,s-1)] \\
&\qquad = 1 - \Pr[\neg\textsc{match}(R,1) \wedge \ldots \wedge \neg\textsc{match}(R,s-1)] \\
&\qquad = 1 - \prod_{i=1}^{s-1}(1 - \Pr[\textsc{match}(R,i)]), \text{ by Lemma 14} \\
&\qquad \geq 1 - (1-1/2^n)^{s-1}, \text{ by Lemma 13.}
\end{aligned}$$

Since $\lim_{s\to\infty}(1-1/2^n)^{s-1} = 0$ we are done. □

3.2.3 Liveness Condition

Lemmas 16 and 17 convert Lemma 15 into a statement about the predicate DONE, in order to show the t-nonblocking property in Theorem 18.

Lemma 16: *In any run from an initial configuration, each processor that completes stage 0 without having decided is in at most asynchronous round 6.*

Proof: Suppose p completes stage 0 without having decided. Then p obtains the n random bits in some message by its $2K^{th}$ step, and broadcasts its coins message. At most $4K$ steps later, p completes stage 0. Since each asynchronous round lasts at least K steps, at most 6 rounds elapse. □

The next lemma shows that each stage $s \geq 1$ takes only a bounded number of asynchronous rounds.

Lemma 17: *In any run from an initial configuration, if each processor that completes stage $s \geq 0$ is in at most asynchronous round r when it completes stage s,* *then each processor that completes stage $s + 1$ is in at most asynchronous round $r + 2$ when it completes stage $s + 1$.*

Proof: Let p be any processor that broadcasts a stage $(s + 1, 1)$ message. This happens when p completes stage s, so all stage $(s + 1, 1)$ messages are at most round r messages.

Let p be any processor that broadcasts a stage $(s + 1, 2)$ message. Processor p cannot enter round $r + 1$ until it has received the last of the round r messages, including all the stage $(s + 1, 1)$ messages. Immediately after receiving the last of these (if not before), p broadcasts its stage $(s + 1, 2)$ message, so all stage $(s + 1, 2)$ messages are at most round $r + 1$ messages.

No processor p can enter round $r + 2$ until it has received the last of the round $r + 1$ messages, including all the stage $(s + 1, 2)$ messages. Yet by the time p receives all the stage $(s + 1, 2)$ messages, p has completed stage $s + 1$. □

Theorem 18: *Protocol 1 is t-nonblocking.*

Proof: Pick any t-admissible run R. Suppose no nonfaulty processor p receives a coins message in R. Then p decides 0 by time $2K$ on its clock, i.e., by round 2. Now suppose some nonfaulty processor receives a coins message in R. Then, since R is t-admissible, every nonfaulty processor receives a coins message in R, and completes stage s, for all $s \geq 0$. By Lemmas 16 and 17, $\textsc{decide}(R, s)$ implies $\textsc{done}(R, 6 + 2s)$ for any t-admissible run R. Lemma 15 gives the result. □

3.3 Time Complexity

Recall that expectation is defined in Section 2.4 to be taken over t-admissible adversaries and initial configurations. First, we show that the expected number of stages is less than 4.

Lemma 19: *Let X be a random variable giving the least s such that all processors that complete stage s decide by stage s. Then $EX < 4$.*

Proof: Fix t-admissible adversary A and initial configuration I. Let $R = run(A, I, F)$, for F in $\mathcal{F}$. Let $q_s = \Pr[\neg\textsc{match}(R, s)]$. Let Y be a random variable giving the least number s such that all processors that complete stage s have the same vote at the end of stage s. By Lemma 3, $X \le Y + 1$.

$$\begin{aligned}
EX &\le E(Y+1) = 1 + EY = 1 + \sum_{s=1}^{\infty} s \cdot \Pr[Y = s] \\
&\le 1 + \sum_{s=1}^{\infty} s \cdot \Pr\left[\left(\bigwedge_{i=1}^{s-1} \neg\textsc{match}(R,i)\right) \wedge \textsc{match}(R,s)\right] \\
&= 1 + \sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_{s-1}(1 - q_s), \text{ by Lemma 14} \\
&= 1 + \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_{s-1}\right) - \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_s\right) \\
&= 1 + 1 + \left(\sum_{s=1}^{\infty} (s+1) \cdot q_1 q_2 \cdots q_s\right) - \left(\sum_{s=1}^{\infty} s \cdot q_1 q_2 \cdots q_s\right) \\
&= 2 + \sum_{s=1}^{\infty} (s + 1 - s) \cdot q_1 q_2 \cdots q_s \\
&= 2 + \sum_{s=1}^{\infty} q_1 q_2 \cdots q_s \\
&= 2 + \left(\sum_{s=1}^{n} q_1 \cdots q_s\right) + \left(q_1 \cdots q_n \cdot \sum_{s=n+1}^{\infty} q_{n+1} \cdots q_s\right).
\end{aligned}$$

We simplify using specific values for q_s. For $1 \le s \le n$, $q_s = 1/2$, and for $s > n$, $q_s = 1 - 1/2^n$, by Lemma 13.

$$\begin{aligned}
EX &\le 2 + \sum_{s=1}^{n} \frac{1}{2^s} + \frac{1}{2^n} \cdot \sum_{s=n+1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s-n} \\
&< 2 + 1 + \frac{1}{2^n} \cdot \sum_{s=1}^{\infty} \left(1 - \frac{1}{2^n}\right)^{s} \\
&= 3 + \frac{1}{2^n}\left(\frac{1 - 1/2^n}{1 - (1 - 1/2^n)}\right) \\
&= 3 + \frac{1}{2^n}(2^n - 1) \\
&< 4. \qquad \square
\end{aligned}$$

Theorem 20: *All nonfaulty processors decide in a constant expected number of asynchronous rounds.*

Proof: Let $R = run(A, I, F)$ for some t-admissible adversary A, initial configuration I, and $F \in \mathcal{F}$. If no nonfaulty processor receives a coins message in R, then every nonfaulty processor decides by round 2.

Suppose some nonfaulty processor receives a coins message in R. Then, since R is t-admissible, every nonfaulty processor p receives a coins message in R, and completes stage s, for all $s \geq 0$. By Lemma 16, p is in at most asynchronous round 6 when it completes stage 0. By Lemma 17, when p completes stage s of Protocol 1, it is in at most asynchronous round $6 + 2s$. The expected number of stages is 4, by Lemma 19. Therefore all nonfaulty processors decide in 14 expected asynchronous rounds. □

4. Lower Bound on Number of Processors

The lower bounds proved in the next two sections hold even if processors run in lockstep synchrony and possess an atomic broadcast capability. In this section, we first give relevant details of this stronger model, and then show that the number of faults tolerated by our transaction commit protocol is optimal.

A processor failure is represented by an explicit *failure step,* denoted $(p, \perp, b)$. After a failure step for p, p is in a distinguished *failed* state. Thus failures can be evidenced in finite runs. (Of course, processors cannot detect failures because message delivery is asynchronous.) A processor is *faulty* in a run if it takes a failure step, otherwise it is *nonfaulty.*

Processors take steps in round-robin order, 0 through $n - 1$; a schedule of the form $(0, M_1, f_1) \dots (n - 1, M_n, f_n)$ is a *cycle.* To enforce the round-robin behavior, each configuration has a *turn* component, designating which processor's turn it is to take a step. An initial configuration has $turn = 0$. In order for an event $e = (p, *, b)$ to be applicable to a configuration C, $turn(C)$ must equal p, and if p is in the failed state in C, then e must be a failure step. After an event is applied, the resulting configuration's *turn* component is incremented by 1 (modulo n).

The *guarantee* definition is no longer needed, since atomic broadcast is allowed. The *delay* of message m that is received in run R is the number of the cycle to which the receiving event belongs minus the sending time of m. An infinite run R is *t-admissible*, for $0 \leq t \leq n$, if

- the first configuration is an initial configuration,
- at most t processors are faulty,
- all messages sent to a nonfaulty processor are received, and

- all received messages have delay at least 1.

In this model, the adversary cannot schedule when processors take steps, but can only determine when a processor fails and what the message delays are.

In this section we show that no protocol, even a randomized one, can solve the transaction commit problem unless more than half the processors are nonfaulty. The intuition behind the proof is similar to that for the coordinated attack problem (first posed in [Gr]; also analyzed in [HM]). We partition the processors into two groups, each of size at most t. Given a run that decides 1 (in which all processors begin with 1), we work backwards from the end of the run to the beginning, delaying messages between the two groups and showing that the resulting runs must still decide 1. Eventually we get a run in which no messages between the groups are received, yet the processors decide 1. This situation leads to a contradiction, since one group could have started with 0's, in which case the decision should be 0.

The actual construction of the runs is fairly involved, and is facilitated by the following definitions and lemmas.

Let $state(p, C)$ be the state of processor p in configuration C, and $buff(p, C)$ be the state of p's buffer in C. Given a schedule σ and a subset S of the processors, define $\sigma|S$ to be the subsequence of σ consisting of exactly those events that are steps for processors in S. Also define $kill(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, b)$ (where $*$ can be M or $\perp$) with $(p, \perp, b)$ whenever p is in S; similarly, define $deafen(S, \sigma)$ to be the schedule obtained from σ by replacing every event $(p, *, b)$ (where $*$ can be M or $\perp$) with $(p, \emptyset, b)$ whenever p is in S.

Lemma 21: *Let σ be a schedule applicable to configuration C and τ be a schedule applicable to configuration D. Let S be a set of processors. If $state(p, C) = state(p, D)$ for all processors p in S and if $\sigma|S = \tau|S$, then for any processor p in S, $state(p, \sigma(C)) = state(p, \tau(D))$.*

Proof: Use induction on the length of $\sigma|S$, and the fact that the transition functions are deterministic, given states, messages and random numbers. □

Given a partition of the set of processors P into two sets S and S', define an *intergroup message* (relative to S and S') to be a message sent from a processor in S to a processor in S' or vice versa.

Lemma 22: *Let S and S' be a partition of the set of processors, and let C and D be two configurations such that $state(p, C) = state(p, D)$ and $buff(p, C) \subseteq buff(p, D)$*

for all p in S. Let σ be a schedule applicable to C in which any intergroup message that is received by $p \in S$ in σ is in $buff(p,C)$. Then

(a) *the schedule $\phi = kill(S',\sigma)$ is applicable to D;*

(b) *if no processor in S' is in a failed state in D, then the schedule $\tau = deafen(S',\sigma)$ is applicable to D.*

Proof: We show (b); (a) is similar. We proceed by induction on the length l of σ.

Basis: $l = 1$. Let $\sigma = e$ and $\tau = e'$. If e is an event for p in S', then p receives no messages in e'. This event is clearly applicable to D since p has not failed in D. If e is an event for p in S, then since $\tau = \sigma$ and $buff(p,C) \subseteq buff(p,D)$, the fact that σ is applicable to C implies that τ is applicable to D.

Induction: $l > 1$. Suppose the lemma is true for schedules of length $l-1$ and show for length l. Let $\sigma = \sigma' e$ be a schedule of length l. Since σ' has length $l-1$, by the induction hypothesis $\tau' = deafen(S',\sigma')$ is applicable to D. We must show that $e' = deafen(S',e)$ is applicable to $\tau'(D) = E$. If e is an event for p in S', then p receives no messages. This event is clearly applicable to E since p has not failed in D and no subsequent steps are failure steps.

Suppose $e = (p, M, b)$ for p in S. We must show that each m in M is in $buff(p,E)$. Choose m in M and let q be the sender.

If m is in $buff(p,C)$, then m is in p's buffer in every configuration from C to $\sigma'(C)$. Since $buff(p,C) \subseteq buff(p,D)$,and no message is removed from a buffer by τ' that is not removed by σ', m is still in $buff(p,E)$.

Suppose m is not in $buff(p,C)$. Then by assumption on σ, q is in S. Let $\sigma'' g$ be the prefix of σ' such that $(\sigma'' g)(C)$ is when m first appears in p's buffer. Thus, q sends m as a result of event g in $run(C,\sigma')$. Since q is in S, $\tau'' g$ is a prefix of τ', where $\tau'' = deafen(S',\sigma'')$. By the induction hypothesis, τ'' is applicable to D, so by Lemma 21, $state(q,\sigma''(C)) = state(q,\tau''(D))$. By the inductive hypothesis, since the length of $\sigma'' g$ is less than l, g is applicable to $\tau''(D)$. Thus m is also sent in $run(D,\tau')$, and m is in p's buffer in E. □

The next theorem shows that for any protocol, there is some finite run that computes the wrong decision value, if no more than half the processors are nonfaulty.

Theorem 23: *There is no t-nonblocking transaction commit protocol if $n \leq 2t$.*

Proof: Suppose $n \leq 2t$ and that there is a t-nonblocking transaction commit protocol with processors 0 through $n-1$.

Let $A = \{0, \ldots, t-1\}$ and $B = \{t, \ldots, n-1\}$. Each of A and B has at most t elements. The first t events of a cycle form an *A-semicycle* (each processor in A takes a step); the remaining events of a cycle form a *B-semicycle* (each processor in B takes a step). An infinite schedule applicable to an initial configuration consists of alternating A- and B-semicycles.

Let I_{11} be the initial configuration in which all processors have initial value 1. Since the protocol is a t-nonblocking transaction commit protocol, given an adversary that kills no processors and delivers in cycle $j+1$ any message sent in cycle j (so every run is failure-free and on-time), there is at least one finite deciding run $run(\alpha, I_{11})$ such that all processors have decided 1 in $\alpha(I_{11})$. Let $\alpha = \pi_1 \ldots \pi_y$ where each π_i is a semicycle.

Claim: There exist $y+1$ finite failure-free schedules α_1 through α_{y+1} such that for each i, (1) $\alpha_i = \pi_1 \ldots \pi_{i-1}\gamma_i$, (2) α_i is applicable to I_{11}, (3) all processors have decided 1 in $\alpha_i(I_{11})$, and (4) no intergroup message is received in γ_i.

Proof of Claim: We show the claim by descending induction on i. Let $C_i = (\pi_1 \ldots \pi_i)(I_{11})$ for $i \geq 1$, and $C_0 = I_{11}$.

Basis: $i = y+1$. Letting $\alpha_{y+1} = \alpha$ (so that γ_{y+1} is empty) proves the claim.

Induction: $i < y+1$. We assume the claim is true for $i+1$ and show it for i.

Assume π_i is a B-semicycle, i.e., i is even. (We will indicate in parentheses the changes necessary when π_i is an A-semicycle, i.e., when i is odd.) If no processor in B receives any message from a processor in A in π_i, then letting $\gamma_i = \pi_i\gamma_{i+1}$ satisfies properties (1) through (4).

Suppose some processor in B receives a message from some processor in A in π_i. We construct γ_i in two steps; first we construct β_1, after which all processors in A have decided, and then we construct β_2, in which all processors in B decide. Then γ_i will be $\beta_1\beta_2$.

Define β_1 to be $deafen(B, \pi_i\gamma_{i+1})$. (See Figure 1.) By Lemma 22, β_1 is applicable to C_{i-1}. Since $\beta_1|A = \pi_i\gamma_{i+1}|A$, Lemma 21 applies and each processor in A has the same state in $\beta_1(C_{i-1}) = F$ as it does in $(\pi_i\gamma_{i+1})(C_{i-1})$, so each decides 1 in F. No intergroup message is received in β_1 because processors in B receive no messages in β_1, and processors in A receive no intergroup messages in $\pi_i\gamma_{i+1}$ or in β_1.

$$\begin{tikzcd}
I_{11} = C_0 \cdots & C_{i-2} \arrow[r, "\pi_{i-1}", "A\text{-s.c.}"'] & C_{i-1} \arrow[r, "\pi_i", "B\text{-s.c.}"'] \arrow[d, "\beta_1 = deafen(B, \pi_i\gamma_{i+1})"'] & C_i \arrow[d, "\gamma_{i+1} \; A \not\leftarrow B \; B \not\leftarrow A"] & \cdots \; C_y = \alpha(I_{11}) \\
 & & F \; \boxed{A:1} & \boxed{A,B:1} & \boxed{A,B:1}
\end{tikzcd}$$

Figure 1: Construction of β_1

Now we must give a schedule β_2 that causes processors in B to decide 1 without hearing from any processors in A. The intuition is that processors in B must be able to decide without hearing from processors in A, because it is possible that all the processors in A have died. By the agreement condition, the processors in B must decide 1 also. The problem with applying this argument is that there may be leftover messages sent by processors in A before the point at which the processors in B think they died, and thus processors in B could wait to receive these messages before deciding. Thus, we must show that processors in A might have died even earlier.

Semicycle π_i is part of cycle number $\lceil i/2 \rceil = j$ in α_i. (See Figure 2.) Let D be the configuration in $run(\alpha_i, I_{11})$ immediately preceding the $(j-1)^{st}$ cycle of α_i. (If $j = 1$, then let $D = I_{11}$.) Let τ be the substring of α_i between I_{11} and D. Let ρ be the substring of α_i between D and C_{i-1}. There are two possibilities for ρ.

- If $i = 2$, then $D = I_{11}$ and $\rho = \pi_1$. Thus, ρ is an A-semicycle.
- If $i > 2$, then $D = C_{i-4}$ and $\rho = \pi_{i-3}\pi_{i-2}\pi_{i-1}$. Thus, ρ consists of all of cycle

$j-1$ and the first half of cycle j (an A-semicycle followed by a B-semicycle followed by another A-semicycle). (Pictured in Figure 2.)

(If π_i is an A-semicycle, i.e., if i is odd, then there are the following two possibilities for ρ.

- If $i=1$, then $D=I_{11}$ and ρ is empty.
- If $i>1$, then $D=C_{i-3}$ and $\rho=\pi_{i-2}\pi_{i-1}$. Thus, ρ consists of cycle $j-1$ (an A-semicycle, followed by a B-semicycle).)

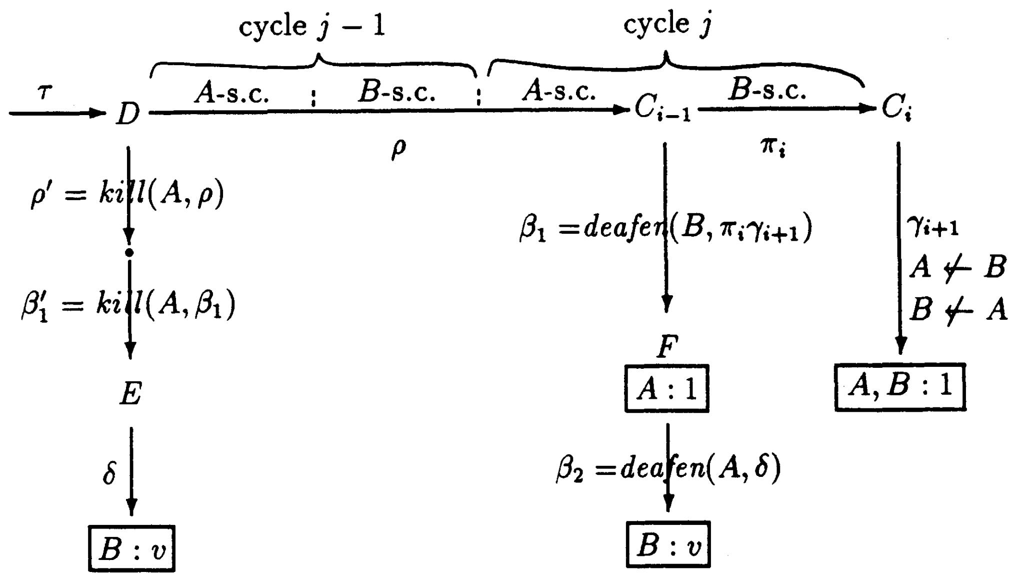

Figure 2: Construction of β_2

Let $\rho' = kill(A,\rho)$. Since no message is sent and received in the same cycle in α (and hence in ρ), any message received in ρ by a processor p in B from a processor in A is sent in $run(\tau, I_{11})$, i.e., prior to cycle $j-1$, and is in $buff(p,D)$. By Lemma 22, ρ' is applicable to D. Since $\rho|B = \rho'|B$, Lemma 21 implies that $state(p,\rho'(D)) = state(p,C_{i-1})$ for all p in B.

Consider the schedule $\beta_1' = kill(A,\beta_1)$. (See Figure 2.) Since the processors in A are failed and the processors in B receive no messages, β_1' is obviously applicable to $\rho'(D)$. Let $E=\beta_1'(\rho'(D))$. Since $\beta_1'|B = \beta_1|B$ and $state(p,\rho'(D)) = state(p,C_{i-1})$ for all p in B, Lemma 21 implies that $state(p,E) = state(p,F)$ for all p in B.

By the t-nonblocking property, since $|A| \leq t$, there must exist a finite deciding run from E with schedule δ. Suppose the decision value is v. Thus, all processors in B decide v in $\delta(E)$. By choice of α, all messages sent in $run(\tau, I_{11})$, i.e., before cycle $j-1$, are received by the end of cycle $j-1$, i.e., by the end of ρ or earlier. Since $\rho'|B = \rho|B$, every processor in B receives in ρ' all messages sent to it in $run(\tau, I_{11})$, i.e., before cycle $j-1$. Thus in δ, processors in B receive only messages sent in $run(\rho'\beta_1'\delta, \rho'(D))$. Since all processors in A are dead in $\rho'\beta_1'\delta$, B receives no intergroup messages in δ.

Let $\beta_2 = deafen(A, \delta)$. Pick p in B. From above, $state(p, E) = state(p, F)$. Let m be any message in $buff(p, E)$; m could only have been sent by a processor q in B in $run(\rho'\beta_1', D)$, i.e., in cycle $j-1$ or later. Lemma 21 implies that q has the same state in corresponding configurations in $run(\rho'\beta_1', D)$ and $run(\rho\beta_1, D)$. Thus q sends the same messages in the two runs, and m is also in $buff(p, F)$. Now we can apply Lemma 22 to show that β_2 is applicable to F.

Since $\beta_2|B = \delta|B$ and $state(p, F) = state(p, E)$ for all p in B, Lemma 21 implies that each processor p in B is in the same state in $\beta_2(F)$ as in $\delta(E)$. So each processor in B decides v in $\beta_2(F)$; by the agreement condition, $v = 1$, because processors in A have already decided 1 in F. No intergroup message is received in β_2 because none is received in δ.

Let $\gamma_i = \beta_1\beta_2$. We have shown that $\alpha_i = \pi_1 \ldots \pi_{i-1}\gamma_i$ satisfies properties (1), (2), (3) and (4). *End of Claim.*

Note that α_1 is a finite schedule in which no intergroup messages are received. Construct schedule $\sigma = kill(A, \alpha_1)$. By Lemma 22, σ is applicable to I_{11}. Since $\sigma|B = \alpha_1|B$, Lemma 21 implies that each processor in B has the same state in $\sigma(I_{11})$ as it does in $\alpha_1(I_{11})$, and thus also decides 1 in $\sigma(I_{11})$.

Let I_{01} be the initial configuration in which all processors in A have initial value 0 and all processors in B have initial value 1. By Lemma 22, σ is applicable to I_{01}. Since each processor in B begins with the same state in I_{01} as in I_{11}, by Lemma 21 each has the same state in $\sigma(I_{01})$ as it does in $\sigma(I_{11})$, and thus also decides 1 in $\sigma(I_{01})$. But this violates the abort validity condition. □

5. Lower Bound on Time

One might imagine a transaction commit protocol for our model such that each processor could decide in a constant number of its own steps, at least in many runs.

For instance, in the protocol presented in Section 3, at most $6K$ steps are required for a processor to complete stage 0 — a processor need not wait arbitrarily long for messages since the existence of a late message means that the processor is allowed to abort. Yet in the subsequent stages, no advantage is taken of this flexibility, and processors wait potentially unbounded time for messages. Unfortunately, the intuition that it may be possible to use the detection of late messages in order to shorten the running time (as measured in processor steps) is incorrect. In fact, in this section we prove that no protocol can guarantee that each processor terminate in a constant expected number of its own steps, even if processors run in lockstep synchrony, and even if only one processor can fail.

In particular, we show that for any constant B, there is a 1-admissible adversary and an initial configuration such that the expected number of cycles needed for all nonfaulty processors to decide is more than B. The proof is constructed as follows. We consider the initial configuration in which all processors begin with 1, and the adversary that kills no processors and delivers all messages with delay 1. If no run from this initial configuration with this adversary is deciding by cycle B, we are done. Suppose there is such a B-cycle run that is deciding. We find a point in this run that has the property there are some very long runs extending from this point that are not deciding. These runs are kept undeciding by delaying the delivery of all messages. These runs are so long that they cause the expected value to exceed B, when calculated with the appropriate initial configuration and adversary.

Thus, we must solve two subproblems. First, we must find the appropriate point in the run from which the long runs branch off (cf. Lemma 24); second, we must show that the long runs extending from this point are undeciding (cf. Lemma 25).

We need the following definitions in addition to the definitions and Lemmas 21 and 22 from Section 4.

If p is a processor, then schedule σ is *p-free* if p only takes failure steps in σ.

A run is *x-slow* for some constant x if every message received in the run has delay at least x. Given a configuration C, a schedule σ is *x-slow relative to C* if the run obtained by applying σ to C is x-slow.

A *seed* (for protocol P) is an n-tuple of sequences of n-bit strings, such that either each sequence is infinite or each sequence has the same number of elements. The *length* of a seed is the length of one sequence. If seed F has infinite length, then F is in $\mathcal{F}$. There is a finite number of seeds of any finite length.

A run is ***F*-compatible,** for seed F, if for all processors p and all i not exceeding the length of F, the random string that p receives in its i^{th} step is the same as the i^{th} element of p's sequence in F. Given configuration C, a schedule σ is ***F*-compatible relative to *C*** if C is reachable by an F-compatible run and $run(C,\sigma)$ is F-compatible.

For the remainder of this section, we fix an arbitrary 1-nonblocking transaction commit protocol P. From now on, "run" means a 1-admissible run of P, and "configuration" means a configuration reachable from some initial configuration of P by a 1-admissible run of P.

Let V be a subset of $\{0,1\}$, x an integer, and F a seed. Configuration C is $\{x,F,V\}$***-valent*** if V is the set of decision values of all configurations that are reachable from C by an x-slow F-compatible run.

For the rest of this section, let I_1 be the initial configuration in which all processors have initial value 1.

The next lemma shows that in an F-compatible run that decides 1, there exists a configuration from which some F-compatible, x-slow run decides 1, and from which some other F-compatible, x-slow run decides 0.

Lemma 24: *If $run(I_1,\tau)$ is a finite failure-free on-time deciding run that is F-compatible for finite seed F, then for any integer $x > 0$ there exists a configuration in $run(I_1,\tau)$ that is $(x,F,\{0,1\})$-valent.*

Proof: Pick such a run $run(I_1,\tau)$ that is F-compatible, and fix x. By the commit validity condition, $\tau(I_1) = C$ has decision value 1. Thus all runs starting at C, including x-slow F-compatible runs, have decision value 1, and hence C is $(x,F,\{1\})$-valent.

Let I_{01} be the initial configuration in which some processor q has initial value 0 and the rest have initial value 1. Since the protocol is 1-nonblocking and since F is finite, there is a finite q-free x-slow F-compatible run $run(\sigma,I_{01})$ such that $\sigma(I_{01})$ has decision value 0, and by the agreement condition, $\sigma(I_{01})$ is $(x,F,\{0\})$-valent.

By Lemma 22, σ is also applicable to I_1. By Lemma 21, all processors except q have the same state in $\sigma(I_1)$ as in $\sigma(I_{01})$, and decide 0 in $\sigma(I_1)$. Thus I_1 is either $(x,F,\{0\})$-valent or $(x,F,\{0,1\})$-valent. If the latter is true, we are done. Suppose the former is true.

Since F is finite, by the 1-nonblocking property no configuration in $run(I_1, \tau)$ is $(x, F, \emptyset)$-valent. The valencies of I_1 and C imply that there must be an event $e = (p, M, b)$ and two adjacent configurations in $run(I_1, \tau)$, C_0 and C_1 with $C_1 = e(C_0)$, such that C_0 is either $(x, F, \{0\})$-valent or $(x, F, \{0, 1\})$-valent, and C_1 is either $(x, F, \{1\})$-valent or $(x, F, \{0, 1\})$-valent. (See Figure 3.)

$$\begin{tikzcd}
I_1 \arrow[r] \arrow[dd, "\sigma"'] & C_0 \arrow[r, "e = (p, M, b)"] & C_1 \arrow[r] & C = \tau(I_1) \\
 & \boxed{0} \arrow[d, "e' = (p, \perp, b')"'] & \boxed{1} \arrow[d, "\alpha'"'] & \boxed{1} \\
\boxed{0} & \bullet \arrow[d, "\alpha'"'] & \boxed{0} & \\
 & \boxed{0} & &
\end{tikzcd}$$

Figure 3: Demonstrating the existence of an $(x, F, \{0, 1\})$-valent configuration

If either configuration is $(x, F, \{0, 1\})$-valent, we are done. Say neither is. Since the protocol is 1-nonblocking, F is finite, no processor has failed so far, and C_0 is $(x, F, \{0\})$-valent, there is a finite p-free x-slow F-compatible run $run(\alpha, C_0)$ in which the nonfaulty processors decide 0. Say $\alpha = (p, \perp, b')\alpha'$. (If F is long enough to extend past C_0, then $b' = b$; otherwise, b' could differ from b.) Since α' is applicable to C_1, Lemma 21 implies that all the processors except p have the same state in $\alpha'(C_1)$ as they do in $\alpha(C_0)$. But since they decide 0 in $\alpha(C_0)$, and since α' is F-compatible and x-slow relative to C_1, this is a contradiction to the hypothesis that C_1 is $(x, F, \{1\})$-valent. □

The next lemma shows that in a certain situation, processors must remain undecided as long as no messages are received.

Lemma 25: *Let A be the adversary that kills no processors, and that for the first l events delivers messages after delay 1 and subsequently delivers messages after delay*

x, for some $x > l$. Let F be a seed of length x. If the configuration C following the l^{th} event in $run(A, I_1, F)$ is $(x, F, \{0,1\})$-valent, then the final configuration in $run(A, I_1, F)$ is $(x, F, \{0,1\})$-valent.

Proof: Let $run(A, I_1, F) = run(\alpha\sigma, I_1)$, where $C = \alpha(I_1)$. Assume in contradiction that $\sigma(C)$ is not $(x, F, \{0,1\})$-valent. Since F is finite, by the 1-nonblocking property, $\sigma(C)$ cannot be $(x, F, \emptyset)$-valent. Assume $\sigma(C)$ is $(x, F, \{v\})$-valent. Then there is a configuration D in $run(\sigma, C)$ and some event $e = (p, M, b)$ in σ such that D is $(x, F, \{0,1\})$-valent and $e(D)$ is $(x, F, \{w\})$-valent. M must be the empty set, since no messages are received in $run(\sigma, C)$. Suppose $w = 0$. (The argument is analogous if $w = 1$.) The only other event applicable to D that can be part of an x-slow F-compatible run is $(p, \perp, b) = e'$, because all messages sent more than x cycles ago have delay 1 and have already been received, and because F is long enough to extend to e. (See Figure 4.)

$$\begin{tikzcd}
I_1 \arrow[r, "\text{delay } 1"] & C \arrow[r] & D \arrow[r, "e = (p,\emptyset,b)"] & e(D) \arrow[r] & \sigma(C) \\
& \boxed{0/1} & \boxed{0/1} \arrow[d, "e' = (p,\perp,b)"] & \boxed{0} \arrow[d, "\tau"'] & \boxed{0} \\
& & \boxed{0/1} \text{ or } \boxed{1} \arrow[d, "\tau"'] & \boxed{1} & \\
& & \boxed{1} & &
\end{tikzcd}$$

Figure 4: Demonstrating that $\sigma(C)$ is $(x, F, \{0,1\})$-valent

Since D is $(x, F, \{0,1\})$-valent, $e'(D)$ must be either $(x, F, \{0,1\})$-valent or $(x, F, \{1\})$-valent. Thus there is some finite p-free x-slow F-compatible run from $e'(D)$ that has decision value 1; let τ be its schedule. Now τ is also applicable, x-slow and F-compatible relative to $e(D)$, and all processors except p have the same state in $\tau(e(D))$ as in $\tau(e'(D))$ (by Lemma 21), so they decide 1, contradicting the valency of $e(D)$. □

Given infinite run R, let $T(R)$ be the cycle when the last nonfaulty processor decides.

Theorem 26: *For any constant B, there is a 1-admissible adversary A and an initial configuration I such that $E(T_{A,I}) \geq B$.*

Proof: Fix B. Let $\mathcal{R}$ be the set of all runs of the form $run(A_1, I_1, F)$, where F is a seed of length B, and A_1 is the adversary that kills no processors and delivers all messages with delay 1. Let $|\mathcal{R}| = j$. Thus, j is also the number of seeds of length B.

Case 1: No run in $\mathcal{R}$ is deciding. Let $A = A_1$ and $I = I_1$. Then $E(T_{A,I}) \geq B$.

Case 2: There is some run R in $\mathcal{R}$ that is deciding. Let $\mathcal{C}$ be the set of all configurations in run R, and let $m = |\mathcal{C}|$. Let $\mathcal{S}$ be the collection of all seeds with length jmB that extend the seed of R. $\mathcal{S}$ is finite; in fact, $|\mathcal{S}| = z/j$, where z is the total number of seeds of length jmB.

We will associate each seed in $\mathcal{S}$ with a configuration in $\mathcal{C}$ in such a way that all runs from a configuration in $\mathcal{C}$, using a particular adversary and any of the associated seeds, is undeciding. The extreme length of these undeciding runs will cause the desired expected value to exceed B.

For each $C \in \mathcal{C}$, define $\mathcal{S}(C)$ to be the set of all $F \in \mathcal{S}$ such that C is the first $(jmB, F, \{0,1\})$-valent configuration in R. By Lemma 24, at least one $(jmB, F, \{0,1\})$-valent configuration exists in R; thus, each $F \in \mathcal{S}$ is in $\mathcal{S}(C)$ for exactly one configuration C.

Fix C to be a configuration in $\mathcal{C}$ with $|\mathcal{S}(C)| \geq \frac{1}{m} \cdot |\mathcal{S}|$. Such a configuration exists by the pigeonhole principle, since $|\mathcal{C}| = m$. Thus, $|\mathcal{S}(C)| \geq \frac{1}{jm} \cdot z$.

Let l be the number of events that precede C in run R. Let A be the adversary that for the first l events delivers messages after delay 1 and that subsequently delivers messages after delay jmB. By Lemma 25, for every F in $\mathcal{S}(C)$, the final configuration of $run(A, I_1, F)$ is $(jmB, F, \{0,1\})$-valent. Thus, no processor has decided in that final configuration, and $T(R') > jmB$, for any infinite run R' that is an extension of $run(A, I_1, F)$.

Let $I = I_1$. By choice of C, at least a $\frac{1}{jm}$ fraction of all the seeds of length jmB are in $\mathcal{S}(C)$. Thus, at least a $\frac{1}{jm}$ fraction of all infinite seeds have a prefix in

$S(C)$. For any infinite seed F with a prefix in $S(C)$, $T(run(A, I, F)) > jmB$, by the argument above. As a result,

$$E(T_{A,I}) \geq \frac{1}{jm} \cdot jmB = B. \qquad \square$$

3 Simulating Synchronous Processors

In this chapter we show how a distributed system with synchronous processors and asynchronous message delays can be simulated by a system with both asynchronous processors and asynchronous message delays in the presence of various types of processor faults. Consequently, the result of Fischer, Lynch and Paterson, that no consensus protocol for asynchronous processors and communication can tolerate one failstop fault, implies a result of Dolev, Dwork and Stockmeyer, that no consensus protocol for synchronous processors and asynchronous communication can tolerate one failstop fault.

1. Introduction

In this chapter we show how a distributed system with synchronous processors and asynchronous message delays can be simulated by a system in which both processors and messages are asynchronous, in the presence of various types of processor failures. One application of this result is that now a result in [DDS], that no fault-tolerant consensus protocol is possible in a distributed system with asynchronous communication even if processors are synchronous, follows easily from the result in [FLP], that no fault-tolerant consensus protocol is possible when communication and processors are asynchronous.

The equivalence of a system with synchronous processors and asynchronous communication to one in which both processors and communication are asyn-

This chapter was published in *Information and Computation* [W].

chronous has been a folk theorem in distributed computing circles for some time. One of the contributions of this chapter is to present a careful statement and proof of this result, using a variant of Lamport clocks [La1]. We have made precise a notion of simulation particularly suited to showing impossibility results. The novel feature of this chapter is applying the simulation result to obtain an easy proof of the impossibility of fault-tolerant consensus for synchronous processors and asynchronous communication.

The sense in which we show that the two systems are equivalent is that no processor can tell if it is in one system or the other. Of course, an outside observer can tell the difference. For instance, if all the processors are to perform some action at their tenth step, the effect could be quite different with synchronous processors (where the actions would happen at the same real time) than with asynchronous processors (where the actions do not necessarily happen at the same real time). Thus, the notion of simulation that we define preserves local views, but not global views.

We observe that the only situation visible to a processor in the system with asynchronous processors that cannot happen in the system with synchronous processors is for the processor to receive a message at its i^{th} step that was sent at the sender's j^{th} step, where $j \geq i$. To avoid this anomalous situation, our simulation tags all messages with the sender's current step number; then processors save messages that arrive too early, and wait to process them until they are no longer early. (Compare Lamport clocks, which cause the local clock, or step counter, to skip ahead when a message with too large a timestamp arrives.)

Neiger and Toueg [NT] have independently developed the same simulation technique. However, they do not consider faults, and they apply the simulation to different problems, namely, determining when one can substitute these modified Lamport clocks for real time clocks while maintaining correctness, and determining when a variant of common knowledge, achieved with the help of this simulation, can be substituted for the standard notion of common knowledge. Their paper formally characterizes types of behavior that can be preserved by this simulation.

Our formal model is presented in Section 2. In Section 3 we show how to do the simulation for Byzantine processor faults. Simplifications for weaker fault models are presented in Section 4. Finally, Section 5 demonstrates that the result in [DDS] follows from that in [FLP].

2. Model

We model a general distributed system in which processors communicate by sending messages. Conceptually, there is a global clock that measures time in integer ticks. At each tick, some processors take steps, in which they can atomically receive messages, change state and send messages. A message buffer holds messages between the sending and receiving times. A protocol determines for each processor the state changes and messages sent, given the old state and messages received. A run of the protocol specifies at each tick which processors take steps and which messages are received. Various kinds of faulty processor behaviors are introduced next. After formally defining what a system is in this general model, we define the type of simulation we are concerned with.

2.1 Basic Model

Messages are assumed to be unique and are tagged with both the sender's and recipient's names by the message system. The *message buffer* holds messages that have been sent but not yet received. It is modeled as a set of messages. A *processor* is a deterministic state machine with a set of states, and a transition function that uses the current state and messages received to compute the new state and messages to be sent (at most one message to each processor). Certain states are designated *initial* states. A *protocol* is a set of n processors. In our terminology, a processor is more than just bare hardware — it includes the local algorithm for changing state and sending messages. A protocol is the collection of all the local algorithms.

A *step* of processor p is designated either α, indicating that p does some computation, or λ, indicating that p does nothing. An α step is an *active* step. A *processor history* for processor p, H_p, consists of an infinite sequence $d_1 s_1 d_2 s_2 \ldots$ of states d_i of p alternating with steps s_i of p such that d_1 is an initial state, and if $s_i = \lambda$, then $d_i = d_{i+1}$. The i^{th} state of H_p is denoted $state(H_p, i)$, and the i^{th} step $step(H_p, i)$. Given processor history H_p and integer i, define $active(H_p, i)$ to be the number of active steps in H_p up to and including the i^{th} step. A *message buffer history* H_B is an infinite sequence $M_1 M_2 \ldots$, where each M_i is a set of messages and $M_1 = \emptyset$, such that if message m is in M_i and not in M_{i+1}, then m is not in M_j for any $j > i$. The i^{th} element of H_B is denoted by $msgs(H_B, i)$.

A *run* R of protocol P consists of n processor histories H_p, one for each processor p in P, and a message buffer history H_B such that the following are true. Suppose message m has sender p and recipient q, and i is the smallest integer such that m is in $msgs(H_B, i)$. (1) Then $step(H_p, i-1)$ is active. We say m is *sent* by p at step $i - 1$. (2) Furthermore, if j is the greatest integer such that m is in $msgs(H_B, j)$, then $step(H_q, j)$ is active. We say m is *received* by q at step j.

Given a processor history H_p, define $states(H_p)$ to be the (finite or infinite) sequence of states $d_1 d_2 \ldots$, where $d_1 = state(H_p, 1)$ and d_{i+1} is the state following the i^{th} active step in H_p. (The do-nothing steps have been eliminated and the state transitions isolated.) For a run $R = \langle H_B, \{H_p\}_{p \in P} \rangle$, define $states(R)$ to be $\{states(H_p)\}_{p \in P}$.

Various types of processor faults are now considered, classified by their observable effects. Suppose processor p has processor history $H_p = d_1 s_1 d_2 s_2 \ldots$ in run R. Fix i and let M be the set of messages received by p at step s_i, and let M' be the set of messages sent by p at step s_i. Processor p *operates correctly* at step s_i, if d_{i+1} is the result of p's transition function applied to d_i and M, and if M' is exactly the set of messages returned by p's transition function applied to d_i and M. Processor p *exhibits an omission failure upon sending* at s_i if d_{i+1} is the result of p's transition function applied to d_i and a subset S of M, and M' is a strict subset of the set of messages returned by p's transition function applied to d_i and S. Processor p *exhibits an omission failure upon receiving* at s_i if p does not operate correctly at s_i, but p's transition function applied to d_i and a strict subset of M produces d_{i+1} and a set of messages of which M' is a subset. A message not used by the transition function, or not placed in the message buffer is *omitted.* (Note that these definitions allow a processor to exhibit an omission failure upon both sending and receiving at the same step.) Processor p *exhibits a Byzantine failure* at s_i if d_{i+1} and M' cannot be described as the result of p's operating correctly, or p's exhibiting an omission failure upon sending or receiving.

Processor p is *nonfaulty* in run R if it takes an infinite number of active steps and operates correctly at each one; otherwise p is *faulty.* Faulty processor p is *failstop-faulty* in run R if it takes only a finite number of active steps and operates correctly at each one. Faulty processor p is *omission-faulty* in run R if p is not failstop-faulty, and at each active step p either operates correctly or exhibits an omission failure upon sending or receiving. Faulty processor p is *Byzantine-faulty* in run R if p is not failstop-faulty or omission-faulty, and at each active step p operates correctly, exhibits an omission failure, or exhibits a Byzantine failure.

The next definition concerns communication faults. A message m sent in an infinite run is *lost* if the recipient takes infinitely many active steps but never receives m.

2.2 Systems

We are interested in restricting the allowable runs (of any protocol) in different ways. Fix a protocol P. Let $runs(P)$ be the set of all runs of P. Define the universe of all runs, U, to be $\bigcup_{\text{all } P} runs(P)$. A *system* is a subset of U. The system U can be characterized as having unreliable, asynchronous communication, since it includes runs in which messages are lost and runs in which messages remain in the buffer for arbitrarily long periods of time. Similarly, U has asynchronous processors, since there is no restriction on the number of λ steps between consecutive active steps in a processor history. There is also no restriction on the number or types of processor faults exhibited, when all the runs of U are considered.

The following systems are used as building blocks in this chapter.

- *System SP*: the set of all runs such that if a processor takes a λ step, then all subsequent steps of that processor are λ steps. This system has synchronous processors. The processors can know the global clock value, because it is the same as the number of active steps they have taken.

- *System RC*: the set of all runs such that no messages are lost. This system has asynchronous, but reliable, communication.

We can restrict the number and type of faults to be considered by defining:

- *System FS(t)*: the set of all runs such that at most t processors are failstop-faulty, and the rest are nonfaulty.

- *System OM(t)*: the set of all runs such that at most t processors are omission-faulty or failstop-faulty, and the rest are nonfaulty.

- *System BZ(t)*: the set of all runs such that at most t processors are Byzantine-faulty, omission-faulty or failstop-faulty, and the rest are nonfaulty.

2.3 Simulations

A *simulation function* $f_{p'}$ for processors p' and p is a function from states of p' to states of p. Extend $f_{p'}$ to map sequences of states of p' to sequences of states of p by defining $f_{p'}(d_1 d_2 \ldots) = f_{p'}(d_1) f_{p'}(d_2) \ldots$.

Run $R' = \langle H_{B'}, \{H_{p'}\}_{p' \in P'} \rangle$ of protocol P' *simulates* run $R = \langle H_B, \{H_p\}_{p \in P} \rangle$ of protocol P via set $F = \{f_{p'} : p' \in P'\}$ of simulation functions, if there exists a one-to-one correspondence c between processors of P' and processors of P with the following properties. Fix p' in P', and let $p = c(p')$. (1) The simulation function $f_{p'}$ for p' and p satisfies $f_{p'}(states(H_{p'})) = states(H_p)$. (2) If p' is nonfaulty in R', then p is nonfaulty in R. We say processor p' *simulates* processor p for runs R and R' via $f_{p'}$. (The simulation function $f_{p'}$ does not necessarily cause p' to simulate p for other pairs of runs.)

Protocol P' in system A' *simulates* protocol P in system A if there exists a set F of simulation functions such that (1) for every run R' of P' in system A', there exists a run R of P in system A such that R' simulates R via F, and (2) for every run R of P in system A, there is a run R' of P' in system A' such that R' simulates R via F. We call P' a *simulation protocol* for P relative to A' and A.

System A' *simulates* system A if, for any protocol P, there exists a protocol P' such that protocol P' in system A' simulates protocol P in system A.

This definition of simulation is very strong, since the correspondence between runs of the simulation protocol and runs of the original protocol must be onto. However, for showing lower bounds or impossibility results, this strength is good, and in fact is necessary for the application in Section 5. A more appropriate definition for upper bounds would not require the correspondence to be onto, but would need some condition on the responses of the simulation protocol to various inputs of the original protocol, in order to rule out trivial solutions. As discussed in the introduction, this definition of simulation concentrates on the sequences of individual processors' state transitions, and is not concerned with global behavior that is only detectable by an observer outside the system.

3. Simulating Synchronous Processors with Byzantine Faults

Our goal is to show that if the communication system is asynchronous, then synchronous processors "don't help" — *i.e.,* a system with asynchronous processors and asynchronous communication can simulate (the state transitions of) a system with synchronous processors and asynchronous communication, even if there is any number of Byzantine-faulty processors. The main idea of the simulation is for each asynchronous processor to keep track of how many active steps it has taken and append this number on each message (of the synchronous protocol) sent. The only situation visible to the processors in the asynchronous case that cannot occur in the synchronous case is for a processor at its i^{th} active step to receive a message that was sent at the sender's j^{th} active step, where $j \geq i$. To avoid this anomaly, such "early" messages are simply saved up until the recipient has passed its j^{th} active step, and then they are used in the simulation.

Although the model of computation presented in this chapter gives processors the ability to receive and send messages in the same atomic step, and to send messages to all the processors at one step, this power is not necessary for the simulation to work. If the model is weakened so that processors can send at most one message at a step, or can only send or receive at a step, but not both, (as studied in [DDS]) the same simulation will show that asynchronous processors can simulate synchronous processors when communication is asynchronous.

Subsection 3.1 describes the simulation protocol for a given synchronous protocol in more detail. In Subsection 3.2, we show how to map a run of the simulation protocol to a run of the simulated protocol. The proof of the main result is presented in Subsection 3.3.

3.1 Simulation Protocol

Fix t between 1 and n. Let system S1(t) be the intersection of systems BZ(t) and RC and SP. This is the system with at most t Byzantine-faulty processors, reliable asynchronous communication and synchronous processors. Let system A1(t) be the intersection of systems BZ(t) and RC. This is the system with at most t Byzantine-faulty processors, reliable asynchronous communication and asynchronous processors.

Fix a protocol P. We define a simulation protocol P' for P relative to A1(t) and S1(t) as follows. Each processor p' in P' is assigned a processor p in P to simulate; it knows the states and transition function for p as well as the processor correspondence c. Each state d of p' has a component $d.sim$. It also has components $d.early$, which is a set of messages (to be described below), and $d.counter$, which tells the sequence number of the next active step p' will take. Every message m that p' sends in the step following state d has the value of $d.counter$ appended to it, in a tag called $m.tag$. Each processor also keeps the necessary information to decide if message m from p' is the first message from p' with the tag value $m.tag$. (More than one such message is only sent if p' is Byzantine-faulty.)

We first describe the states of p'. An initial state d of p' has $d.sim$ equal to an initial state of p, $d.early = \emptyset$ and $d.counter = 1$. There is one initial state of p' for each initial state of p. Non-initial states are obtained by starting from an initial state and applying p''s transition function (some number of times).

We now describe p''s transition function. Suppose that p' is in state d and receives the set of messages M. Let E be the set of all messages m in $M \cup d.early$ such that m is the first message received from the sender with the tag value $m.tag$. Let M' be the set of all messages m in E such that $m.tag < d.counter$. Then p' calculates the result of the transition function for p applied to $d.sim$ and M' (after removing the *tag* components of the messages and applying c to the sender's name). Call the results the state d'' and the message set M''. Let d' be the new state of p'; $d'.sim$ is set equal to d'', $d'.early$ is set equal to $E - M'$, and $d'.counter$ is set equal to $d.counter + 1$. The messages sent are those in M'', each tagged with $d.counter$.

3.2 Constructing Corresponding Runs

Pick a run $R' = \langle H_{B'}, \{H_{p'}\}_{p' \in P'} \rangle$ of P' in system A1(t). We describe a particular run R of protocol P corresponding to R'. (In the next subsection we show that R is in S1(t).)

We define the message buffer history H_B. Suppose processor p', at its a^{th} active step, sends message m' with tag b to processor q'. (As will be discussed in Section 4, if p' is not Byzantine-faulty, then $a = b$.) Let m be the message obtained from m' by deleting the tag and changing the sender to p and the recipient to q. If b is anything other than a positive integer (for instance, missing) or if m' is not the first message received by q' from p' with tag b, then nothing corresponding to m' is present in H_B. Otherwise, let $i = \min(a+1, b+1)$. (The goal is for m to be sent in R either at the same active step when p' actually sends m', or when p' claims, via the tag, to have sent it, whichever is earlier.) Suppose q' receives m' at its l^{th} active step. Let $j = \max(b+1, l)$. If m' is never received in $H_{q'}$, or if q' takes fewer than j active steps, then m is in $msgs(H_B, k)$ precisely for all $k \geq i$. Otherwise m is in $msgs(H_B, k)$ precisely for $i \leq k \leq j$. No other messages are present. Clearly H_B is a message history.

We define inductively the processor history $H_p = d_1 s_1 d_2 s_2 \ldots$ for processor p in P, which is simulated by processor p' in P'. Let $H_{p'} = d'_1 s'_1 d'_2 s'_2 \ldots$. For the basis, $d_1 = d'_1.sim$. Suppose the processor history up to d_i has been defined. If there are fewer than i active steps in $H_{p'}$, then $s_i = \lambda$ and $d_{i+1} = d_i$. Otherwise, $s_i = \alpha$, and $d_{i+1} = d'_j.sim$, where d'_j is the state following the i^{th} active step in $H_{p'}$. Clearly, the sequence H_p is a processor history for p in P.

Lemma 1: *$R = \langle H_B, \{H_p\}_{p \in P} \rangle$, as defined above, is a run of protocol P.*

Proof: We already know that the H_p's are processor histories for P. We must show that the message buffer behaves properly. Suppose message m has sender p

and recipient q, and i is the smallest integer such that m is in $msgs(H_B, i)$. (1) By construction of R, there exists a such that m' (m with *tag* b) is sent at p''s a^{th} active step, and $i - 1 = \min(a, b)$. Thus p' takes at least $i - 1$ active steps, so $step(H_p, i - 1)$ is active. (2) Suppose m is received in R. Let j be the greatest integer such that m is in $msgs(H_B, j)$. By construction of R, there exists l such that m is received at q''s l^{th} active step, $j = \max(b + 1, l)$, and q' takes at least j active steps. Thus, $step(H_q, j)$ is active. □

3.3 Results

This subsection contains the proof that the simulation protocol actually works. For the remainder of this section, fix a run R' of P' in $\mathrm{A1}(t)$, and construct run R from R' as above. Recall that processor p' in P' simulates processor p in P for runs R' and R.

Lemma 2: *Processor p' takes an infinite number of active steps in R' if and only if p takes an infinite number of active steps in R.*

Proof: By construction of R. □

Nonfaulty, sending omission-faulty and failstop-faulty behaviors are preserved by the simulation. However, if a processor p' exhibits an omission failure upon receiving in R' and the message omitted is early, then p in R may exhibit a weaker form of faulty behavior (or perhaps be nonfaulty). Similarly, if a processor p' exhibits a Byzantine failure in R' and the Byzantine nature of the error only affects the tag on a message, then p in R may exhibit a weaker form of faulty behavior (or perhaps be nonfaulty). Lemmas 3 and 4 demonstrate these facts.

Lemma 3: *If p' is not Byzantine-faulty and p' operates correctly at $step(H_{p'}, i)$, then p operates correctly at $step(H_p, j)$, where $j = active(H_{p'}, i)$.*

Proof: Suppose at $step(H_{p'}, i)$, p' applies p's transition function to the set of messages M', and that p receives the set of messages M at $step(H_p, j)$. The following argument shows that $M' = M$. We say that a message m' of R' and a message m of R *correspond* if the text is the same and the senders and recipients are corresponding processors (with respect to the simulation). Message m is in M' if and only if there is some corresponding message m' such that m' is the first message received from the sender in $H_{p'}$ with tag value $m'.tag$, $m'.tag$ is a positive integer, and $m'.tag < j$. These three conditions are true if and only if m is in M.

By construction of R, $state(H_p, j) = state(H_{p'}, i).sim$. Since p' operates correctly at $step(H_{p'}, i)$, and it applies p's transition function to $state(H_p, j)$ and M, and since $state(H_p, j+1) = state(H_{p'}, i+1).sim$, p changes state correctly at $step(H_p, j)$.

Suppose p' sends the set of messages N' at $step(H_{p'}, i)$ and p sends the set of messages N at $step(H_p, j)$. Since p' operates correctly, we can deduce that $state(H_{p'}, i).counter = j$, all the tags of messages in N' are equal to j, there is at most one message sent to each processor, and no other messages from p' have tag j (because p' is not Byzantine-faulty). Thus, if m' is in N', then a corresponding m is in N, and if m is in N, then a corresponding m' is in N'.

Thus, p sends the correct messages at $step(H_p, j)$. □

Lemma 4: *(a) If processor p' is nonfaulty in R', then processor p is nonfaulty in R.*

(b) If processor p' is failstop-faulty in R', then processor p is failstop-faulty in R.

(c) If processor p' is omission-faulty in R', then processor p is omission-faulty, failstop-faulty or nonfaulty in R.

Proof: Parts (a) and (b) follow from Lemmas 2 and 3.

(c) The hypothesis that p' is omission-faulty in R' is equivalent to assuming that at each active step (of which there are either a finite or infinite number), p' either operates correctly or exhibits an omission failure, and there is some active step at which p' exhibits an omission failure.

By Lemma 3, if p' operates correctly at $step(H_{p'}, i)$, then p operates correctly at $step(H_p, j)$, where $j = active(H_{p'}, i)$.

Suppose p' exhibits an omission failure upon sending at $step(H_{p'}, i)$. Then by construction of R, p exhibits an omission failure upon sending at $step(H_p, j)$, where $j = active(H_{p'}, i)$.

Suppose p' exhibits an omission failure upon receiving at $step(H_{p'}, i)$, and one of the messages omitted is m. Let $a = active(H_{p'}, i)$ and $m.tag = b$. If $b < a$, then by construction of R, p exhibits an omission failure upon receiving at $step(H_p, a)$ (p' should have used m in the simulation when m was received). If

$b \geq a$, then by construction of R, p could exhibit an omission failure upon receiving at $step(H_p, b+1)$ (p' should have saved m and used it in the simulation when its counter reached $b+1$). However, it might be the case that the presence or absence of message m is immaterial to p's state change and set of messages sent, in which case p operates correctly at $step(H_p, b+1)$.

Thus, at each active step in R, p either operates correctly, or exhibits an omission failure. The result follows. □

Lemma 5: *R is in system S1(t).*

Proof: R is in system SP since, by construction of R, once a processor takes a λ step, all subsequent steps are λ steps.

Since R' is in system BZ(t), at least $n-t$ processors are nonfaulty in R'. By Lemma 4, at least $n-t$ processors are nonfaulty in R. Thus, R is in system BZ(t).

Next we show that R is in system RC. Suppose message m is sent in R by processor p to processor q, and q takes infinitely many active steps. In R', p' sends message m' (m with *tag* b for some positive integer b) to q'. Since R' is in system RC, and since by Lemma 2 q' takes infinitely many active steps, m' eventually arrives in R', say at q''s l^{th} active step. Then m is received at $step(H_q, j)$, where $j = \max(b+1, l)$. □

Theorem 6: *System A1(t) simulates system S1(t), for any value of t, $1 \leq t \leq n$.*

Proof: Fix any protocol P. Let P' be the protocol defined above. We must show that protocol P' in system A1(t) simulates protocol P in system S1(t). Let the correspondence c between processors in P' and processors in P be that implicit in the construction of P'. Define a set $F = \{f_{p'} : p' \in P'\}$ of simulation functions as follows. Fix p' in P' and let $p = c(p')$. Define simulation function $f_{p'}$ from states of p' to states of p to be $f_{p'}(d') = d'.sim$.

The first direction is showing that for every run R' of P' in system A1(t), there exists a run R of P in system S1(t) such that R' simulates R via F. Given a run R' of P' in system A1(t), let R be the run constructed as above. By Lemma 1, R is a run of P. By Lemma 5, R is in system S1(t). Now we must show that R' simulates R via F. By construction of R, $f_{p'}(states(H_{p'})) = states(H_p)$. Furthermore, if p' is nonfaulty in R', then p is nonfaulty in R, by Lemma 4.

The second direction is showing that given a run R of P in S1(t), there is a run R' of P' in system A1(t) such that R' simulates R via F. The idea of the construction is to let processors in R' take the same steps at exactly the same ticks as do the processors they are simulating in R, and to let the message delays be exactly the same. The key is to observe that a run in which processors are synchronous is also in the system with asynchronous processors (*i.e.*, S1(t) is a subset of A1(t)). The following merely formalizes the idea and adds the appropriate tags to the messages.

Let $R = \langle H_B, \{H_p\}_{p \in P}\rangle$. Define a message buffer history $H_{B'}$ as follows. Suppose message m from processor p to processor q is in $msgs(H_B, i)$ for some i, and let b be the smallest integer such that m is in $msgs(H_B, b)$. Then message m', equal to m with *tag* $b - 1$, from processor p' to processor q', is in $msgs(H_{B'}, i)$. No other messages are in $msgs(H_{B'}, i)$.

Define processor history $H_{p'} = d'_1 s'_1 d'_2 s'_2 \ldots$ as follows. Let d'_1 be the initial state of p' with *sim* component equal to $state(H_p, 1)$. Suppose $H_{p'}$ has been defined up to d'_i. Then $s_i = step(H_p, i)$. If $s_i = \lambda$, then $d'_{i+1} = d'_i$; otherwise let $d'_{i+1}.sim = state(H_p, i+1)$, $d'_{i+1}.counter = d'_i.counter + 1$, and $d'_{i+1}.early = \emptyset$. This defines the states of $H_{p'}$.

It is straightforward to show that $R' = \langle H_{B'}, \{H_{p'}\}_{p' \in P'}\rangle$ is a run of P' in system A1(t), and that R' simulates R via F. □

4. Simulating Synchronous Processors with Weaker Faults

If the strongest type of processor fault allowed is omission, then the simulation and proofs can be slightly simplified. Fix t between 1 and n. Let system S2(t) be the intersection of systems OM(t) and RC and SP. Let system A2(t) be the intersection of systems OM(t) and RC. The same simulation as in Section 3 can be used, except it is no longer necessary to check if a message is the first one with that tag value. Since no Byzantine faults are considered, the message tag is always the correct active step count, so in constructing a run of the simulated protocol, variables a and b are always equal. Furthermore, Lemma 4 implies that each simulated processor has the same behavior (or better) as its simulating processor.

Theorem 7: *System A2(t) simulates system S2(t), for any value of t,* $1 \le t \le n$.

The same simplifications apply if the only type of faults is failstop. Fix t between 1 and n. Let system S3(t) be the intersection of systems FS(t) and RC and SP. Let system A3 be the intersection of systems FS(t) and RC.

Theorem 8: *System A3(t) simulates system S3(t), for any value of t,* $1 \le t \le n$.

5. Application

An important result in the theoretical study of distributed systems is that no consensus protocol operating in a system with asynchronous processors and asynchronous communication can be guaranteed to terminate, if it must tolerate even one failstop processor fault [FLP]. This result was subsequently extended [DDS] to show that no consensus protocol operating in a system with asynchronous communication, but with processors in lockstep synchrony, can be guaranteed to terminate, if it must tolerate even one failstop processor fault. The proof in [DDS] followed the spirit of the proof in [FLP], but required additional machinery and a more involved argument.

The result in [DDS] can be seen to be a corollary of the result in [FLP] using Theorem 8 of this chapter.

Given a system S, a *consensus protocol* P for S is a protocol that satisfies the following. (1) Each processor's set of non-initial states has two disjoint subsets, the *0-final* states and the *1-final* states. Once a processor enters a v-final state, it is always in a v-final state. (2) There exists a run of P in S in which a processor enters a 0-final state, and there exists a run of P in S in which a processor enters a 1-final state. (3) For every run of P in system S, if some processor enters a v-final state, then no processor enters a w-final state for $w \neq v$. (4) For every run of P in system S, some processor enters a v-final state, for some v.

The model in [FLP] corresponds in our model to the system A3(1) obtained from the intersection of systems FS(1) and RC, *i.e.*, the system with asynchronous processors, at most one of which is failstop-faulty, and reliable but asynchronous communication.

Theorem 9: *(Theorem I in [FLP]) There is no consensus protocol for system A3(1).*

The model in [DDS] corresponds in our model to the system S3(1) obtained from the intersection of systems FS(1) and SP and RC, *i.e.*, the system with lockstep-synchronous processors, at most one of which is failstop-faulty, and reliable but asynchronous communication.

Theorem 10: *(Theorem I0 in [DDS]) There is no consensus protocol for system S3(1).*

We now show that Theorem 10 follows from Theorem 9 using the results of this chapter.

Theorem 11: *If there is no consensus protocol for system A3(1), then there is no consensus protocol for system S3(1).*

Proof: Suppose in contradiction that there is a consensus protocol P for system S3(1). By Theorem 8, system A3(1) simulates system S3(1). Thus, there exists a simulation protocol P' such that P' in system A3(1) simulates P in system S3(1). The protocol P' can be used to construct a consensus protocol for system A3(1) simply by letting v-final states of P' be those states d such that $d.sim$ is a v-final state of P. Since P is a consensus protocol for system S3(1), there is a run R_0 of P in system S3(1) in which some processor enters a 0-final state and another run R_1 of P in system S3(1) in which some processor enters a 1-final state. Since P' in A3(1) simulates P in S3(1), there is a run R'_0 of P' in system A3(1) that simulates R_0, *i.e.,* in which some processor enters a 0-final state, and another run R'_1 of P' in system A3(1) that simulates R_1, *i.e.,* in which some processor enters a 1-final state. Since P is a consensus protocol for S3(1), and since P is simulated by P', there is no run of P' in system A3(1) with processors in conflicting final states, and some processor eventually enters a final state in every run in system A3(1). Thus there is a consensus protocol for system A3(1), contradicting the hypothesis. □

4

Synthesis of Efficient Drinking Philosophers Algorithms

A variant of the drinking philosophers algorithm of Chandy and Misra is described and proved correct in a modular way, using the I/O automaton model of Lynch and Tuttle. The algorithm of Chandy and Misra is based on a particular dining philosophers algorithm, and relies on certain properties of its implementation. The drinking philosophers algorithm presented in this chapter is able to use an arbitrary dining philosophers algorithm as a true subroutine; nothing about the implementation needs to be known, only that it solves the dining philosophers problem. An important advantage of this modularity is that by substituting a more time-efficient dining philosophers algorithm than the one used by Chandy and Misra, a drinking philosophers algorithm with $O(1)$ worst-case waiting time is obtained, whereas the drinking philosophers algorithm of Chandy and Misra has $O(n)$ worst-case waiting time (for n philosophers). Formal definitions are given to distinguish the drinking and dining philosophers problems and to specify precisely varying degrees of concurrency.

1. Introduction

In this chapter, we present a modular description and proof of correctness for an algorithm to solve the drinking philosophers problem in a message-passing distributed system. Our algorithm uses an arbitrary solution to the dining philosophers problem as a subroutine; by using a time-efficient subroutine, one can obtain

This chapter is joint work with Nancy Lynch.

a drinking philosophers algorithm with $O(1)$ worst-case waiting time. Another contribution of this chapter is a complete and rigorous problem statement and proof of correctness, which are often lacking in work in this area.

The drinking philosophers problem is a dynamic variant due to Chandy and Misra [CM] of the dining philosophers problem, a much-studied resource allocation problem [D2] [Ly] [RL]. In the original dining philosophers problem of Dijkstra [D2], five philosophers (processes) are arranged in a ring with one fork (resource) between them, and in order to eat (do work), a philosopher must have exclusive access to both of its adjacent forks. A more general version of the problem allows any number of processes and puts no restrictions on which processes share resources, as studied in [Ly] and [CM]. In the drinking philosophers problem, for each process there is a maximum set of resources that it can request, and each time a process wishes to do some work, it may request an arbitrary subset of its maximum set.

Our drinking philosophers algorithm is a variant of the one in [CM]. Their algorithm is based on a particular dining philosophers algorithm, and relies on certain properties of its implementation. Our drinking philosophers algorithm is able to use an arbitrary dining philosophers algorithm as a true subroutine; nothing about the implementation needs to be known, only that it solves the dining philosophers problem. We show that the maximum waiting time for a drinking philosopher to enter its critical region is roughly equal to the maximum waiting time for a dining philosopher to enter its critical region in the subroutine. Thus, by replacing the dining philosophers algorithm of [CM], which has waiting time $O(n)$, with a dining philosophers algorithm such as [Ly], which has waiting time $O(1)$, we obtain a more efficient drinking philosophers algorithm.

We provide definitions that distinguish the drinking and dining philosophers problem, and that specify precisely varying degrees of concurrency. We use the model in [LT], which is useful for stating properties that concern the infinite behavior of a system, such as no-deadlock and no-lockout, and encourages modular algorithm design and verification. This model, together with the particular definitions developed in this chapter for expressing the safety and liveness properties for resource allocation problems, make possible a clear and precise proof of correctness for our construction. In fact, the same combination has been used in work on modular decomposition of other resource allocation problems [W2].

All the program modules in this chapter are specified using implications, essentially of the form: "If the input is well-behaved, then the output is well-behaved." Such conditional specifications must have the property that the consequent is fully under the control of the module being specified, assuming the input satisfies the antecedent. Safety and liveness properties receive a uniform treatment in our conditional specifications; both kinds of conditions are described in English as predicates on the desired sequences of actions. The basis of our method for proving the correctness of the drinking philosophers algorithm is a general theorem that identifies a sufficient condition for when the composition of several modules, each satisfying a conditional specification, itself satisfies a conditional specification. Composable specifications for concurrent and/or distributed systems have been widely studied. In [HO], [La2] and [NDGO], specifications are described using temporal logic, but no method is presented that is tailored for combining conditional specifications. Theorems similar to ours do appear in [MC] and [J] (only for safety properties) and in [MCS] and [St] (for safety and liveness properties).

Section 2 consists of the general theorems used to do modular correctness proofs. In Section 3, the dining philosophers and drinking philosophers problems are defined. In Section 4, we describe our algorithm, as an automaton. Section 5 contains the proof of correctness of our algorithm, and Section 6 analyzes the performance of our algorithm with respect to various complexity measures.

2. Theorems for Modular Correctness Proofs

For the rest of this chapter, we assume the model of [LT]. (See the Appendix for a summary of its relevant features.) The following theorem provides the formal basis for the method we use to prove that an automaton that "calls a subroutine" solves a problem. Suppose automaton S solves a problem whose schedules are all sequences of actions that satisfy some set of m implications. We would like to show that the automaton A formed by composing S with another automaton solves the problem whose schedules are all sequences of actions that satisfy some other implication. A sufficient condition for showing this is that there is a subset of the m implications for S such that the antecedent of the desired implication for A implies the antecedents of the subset of implications for S, and the antecedent of the desired implication together with the consequents of the subset of implications for S imply the consequent of the desired implication.

Theorem 1: *Let automaton A be the composition of two automata E and S, where S solves the problem P' consisting of all sequences from $ext(S)$ satisfying the m implications $p'_i \supset q'_i$, $1 \leq i \leq m$, where p'_i and q'_i are predicates on strings from $ext(S)$. Let p and q be predicates on strings from $ext(A)$. Suppose that for any fair execution e of A, with schedule α, the following is true: There exists a subset I of $\{i : 1 \leq i \leq m\}$ such that*

(1) if p is true of $\alpha|ext(A)$, then $\bigwedge_{i\in I} p'_i$ is true of $\alpha|ext(S)$, and

(2) if p is true of $\alpha|ext(A)$ and $\bigwedge_{i\in I} q'_i$ is true of $\alpha|ext(S)$, then q is true of $\alpha|ext(A)$.

Then A solves the problem P consisting of all sequences from $ext(A)$ satisfying the implication $p \supset q$.

Proof: Let e be a fair execution of A with schedule α. Since e is fair, $e|S$ is fair, by a result in [LT]. Thus, $\alpha|ext(S)$ is in *Fbeh*(S). Since S solves P', every schedule in *Fbeh*(S) satisfies $p'_i \supset q'_i$, for $1 \leq i \leq m$. Assume p is true of $\alpha|ext(A)$. By assumption (1), there exists a subset I of $\{i : 1 \leq i \leq m\}$ such that $\bigwedge_{i\in I} p'_i$ is true of $\alpha|ext(S)$. Thus, $\bigwedge_{i\in I} q'_i$ is true of $\alpha|ext(S)$. By assumption (2), q is true of $\alpha|ext(A)$. □

The previous theorem can be easily generalized to show that A satisfies a set of implications.

Corollary 2: *Let automaton A be the composition of two automata E and S, where S solves the problem P' consisting of all sequences from $ext(S)$ satisfying the m implications $p'_i \supset q'_i$, $1 \leq i \leq m$, where p'_i and q'_i are predicates on strings from $ext(S)$. Let p_j and q_j, $1 \leq j \leq n$, be predicates on strings from $ext(A)$. Suppose that for any fair execution e of A, with schedule α, the following is true for all j, $1 \leq j \leq n$: There exists a subset I of $\{i : 1 \leq i \leq m\}$ such that*

(1) if p_j is true of $\alpha|ext(A)$, then $\bigwedge_{i\in I} p'_i$ is true of $\alpha|ext(S)$, and

(2) if p_j is true of $\alpha|ext(A)$ and $\bigwedge_{i\in I} q'_i$ is true of $\alpha|ext(S)$, then q_j is true of $\alpha|ext(A)$.

Then A solves the problem P consisting of all sequences from $ext(A)$ satisfying the n implications $p_j \supset q_j$, $1 \leq j \leq n$.

The previous corollary can be generalized to prove that an automaton that calls several subroutines solves a problem.

Corollary 3: *Let automaton A be the composition of $l+1$ automata $E, S_1, \ldots, S_l$, where for $1 \leq h \leq l$, S_h solves the problem P_h consisting of all sequences from $ext(S_h)$ satisfying the m_h implications $p^h_i \supset q^h_i$, $1 \leq i \leq m_h$, where p^h_i and q^h_i are predicates on strings from $ext(S_h)$. Let p_j and q_j, $1 \leq j \leq n$, be predicates on strings from $ext(A)$. Suppose that for any fair execution e of A, with schedule α, the following is true for all j, $1 \leq j \leq n$: For all h, $1 \leq h \leq l$, there exists a subset I of $\{i : 1 \leq i \leq m_h\}$ such that*

(1) if p_j is true of $\alpha|ext(A)$, then $\bigwedge_{i\in I} p_i^h$ is true of $\alpha|ext(S_h)$ for all h, $1 \leq h \leq l$, and

(2) if p_j is true of $\alpha|ext(A)$ and $\bigwedge_{i\in I} q_i^h$ is true of $\alpha|ext(S_h)$ for all h, $1 \leq h \leq l$, then q_j is true of $\alpha|ext(A)$.

Then A solves the problem P consisting of all sequences from $ext(A)$ satisfying the n implications $p_j \supset q_j$, $1 \leq j \leq n$.

3. Problem Statement

There are n user processes in the system being modeled, and at various times, each one needs some of the system resources. Only one process at a time may have access to a resource. Each user's code is divided into four parts. In its *trying region*, the user is vying for access to its required resources. Once the resources are obtained, the user may enter its *critical region*. When the user is through with the resources, it enters its *exit region*, which usually involves some "cleaning up" activities. Otherwise, the user is in its *remainder region*. A resource allocation algorithm specifies the code of the trying and exit regions. Our goal is to define two resource allocation problems, dining philosophers and drinking philosophers, as external schedule modules. We imagine an automaton that, given input from some number of users informing the automaton of their desire to gain or give up a set U of resources (with input actions $T_i(U)$ and $E_i(U)$ for each user i), decides which users are allowed to enter their critical and remainder regions at which times (with output actions $C_i(U)$ and $R_i(U)$). The automaton, then, represents the algorithm used to allocate the resources.

In the dining philosophers problem, each user (or philosopher) always requests the same set of resources. In the drinking philosophers problem, each user can request an arbitrary subset of the total set of resources that it ever will want.

We consider several versions of the dining and drinking philosophers problems, each satisfying successively stronger liveness properties. First we define the basic dining and drinking philosophers problems, which only satisfy safety conditions. Then the no-deadlock versions are defined, in which as long as a philosopher wants to eat (or drink), eventually some philosopher succeeds, but not necessarily the original one. In the no-lockout versions, any philosopher that wants to eat (or drink) eventually does so. The no-deadlock and no-lockout conditions assume that no user keeps resources forever.

Up to this point, no real distinction has been made between the dining and drinking philosophers problems; in fact, a dining philosophers algorithm can be used to solve the drinking philosophers problem. However, drinkers may be blocked unnecessarily. A preferable solution would not rule out two drinkers that share a resource from entering their critical regions simultaneously, if their current resource requirements are disjoint. We capture part of this intuition by defining the "more-concurrent" condition — if a drinker requests a set of resources, none of which is currently being sought or used by another drinker, then the drinker enters its critical region, even if some other resources are never relinquished. (One might imagine being interested in defining even stronger forms of the drinking philosophers problem. In Section 5.4, we explore a few such definitions and discuss why or why not our algorithm and that in [CM] satisfy them.)

Let *Utensils* be a finite set of resources, or *utensils*. Define an *n-process utensil requirement* to a collection of n sets U_i, for $1 \leq i \leq n$, where each U_i is a subset of *Utensils*, such that no utensil is in more than two U_i's. This restriction makes the algorithm much simpler to describe and reason about, but is not substantial. If a resource is shared by k users, then it can be represented by k choose 2 virtual resources, one shared between each pair of the original k users; to gain the "real" resource, a user must gain the $k-1$ virtual resources shared with it.

In the context of the dining philosophers problem, utensils will be referred to as *forks*; in the context of the drinking philosophers problem, utensils will be referred to as *bottles*.

3.1 Dining Philosophers

In order to specify the external schedule module for the dining philosophers problem, we define several properties on strings. Conditions (I1-F), (I2-F), (O1-F) and (O2-F) state that input and output actions alternate correctly, in the order T_i, C_i, E_i, R_i. Condition (I3-F) models that fact that no user process keeps the resources forever. (EX-F), (ND-F), and (NL-F) are the exclusion, no-deadlock and no-lockout conditions. Conditions (I1-F), (I2-F), (O1-F), (O2-F) and (EX-F) are safety properties; (I3-F), (ND-F) and (NL-F) are liveness properties.

Fix an n-process fork requirement $\mathcal{F} = \{F_i : 1 \leq i \leq n\}$. The following definitions are all made relative to this fork requirement.

For a positive integer i, let the set $\{T_i, C_i, E_i, R_i\}$ be denoted $F\text{-}TCER_i$. Since each user i must request the same set F_i of forks each time, we don't explicitly include the set of utensils in the action names. (The letter F stands for "fork.")

Given a sequence α from $F\text{-}TCER_i$, we define the following conditions, which α may or may not satisfy.

- (I1-F) For any prefix βa of α, if $a = T_i$, then β is empty or β ends in R_i.
- (I2-F) For any prefix βa of α, if $a = E_i$, then β ends in C_i.
- (I3-F) If α is finite, then α does not end in C_i.
- (O1-F) For any prefix βa of α, if $a = C_i$, then β ends in T_i.
- (O2-F) For any prefix βa of α, if $a = R_i$, then β ends in E_i.

Given a sequence α from an alphabet $\{T_i, C_i, E_i, R_i : 1 \leq i \leq n\}$, where n is some positive integer, we define the following conditions, which α may or may not satisfy.

- (EX-F) For any i and j, if $\alpha = \beta_1 C_i \beta_2 C_j \beta_3$ and if $F_i \cap F_j \neq \emptyset$, then β_2 contains E_i.
- (ND-F) For any i, if α is finite, then $\alpha | F\text{-}TCER_i$ ends in R_i.
- (NL-F) For any i, if $\alpha | F\text{-}TCER_i$ is finite, then $\alpha | F\text{-}TCER_i$ ends in R_i.

The *dining philosophers* problem for $\mathcal{F}$ is the external schedule module M_{Dine} with input actions $\{T_i, E_i : 1 \leq i \leq n\}$ and output actions $\{C_i, R_i : 1 \leq i \leq n\}$; the schedules of M_{Dine} are all sequences α of actions of M_{Dine} satisfying the (EX-F) implication: namely, if (I1-F) and (I2-F) are true of $\alpha | F\text{-}TCER_i$ for all i, then (O1-F) and (O2-F) are true of $\alpha | F\text{-}TCER_i$ for all i, and (EX-F) is true of α.

The *no-deadlock* dining philosophers problem for $\mathcal{F}$ is the external schedule module M_{NDDine} that is the sub-schedule module of M_{Dine} consisting of all sequences α of actions that, in addition to satisfying the (EX-F) implication, also satisfy the (ND-F) implication: namely, if (I1-F), (I2-F) and (I3-F) are true of $\alpha | F\text{-}TCER_i$ for all i, then (O1-F) and (O2-F) are true of $\alpha | F\text{-}TCER_i$ for all i, and (EX-F) and (ND-F) are true of α.

The *no-lockout* dining philosophers problem for $\mathcal{F}$ is the external schedule module M_{NLDine} that is the sub-schedule module of M_{Dine} consisting of all sequences α of actions that, in addition to satisfying the (EX-F) implication, also satisfy the (NL-F) implication: namely, if (I1-F), (I2-F) and (I3-F) are true of $\alpha | F\text{-}TCER_i$ for all i, then (O1-F) and (O2-F) are true of $\alpha|F\text{-}TCER_i$ for all i, and (EX-F) and (NL-F) are true of α.

3.2 Drinking Philosophers

Fix an n-process bottle requirement $\mathcal{B} = \{B_i : 1 \leq i \leq n\}$. The following definitions are made relative to this bottle requirement, and most are similar to those in Section 3.1. Two new conditions, (MC-B) and (I4-B)$_i$ are used to create implications to distinguish the drinking philosophers problem from the dining philosophers problem. (I4-B)$_i$ is indexed by process id i, and is a safety condition imposed on the input, stating that whenever user i requests a set of resources, no other user is currently requesting or using any of those resources. (MC-B) is a liveness condition imposed on the output, that the process is never stuck in its trying or exit regions.

For a positive integer i, let the set $\{T_i(B), C_i(B), E_i(B), R_i(B) : B \subseteq B_i\}$ be denoted $B\text{-}TCER_i$. (The letter B stands for "bottles.") Given a sequence α from $B\text{-}TCER_i$, we define the following conditions, which α may or may not satisfy.

- (I1-B) For any prefix βa of α, if $a = T_i(B)$ for some $B \subseteq B_i$, then β is empty or β ends in $R_i(B')$ for some $B' \subseteq B_i$.
- (I2-B) For any prefix βa of α, if $a = E_i(B)$ for some $B \subseteq B_i$, then β ends in $C_i(B)$.
- (I3-B) If α is finite, then α does not end in $C_i(B)$ for any $B \subseteq B_i$.
- (O1-B) For any prefix βa of α, if $a = C_i(B)$ for some $B \subseteq B_i$, then β ends in $T_i(B)$.
- (O2-B) For any prefix βa of α, if $a = R_i(B)$ for some $B \subseteq B_i$, then β ends in $E_i(B)$.
- (MC-B) If α is finite, then α does not end in $T_i(B)$ or $E_i(B)$ for any $B \subseteq B_i$.

Given a sequence α from an alphabet $\bigcup_{i=1}^{n} B\text{-}TCER_i$, we define the following conditions, which α may or may not satisfy.

- (EX-B) For any i and j, if $\alpha = \beta_1 C_i(B)\beta_2 C_j(B')\beta_3$ for some $B \subseteq B_i$ and $B' \subseteq B_j$, and if $B \cap B' \neq \emptyset$, then β_2 contains $E_i(B)$.
- (ND-B) For any i, if α is finite, then $\alpha|B\text{-}TCER_i$ ends in $R_i(B)$ for some $B \subseteq B_i$.

- (NL-B) For any i, if $\alpha|B\text{-}TCER_i$ is finite, then $\alpha|B\text{-}TCER_i$ ends in $R_i(B)$ for some $B \subseteq B_i$.

- (I4-B)$_i$ If there is an action $T_i(B)$ in α and there is an action $T_j(B')$ in α for any j with $B \cap B' \neq \emptyset$, then either there is an action $E_j(B')$ between the $T_j(B')$ and $T_i(B)$ actions, or there is an action $C_i(B)$ between the $T_i(B)$ and $T_j(B')$ actions.

The *drinking philosophers* problem for $\mathcal{B}$ is the external schedule module M_{Drink} with input actions $\bigcup_{i=1}^{n}\{T_i(B), E_i(B) : B \subseteq B_i\}$ and output actions $\bigcup_{i=1}^{n}\{C_i(B), R_i(B) : B \subseteq B_i\}$; the schedules of M_{Drink} are all sequences α of actions of M_{Drink} satisfying the (EX-B) implication: namely, if (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_i$ for all i, then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_i$ for all i, and (EX-B) is true of α.

The *no-deadlock* drinking philosophers problem for $\mathcal{B}$ is the external schedule module $M_{NDDrink}$ that is the sub-schedule module of M_{Drink} consisting of all sequences α of actions that, in addition to satisfying the (EX-B) implication, also satisfy the (ND-B) implication: namely, if (I1-B), (I2-B) and (I3-B) are true of $\alpha|B\text{-}TCER_i$ for all i, then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_i$ for all i, and (EX-B) and (ND-B) are true of α.

The *no-lockout* drinking philosophers problem for $\mathcal{B}$ is the external schedule module $M_{NLDrink}$ that is the sub-schedule module of M_{Drink} consisting of all sequences α of actions that, in addition to satisfying the (EX-B) implication, also satisfy the (NL-B) implication: namely, if (I1-B), (I2-B) and (I3-B) are true of $\alpha|B\text{-}TCER_i$ for all i, then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_i$ for all i, and (EX-B) and (NL-B) are true of α.

The *more-concurrent* drinking philosophers problem for $\mathcal{B}$ is the external schedule module $M_{MCDrink}$ that is the sub-schedule module of $M_{NLDrink}$ consisting of all sequences α of actions that, in addition to satisfying the (EX-B) and (NL-B) implications, also satisfy the following n implications, (MC-B) for i, $1 \leq i \leq n$: namely, if (I1-B) and(I2-B) are true of $\alpha|B\text{-}TCER_k$ for all k, and if (I4-B)$_i$ is true of α, then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_k$ for all k, (EX-B) and (NL-B) are true of α, and (MC-B) is true of $\alpha|B\text{-}TCER_i$.

4. Drinking Philosophers Automaton

In this section we describe an automaton $Drink(\mathcal{B})$ to solve the drinking philosophers problem for the n-process bottle requirement $\mathcal{B}$, in a message-passing distributed system. It is created by composing several automata, to be described, and then hiding most of the actions, in order for the external actions to be consistent with the definition of the problem. The component automata are *Drinker*(i), for $1 \leq i \leq n$, and any automaton *Dine*($\mathcal{F}$) that solves the dining philosophers problem for $\mathcal{F} = \mathcal{B}$. After describing the *Drinker* automata, we identify the actions that are hidden in the creation of *Drink*($\mathcal{B}$). But first, we describe the algorithm informally.

As soon as a drinker enters its trying region, it does the following two tasks: (1) It sends request messages for all bottles that it needs but lacks. (2) If the dining subroutine is in its remainder region, the drinker sends it to its trying region. Any requests received for needed bottles while the drinker is in its trying region are deferred; other requests are satisfied. If a demand for a bottle is received, it is satisfied, unless the dining subroutine is in its critical region, in which case the demand is deferred. (If the demand is for an unneeded bottle, it is satisfied.) If the dining subroutine enters its critical region, the drinker sends demand messages for all the bottles it still lacks.

Eventually, the drinker obtains all needed bottles and can enter its critical region. Any requests or demands for bottles in use are deferred; others are satisfied. If the dining subroutine ever is in its critical region, it is sent to its exit region.

As soon as the drinker enters its exit region, all deferred requests and demands are satisfied. If the dining subroutine enters its critical region, it is immediately sent to its exit region.

In its remainder region, the drinker satisfies all requests and demands for bottles. If the dining subroutine enters its critical region, it is immediately sent to its exit region.

Our algorithm manipulates the dining philosophers subroutine in the same way as the algorithm in [CM], except that the dining critical region is left as soon as the drinking critical region is entered, instead of when the drinking critical region is left. Also, the rules for relinquishing and keeping bottles, which in [CM] require knowing the current state of the particular dining philosophers algorithm, are different in our solution.

The set of possible *messages* is $\{req(b), sat(b), dem(b) : b \in \mathit{Utensils}\}$.

The state of *Drinker*(i), $1 \leq i \leq n$, consists of values for the following variables: *drink-region*, *dine-region*, *deferred*, *bottles*, *req-bottles*, *buff*[j] for all $j \neq i$, *send-*T_i, and *send-*E_i. The *region* variables take on the values T, C, E and R, and indicate which region the i^{th} dining and i^{th} drinking philosophers are in. The *deferred* variable is a set of (bottle,id) pairs, indicating whose requests for which bottles have been deferred. The *bottles* and *req-bottles* variables are sets of bottles, and indicate which bottles the i^{th} drinking philosopher has and which it requires, respectively. For each $j \neq i$, the variable *buff*[j] is a FIFO queue of messages to send to *Drinker*(j), and is manipulated with operations enqueue and dequeue. The *send* variables are Booleans and control when the named output actions are enabled. Initially, the *regions* are R; *deferred*, *req-bottles*, and all the *buff*[j] are empty; and the *sends* are false. Each bottle shared by *Drinker*(i) and *Drinker*(j) is initially in the *bottles* variable of either *Drinker*(i) or *Drinker*(j) but not both.

The actions of *Drinker*(i) are listed below, together with their preconditions and effects. (There are no internal actions.) First we define two macros, *SAT* and *DEFER*.

$SAT(b,j)$ == enqueue(*buff*[j],*sat*(b)); *bottles* $\leftarrow$ *bottles* $-\{b\}$

$DEFER(b,j)$ == if $b \in$ *req-bottles* then *deferred* $\leftarrow$ *deferred* $\cup\{(b,j)\}$ else $SAT(b,j)$

Input actions:

```
• $T_i(B)$, $B \subseteq B_i$
      Effects:
          drink-region $\leftarrow T$
          req-bottles $\leftarrow B$
          for all $j \neq i$ and $b \in B \cap B_j$: if $b \notin$ bottles then enqueue(buff[$j$],req($b$))
          if dine-region $= R$ then send-$T_i \leftarrow$ true

• $E_i(B)$, $B \subseteq B_i$
      Effects:
          drink-region $\leftarrow E$
          for all $(b,j) \in$ deferred: $SAT(b,j)$
          deferred $\leftarrow \emptyset$

• deliver(sat($b$),$j$,$i$) for all $j \neq i$, $b \in B_i \cap B_j$
      Effects:
          bottles $\leftarrow$ bottles $\cup\{b\}$

• deliver(req($b$),$j$,$i$) for all $j \neq i$, $b \in B_i \cap B_j$
      Effects:
          case drink-region of
```

```
        $T$ or $C$: $DEFER(b,j)$
        $E$ or $R$: $SAT(b,j)$
```

- $deliver(dem(b), j, i)$ for all $j \neq i$, $b \in B_i \cap B_j$

```
    Effects:
      if $b \in bottles$ then
        if $drink\text{-}region = C$ or $(drink\text{-}region = T$ & $dine\text{-}region = C)$ then
          $DEFER(b,j)$
          else [ $SAT(b,j)$; $deferred \leftarrow deferred - \{(b,j)\}$ ]
```

- C_i

```
    Effects:
      $dine\text{-}region \leftarrow C$
      if $drink\text{-}region = T$ then
        for all $j \neq i$ and $b \in req\text{-}bottles \cap B_j$ : if $b \notin bottles$ then
          enqueue($buff[j], dem(b)$)
        else $send\text{-}E_i \leftarrow$ true
```

- R_i

```
    Effects:
      $dine\text{-}region \leftarrow R$
      if $drink\text{-}region = T$ then $send\text{-}T_i \leftarrow$ true
```

Output actions:

- $C_i(B)$, $B \subseteq B_i$

```
    Preconditions:
      $drink\text{-}region = T$ and $req\text{-}bottles \subseteq bottles$ and $send\text{-}T_i = send\text{-}E_i =$ false
    Effects:
      $drink\text{-}region \leftarrow C$
      if $dine\text{-}region = C$ then $send\text{-}E_i \leftarrow$ true
```

- $R_i(B)$, $B \subseteq B_i$

```
    Preconditions:
      $drink\text{-}region = E$ and $send\text{-}T_i = send\text{-}E_i =$ false
    Effects:
      $drink\text{-}region \leftarrow R$
```

- $deliver(m, i, j)$ for all $j \neq i$, $m \in \{req(b), sat(b), dem(b) : b \in B_i \cap B_j\}$

```
    Preconditions:
```

```
    m is at head of buff[j]
  Effects:
    dequeue(buff[j])
```

- T_i

```
  Preconditions:
    send-$T_i$ = true
  Effects:
    dine-region ← T
    send-$T_i$ ← false
```

- E_i

```
  Preconditions:
    send-$E_i$ = true
  Effects:
    dine-region ← E
    send-$E_i$ ← false
```

The output actions are partitioned into n classes, one for the delivery of messages in each *buff*[j], and one for all the other actions. Formally, the subsets of the output actions are $\{C_i(B), R_i(B), T_i, E_i : B \subseteq B_i\}$, and for each $j \neq i$, $\{deliver(m, i, j) : m = req(b), dem(b)$, or $sat(b), b \in B_i \cap B_j\}$. This partition guarantees that messages are eventually delivered in fair executions, since the message queues are FIFO. In essence, the *buff* variables are modeling separate pieces of hardware, the communication links.

The automaton *Drink($\mathcal{B}$)* is formed by composing *Drinker*(i), $1 \leq i \leq n$, and *Dine($\mathcal{F}$)*, where $\mathcal{F} = \mathcal{B}$, and then hiding all actions except $\bigcup_{i=1}^{n} B\text{-}TCER_i$.

5. Proof of Correctness

5.1 Drinking Philosophers

Our goal is to use Corollary 2 to prove that *Drink($\mathcal{B}$)* solves the drinking philosophers problem. The drinking philosophers problem is defined by the (EX-B) implication. Interestingly enough, showing that *Drink($\mathcal{B}$)* solves the drinking philosophers problem requires nothing about the behavior of *Dine($\mathcal{F}$)*. Thus, in using Corollary 2, the set of implications of the subroutine used to verify the (unique) implication of the composition is empty. Consequently, condition (1) of the theorem is trivially true. Lemmas 4 and 5 are used to verify condition (2). Theorem 6 puts the pieces of the argument together.

(Throughout this paper, Greek letters stand for sequences of actions, and Roman letters for single actions.)

Lemma 4: *Let e be an execution of Drink($\mathcal{B}$) with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_i$ for all i. Then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_i$ for all i.*

Proof: By induction on the length of the prefixes π of α.

Basis: $|\pi| = 0$. The lemma is vacuously true.

Induction: $|\pi| > 0$. Assume the lemma is true for all prefixes of α shorter than π. Let $\pi = \pi' a$. Then $\pi'|B\text{-}TCER_i$ satisfies (O1-B) and (O2-B). If a is not in $B\text{-}TCER_i$ for any i, then the inductive hypothesis gives the result for π. Suppose a is in $B\text{-}TCER_i$. Then the inductive hypothesis implies that $\pi|B\text{-}TCER_j$ satisfies (O1-B) and (O2-B) for all $j \neq i$.

For now, assume that $\pi'|B\text{-}TCER_i = \delta b$. The following four cases show that $\pi|B\text{-}TCER_i = \delta b a$ satisfies (O1-B) and (O2-B).

Case 1: $b = T_i(B)$ for some $B \subseteq B_i$.

By assumption that $\alpha|B\text{-}TCER_i$ satisfies (I1-B) and (I2-B), a cannot be $T_i(B')$ or $E_i(B')$ for any $B' \subseteq B_i$.

After b, *drink-region* is T and does not change until further actions in $B\text{-}TCER_i$ occur. Since $R_i(B')$ is not enabled unless *drink-region* is E, a cannot be $R_i(B')$ for any $B' \subseteq B_i$.

$C_i(B')$ is only enabled if *req-bottles* $= B'$, which is only true if the most recent $B\text{-}TCER_i$ action is $T_i(B')$. Thus $C_i(B)$ is the only possibility.

Case 2: $b = C_i(B)$ for some $B \subseteq B_i$.

By assumption that $\alpha|B\text{-}TCER_i$ obeys (I1-B), a cannot be $T_i(B')$ for any $B' \subseteq B_i$.

After b, *drink-region* is C and does not change until further actions in $B\text{-}TCER_i$ occur. Since $R_i(B')$ is not enabled unless *drink-region* is E, and $C_i(B')$

is not enabled unless *drink-region* is T, a can be neither of these actions for any $B' \subseteq B_i$.

Thus $a = E_i(B)$ is the only possibility, since $E_i(B')$ for any $B' \neq B$ is ruled out by (I2-B).

Case 3: $b = E_i(B)$ for some $B \subseteq B_i$.

By assumption that $\alpha|B\text{-}TCER_i$ obeys (I1-B) and (I2-B), a cannot be $T_i(B')$ or $E_i(B')$ for any $B' \subseteq B_i$.

After b, *drink-region* is E and does not change until further actions in $B\text{-}TCER_i$ occur. Since $C_i(B')$ is not enabled unless *drink-region* is T, a cannot be $C_i(B')$ for any $B' \subseteq B_i$.

$R_i(B')$, for any $B' \subseteq B_i$, is only enabled if the most recent $B\text{-}TCER_i$ action is $E_i(B')$. Thus $a = R_i(B)$ is the only possibility.

Case 4: $b = R_i(B)$ for some $B \subseteq B_i$.

By assumption that $\alpha|B\text{-}TCER_i$ obeys (I2-B), a cannot be $E_i(B')$ for any $B' \subseteq B_i$.

After b, *drink-region* is R and does not change until further actions in $B\text{-}TCER_i$ occur. Since $C_i(B')$ is not enabled unless *drink-region* is T and $R_i(B')$ is not enabled unless *drink-region* is E, a cannot be either of these actions for any $B' \subseteq B_i$.

Thus $a = T_i(B')$, for some $B' \subseteq B_i$, is the only possibility.

End of Cases.

If there is no such b, i.e., a is the first action from $B\text{-}TCER_i$ in π, then essentially the same argument as Case 4 shows that a must be $T_i(B')$ for some $B' \subseteq B_i$. □

We say that bottle b *resides* at *Drinker*(i) in state s if b is in *Drinker*(i)'s *bottles* variable in s. The following lemma implies that b resides at no more than one *Drinker* in any state.

Lemma 5: *Let e be an execution of Drink($\mathcal{B}$) with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_i$ for all i. Let b be a bottle shared between Drinker(i) and Drinker(j). Then in every state of e, the following are true:*

(a) Exactly one of the following is true: b resides at Drinker(i), b resides at Drinker(j), sat(b) is in buff[i] at Drinker(j), or sat(b) is in buff[j] at Drinker(i).

(b) At most one sat(b) message is in buff[i] at Drinker(j) and buff[j] at Drinker(i).

(c) If (b, j) is in deferred at Drinker(i), then b resides at Drinker(i).

(d) If req(b) is at the head of buff[i] at Drinker(j), then b resides at Drinker(i) and (b, j) is not in deferred at Drinker(i).

Proof: Let $e = s_0 a_1 s_1 \ldots a_m s_m \ldots$. We proceed by induction on m, which indexes the states of e.

(a), (b), (c) and (d) are obviously true of s_0, since it is a start state. Assuming (a), (b), (c) and (d) are true of all states of e up to and including s_{m-1}, we show they are true of s_m.

We show that (a), (b) and (c) are true in s_m by considering all possible values for the action a_m.

Case 1: $a_m = T_i(B)$ for some $B \subseteq B_i$. Inspecting the code reveals that this action causes no change to either *deferred* or *bottles* at *Drinker*(i), and causes no *sat*(b) to be placed in any *buff*. Thus, the inductive hypothesis that (a),(b) and (c) are true in s_{m-1} implies that they are true in s_m.

Case 2: $a_m = E_i(B)$ for some $B \subseteq B_i$.

(a) and (b) If (b, j) is in *deferred* at *Drinker*(i) in s_{m-1}, then by the inductive hypothesis for (c), b resides at *Drinker*(i) in s_{m-1}, and by the inductive hypothesis for (a), b does not reside at *Drinker*(j) and there is no *sat*(b) in either *buff* in s_{m-1}. Thus, in s_m, b does not reside at *Drinker*(i) or *Drinker*(j), exactly one *sat*(b) is in *buff*[j] at *Drinker*(i), and no *sat*(b) is in *buff*[i] at *Drinker*(j).

If (b, j) is not in *deferred* at *Drinker*(i) in s_{m-1}, then there are no changes in going from s_{m-1} to s_m that affect the truth of (a) and (b).

(c) Since *deferred* at *Drinker*(i) is empty in s_m, (c) is vacuously true.

Case 3: a_m = *deliver(sat(b),j,i)*.

(a) and (b) In s_{m-1}, b does not reside at *Drinker*(i) or *Drinker*(j), and there is no *sat*(b) in *buff*[j] at *Drinker*(i), since there is exactly one *sat*(b) in *buff*[i] at

Drinker(j). Thus, in s_m, b resides at *Drinker*(i) and not at *Drinker*(j), and there is no $sat(b)$ in either *buff* variable.

(c) Since b resides at *Drinker*(i) in s_m by the argument for (a) and (b), (c) is true whether or not (b, j) is in *deferred*.

Case 4: a_m = *deliver(req(b),j,i).*

Thus, $req(b)$ is at the head of *buff*[i] at *Drinker*(j) in s_{m-1}. By the inductive hypothesis for (d), b resides at *Drinker*(i) and (b, j) is not in *deferred* in s_{m-1}.

(a) and (b) Since b resides at *Drinker*(i) in s_{m-1}, b does not reside at *Drinker*(j) and there is no $sat(b)$ in either *buff* in s_{m-1}. Thus, in s_m, either the same is true, or else b does not reside at either *Drinker* and there is exactly one $sat(b)$ in one *buff*.

(c) If the request is deferred as a result of a_m, then b still resides at *Drinker*(i) in s_m. If the request is not deferred, then b is removed from *bottles* and (b, j) is not in *deferred* in s_m, because it is not in s_{m-1}.

Case 5: a_m = *deliver(dem(b),j,i).*

(a), (b) and (c) are true in s_m because they are true in s_{m-1} and the code checks if b resides at *Drinker*(i) before making any changes.

Case 6: a_m is T_i, C_i, E_i, R_i, $C_i(B)$ or $R_i(B)$.

(a), (b) and (c) are true in s_m because they are true in s_{m-1} and no relevant changes are made as a result of any of these actions.

End of (a), (b) and (c).

We prove that (d) is true in s_m by contradiction. Assume that in s_m, $req(b)$ is at the head of *buff*[i] at *Drinker*(j), but that either b does not reside at *Drinker*(i) or that (b, j) is in *deferred* at *Drinker*(i).

Suppose b does not reside at *Drinker*(i). The $req(b)$ at the head of the *buff* is triggered by a most recent $T_j(B)$ action in e, for some $B \subseteq B_j$. If, prior to this $T_j(B)$, b never resides at *Drinker*(j), then it is easy to see that this is the first $req(b)$ sent by *Drinker*(j) and that no $dem(b)$ is sent by *Drinker*(j) up to this point. Thus, b initially resides at *Drinker*(i). There are two ways that b can leave *Drinker*(i), either by the delivery of $req(b)$ or $dem(b)$, or by the occurrence of some $E_i(B')$ action when (b, j) is in *deferred*. But (b, j) can only be in *deferred* if previously a *req*(b) or *dem*(b) is delivered. Thus, b still resides at *Drinker*(i) as long as this first *req*(b) has not been delivered.

Assume that at some prior point, b does reside at *Drinker*(j). (See Figure 1.)

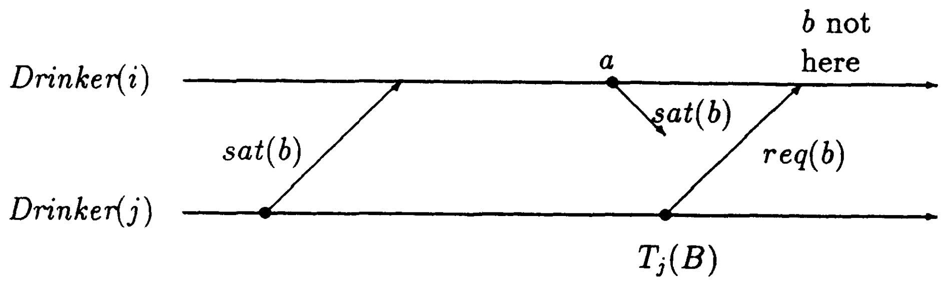

Figure 1

Some action occurs before the $T_j(B)$ (triggering the *req*(b) at the head of *buff*[i]) that causes *sat*(b) to be put in *buff*[i] at *Drinker*(j). Since *req*(b) is at the head of the *buff*, this *sat*(b) is delivered before s_m, at which point b resides at *Drinker*(i). Yet, b does not reside at *Drinker*(i) in s_m. How can this be? Some action a occurring between the delivery of the *sat*(b) to *Drinker*(i) and s_m causes a *sat*(b) to be placed in *buff*[j] at *Drinker*(i).

Case 1: a is the delivery of a *req*(b) or *dem*(b) message. (See Figure 2.) When this message is sent, *Drinker*(j) is in ***drink-region*** T and remains there until b resides at *Drinker*(j). Yet b does not reside at *Drinker*(j) until after the $T_j(B)$. This contradicts (I1-B).

Case 2: a is $E_i(B')$ for some $B' \subseteq B_i$, and (b, j) is in ***deferred*** at *Drinker*(i). Now (b, j) is in ***deferred*** because of the delivery of some *req*(b) or *dem*(b) message from *Drinker*(j). Call this message m.

Case 2a: The message m follows *sat*(b) in *buff*[i] (at *Drinker*(j)). The same argument as in Case 1 gives a contradiction.

Case 2b: The message m precedes *sat*(b) in *buff*[i]. (See Figure 3.) Then (b, j) is in ***deferred*** at the same time as *sat*(b) is in *buff*[i] at *Drinker*(j). By the inductive hypothesis for (c), as long as (b, j) is in ***deferred***, b resides at *Drinker*(i). This contradicts the inductive hypothesis for (a).

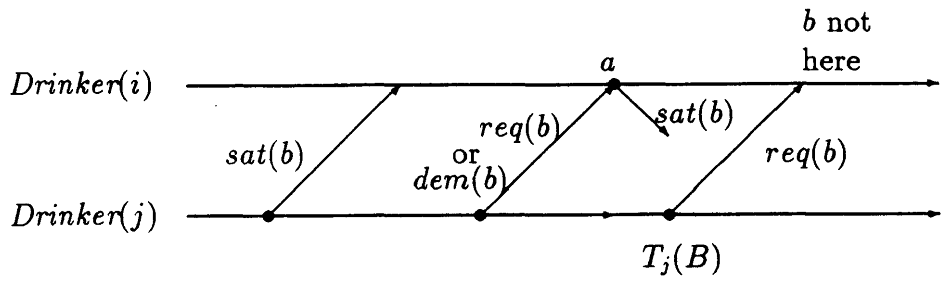

Figure 2

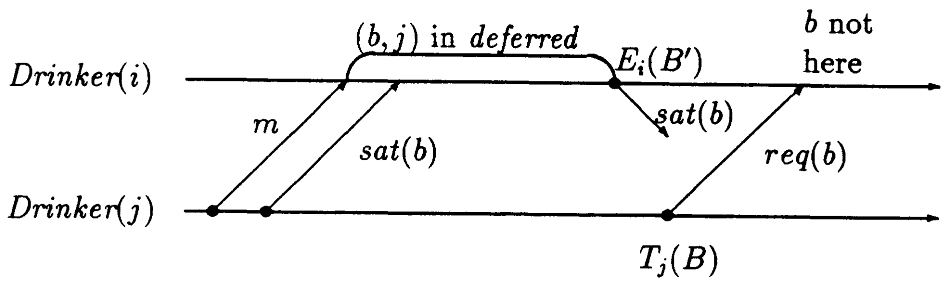

Figure 3

We have now shown that b resides at *Drinker*(i) in s_m.

It remains to show that (b, j) is not in ***deferred*** at ***Drinker***(i) in s_m. Suppose it is. (See Figure 4.)

Then (b, j) is put in *deferred* at *Drinker*(i) by the delivery of either a $req(b)$ or a $dem(b)$ message from *Drinker*(j). Call this message m. When the message m is placed in ***buff***$[i]$, due to action a, ***Drinker***(j) is in ***drink-region*** T and b does not reside at ***Drinker***(j). By Lemma 4, there is an action $C_j(B')$ (with b in B') between a and the $T_j(B)$ that triggers the $req(b)$ message at the head of ***buff***$[i]$ in s_m. When the $C_j(B')$ occurs, b resides at ***Drinker***(j). Since b does not reside at ***Drinker***(j) when the $T_j(B)$ occurs, a $sat(b)$ message is put in ***buff***$[i]$ at ***Drinker***(j) after the message m. But since ***buff***$[i]$ is FIFO, $sat(b)$ is delivered after m is and before the $req(b)$ is. Thus, there is a state in which (b, j) is in ***deferred*** at ***Drinker***(i), and by the inductive hypothesis for (c), b resides at ***Drinker***(i), yet simultaneously, $sat(b)$

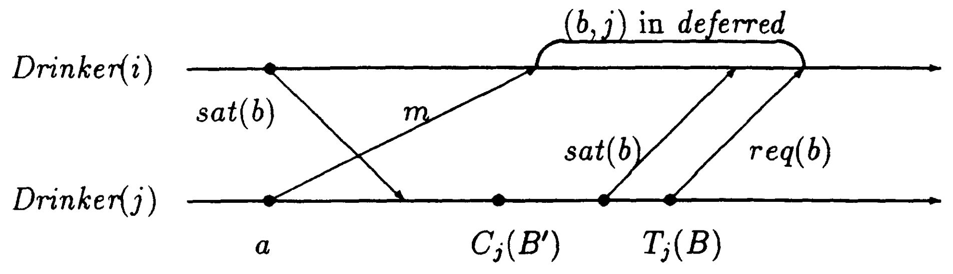

Figure 4

is in *buff*[*i*]. This situation contradicts the inductive hypothesis for (a). □

Here is the main theorem.

Theorem 6: *Drink($\mathcal{B}$) solves the drinking philosophers problem for $\mathcal{B}$.*

Proof: For the (EX-B) implication defining the drinking philosophers problem, we verify that conditions (1) and (2) of Corollary 2 are true, where the corresponding set of implications of the subroutine is empty. Condition (1) is vacuously true.

Condition (2) is reduced to showing the (EX-B) implication. Let e be a fair execution of *Drink($\mathcal{B}$)* with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_i$ for all i. By Lemma 4, (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_i$ for all i.

We show (EX-B) is true of $\alpha|$*ext(Drink($\mathcal{B}$))*. Let $\alpha = \alpha_1 C_i(B)\alpha_2 C_j(B')\alpha_3$ for some i and j, with $B \subseteq B_i$, $B' \subseteq B_j$, and $B \cap B' \neq \emptyset$. Choose b in $B \cap B'$. By Lemma 5, in every state of e, b resides at no more than one drinker. The $C_j(B')$ is not enabled until all bottles in B' reside at *Drinker*(j), yet b resides at *Drinker*(i) from before the $C_i(B)$ until an $E_i(B)$ action occurs. Thus α_2 contains $E_i(B)$, and (EX-B) is true of $\alpha|$*ext(Drink($\mathcal{B}$))* since $C_i(B)$, $E_i(B)$ and $C_j(B')$ are in *ext(Drink($\mathcal{B}$))*.

By Corollary 2, *Drink($\mathcal{B}$)* solves the drinking philosophers problem. □

5.2 No Deadlock and No Lockout

In this subsection we show that *Drink($\mathcal{B}$)* solves the no-deadlock (resp., no-lockout) drinking philosophers problem if *Dine($\mathcal{F}$)* solves the no-deadlock (resp., no-lockout) dining philosophers problem. Again, we use Corollary 2. The no-deadlock and no-lockout versions of the drinking philosophers problem are both defined by two implications, the (EX-B) and (ND-B) implications for no-deadlock, and the (EX-B) and (NL-B) implications for no-lockout. The (EX-B) implication has been taken care of in the previous section. First we prove in Lemma 7 that the subroutine behaves properly in the composition. Lemmas 8 and 9 verify condition (1) of Corollary 2 for the (ND-B) and (NL-B) implications, and Lemma 10 verifies condition (2). Theorems 11 and 12 finish the argument.

Lemma 7 shows that *Dine($\mathcal{F}$)* behaves properly in the composition. (This lemma and the next one do not rely on any liveness properties such as fairness or (I3-B), and thus could have been in the previous section, but the results were not needed there.)

Lemma 7: *Let e be a fair execution of Drink($\mathcal{B}$) with schedule α such that (I1-F) and (I2-F) are true of $\alpha|F\text{-}TCER_i$ for all i. If Dine($\mathcal{F}$) solves the dining philosophers problem for $\mathcal{F}$, then*

(a) $\alpha|F\text{-}TCER_i$ satisfies (O1-F) and (O2-F) for all i.

(b) $\alpha|ext(Dine(\mathcal{F}))$ satisfies (EX-F).

Proof: Since e is fair, $e|Dine(\mathcal{F})$ is fair, by a result in [LT]. Thus $\alpha|ext(Dine(\mathcal{F})) = \beta$ is in *Fbeh(Dine($\mathcal{F}$))*. Since (I1-F) and (I2-F) are true of $\beta|F\text{-}TCER_i$ for all i, (O1-F) and (O2-F) are true of $\beta|F\text{-}TCER_i = \alpha|F\text{-}TCER_i$ for all i, and (EX-F) is true of β. □

The next lemma shows that (I1-F) and (I2-F) are true for *Dine($\mathcal{F}$)* in the composition. This fact is true simply because of local code and requires nothing about the input actions to the composition.

Lemma 8: *Let e be an execution of Drink($\mathcal{B}$) with schedule α. Then (I1-F) and (I2-F) are true of $\alpha|F\text{-}TCER_i$ for all i.*

Proof: By induction on the length of the prefixes π of α.

Basis: $|\pi| = 0$. The lemma is vacuously true.

Induction: $|\pi| > 0$. Assume the lemma is true for all prefixes of α shorter than π. Let $\pi = \pi' a$. Then $\pi'|F\text{-}TCER_i$ satisfies (I1-F) and (I2-F) for all i. If a is not in $F\text{-}TCER_i$ for any i, then the inductive hypothesis gives the result for π. Suppose a

is in $F\text{-}TCER_i$. The inductive hypothesis implies that $\pi|F\text{-}TCER_j$ satisfies (I1-F) and (I2-F) for all $j \neq i$.

For now, assume that $\pi'|F\text{-}TCER_i = \delta b$. The following four cases show that $\pi|F\text{-}TCER_i = \delta ba$ satisfies (I1-F) and (I2-F).

Case 1: $b = T_i$.

After b, *dine-region* is T and does not change until further actions in $F\text{-}TCER_i$ occur. Since T_i is only enabled when *dine-region* is R, and E_i is only enabled when *dine-region* is C, a can be neither of these actions.

Case 2: $b = C_i$.

After b, *dine-region* is C and does not become R until R_i occurs. Since T_i is only enabled when *dine-region* is R, a cannot be T_i.

Case 3: $b = E_i$.

After b, *dine-region* is E and does not change until further actions in $F\text{-}TCER_i$ occur. Since T_i is only enabled when *dine-region* is R, and E_i is only enabled when *dine-region* is C, a can be neither of these actions.

Case 4: $b = R_i$.

After b, *dine-region* is R and does not change until further actions in $F\text{-}TCER_i$ occur. Since E_i is only enabled when *dine-region* is C, a cannot be E_i.

End of Cases.

If there is no such b, i.e., a is the first action of $F\text{-}TCER_i$ in π, then initially *dine-region* is R. The same argument as in Case 4 shows that a cannot be E_i. □

Next we show that (I3-F) is true for the *Dine($\mathcal{F}$)* subroutine. One of the hypotheses is that *Dine($\mathcal{F}$)* solve the dining philosophers problem; this assumption is sufficient for the results of this subsection, since an automaton that solves the no-deadlock (or no-lockout) dining philosophers problem also solves the dining philosophers problem.

Lemma 9: *Let e be a fair execution of Drink($\mathcal{B}$) with schedule α such that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i. If Dine($\mathcal{F}$) solves the dining philosophers problem for $\mathcal{F} = \mathcal{B}$, then $\alpha|F\text{-}TCER_i$ satisfies (I3-F) for all i.*

Proof: Suppose in contradiction that $\alpha|F\text{-}TCER_i$ ends in C_i for some i.

If *Drinker(i)* is in *drink-region* C, E or R when the final C_i occurs, E_i becomes enabled and remains enabled throughout e. T_i never becomes enabled once E_i is (by Lemma 8), and $C_i(B)$ and $R_i(B)$ are not enabled for any B (since *send-*E_i is true). Thus, no output action in this class of the partition occurs after the final C_i, giving a contradiction since e is fair.

Assume *Drinker(i)* is in *drink-region* T when the final C_i occurs. Then for some $B \subseteq B_i$, $\alpha = \alpha_1 T_i(B) \alpha_2 C_i \alpha_3$, where there is no $C_i(B)$ in α_2, and no action from $F\text{-}TCER_i$ in α_3. We show that eventually $C_i(B)$ occurs in α_3. When the final C_i occurs, if there is any b in B that is not in *bottles* at *Drinker(i)*, *dem(b)* is placed in *buff[j]*, where $b \in B_j$. By fairness, eventually *deliver(dem(b),i,j)* occurs. If *Drinker(j)* is in *drink-region* C at that time, then (b, i) is put in its *deferred* variable. By (I3-B), there is a subsequent $E_j(B')$ actions, for some $B' \subseteq B_j$, and at that time, *sat(b)* is put in *buff[i]* at *Drinker(j)*. Suppose *Drinker(j)* is not in *drink-region* C when the demand is delivered. By Lemma 8 and part (b) of Lemma 7, *Drinker(j)* is not in *dine-region* C, and *sat(b)* is immediately placed in *buff[i]*. In either case, by fairness, eventually *deliver(sat(b),j,i)* occurs. Thus, all bottles in B eventually reside at *Drinker(i)* and remain there, since all subsequent requests are deferred. Thus, $C_i(B)$ is enabled, and occurs, causing E_i to become enabled, again giving a contradiction. □

The next lemma verifies condition (2) of Corollary 2 for showing no-deadlock and no-lockout.

Lemma 10: *Let e be a fair execution of Drink($\mathcal{B}$) with schedule α such that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i. Assume that $\alpha|F\text{-}TCER_i$ satisfies (O1-F) and (O2-F) for all i, and $\alpha|ext(Dine(\mathcal{F}))$ satisfies (EX-F) and (ND-F) (resp., (NL-F)), with $\mathcal{F} = \mathcal{B}$. For any i, if α (resp., $\alpha|B\text{-}TCER_i$) is finite, then $\alpha|B\text{-}TCER_i$ ends in $R_i(B)$ for some $B \subseteq B_i$.*

Proof: Suppose in contradiction that there exists an i such that α (resp., $\alpha|B\text{-}TCER_i$) is finite and $\alpha|B\text{-}TCER_i$ does not end in $R_i(B)$ for any $B \subseteq B_i$.

We now show that $\alpha|F\text{-}TCER_i$ ends in R_i. Let $\beta = \alpha|ext(Dine(\mathcal{F}))$.

No-deadlock: Since α is finite, β is also finite. Since β satisfies (ND-F), $\beta|F\text{-}TCER_i = \alpha|F\text{-}TCER_i$ ends in R_i.

No-lockout: Since $\alpha|B\text{-}TCER_i$ is finite, $\alpha|F\text{-}TCER_i = \beta|F\text{-}TCER_i$ is also finite, because T_i is only enabled if some $T_i(B)$ occurs. Since β satisfies (NL-F), $\beta|F\text{-}TCER_i = \alpha|F\text{-}TCER_i$ ends in R_i.

We now show that any choice other than $R_i(B)$ for the final action in $\alpha|B\text{-}TCER_i$ leads to a contradiction.

Case 1: $\alpha|B\text{-}TCER_i$ ends in $T_i(B)$ for some $B \subseteq B_i$. Then *Drinker(i)* remains in *drink-region* T for the rest of e. We show that E_i becomes enabled after the final $T_i(B)$, and use this fact to produce a contradiction.

Subcase 1a: If *Drinker(i)* is in *dine-region* T when the final $T_i(B)$ occurs, then subsequently E_i becomes enabled, since $\alpha|F\text{-}TCER_i$ ends in R_i.

Subcase 1b: If *Drinker(i)* is in *dine-region* C when the final $T_i(B)$ occurs, then there is a subsequent R_i, since $\alpha|F\text{-}TCER_i$ ends in R_i. T_i is immediately enabled. E_i never becomes enabled once T_i is (by Lemma 8), and $C_i(B')$ and $R_i(B')$ are not enabled for any B' (since *send-*T_i is true). T_i eventually occurs, since e is fair. Since $\alpha|F\text{-}TCER_i$ ends in R_i, eventually E_i becomes enabled.

Subcase 1c: If *Drinker(i)* is in *dine-region* E when the final $T_i(B)$ occurs, then there is a following R_i, since $\alpha|F\text{-}TCER_i$ ends in R_i. T_i is immediately enabled, and the same argument as in Subcase 1b shows that subsequently E_i becomes enabled.

Subcase 1d: If *Drinker(i)* is in *dine-region* R when the final $T_i(B)$ occurs, then T_i is immediately enabled. The same argument as in Subcase 1b shows that subsequently E_i becomes enabled.

End of Subcases.

We have shown that E_i becomes enabled after the final $T_i(B)$. But E_i becomes enabled in one of two ways: either *drink-region* is C, E or R when some C_i action occurs, which is not possible since *drink-region* remains T, or else some $E_i(B)$ action occurs, contradicting our assumption that $T_i(B)$ is the last action in $\alpha|B\text{-}TCER_i$.

Case 2: $\alpha|B\text{-}TCER_i$ ends in $C_i(B)$ for some $B \subseteq B_i$. This case is ruled out by (I3-B).

Case 3: $\alpha|B\text{-}TCER_i$ ends in $E_i(B)$ for some $B \subseteq B_i$. $C_i(B)$ is never enabled after this point, because *Drinker(i)* remains in *drink-region* E. If T_i is enabled when the final $E_i(B)$ occurs, it eventually occurs (cf. the argument in Subcase 1b), and since $\alpha|F\text{-}TCER_i$ ends in R_i, there are subsequent C_i, E_i and R_i actions. If E_i is enabled when the final $E_i(B)$ occurs, there are subsequent E_i and R_i actions since $\alpha|F\text{-}TCER_i$ ends in R_i. But after this R_i action, T_i, and thus E_i, are never again enabled. So after some point, $R_i(B)$, and no other action in the same class of the partition, is enabled for the rest of e, contradicting e being fair. □

The main theorems follow.

Theorem 11: *If Dine($\mathcal{F}$) solves the no-deadlock dining philosophers problem for $\mathcal{F}$, then Drink($\mathcal{B}$) solves the no-deadlock drinking philosophers problem for $\mathcal{B} = \mathcal{F}$.*

Proof: We verify conditions (1) and (2) of Corollary 2 for the (EX-B) and (ND-B) implications.

The same argument as in the proof of Theorem 6 verifies conditions (1) and (2) for the (EX-B) implication.

Consider the (ND-B) implication. The corresponding set of implications of the subroutine consists solely of the (ND-F) implication. Let e be a fair execution of *Drink($\mathcal{B}$)* with schedule α.

(1) Suppose that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i. By Lemmas 8 and 9, $\alpha|F\text{-}TCER_i$ satisfies (I1-F), (I2-F) and (I3-F) for all i.

(2) Suppose that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i, and that $\alpha|F\text{-}TCER_i$ satisfies (O1-F) and (O2-F) for all i, and $\alpha|ext(Dine(\mathcal{F}))$ satisfies (EX-F) and (ND-F). The same argument as in the proof of Theorem 6 shows that $\alpha|B\text{-}TCER_i$ satisfies (O1-B) and (O2-B) for all i and that $\alpha|ext(Drink(\mathcal{B}))$ satisfies (EX-B). By Lemma 10, if α is finite, then $\alpha|B\text{-}TCER_i$ ends in R_i for all i, thus showing that $\alpha|ext(Drink(\mathcal{B}))$ satisfies (ND-B).

By Corollary 2, *Drink($\mathcal{B}$)* solves the no-deadlock drinking philosophers problem. □

Theorem 12: *If Dine($\mathcal{F}$) solves the no-lockout dining philosophers problem for $\mathcal{F}$, then Drink($\mathcal{B}$) solves the no-lockout drinking philosophers problem for $\mathcal{B} = \mathcal{F}$.*

Proof: We verify conditions (1) and (2) of Corollary 2, for the (EX-B) and (NL-B) implications.

The same argument as in the proof of Theorem 6 verifies conditions (1) and (2) for the (EX-B) implication.

Consider the (NL-B) implication. The corresponding set of implications of the subroutine consists solely of the (NL-F) implication. Let e be a fair execution of *Drink($\mathcal{B}$)* with schedule α.

(1) Suppose that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i. By Lemmas 8 and 9, $\alpha|F\text{-}TCER_i$ satisfies (I1-F), (I2-F) and (I3-F) for all i.

(2) Suppose that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i, and that $\alpha|F\text{-}TCER_i$ satisfies (O1-F) and (O2-F) for all i, and $\alpha|ext(Dine(\mathcal{F}))$ satisfies (EX-F) and (NL-F). The same argument as in the proof of Theorem 6 shows that $\alpha|B\text{-}TCER_i$ satisfies (O1-B) and (O2-B) for all i and that $\alpha|ext(Drink(\mathcal{B}))$ satisfies (EX-B). By Lemma 10, if $\alpha|B\text{-}TCER_i$ is finite for some i, then $\alpha|B\text{-}TCER_i$ ends in R_i, thus showing that $\alpha|ext(Drink(\mathcal{B}))$ satisfies (NL-B).

By Corollary 2, *Drink($\mathcal{B}$)* solves the no-lockout drinking philosophers problem. □

5.3 More Concurrent

In this subsection we show that *Drink($\mathcal{B}$)* solves the more-concurrent drinking philosophers problem if *Dine($\mathcal{F}$)* solves the no-lockout dining philosophers problem. Again, we want to use Corollary 2. The more-concurrent drinking philosophers problem is defined by $n+2$ implications, (EX-B), (NL-B), and (MC-B) for i, $1 \leq i \leq n$. The (EX-B) and (NL-B) implications have been taken care of in previous sections. In this section we pick an arbitrary i and concentrate on the implication (MC-B) for i. Showing this implication is true requires nothing about the *Dine($\mathcal{F}$)* subroutine. Thus, in using Corollary 2, the set of implications of the subroutine used to verify (MC-B) for i is empty. Consequently, condition (1) of Corollary 2 is trivially true. Lemma 13 is used to verify condition (2). Theorem 14 puts the pieces together.

Lemma 13: *Let e be a fair execution of Drink($\mathcal{B}$) with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_k$ for all k, and (I4-B)$_i$ is true of $\alpha|ext(Drink(\mathcal{B}))$ for some fixed i. Then $\alpha|B\text{-}TCER_i$ satisfies (MC-B).*

Proof: Suppose in contradiction that $\alpha|B\text{-}TCER_i$ ends in $T_i(B)$ or $E_i(B)$ for some $B \subseteq B_i$.

Case 1: $\alpha|B\text{-}TCER_i$ ends in $T_i(B)$. When the final $T_i(B)$ occurs, request messages for any bottles in B that do not reside at *Drinker*(i) are placed in *Drinker*(i)'s *buff* variables. Since e is fair, they are eventually delivered. By $(\text{I4-B})_i$, every recipient is in *drink-region* E or R when the request arrives, and immediately satisfies the request. Since e is fair, the satisfy messages are eventually delivered to *Drinker*(i). Again by $(\text{I4-B})_i$, no request or demand message for any bottle in B is delivered to *Drinker*(i) as long as it is in (the current) *drink-region* T. Thus, once a bottle in B resides at *Drinker*(i) after the final $T_i(B)$, it remains there.

If E_i is enabled when the final $T_i(B)$ occurs, it eventually occurs by fairness of e, since no more actions in $B\text{-}TCER_i$ occur and T_i cannot be enabled. It is possible that an R_i action occurs after the E_i, in which case T_i becomes enabled. If T_i is enabled in any state after the final $T_i(B)$, it eventually occurs, since no more actions in $B\text{-}TCER_i$ occur and E_i cannot be enabled. It is possible that a C_i action occurs after the T_i, however, no E_i action is subsequently enabled, because *Drinker*(i) remains in *drink-region* T. Thus, eventually $C_i(B)$ and no other action in the same class of the partition is enabled and remains enabled, yet never occurs, contradicting e being fair.

Case 2: $\alpha|B\text{-}TCER_i$ ends in $E_i(B)$. If T_i is enabled when the final $R_i(B)$ occurs, it eventually occurs by fairness of e, since no more actions in $B\text{-}TCER_i$ occur and E_i cannot be enabled. It is possible that a C_i action occurs after the T_i, in which case E_i becomes enabled. If E_i is enabled in any state after the final $E_i(B)$, it eventually occurs, since no more actions in $B\text{-}TCER_i$ occur, and T_i cannot be enabled. It is possible that an R_i action occurs after the E_i; however, no T_i action is subsequently enabled, because *Drinker*(i) remains in *drink-region* E. Thus, eventually $R_i(B)$ and no other action in the same class of the partition is enabled and remains enabled, yet never occurs, contradicting e being fair. □

Theorem 14: *If Dine($\mathcal{F}$) solves the no-lockout dining philosophers problem for $\mathcal{F}$, then Drink($\mathcal{B}$) solves the more-concurrent drinking philosophers problem for $\mathcal{B} = \mathcal{F}$.*

Proof: We verify conditions (1) and (2) of Corollary 2 for the $n + 2$ implications (EX-B), (NL-B), and (MC-B) for i, $1 \le i \le n$. The same argument as in the proof of Theorem 12 verifies conditions (1) and (2) for the (EX-B) and (NL-B) implications.

Choose (MC-B) for any i, $1 \le i \le n$. The corresponding set of implications of the subroutine is empty. Condition (1) is vacuously true. Condition (2) is reduced to showing (MC-B) for i. Let e be fair execution of *Drink($\mathcal{B}$)* with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_k$ for all k, and $(\text{I4-B})_i$ is true of $\alpha|ext(Drink(\mathcal{B}))$. The same argument as in the proof of Theorem 6 shows that

$\alpha|B\text{-}TCER_k$ satisfies (O1-B) and (O2-B) for all k, and that $\alpha|extDrink(\mathcal{B}))$ satisfies (EX-B). By Lemma 13, $\alpha|B\text{-}TCER_i$ satisfies (MC-B).

By Corollary 2, *Drink($\mathcal{B}$)* solves the more-concurrent drinking philosophers problem for $\mathcal{B}$. □

5.4 Stronger Problem Definitions

In this subsection, we explore two other versions of the drinking philosophers problem, that demand a higher degree of concurrency than does the more-concurrent formulation. One version, the strongest, is not satisfied by either our algorithm or that in [CM]. The second, intermediate between the strongest and the more-concurrent, is not satisfied by our algorithm but is by [CM]. Our contribution in this section is providing precise definitions of the degree of concurrency demanded.

The strongest possible version of the drinking philosophers problem would require that if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses any of the bottles in B forever. (We could also allow the bottles in B to be kept forever after this request is satisfied.) Unfortunately, neither the algorithm in this paper nor that in [CM] satisfies this conditions. It would be interesting to devise one that does.

We now formalize this "strongest" definition. We introduce a new condition (I5-B)$_i$ which, similarly to (I4-B)$_i$, is indexed by process id i. It is a liveness condition imposed on the input, stating that whenever user i requests a set of resources, none of those resources is kept forever until (possibly) after i enters its critical region.

Given a sequence α from the set $\bigcup_{i=1}^{n} B\text{-}TCER_i$, define condition (I5-B)$_i$ for a fixed i, which α may or may not satisfy.

- (I5-B)$_i$ If a $T_i(B)$ action and a $C_j(B')$ action occur in α, for any $j \neq i$ with $B \cap B' \neq \emptyset$, then either an $E_j(B')$ action occurs after the $C_j(B')$ action, or a $C_i(B)$ action occurs between the $T_i(B)$ action and the $C_j(B')$ action.

The *strongest* drinking philosophers problem for $\mathcal{B}$ is the external schedule module $M_{STDrink}$ that is the sub-schedule module of $M_{NLDrink}$ consisting of all sequences α of actions that, in addition to satisfying the (EX-B) and (NL-B) implications, also satisfy the following n implications (ST-B) for i, $1 \leq i \leq n$: namely, if (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_k$ for all k and if (I5-B)$_i$ is true of α, then (O1-B) and (O2-B) are true of $\alpha|B\text{-}TCER_k$ for all k, (EX-B) and (NL-B) are true of α, and (MC-B) is true of $\alpha|B\text{-}TCER_i$.

The following situation shows that our algorithm does not solve the strongest drinking philosophers problem. (Essentially the same scenario shows that the algorithm in [CM] also does not.) Suppose there are three drinkers, 1, 2 and 3; 1 and 2 share bottle a, 2 and 3 share bottle b. First, 1 gets bottle a, enters its drinking critical region, and stays there forever. Then 2 requests a and b, obtains b, and enters its dining critical region. Since 2 can never obtain a, it stays in its dining critical region forever. Finally, 3 requests b. Drinker 2 does not relinquish b upon a mere request, and 3 can never demand b, because it can never enter its dining critical region. Thus, even though 3's bottle request includes no bottle that is ever in use, it can never enter its drinking critical region.

There is a version of the drinking philosophers problem specifying a degree of concurrency intermediate between strongest and more-concurrent, that the algorithm in [CM] solves and ours does not. The informal description is that if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses *or wants* any of the bottles in B forever. The formal statement is the same as the strongest definition, with (I5-B)$_i$ replaced with (I6-B)$_i$:

- (I6-B)$_i$ If a $T_i(B)$ action and a $T_j(B')$ action occur in α, for any $j \neq i$ with $B \cap B' \neq \emptyset$, then either an $E_j(B')$ action occurs after the $T_j(B')$ action, or a $C_i(B)$ action occurs between the $T_i(B)$ action and the $T_j(B')$ action.

The following scenario shows that our algorithm does not solve this problem. Suppose there are five drinkers, 1 through 5. Drinkers 1 and 2 share bottle a, 2 and 3 share b, 3 and 4 share c, and 3 and 5 share d. First, 1 gets a, enters its drinking critical region and stays there forever. Then 2 requests a and b, obtains b and enters its dining critical region. As in the previous scenario, 2 remains in its dining critical region forever. Next, 3 requests c and d. It obtains c from 4. Then 4 requests c from 3, the request is deferred, 4 demands c from 3, and the request is satisfied. Now 3 obtains d from 5. But 3 will never get c from 4, because it can never demand it. Thus, although none of the bottles required by 3 are ever wanted forever by another drinker, 3 cannot enter its drinking critical region.

In this particular case, a small change to the algorithm can solve the problem — whenever a demand for a needed bottle is satisfied, another request for that bottle is sent. (Almost the identical analysis as already given can be used to verify the correctness of the algorithm with this change.) However, 3 can be locked out if 4 and 5 alternate forever using c and d even with this change.

In contrast, the algorithm in [CM] will allow 3 to enter its drinking critical region, even if 4 and 5 alternate forever using c and d. The forks in the dining philosophers algorithm provide a priority for the use of the corresponding bottles by the drinkers. The priority alternates between the two processes sharing the resource. Thus, once 3 obtains c it will not relinquish it until it has gotten to use it. In general, priority is broken down on a link-by-link basis, whereas in our (more modular) algorithm, the priority comes only with entering the dining critical region.

6. Complexity Analysis

In this section, we evaluate our no-lockout algorithm using the criteria listed in [CM], as well as analyzing the worst-case waiting time. The analysis of the worst-case waiting time shows that the limiting factor is the no-lockout dining philosophers subroutine. By replacing the $O(n)$ subroutine of [CM] with an $O(1)$ subroutine (for instance, that of [Ly]), we obtain an $O(1)$ drinking philosophers algorithm.

We would like to bound how long a process must wait after requesting to enter its critical region until it does so. The following definitions provide a measure of time complexity for our model that is analogous to that in [PF], in which an upper bound on process step time, but no lower bound, is assumed. (Thus, all interleavings of system events are still possible.)

Given an execution e of automaton A, where $e = s_0 a_1 s_1 a_2 \ldots$, a *timing function* for e is an increasing function t_e mapping positive integers to nonnegative real numbers such that for each real number t, only a finite number of integers i satisfy $t_e(i) < t$. Intuitively, $t_e(i)$ is the real time at which a_i occurs; we rule out an infinite number of actions occurring before a finite real time.

Execution e is *c-bounded*, for positive real c, if the following condition is true for each class C of the partition $part(A)$. For each $i \geq 0$, either

(1) there exists $j > i$ such that a_j is in C and $t_e(j) - t_e(i) \leq c$, or

(2) there exists $j \geq i$ such that no action of C is enabled in s_j and $t_e(j) - t_e(i) \leq c$.

That is, starting at any point in the execution, within time c either some output action in C occurs, or else the automaton passes through a state in which no output action in C is enabled. Each class of the partition is considered separately, since each class corresponds, in some sense, to a distinct entity in a larger system.

Now we analyze the worst-case time behavior of the no-lockout drinking philosophers algorithm, automaton *Drink($\mathcal{B}$)*, which uses any no-lockout dining philosophers subroutine *Dine($\mathcal{F}$)*, with $\mathcal{F} = \mathcal{B}$. For the following definitions, we only consider fair c-bounded executions of *Drink($\mathcal{B}$)* with schedules α such that $\alpha|B\text{-}TCER_i$ satisfies (I1-B), (I2-B) and (I3-B) for all i.

Let try_{Drink} be the maximum time, over all i and all $B \subseteq B_i$, between any $T_i(B)$ action and the subsequent $C_i(B)$ action, in any execution. Let $crit_{Drink}$ be the maximum time, over all i and all $B \subseteq B_i$, between any $C_i(B)$ action and the subsequent $E_i(B)$ action, in any execution.

Let try_{Dine} be the maximum time over all i between any T_i action and the subsequent C_i action, in any execution. Let $exit_{Dine}$ be the maximum time over all i between any E_i action and the subsequent R_i action, in any execution.

We assume that $crit_{Drink}$ and $exit_{Dine}$ are constants.

Our goal is to find an upper bound on try_{Drink}, the maximum time a user process must wait after requesting to enter its critical region until it is allowed to do so. First we show that in any state, there are a bounded number of messages in any *buff*. Let r be the maximum number of bottles shared by any two drinkers.

Lemma 15: *Let e be a fair execution of Drink($\mathcal{B}$) with schedule α such that (I1-B) and (I2-B) are true of $\alpha|B\text{-}TCER_i$ for all i. If Dine($\mathcal{F}$) solves the dining philosophers problem for $\mathcal{F} = \mathcal{B}$, then in any state of e, there are at most $4r$ messages in buff[j] at Drinker(i), for any i and j.*

Proof: Let s be any state in e, and b a bottle shared between *Drinker*(i) and *Drinker*(j). The following four facts imply the result.

(1) First we show that there cannot be two $req(b)$ messages in *buff*[i] at *Drinker*(j) in s. Suppose such is the case. (See Figure 5.)

Let $T_j(B)$ be the action triggering the first $req(b)$, and $T_j(B')$ the action triggering the second. (Thus, b is in both B and B'.) By Lemma 4, there is an action $C_j(B)$ between $T_j(B)$ and $T_j(B')$. Thus, at some point between the $T_j(B)$ and the $C_j(B)$, b resides at *Drinker*(j). Since b does not reside at *Drinker*(j) when the $T_j(B')$ occurs, there is a $sat(b)$ message between the two $req(b)$ messages in *buff*[i]. Consider the state s immediately before the delivery of the first $req(b)$ message. By Lemma 5, part (d), b resides at *Drinker*(i) in s. Yet there is also a $sat(b)$ message in *buff*[i] at *Drinker*(j), contradicting Lemma 5, part (a).

(2) By Lemma 5, part (b), there cannot be two $sat(b)$ messages in *buff*[i] at *Drinker*(j) in s.

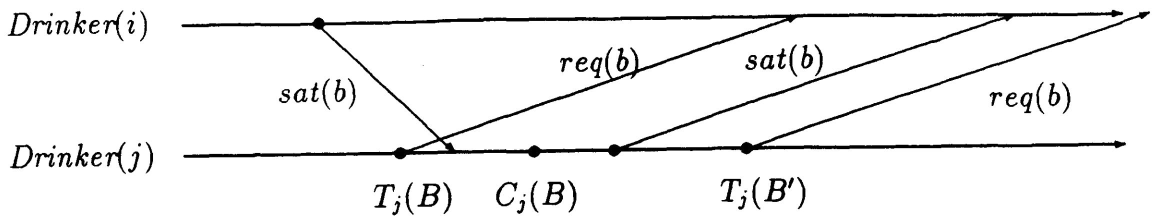

Figure 5

(3) Next we show that there cannot be a *sat*(b) message immediately followed by a *dem*(b) message in *buff*[i] at *Drinker*(j) in s. Suppose such is the case. Some action C_j causes the *dem*(b) to be put in *buff*[i], and at that time *Drinker*(j) is in *drink-region* T.

If the $T_j(B)$ action causing *Drinker*(j) to be in *drink-region* T occurs when the *sat*(b) message is in *buff*[i], then at that time, b does not reside at *Drinker*(j) (by Lemma 5, part (a)), and so there is a *req*(b) message in between the *sat*(b) and the *dem*(b), contradicting our assumption.

Thus, the $T_j(B)$ action must precede the action that triggers the *sat*(b) message. (See Figure 6.)

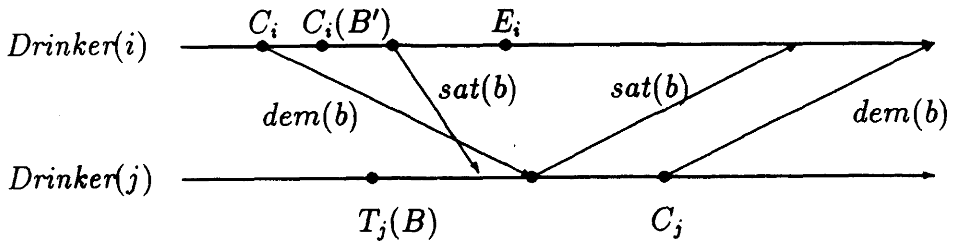

Figure 6

The only action that can trigger the *sat*(b) message while *Drinker*(j) is in *drink-region* T is the delivery of a *dem*(b) message to *Drinker*(j) while b resides at *Drinker(j)*. The *dem(b)* message to *Drinker(j)* is triggered by a C_i action. By Lemma 8 and part (b) of Lemma 7, there is an E_i action between the C_i and the C_j. Yet this E_i cannot happen until some $C_i(B')$ action occurs (for some $B' \subseteq B_i$), which cannot happen until b resides at *Drinker(i)*. Yet once the *sat(b)* is in *buff[i]* at *Drinker(j)*, by part (a) of Lemma 5, b cannot reside at *Drinker(i)*, and *sat(b)* cannot be in *buff[j]* at *Drinker(i)*. Thus, a *sat(b)* message from *Drinker(i)* to *Drinker(j)* is triggered by some action that follows the $C_i(B')$ (and thus follows the C_i), yet is delivered before the *dem(b)* message from *Drinker(i)* to *Drinker(j)*, contradicting the FIFO nature of *buff[j]*.

(4) Finally we show that there cannot be two consecutive *dem(b)* messages in *buff[i]* at *Drinker(j)* in s. Suppose there are. Each of the two *dem(b)* messages is triggered by a C_j action occurring when *Drinker(j)* is in *drink-region* T. Thus there is a $T_j(B)$ action preceding the first C_j, and a $T_j(B')$ action between the two C_j actions. By Lemma 4, there is a $C_j(B)$ action between the first C_j action and the $T_j(B')$. When the $C_j(B)$ occurs, b resides at *Drinker(j)*. Yet at the second C_j, b does not reside at *Drinker(j)*, so there is a *sat(b)* message in between the two *dem(b)* messages, contradicting our assumption. □

The main theorem follows. (Recall that r is the maximum number of bottles shared by any two drinkers.)

Theorem 16: $try_{Drink} \leq (3 + 8r)c + exit_{Dine} + try_{Dine} + crit_{Drink}$.

Proof: Let e be a fair c-bounded execution of *Drink($\mathcal{B}$)* with schedule α such that (I1-B), (I2-B) and (I3-B) are true of $\alpha | B\text{-}TCER_i$ for all i. Choose any i. Suppose $T_i(B)$ occurs at time t, for some $B \subseteq B_i$. First we bound the time until the next C_i action occurs, considering all possible values of *dine-region* for *Drinker(i)* in the state s immediately before the action $T_i(B)$.

Case 1: dine-region is T in s. By time $t + try_{Dine}$, C_i occurs.

Case 2: dine-region is C in s. Thus, there is a most recent C_i action preceding the $T_i(B)$. We show that *send-E_i* is already true. If there is some $E_i(B')$ action between the preceding C_i and the $T_i(B)$, then *send-E_i* is set to true when the $E_i(B')$ occurs. Otherwise, *drink-region* is E or R when the preceding C_i occurs, and *send-E_i* is set to true then. Since no other output action in the same class of the partition can be enabled while E_i is, an E_i action occurs by time $t + c$. By time $t + c + exit_{Dine}$, an R_i action occurs and *send-T_i* is set to true. Since no other output action in the same class of the partition can be enabled while T_i is, a T_i

action occurs by time $t + 2c + exit_{Dine}$. Finally, by time $t + 2c + exit_{Dine} + try_{Dine}$, a C_i action occurs.

Case 3: dine-region is E in s. By time $t + exit_{Dine}$, R_i occurs and since *drink-region* is T, *send-*T_i is set to true. By time $t + c + exit_{Dine}$, T_i occurs. Finally, by time $t + c + exit_{Dine} + try_{Dine}$, C_i occurs.

Case 4: dine-region is R in s. Then *send-*T_i is set to true when $T_i(B)$ occurs. By time $t + c$, T_i occurs, and by time $t + c + try_{Dine}$, C_i occurs.

End of Cases.

Thus, in the worst case, the next C_i action occurs by time $t + 2c + exit_{Dine} + try_{Dine}$.

Now we analyze how long it can take some bottle $b \in B$ to reside at *Drinker*(i) after this next C_i action has occurred. Pick any $b \in B$ that does not reside at *Drinker*(i) when the C_i occurs. Then *dem*(b) is put in *buff*$[j]$ at *Drinker*(i), where $b \in B_j$. By Lemma 15, there are at most $4r$ messages in *buff*$[j]$ in any state of e; by $4rc$ time later, *deliver*$(dem(b), i, j)$ occurs. If *Drinker*(j) is not in *drink-region* C when the demand is delivered, *sat*(b) is immediately put in *buff*$[i]$ at *Drinker*(j) (since, by Lemma 8 and part (b) of Lemma 7, *Drinker*(j) cannot be in *dine-region* C). Otherwise, (b, i) is put in *deferred*; by $crit_{Drink}$ later, $E_j(B')$ occurs, for some $B' \subseteq B_j$, and then *sat*(b) is added to *buff*$[i]$. In either case, when *sat*(b) is added to *buff*$[i]$, there are at most $4r$ messages in *buff*$[i]$ (by Lemma 15), and by $4rc$ later, *deliver*$(sat(b), j, i)$ occurs.

Thus, by $8rc + crit_{Drink}$ time after the C_i, all bottles in B reside at *Drinker*(i) and stay there. Thus $C_i(B)$ is enabled and remains enabled until it occurs. By Lemma 4, $R_i(B)$ cannot be enabled before $C_i(B)$ occurs; by Lemma 8 and part (a) of Lemma 7, T_i cannot be enabled as long as C_i is the most recent action from $F\text{-}TCER_i$; by inspecting the code, E_i cannot be enabled before $C_i(B)$ occurs. Thus, by c time after all bottles in B reside at *Drinker*(i), $C_i(B)$ occurs. Summing all these bounds gives the result. □

Since we assume that $crit_{Drink}$, $exit_{Dine}$, r and c are constants, the worst-case waiting time of this solution depends on try_{Dine}, the worst-case waiting time of the dining philosophers subroutine. The dining philosophers subroutine used in [CM] has try_{Dine} of $O(n)$. By replacing it with, for instance, the dining philosophers algorithm in [Ly], which has worst-case waiting time of $O(1)$, we obtain a more efficient drinking philosophers algorithm. (The algorithm in [Ly] has time $O(1)$ in the sense that the worst-case waiting time is a function of local information, including the maximum number of users for each resource, and the maximum number of resources for each user, and is not necessarily a function of the total number of users.)

Our drinking philosophers algorithm could be modified to replace r with a small constant, if the request, demand, and satisfy messages took a set of bottles as arguments instead of a single bottle.

Five criteria for evaluating resource allocation algorithms are given in [CM], fairness, symmetry, economy, concurrency and boundedness. We discuss each in turn.

Fairness corresponds to our definition of no-lockout. Our drinking philosophers solution has the no-lockout property as long as the dining philosophers subroutine has it.

Symmetry means that each process runs the identical program. This property is true of our solution, as long as it is true of the subroutine.

Economy means that processes send and receive a finite number of messages between subsequent entries to their critical regions, and a process that enters its critical region a finite number of times does not send or receive an infinite number of messages. Our solution has this property: Recall that when a drinker first receives $T_i(B)$, it sends $req(b)$ messages for all missing resource. It defers any $req(b)$ messages it receives when in ***drink-region*** T, but yields to $dem(b)$ messages. When it enters ***dine-region*** C, it sends $dem(b)$ messages for any missing resources. Thus at most four messages ($req(b)$, $sat(b)$, $dem(b)$, $sat(b)$) are sent on behalf of any bottle for any one trying attempt. Furthermore, once a drinker stops wanting to enter its critical region, it may receive a request for each of its bottles, but after satisfying the requests, it never sends or receives any more messages.

Concurrency means that "the solution does not deny the possibility of simultaneous drinking from different bottles by different philosophers." This is certainly true of our algorithm, since it satisfies the more-concurrent condition. More precise formulations of "concurrency" were given in our definitions (see Sections 3 and 5.4).

Boundedness means that the number of messages in any $buff[j]$ variable is bounded, and the size of each message is bounded. This is certainly true of our solution, by Lemma 15.

5 A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm

A complete and rigorous proof of the correctness of Gallager, Humblet and Spira's distributed minimum spanning tree algorithm is presented. This important algorithm has been difficult to prove, because it does not break down naturally into pieces that may be proved separately and then composed. Our lattice-structured technique provides a way to use modularity to prove such not-very-modular algorithms. The method of proof is to represent the entire algorithm at different levels of abstraction and demonstrate relationships between the representations that preserve correctness. Each representation is a complete description of the algorithm, but with different aspects of the algorithm's states and actions described at different levels of detail. The representations are arranged in a lattice instead of a chain; representations that are not comparable in the lattice concentrate on different tasks performed by the algorithm. A framework of definitions and theorems is given, using the I/O automaton model of Lynch and Tuttle, for showing that both safety and liveness properties of the algorithm are preserved throughout the lattice.

1. Introduction

A complete and rigorous proof of the correctness of Gallager, Humblet and Spira's distributed minimum spanning tree algorithm [GHS] is presented. This particular algorithm has been of great interest for some time; although an intuitive description of why it should work is given, there is no formal proof of its correctness

This chapter is joint work with Leslie Lamport and Nancy Lynch.

in [GHS]. It is important to have such a proof for several reasons. First, this algorithm solves a significant problem. Minimum spanning trees are very useful in many distributed applications, such as implementing efficient broadcast in communication networks, and electing a leader. In turn, the election of a leader can facilitate crash recovery. Second, the algorithm of [GHS] often appears as a subroutine in other algorithms [A2] [AG]; the correctness of these algorithms thus depends directly on the correctness of [GHS]. Finally, many of the individual concepts and techniques are taken from [GHS], out of context, and used in other algorithms [A2] [CT] [Ga]. Yet the pieces of the [GHS] algorithm interact in subtle ways, some of which are not explained in the original paper. A careful proof of the entire algorithm can illuminate the dependencies between the pieces.

We present a way to generalize existing verification techniques for a totally ordered hierarchy to a partially ordered one. Our method of proof is to represent the algorithm at different levels of abstraction and demonstrate relationships between the representations that preserve correctness. The representations are arranged in a lattice instead of a chain. Each representation is a complete description of the algorithm, but with different aspects of the algorithm's actions and state described at different levels of detail. Representations that are not comparable in the lattice concentrate on different functions performed by the algorithm. The advantage is that one can concentrate on a single function at a time, without becoming bogged down by irrelevant details.

A framework of definitions and theorems is given, using the I/O automaton model of Lynch, Merritt, and Tuttle [LT] [LM], for verifying both safety and liveness properties of the algorithm. (Any state-based assertional method could be used; we chose I/O automata because of the support it gives for describing the interaction between the algorithm and its environment.) Once a more concrete version of the algorithm and one (or more) more abstract versions of the algorithm have been presented, state and action mappings from the more concrete to the more abstract version(s) are posited, together with a predicate about the state of the more concrete version. To verify safety properties, it is necessary to show that the predicate is an invariant (i.e., true in every reachable state), and that steps of the more concrete version simulate steps of the more abstract version(s). Generally, an invariant about the more abstract version is used in the verification. We provide three techniques for verifying liveness properties; all are similar to the safety technique in that they only involve checking that certain conditions are true in certain states, and that certain steps preserve certain properties.

The method seems useful for developing modular proofs of algorithms that are not very modular, even very complicated ones (i.e., many practical algorithms). It would have been very hard for us to come up with a proof of [GHS] if we had to work with the whole algorithm at once — this method let us reason about each facet of the algorithm alone, which we felt was necessary to allow us to discover the correct invariants. One can sometimes describe a more modular algorithm as a true composition of automata. (See, for example, the proof in [FLS] of the synchronizer from [A1].) Such a complete decomposition allows for a simpler, shorter proof than the one we have here — the lowest level is essentially just a product of the subalgorithms, and the mappings from the lowest level to the subalgorithms are immediate. The [GHS] algorithm is not quite modular enough for such a complete decomposition, so there is some duplication in the various subalgorithms. Yet many of the benefits of modularity are retained — the proof has high-level modular structure that follows one's intuition; also, whenever a bug surfaced, it was easy to isolate the affected places and determine what needed to be modified.

The complete proof of the correctness of the [GHS] algorithm is very long. Part of the reason for the length is the duplication resulting from our decomposition, as discussed above. We believe the extra length is worthwhile because of the greater understanding it provides. The full proof also includes extremely detailed arguments, at a level of detail that is machine-checkable. It seems necessary to go through the exercise of doing these proofs, though, in order to catch small bugs, and to formulate the proper invariants. Perhaps in the future the checking could be done mechanically. Another point to bear in mind when considering the length of the proof is that the algorithm is delicate and complex, even though it is not very long.

In the future, it would be interesting to see if our method applies to related algorithms, such as [A2] [AG] [CT] [Ga], and to see which, if any, of the pieces of the proof can be reused. This proof took us a year – the others we expect would be quicker. Machine verification of this and related proofs seems tractable and could be investigated.

Related work falls into two categories: other proofs of the [GHS] algorithm, and similar proof techniques. Stomp and de Roever [SR] are currently working on a proof of the algorithm using the notion of communication-closed layers. Their proof is at the level of "informal mathematics."

Lam and Shankar [LS] have developed the "method of projections" for analyzing communication protocols. A lattice structure is used in this work as well. A specific function of the original protocol can be verified by aggregating certain states and events of the original protocol to produce a "projection" protocol, which focuses on the desired function. However, their definitions and theorems are specialized for communication protocols, whereas our technique is more general.

Kurshan [K] has an automata-based model and a method for describing systems at different levels of abstraction and defining mappings between the levels that preserve correctness. No support for lattice-structured proofs is given.

Work of Lamport [La2 [La3] uses temporal logic to specify, in a state-based, axiomatic way, concurrent programs at different levels of abstraction. Each level is specified by a set of state functions that satisfy a set of temporal logic assertions. One level is a refinement of another if each high-level state function can be defined in terms of the low-level state functions such that the high-level assertions are provable from the low-level assertions. No provision is made for lattice-structured proofs.

One of our techniques for proving liveness (see definition of "progressive" in Section 2) is very similar to a technique presented in [LPS], called the "method of helpful directions" in [F]. However, in those works, the method is used to prove termination of a program, considered at a single level of abstraction, whereas we use it to show that a low-level algorithm causes high-level actions to be simulated.

Section 2 contains the definitions and theorems upon which our method of proof is based. Section 3 formally defines the minimum spanning tree problem in our model. Section 4 consists of the proof of correctness of the algorithm. Following an overview of the proof, Section 4.1 shows that a very high level description of the algorithm solves the problem. Section 4.2 verifies the necessary safety properties throughout the lattice, and Section 4.3 does the same with liveness properties. All the results are pulled together in Section 4.4.

2. Foundations

This section contains the definitions and theorems used to relate descriptions of an algorithm at different levels of abstraction. The I/O automaton model of Lynch and Tuttle [LT] is the basis of our framework. (See Appendix for a summary.) We define a mapping from a more concrete algorithm to a more abstract algorithm that preserves the safety and liveness properties necessary to show that the automaton modeling the more concrete algorithm *satisfies* the automaton modeling the more abstract algorithm.

Section 2.1 deals with safety properties. First, suppose there are two automata, A and B, where B is offered as a "more abstract" version of A. We define a mapping from executions of A to sequences of alternating states and actions of B; if the mapping obeys certain conditions, we say A *simulates* B. Lemma 1 proves that this definition preserves important safety properties, namely that executions of A map to executions of B, and that a certain predicate is an invariant for A. Next we suppose that there are several higher-level versions, A_1, A_2, etc., of one more concrete automaton A. There are situations in which it is difficult to show independently that A simulates A_1 and A simulates A_2, but invariants about states of A_2 can help show a mapping from A to A_1, and invariants about states of A_1 can help show a mapping from A to A_2. To capture this, we define a notion of *simultaneously simulates*, which Lemma 2 proves preserves the same safety properties as in Lemma 1. Of course, to be able to apply Lemma 2, we must know what the invariants of A_1 and A_2 are, which may require having already shown that A_1 and A_2 simulate other automata.

Section 2.2 considers liveness properties. Given automata A and B, and a locally-controlled action φ of B, a definition of A being *equitable* for φ is given; Lemmas 3 and 4 show that this definition implies that in the execution of B obtained from a fair execution of A by either of the simulation mappings, once φ becomes enabled, it either occurs or becomes disabled. We are on our way to verifying the fairness of the induced execution of B.

Three methods of showing that A is equitable for locally-controlled action φ of B are described. The first method is to show that there is an action ρ of A that is enabled whenever φ is, and whose occurrence implies φ's occurrence. (Cf. Lemma 5.)

The second method uses a definition of A being *progressive* for φ. The intuition behind the definition is that there is a set of "helping" actions of A that are guaranteed to occur, and which make progress toward an occurrence of φ in the induced execution of B. Lemma 6 shows that progressive implies equitable.

The third method for checking the equitable condition can be useful when various automata are arranged in a lattice. (See Figure 1.) Suppose B and C are more abstract versions of A, and D is a more abstract version of C. In order to show that A is equitable for action φ of B, we demonstrate an action ρ of D that is "similar" to φ, such that C is progressive for ρ using a set Ψ of helping actions, and A is equitable for all the helping actions in Ψ. (Cf. Lemma 7.)

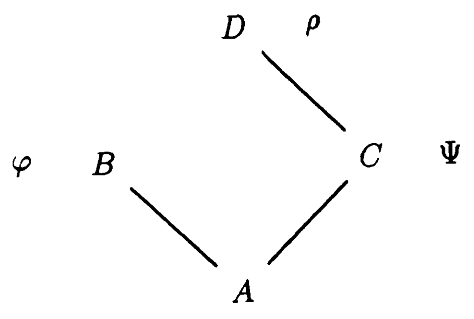

Figure 1

Theorems 8 and 9 in Section 2.3 relate the definitions of simulates, simultaneously simulates, and equitable to the notion of satisfaction.

2.1 Safety

Let A and B be automata. Throughout this chapter, we only consider automata such that each locally-controlled action is in a separate class of the action partition. (The definitions and results of this section can be generalized to avoid this assumption, but the statements and proofs are more complicated, and the generalization is not needed for the proof of the [GHS] algorithm.) Let $alt\text{-}seq(B)$ be the set of all finite sequences of alternating actions of B and states of B that begin and end with an action, including the empty sequence (and the sequence of a single action). An ***abstraction mapping*** $\mathcal{M}$ from A to B is a pair of functions, $\mathcal{S}$ and $\mathcal{A}$, where $\mathcal{S}$ maps $states(A)$ to $states(B)$ and $\mathcal{A}$ maps pairs (s, π), of states s of A and actions π of A enabled in s, to $alt\text{-}seq(B)$.

Given execution fragment $e = s_0\pi_1 s_1 \ldots$ of A, define $\mathcal{M}(e)$ as follows.

- If $e = s_0$, then $\mathcal{M}(e) = \mathcal{S}(s_0)$.
- Suppose $e = s_0 \ldots s_{i-1}\pi_i s_i$, $i > 0$. If $\mathcal{A}(s_{i-1}, \pi_i)$ is empty, then $\mathcal{M}(e) = \mathcal{M}(s_0 \ldots s_{i-1})$. If $\mathcal{A}(s_{i-1}, \pi_i) = \varphi_1 t_1 \ldots t_{m-1}\varphi_m$, then $\mathcal{M}(e) = \mathcal{M}(s_0 \ldots s_{i-1})$ $\varphi_1 t_1 \ldots t_{m-1}\varphi_m \mathcal{S}(s_i)$. The t_j are called ***interpolated*** states of $\mathcal{M}(e)$.
- If e is infinite, then $\mathcal{M}(e)$ is the limit of $\mathcal{M}(s_0\pi_1 s_1 \ldots s_i)$ as i increases without bound.

We now define a particular kind of abstraction mapping, one tailored for showing inductively that a certain predicate is an invariant of A, and that executions of

A map to (nontrivial) executions of B. (A *predicate* is a Boolean-valued function. If Q is a predicate on $states(B)$, and $\mathcal{S}$ maps $states(A)$ to $states(B)$, then $(Q \circ \mathcal{S})$, applied to state s of A, is the predicate "Q is true in $\mathcal{S}(s)$," and is also written $(Q(\mathcal{S}(s)))$.)

Definition: Let A and B be automata with the same external action signature. Let $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ be an abstraction mapping from A to B, P be a predicate on $states(A)$, and Q be a predicate true of all reachable states of B. We say A *simulates* B *via* $\mathcal{M}$, P, and Q if the following three conditions are true.

(1) If s is in $start(A)$, then
  (a) $P(s)$ is true, and
  (b) $\mathcal{S}(s)$ is in $start(B)$.

(2) If s is a state of A such that $Q(\mathcal{S}(s))$ and $P(s)$ are true, and π is any action of A enabled in s, then $\mathcal{A}(s,\pi)|ext(B) = \pi|ext(A)$.

(3) Let (s', π, s) be a step of A such that $Q(\mathcal{S}(s'))$ and $P(s')$ are true. Then
  (a) $P(s)$ is true,
  (b) if $\mathcal{A}(s',\pi)$ is empty, then $\mathcal{S}(s) = \mathcal{S}(s')$, and
  (c) if $\mathcal{A}(s',\pi) = \varphi_1 t_1 \ldots t_{m-1}\varphi_m$, then $\mathcal{S}(s')\varphi_1 t_1 \ldots t_{m-1}\varphi_m \mathcal{S}(s)$ is an execution fragment of B. □

The first lemma verifies that if A simulates B via $\mathcal{M}$, then $\mathcal{M}(e)$ is an execution of B and a certain predicate is true of all states of e.

Lemma 1: *If A simulates B via $\mathcal{M} = (\mathcal{S}, \mathcal{A})$, P and Q, then the following are true for any execution e of A.*

(1) $\mathcal{M}(e)$ is an execution of B.

(2) $(Q \circ \mathcal{S}) \wedge P$ is true in every state of e.

Proof: Let $e = s_0\pi_1 s_1 \ldots$. If (1) and (2) are true for every finite prefix $e_i = s_0 \ldots s_i$ of e, then (1) and (2) are true for e. We proceed by induction on i. We need to strengthen the inductive hypothesis for (1) to be the following:

(1) $\mathcal{M}(e_i)$ is an execution of B and $\mathcal{S}(s_i) = t$, where t is the final state in $\mathcal{M}(e_i)$.

(Throughout this proof, "conditions (1), (2) and (3)" refer to the conditions in the definition of "simulates".)

Basis: $i = 0$. (1) $\mathcal{M}(e_0) = \mathcal{S}(s_0)$. Since e_0 is an execution of A, s_0 is in $start(A)$. Condition (1b) implies that $\mathcal{S}(s_0)$ is in $start(B)$, so $\mathcal{M}(e_0)$ is an execution of B. Obviously, the assertion about the final states is true.

(2) Condition (1a) states that P is true in s_0. Since $\mathcal{S}(s_0)$ is in $start(B)$, it is a reachable state of B, and $Q(\mathcal{S}(s_0))$ is true.

Induction: $i > 0$. By the inductive hypothesis for (2), $Q(\mathcal{S}(s_{i-1}))$ and $P(s_{i-1})$ are true. Thus, conditions (3a), (3b) and (3c) are true.

(1) Let $\mathcal{M}(e_{i-1}) = t_0\varphi_1 t_1 \ldots t_j$ and $\mathcal{M}(e_i) = t_0\varphi_1 t_1 \ldots t_m$. Obviously, $m \geq j$.

Suppose $m = j$. Then $\mathcal{M}(e_i) = \mathcal{M}(e_{i-1})$ and is an execution of B by the inductive hypothesis for (1). We deduce that $\mathcal{A}(s_{i-1}, \pi_i)$ is empty, so by condition (3b), $\mathcal{S}(s_i) = \mathcal{S}(s_{i-1})$, and by the inductive hypothesis for (1), $\mathcal{S}(s_{i-1}) = t_j$.

Suppose $m > j$. By construction of $\mathcal{M}(e_i)$, $\mathcal{A}(s_{i-1}, \pi_i) = \varphi_{j+1}t_{j+1} \ldots t_{m-1}\varphi_m$, and $t_m = \mathcal{S}(s_i)$. By the inductive hypothesis for (1), $\mathcal{S}(s_{i-1}) = t_j$. By condition (3c), $t_j\varphi_{j+1} \ldots \varphi_m t_m$ is an execution fragment of B. Thus, $\mathcal{M}(e_i)$ is an execution of B. Obviously, the assertion about the final states is true.

(2) By the inductive hypothesis for (2), $(Q \circ \mathcal{S}) \wedge P$ is true in every state of e_i, except (possibly) s_i. By condition (3a), $P(s_i)$ is true. The final state in $\mathcal{M}(e_i)$ is $\mathcal{S}(s_i)$. Since, by part (1), $\mathcal{M}(e_i)$ is an execution of B and $\mathcal{S}(s_i)$ equals the final state of $\mathcal{M}(e_i)$, $\mathcal{S}(s_i)$ is a reachable state of B. By definition of Q, $Q(\mathcal{S}(s_i))$ is true. □

The next definition maps one automaton to multiple higher-level automata.

Definition: Let I be an index set. Let A and A_r, $r \in I$, be automata with the same external action signature. For all $r \in I$, let $\mathcal{M}_r = (\mathcal{S}_r, \mathcal{A}_r)$ be an abstraction mapping from A to A_r, and let Q_r be a predicate true of all reachable states of A_r. Let P be a predicate on $states(A)$. We say A *simultaneously simulates* $\{A_r : r \in I\}$ *via* $\{\mathcal{M}_r : r \in I\}$, P, and $\{Q_r : r \in I\}$ if the following three conditions are true.

(1) If s is in $start(A)$, then
  (a) $P(s)$ is true, and
  (b) $\mathcal{S}_r(s)$ is in $start(A_r)$ for all $r \in I$.

(2) If s is a state of A such that $\bigwedge_{r\in I} Q_r(\mathcal{S}_r(s))$ and $P(s)$ are true, and π is any action of A enabled in s then $\mathcal{A}_r(s, \pi)|ext(A_r) = \pi|ext(A)$ for all $r \in I$.

(3) Let (s', π, s) be a step of A such that $\bigwedge_{r \in I} Q_r(\mathcal{S}_r(s'))$ and $P(s')$ are true. Then
(a) $P(s)$ is true,
(b) if $\mathcal{A}_r(s', \pi)$ is empty, then $\mathcal{S}_r(s) = \mathcal{S}_r(s')$, for all $r \in I$, and
(c) if $\mathcal{A}_r(s', \pi) = \varphi_1 t_1 \ldots t_{m-1} \varphi_m$, then $\mathcal{S}_r(s') \varphi_1 t_1 \ldots t_{m-1} \varphi_m \mathcal{S}_r(s)$ is an execution fragment of A_r, for all $r \in I$. □

The statement "A simultaneously simulates $\{A_1, A_2\}$ via $\{\mathcal{M}_1, \mathcal{M}_2\}$, P and $\{Q_1, Q_2\}$" is weaker than the statement "A simulates A_1 via $\mathcal{M}_1$, P and Q_1, and A simulates A_2 via $\mathcal{M}_2$, P and Q_2" because the hypotheses of conditions (2) and (3) in the simultaneous definition require that a stronger predicate be true. (By restating Q_2 as a predicate on states of A, one could show that A simulates A_1 via $\mathcal{M}_1$, $P \wedge Q_2$, and Q_1, but a loss of abstraction results.) Lemma 2 shows that the safety properties of interest are still preserved.

Lemma 2: *Let I be an index set. If A simultaneously simulates $\{A_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$, P, and $\{Q_r : r \in I\}$, where $\mathcal{M}_r = (\mathcal{S}_r, \mathcal{A}_r)$ for all $r \in I$, then the following are true of any execution e of A.*

(1) $\mathcal{M}_r(e)$ is an execution of A_r, for all $r \in I$.

(2) $\bigwedge_{r \in I}(Q_r \circ \mathcal{S}_r) \wedge P$ is true in every state of e.

2.2 Liveness

The following notation is introduced to define the basic liveness notion, "equitable", and to verify that this definition has the desired properties.

We define an execution $e = s_0 \pi_1 s_1 \ldots$ of automaton A to satisfy $S \hookrightarrow (T, X)$, where S and T are subsets of *states*(A) and X is a subset of *states*$(A) \times$ *acts*(A), if for all i with $s_i \in S$, there is a $j \geq i$ such that either $s_j \in T$ or $(s_j, \pi_{j+1}) \in X$. In words, starting at any state of e, eventually either a state in T is reached, or a state-action pair in X is reached.

If $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ is an abstraction mapping from A to B, then for each locally-controlled action φ of B, we make the following definitions: E_φ is the set of all states s of A such that φ is enabled in $\mathcal{S}(s)$; D_φ is *states*$(A) - E_\varphi$; D'_φ is the set of all states t of B such that φ is not enabled in t; X_φ is the set of all pairs (s, π) of states s of A and actions π of A such that φ is in $\mathcal{A}(s, \pi)$; and X'_φ is *states*$(B) \times \{\varphi\}$.

Definition: Suppose $\mathcal{M}$ is an abstraction mapping from A to B. Let φ be a locally-controlled action of B. If every fair execution of A satisfies *states*$(A) \hookrightarrow (D_\varphi, X_\varphi)$, then A is *equitable* for φ via $\mathcal{M}$. If A is equitable for φ via $\mathcal{M}$ for every locally-controlled action φ of B, then A is *equitable* for B. $\square$

The next lemma motivates the equitable definition — in the induced execution of B, if φ is ever enabled, then eventually φ either occurs or becomes disabled.

Lemma 3: *Suppose A simulates B via $\mathcal{M}$. Let φ be a locally-controlled action of B. If A is equitable for φ via $\mathcal{M}$, then $\mathcal{M}(e)$ satisfies states(B) $\hookrightarrow (D'_\varphi, X'_\varphi)$, for every fair execution e of A.*

Proof: Let $\mathcal{M} = (\mathcal{S}, \mathcal{A})$. Let $e = s_0\pi_1 s_1 \ldots$ be a fair execution of A, and let $\mathcal{M}(e) = t_0\varphi_1 t_1 \ldots$. For any $i \geq 0$, define *index*(i) to be j such that $\mathcal{M}(s_0 \ldots s_i) = t_0 \ldots t_j$. Choose $i \geq 0$.

Case 1: t_i is not interpolated. Choose any l be such that *index*(l) $= i$. Then $t_i = \mathcal{S}(s_l)$, as argued in the proof of Lemma 1. Suppose there is an $m \geq l$ such that $s_m \in D_\varphi$. Then there is a $j =$ *index*(m) $\geq i$ such that $t_j = \mathcal{S}(s_m)$, and by definition of D_φ, t_j is in D'_φ. Suppose there is an $m \geq l$ such that $(s_m, \pi_{m+1}) \in X_\varphi$. Then there is a $j = index(m) \geq i$ such that $\varphi_j = \varphi$, by definition of X_φ, and (t_j, φ_{j+1}) is in X'_φ.

Case 2: t_i is interpolated. Let i' be the smallest integer greater than i such that $t_{i'}$ is not interpolated. If either a state in D'_φ or φ occurs between i and i' in $\mathcal{M}(e)$, then we are done. Suppose not. Then the argument in Case 1, applied to $t_{i'}$, shows that eventually after $t_{i'}$, and thus after t_i, either a state in D'_φ or φ occurs in $\mathcal{M}(e)$. $\square$

The next lemma is the analog of Lemma 3 for simultaneously simulates. (D'_φ and X'_φ are defined with respect to $\mathcal{M}_r$.)

Lemma 4: *Suppose A simultaneously simulates $\{A_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$. Let φ be a locally-controlled action of A_r for some r. If A is equitable for φ via $\mathcal{M}_r$, then $\mathcal{M}_r(e)$ satisfies states(B) $\hookrightarrow (D'_\varphi, X'_\varphi)$, for every fair execution e of A.*

The rest of this subsection describes three methods of verifying that A is equitable for action φ of B. Lemma 5 describes the first method, which is to identify an action of A that is essentially the "same" as φ.

Lemma 5: *Suppose $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ is an abstraction mapping from A to B, φ is a locally-controlled action of B, and ρ is a locally-controlled action of A such that, for all reachable states s of A,* *(1) ρ is enabled in s if and only if φ is enabled in state $\mathcal{S}(s)$ of B, and*

(2) if ρ is enabled in s, then φ is included in $\mathcal{A}(s, \rho)$.

Then A is equitable for φ via $\mathcal{M}$.

Proof: Let $e = s_0\pi_1s_1\ldots$ be a fair execution of A. Choose $i \geq 0$. If $s_i \in D_\varphi$, we are done. Suppose $s_i \in E_\varphi$. By assumption, ρ is enabled in s_i. Since e is fair, there exists $j > i$ such that either $\pi_j = \rho$, in which case $\mathcal{A}(s_{j-1}, \pi_j)$ includes φ, or else ρ is not enabled in s_j, in which case φ is not enabled in $\mathcal{S}(s_j)$. Thus, e satisfies $states(A) \hookrightarrow (D_\varphi, X_\varphi)$. □

The second method uses the following definition, which is shown in Lemma 6 to imply equitable.

Definition: Suppose $\mathcal{M} = (\mathcal{S}, \mathcal{A})$ is an abstraction mapping from A to B. If φ is a locally-controlled action of B, then we say A is ***progressive*** for φ via $\mathcal{M}$ if there is a set Ψ of pairs (s, ψ) of states s of A and locally-controlled actions ψ of A, and a function v from $states(A)$ to a well-founded set such that the following are true.

(1) For any reachable state $s \in E_\varphi$ of A, some action ψ is enabled in s such that (s, ψ) is in Ψ.

(2) For any step (s', π, s) of A, where s' is reachable and in E_φ, $(s', \pi) \notin X_\varphi$, and $s \in E_\varphi$,

(a) $v(s) \leq v(s')$,

(b) if $(s', \pi) \in \Psi$, then $v(s) < v(s')$, and

(c) if $(s', \pi) \notin \Psi$, ψ is enabled in s', and (s', ψ) is in Ψ, then ψ is enabled in s and (s, ψ) is in Ψ. □

Lemma 6: *If A is progressive for φ via $\mathcal{M}$, then A is equitable for φ via $\mathcal{M}$.*

Proof: Let $\mathcal{M} = (\mathcal{S}, \mathcal{A})$. By assumption, φ is a locally-controlled action of B, and there exist Ψ and v satisfying conditions (1) and (2) in the definition of "progressive".

Let $e = s_0\pi_1s_1\ldots$ be a fair execution of A. Choose $i \geq 0$. If $s_i \in D_\varphi$, we are done. Suppose $s_i \in E_\varphi$. Assume in contradiction that for all $j \geq i$, $(s_j, \pi_{j+1}) \notin X_\varphi$ and $s_j \in E_\varphi$. By condition (1), there is an action ψ enabled in s_i such that (s_i, ψ) is in Ψ. By condition (2c), as long as $(s_j, \pi_{j+1}) \notin \Psi$, ψ is enabled in s_{j+1} and $(s_{j+1}, \psi) \in \Psi$, for $j \geq i$. Since e is fair, there is $i_1 > i$ such that $(s_{i_1-1}, \pi_{i_1}) \in \Psi$.

By conditions (2a) and (2b), $v(s_{i_1}) < v(s_i)$. Similarly, we can show that there is $i_2 > i_1$ such that $v(s_{i_2}) < v(s_{i_1})$. We can continue this indefinitely, contradicting the range of v being a well-founded set. □

The next lemma demonstrates a third technique for showing that A is equitable for locally-controlled action φ of B, in a situation when there are multiple higher-level algorithms. The main idea is to show that there is some action ρ of D that is "similar" to φ (cf. conditions (2) and (3)) such that C is progressive for ρ using certain helping actions (cf. condition (4)), and A is equitable for all the helping actions for ρ (cf. condition (5)). By "similar", we mean that if φ is enabled in the B-image of state s of A, then ρ is enabled in the D-image of the C-image of s; and if ρ occurs in the D-image of the C-image of the pair (s', π), then φ occurs in the B-image of (s', π). Condition (1) is needed for technical reasons. (For convenience, we define abstraction function $\mathcal{M}$ applied to the empty sequence to be the empty sequence. To avoid ambiguity, we add the superscript AB to E_φ, D_φ, and X_φ when they are defined with respect to the abstraction function from A to B.)

Lemma 7: *Let A, B, C and D be automata such that $\mathcal{M}_{AB} = (\mathcal{S}_{AB}, \mathcal{A}_{AB})$ is an abstraction function from A to B, and similarly for $\mathcal{M}_{AC}$ and $\mathcal{M}_{CD}$. Let φ be a locally-controlled action of B. Suppose the following conditions are true.*

(1) $\mathcal{M}_{AC}(e)$ is an execution of C for every execution e of A.

(2) There is a locally-controlled action ρ of D such that for any reachable state s of A, if $s \in E_\varphi^{AB}$, then $\mathcal{S}_{AC}(s) \in E_\rho^{CD}$.

(3) If (s', π, s) is a step of A, s' is reachable, and ρ is in $\mathcal{M}_{CD}(\mathcal{M}_{AC}(s'\pi s))$, then φ is in $\mathcal{A}_{AB}(s', \pi)$.

(4) C is progressive for ρ via $\mathcal{M}_{CD}$, using the set Ψ_ρ and the function v_ρ.

(5) A is equitable for ψ via $\mathcal{M}_{AC}$, for all actions ψ of C such that $(t, \psi) \in \Psi_\rho$ for some state t of C.

Then A is equitable for φ via $\mathcal{M}_{AB}$.

Proof: Let $e = s_0\pi_1 s_1 \ldots$ be a fair execution of A. Let $\mathcal{M}_{AC}(e) = t_0\varphi_1 t_1 \ldots$. By assumption (1), t_m is a reachable state of C for all $m \geq 0$. For any $i \geq 0$, define *index*(i) to be m such that $\mathcal{M}_{AC}(s_0\pi_1 \ldots s_i) = t_0\varphi_1 \ldots t_m$.

Choose $i \geq 0$. If $s_i \in D_\varphi^{AB}$, we are done. Suppose $s_i \in E_\varphi^{AB}$. Assume in contradiction that for all $j \geq i$, $(s_j, \pi_{j+1}) \notin X_\varphi^{AB}$ and $s_j \in E_\varphi^{AB}$. Let $m = $ *index*(i).

By assumption (2), there is a locally-controlled action ρ of D such that $t_n \in E_\rho^{CD}$ for all $n \geq m$. By assumption (3), $(t_n, \varphi_{n+1}) \notin X_\rho^{CD}$ for all $n \geq m$.

By assumption (4), C is progressive for ρ via $\mathcal{M}_{CD}$, using set Ψ_ρ and function v_ρ. Thus, there is a locally-controlled action ψ of C enabled in $\mathcal{S}_{AC}(s_i) = t_m$ such that $(t_m, \psi) \in \Psi_\rho$. By assumption (5), A is equitable for ψ via $\mathcal{M}_{AC}$. Since e is fair and $s_i \in E_\psi^{AC}$, by Lemma 3 there exists $i_1 > i$ such that either $(s_{i_1-1}, \pi_{i_1}) \in X_\psi^{AC}$ or $s_{i_1} \in D_\psi^{AC}$. Let $m_1 = index(i_1)$.

Case 1: $(s_{i_1-1}, \pi_{i_1}) \in X_\psi^{AC}$. Then $\mathcal{A}_{AC}(s_{i_1-1}, \pi_{i_1})$ includes ψ. Since t_n is reachable, $t_n \in E_\rho^{CD}$, and $(t_n, \varphi_{n+1}) \notin X_\rho^{CD}$ for all $n \geq m$, we conclude that $v_\rho(t_{m_1}) < v_\rho(t_m)$, by parts (2a) and (2b) of the definition of "progressive".

Case 2: $s_{i_1} \in D_\psi^{AC}$. Since t_n is reachable, $t_n \in E_\rho^{CD}$, and $(t_n, \varphi_{n+1}) \notin X_\rho^{CD}$ for all $n \geq m$, by part (2c) of the definition of " progressive", the only way ψ can go from enabled in t_m to disabled in t_{m_1} is for some action in Ψ_ρ to occur between φ_{m+1} and φ_{m_1}. By part (2b) of the definition of "progressive", $v_\rho(t_{m_1}) < v_\rho(t_m)$.

Similarly, we can show that there exists $i_2 > i_1$ such that $v_\rho(\mathcal{S}_{AC}(s_{i_2})) < v_\rho(\mathcal{S}_{AC}(s_{i_1}))$. We can continue this indefinitely, contradicting the range of v_ρ being a well-founded set. □

2.3 Satisfaction

The next theorem shows that our definitions of simulate and equitable are sufficient for showing that A satisfies B.

Theorem 8: *If A simulates B via $\mathcal{M}$, P and Q and if A is equitable for B via $\mathcal{M}$, then A satisfies B.*

Proof: We must show that for any fair execution e of A, there is a fair execution f of B such that $sched(e)|ext(A) = sched(f)|ext(B)$. Given e, let f be $\mathcal{M}(e)$. We verify that $\mathcal{M}(e)$ is a fair execution of B with the desired property. Lemma 1, part (1), implies that f is an execution of B. Choose any locally-controlled action φ of B. By Lemma 3, if φ is enabled in any state of f, then subsequently in f, either a state occurs in which φ is not enabled, or φ occurs. Thus, f is fair. Finally, $sched(e)|ext(A) = sched(f)|ext(B)$ because of condition (2) in the definition of "simulates". □

The next theorem is the analog of Theorem 7 for simultaneously simulates.

Theorem 9: *Let I be an index set. If A simultaneously simulates $\{A_r : r \in I\}$ via $\{\mathcal{M}_r : r \in I\}$, P and $\{Q_r : r \in I\}$, and if A is equitable for A_r via $\mathcal{M}_r$ for some $r \in I$, then A satisfies A_r.*

3. Problem Statement

We define the minimum spanning tree problem as an external schedule module.

For the rest of this chapter, let G be a connected undirected graph, with at least two nodes and for each edge, a unique weight chosen from a totally ordered set. Nodes are $V(G)$ and edges are $E(G)$. For each edge (p,q) in $E(G)$, there are two *links* (i.e., directed edges), $\langle p,q\rangle$ and $\langle q,p\rangle$. The set of all links of G is denoted $L(G)$. The set of all links leaving p is denoted $L_p(G)$. The weight of (p,q) is denoted $wt(p,q)$; $wt(\langle p,q\rangle)$ is defined to be $wt(p,q)$; and $wt(nil)$ is defined to be ∞.

The following facts about minimum spanning trees will be useful.

Lemma 10: *(Property 2 in [GHS]) The minimum spanning tree of G is unique.*

Proof: Suppose in contradiction that T_1 and T_2 are both minimum spanning trees of G and $T_1 \neq T_2$. Let e be the minimum-weight edge that is in one of the trees but not both. Without loss of generality, suppose e is in $E(T_1)$. The set of edges $\{e\} \cup E(T_2)$ must contain a cycle, and at least one edge, say e', of this cycle is not in $E(T_1)$. Since $e \neq e'$ and e' is in one but not both of the trees, $wt(e) < wt(e')$. Thus replacing e' with e in $E(T_2)$ yields a spanning tree of G with smaller weight than T_2, contradicting the assumption. □

Let $T(G)$ be the (unique) minimum spanning tree of G.

An *external* edge (p,q) of subgraph F of G is an edge of G such that $p \in V(F)$ and $q \notin V(F)$.

Lemma 11: *(Property 1 in [GHS]) If F is a subgraph of $T(G)$, and e is the minimum-weight external edge of F, then e is in $T(G)$.*

Proof: Suppose in contradiction that e is not in $T(G)$. Then a cycle is formed by e together with some subset of the edges of $T(G)$. At least one other edge e' of this cycle is also an external edge of F. By choice of e, $wt(e) < wt(e')$. Thus, replacing e' with e in the edge set of $T(G)$ produces a spanning tree of G with smaller weight than $T(G)$, which is a contradiction. □

The $MST(G)$ problem is the following external schedule module. Input actions are $\{Start(p) : p \in V(G)\}$. Output actions are $\{InTree(l), NotInTree(l) : l \in L(G)\}$. Schedules are all sequences of actions such that

- no output action occurs unless an input action occurs;
- if an input action occurs, then exactly one output action occurs for each $l \in L(G)$;
- if $InTree(\langle p, q\rangle)$ occurs, then (p, q) is in $T(G)$; and
- if $NotInTree(\langle p, q\rangle)$ occurs, then (p, q) is not in $T(G)$.

4. Proof of Correctness

The verification of Gallager, Humblet and Spira's minimum-spanning tree algorithm [GHS] uses several automata, arranged into a lattice as in Figure 2.

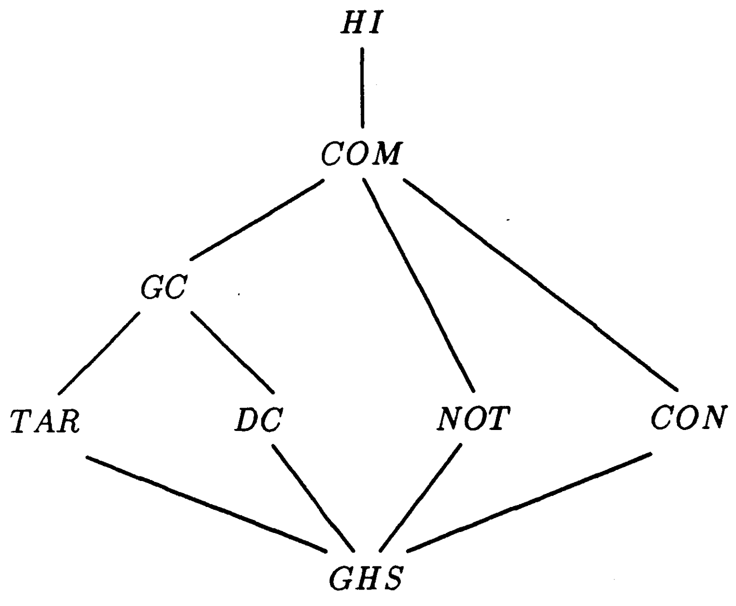

Figure 2: The Lattice

Each element of the lattice is a complete algorithm. However, the level of detail in which the actions and state of the original algorithm are represented varies. Working down the lattice takes us from a description of the algorithm that uses global information about the state of the graph, and powerful, atomic actions, to a fully distributed algorithm, in which each node can only access its local variables, and many actions are needed to implement a single higher level action. A brief overview of each algorithm is given below; a fuller description of each appears later.

HI is a very high-level description of the algorithm, and is easily shown in Section 4.1 to solve the $MST(G)$ problem. GHS is the detailed algorithm from [GHS]. We show a path in the lattice from GHS to HI, where each automaton in the path satisfies the automaton above it. By transitivity of satisfaction, then GHS will have been shown to solve $MST(G)$.

The essential feature of the state of HI is a set of subgraphs of G, initially the set of singleton nodes of G. Subgraphs combine, in a single action, along minimum-weight external edges, until only one subgraph, the minimum spanning tree, remains.

The COM automaton introduces *fragments*, each of which corresponds to a subgraph of HI, plus extra information about the global *level* and *core* (or identity) of the subgraph. Two ways to combine fragments are distinguished, *merging* and *absorbing*, and two milestones that a fragment must reach before combining are identified. The first milestone is computing the minimum-weight external link of the fragment, and the second is indicating readiness to combine.

The GC automaton expands on the process of finding the minimum-weight external link of a fragment, by introducing for each fragment a set *testset* of nodes that are participating in the search. Once a node has found its local minimum-weight external link, it is removed from the testset.

TAR and DC expand on GC in complementary ways. DC focuses on how the nodes of a fragment cooperate to find the minimum-weight external link of the whole fragment in a distributed fashion. It describes the flow of messages throughout the fragments: first a broadcast informs nodes that they should find their local minimum-weight external links, and then a convergecast reports the results back. In contrast, TAR is unconcerned with specifying exactly when each node finds its local minimum-weight external link, and concentrates on the details of the protocol performed by a node to find this link.

NOT is a refinement of COM that expands on the method by which the global level and core information for a fragment is implemented by variables local to each node. Messages attempt to notify nodes of the level and core of the nodes' current fragment.

CON, an orthogonal refinement of COM, concentrates on how messages are used to implement what happens between the time the minimum-weight external link of an entire fragment is computed, and the time the fragment is combined with another one.

Finally, the entire, fully distributed, algorithm is represented in automaton GHS. It expands on and unites TAR, DC, NOT and CON.

The path chosen through the lattice is HI, COM, GC, TAR, GHS. Why this path? Obviously, GHS must be shown to simulate one of TAR, DC, NOT and CON. However, it cannot be done in isolation; that is, invariants about the other three are necessary to show that GHS simulates one. (As mentioned in Section 2.1, the invariants about the other three could be made predicates about GHS, but this approach does not take advantage of abstraction.) Thus, we show that GHS simultaneously simulates those four automata. To show this, however, we need to verify that certain predicates really are invariants for the four. In order to do this, we show that TAR and DC (independently) simulate GC, and that NOT and CON (independently) simulate COM. Likewise, in order to show these facts, we need to know that certain predicates are invariants of GC and COM, and the way we do that is to show that GC simulates COM, and that COM simulates HI. Thus, it is necessary to show safety relationships along every edge in the lattice.

The liveness relationships only need to be shown along one path from GHS to HI. After inspecting GHS and the four automata directly above it, we decided on pragmatic grounds that it would be easiest to show that GHS is equitable for TAR. One consideration was that the output actions have exactly the same preconditions in GHS and in TAR, and thus showing GHS is equitable for those actions is trivial. Once TAR was chosen, the rest of the path was fixed.

First, the necessary safety properties are verified in Section 4.2. We show that COM simulates HI (Section 4.2.1), that GC simulates COM (Section 4.2.2), that TAR simulates GC (Section 4.2.3), that DC simulates GC (Section 4.2.4), that NOT simulates COM (Section 4.2.5), that CON simulates COM (Section 4.2.6), and that GHS simultaneously simulates TAR, DC, NOT and CON (Section 4.2.7).

Section 4.3 contains the liveness arguments. To show the desired chain of satisfaction, we show that COM is equitable for HI (Section 4.3.1), that GC is equitable for COM (Section 4.3.2), that TAR is equitable for GC (Section 4.3.3), and that GHS is equitable for TAR (Section 4.3.6). In Section 4.3.6, the technique of Lemma 7 is used in several places; thus we need to show that DC is progressive for an action of GC (Section 4.3.4), and that CON is progressive for several actions of COM (Section 4.3.5).

Section 4.4 puts the pieces together to show that GHS solves $MST(G)$.

4.1 HI Solves MST(G)

The main feature of the HI state is the data structure FST (for "forest"), which consists of a set of subgraphs of G, partitioning $V(G)$. The idea is that the subgraphs of G are connected subgraphs of the minimum spanning tree $T(G)$. Two subgraphs can combine if the minimum-weight external link of one leads to the other. The *awake* variable is used to make sure that no output action occurs unless an input action occurs. The *answered* variables are used to ensure that at most one output action occurs for each link. *InTree*($\langle p,q\rangle$) can only occur if $\langle p,q\rangle$ is already in a subgraph, or is the minimum-weight external edge of a subgraph (i.e., is destined to be in a subgraph). *NotInTree*($\langle p,q\rangle$) can only occur if p and q are in the same subgraph but the edge between them is not.

Define automaton HI (for "High Level") as follows.

The state consists of a set FST of subgraphs of G, a Boolean variable *answered*(l) for each $l \in L(G)$, and a Boolean variable *awake*.

In the start state of HI, FST is the set of single-node graphs, one for each $p \in V(G)$, every *answered*(l) is false, and *awake* is false.

Input actions:

- *Start*(p), $p \in V(G)$
    - Effects:
        - *awake* := true

Output actions:

- *InTree*($\langle p,q\rangle$), $\langle p,q\rangle \in L(G)$
    - Preconditions:
        - *awake* = true
        - $(p,q) \in F$ or (p,q) is the minimum-weight external edge of F, for some $F \in FST$
        - *answered*($\langle p,q\rangle$) = false
    - Effects:

```
        $answered(\langle p,q \rangle) :=$ true

• $NotInTree(\langle p,q \rangle)$, $\langle p,q \rangle \in L(G)$
    Preconditions:
        $awake =$ true
        $p, q \in F$ and $(p,q) \notin F$, for some $F \in FST$
        $answered(\langle p,q \rangle) =$ false
    Effects:
        $answered(\langle p,q \rangle) :=$ true
```

Internal actions:

```
• $Combine(F, F', e)$, $F, F' \in FST$, $e \in E(G)$
    Preconditions:
        $awake =$ true
        $F \neq F'$
        $e$ is an external edge of $F$
        $e$ is the minimum-weight external edge of $F'$
    Effects:
        $FST := FST - \{F, F'\} \cup \{F \cup F' \cup e\}$
```

Define the following predicates on $states(HI)$. (A *minimum spanning forest* of G is a set of disjoint subgraphs of G that span $V(G)$ and form a subgraph of a minimum spanning tree of G.)

- HI-A: Each F in FST is connected.
- HI-B: FST is a minimum spanning forest of G.

Let $P_{HI} =$ HI-A $\wedge$ HI-B. HI-B implies that the elements of FST form a partition of $V(G)$. Lemma 10 and HI-B imply that FST is a subgraph of $T(G)$.

Theorem 12: *HI solves the $MST(G)$ problem, and P_{HI} is true in every reachable state of HI.*

Proof: First we show that P_{HI} is true in every reachable state of HI. If s is a start state of HI, then P_{HI} is obviously true. Suppose (s', π, s) is a step of HI and P_{HI} is true in s'. If $\pi \neq Combine(F, F', e)$, then, since FST is unchanged, P_{HI} is obviously true in s as well.

Suppose $\pi = Combine(F, F', e)$. By the precondition, $F \neq F'$, e is the minimum-weight external edge of F', and e is an external edge of F in s'. By

HI-A, F and F' are each connected in s'; thus, the new fragment formed in s by joining F and F' along e is connected, and HI-A is true. Since by HI-B and Lemma 10, F and F' are subgraphs of $T(G)$, and since by Lemma 11 e is in $T(G)$, the new FST is a minimum spanning forest of G, and HI-B is true.

We now show that HI solves $MST(G)$. Let e be a fair execution of HI. The use of the variable *awake* ensures that no output action occurs in e unless an input action occurs in e. The use of the variables *answered*(l) ensures that at most one output action occurs in e for each link l. Suppose *InTree*$(\langle p,q\rangle)$ occurs in e. Then in the preceding state, either (p,q) is in F or (p,q) is the minimum-weight external edge of F, for some $F \in FST$. By HI-B and Lemmas 10 and 11, (p,q) is in $T(G)$. Suppose *NotInTree*$(\langle p,q\rangle)$ occurs in e. Then in the preceding state, p and q are in F and (p,q) is not in F, for some $F \in FST$. By HI-A, there is path from p to q in F. By HI-B and Lemma 10, this path is in $T(G)$. Thus (p,q) cannot be in $T(G)$, or else there would be a cycle.

Suppose an input action occurs in e. We show that an output action occurs in e for each link. Let $e = s_0\pi_1 s_1 \ldots$. Obviously, π_1 is an input action. Only a finite number of output actions can occur in e. Choose m such that π_m is the last output action occurring in e. (Let $m = 1$ if there is no output action in e.) It is easy to see that $s_m = s_i$ for all $i \geq m$. Since an input action occurs in e before s_m, *awake* = true in s_m. $|FST| = 1$ in s_m, because otherwise some *Combine*(F, F', e') action would be enabled in s_m, contradicting e being fair. Let $FST = \{F\}$. By HI-A and HI-B, $F = T(G)$ in s_m. Furthermore, *answered*(l) is true in s_m for each l, because otherwise some output action for l would be enabled in s_m, contradicting e being fair. Yet the only way *answered*(l) can be true in s_m is if an output action for l occurs in e. □

4.2 Safety

Each algorithm in the lattice below HI is presented in a separate subsection. Each subsection is organized as follows. First, an informal description of the algorithm is given, together with a discussion of any particularly interesting aspects. Then comes a description of the state of the automaton, both explicit variables, and derived variables (if any). A *derived* variable is a variable that is not an explicit element of the state, but is a function of the explicit variables. We employ the convention that whenever the definition of a derived variable is not unique or sensible, then the derived variable is undefined. The actions of the automaton are specified next. Then predicates to be shown invariant for this automaton are listed. The abstraction mapping to be used for simulating the higher-level automaton is defined next. All our state mappings conform to the rule that variables with the same name have the same value in all the algorithms. The only potential problem that might arise with this rule is if a derived variable is mapped to an explicit variable, but the derived variable is undefined. Although we will prove that this situation never occurs in states we are interested in, for completeness of the definition of state mapping one can simply choose some default value for the explicit variable. Often it is useful to derive some predicates about this automaton's state that follow from the invariant for this automaton and the higher-level one; these predicates are true of any state of this automaton satisfying the invariant and mapping to a reachable state of the higher-level algorithm. The proof of simulation completes the subsection.

4.2.1 COM Simulates HI

The *COM* algorithm still takes a completely global view of the algorithm, but some intermediate steps leading to combining are identified, and the state is expanded to include extra information about the subgraphs. The *COM* state consists of a set of *fragments*, a data structure used throughout the rest of the lattice. Each fragment f has associated with it a subgraph of G, as well as other information: *level*(f), *core*(f), *minlink*(f), and *rootchanged*(f). Two milestones must be reached before a fragment can combine. First, the *ComputeMin*(f) action causes the minimum-weight external link of fragment f to be identified as *minlink*(f), and second, the *ChangeRoot*(f) action indicates that fragment f is ready to combine, by setting the variable *rootchanged*(f). This automaton distinguishes two ways that fragments (and hence, their associated subgraphs) can combine. The *Merge*(f, g) action causes two fragments, f and g, at the same level with the same minimum-weight external edge, to combine; the new fragment has a higher level and a new core (i.e., identifying edge). The *Absorb*(f, g) action causes a fragment g to be engulfed by the fragment f at the other end of *minlink*(g), provided f is at a higher level than g.

Define automaton *COM* (for "Common") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;

- *level*(f), a nonnegative integer;
- *minlink*(f), a link of G or *nil*;
- *rootchanged*(f), a Boolean.

The state also contains Boolean variables, *answered*(l) one for each $l \in L(G)$, and Boolean variable *awake*.

In the start state of COM, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*(f) $= \{p\}$, *core*(f) $=$ *nil*, *level*(f) $= 0$, *minlink*(f) is the minimum-weight link adjacent to p, and *rootchanged*(f) is false. Each *answered*(l) is false and *awake* is false.

Two fragments will be considered the same if either they have the same single-node subtree, or they have the same nonnil core.

We define the following derived variables.

- For node p, *fragment*(p) is the element f of *fragments* such that p is in *subtree*(f).
- A link $\langle p, q\rangle$ is an *external* link of p and of *fragment*(p) if *fragment*(p) $\neq$ *fragment*(q); otherwise the link is *internal*.
- If *minlink*(f) $= \langle p, q\rangle$, then *minedge*(f) is the edge (p, q), *minnode*(f) $= p$, and *root*(f) is the endpoint of *core*(f) closest to p.
- If $\langle p, q\rangle$ is the minimum-weight external link of fragment f, then *mw-minnode*(f) $= p$ and *mw-root*(f) is the endpoint of *core*(f) closest to p.
- *subtree*(p) is all nodes and edges of *subtree*(*fragment*(p)) on the opposite side of p from *core*(*fragment*(p)).
- q is a ***child*** of p if $q \in$ *subtree*(p) and $(p, q) \in$ *subtree*(*fragment*(p)).

Input actions:

```
• Start(p), $p \in V(G)$
      Effects:
        awake := true
```

Output actions:

- $InTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$

  Preconditions:

  $awake =$ true

  $(p,q) \in subtree(fragment(p))$ or $\langle p,q\rangle = minlink(fragment(p))$

  $answered(\langle p,q\rangle) =$ false

  Effects:

  $answered(\langle p,q\rangle) :=$ true

- $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$

  Preconditions:

  $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$

  $answered(\langle p,q\rangle) =$ false

  Effects:

  $answered(\langle p,q\rangle) :=$ true

Internal actions:

- $ComputeMin(f)$, $f \in fragments$

  Preconditions:

  $minlink(f) = nil$

  l is the minimum-weight external link of f

  $level(f) \leq level(fragment(target(l)))$

  Effects:

  $minlink(f) := l$

- $ChangeRoot(f)$, $f \in fragments$

  Preconditions:

  $awake =$ true

  $rootchanged(f) =$ false

  $minlink(f) \neq nil$

  Effects:

  $rootchanged(f) :=$ true

- $Merge(f,g)$, $f,g \in fragments$

  Preconditions:

  $f \neq g$

  $rootchanged(f) = rootchanged(g) =$ true

  $minedge(f) = minedge(g)$

  Effects:

  add a new element h to $fragments$

```
        $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
        $core(h) := minedge(f)$
        $level(h) := level(f) + 1$
        $minlink(h) := nil$
        $rootchanged(h) :=$ false
        delete $f$ and $g$ from $fragments$
```

- $Absorb(f, g)$, $f, g \in fragments$

```
    Preconditions:
        $rootchanged(g) =$ true
        $level(g) < level(f)$
        $fragment(target(minlink(g))) = f$
    Effects:
        $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
        delete $g$ from $fragments$
```

Define the following predicates on states of COM. (All free variables are universally quantified.)

- COM-A: If $minlink(f) = l$, then l is the minimum-weight external link of f, and $level(f) \leq level(fragment(target(l)))$.
- COM-B: If $rootchanged(f) =$ true, then $minlink(f) \neq nil$.
- COM-C: If $awake =$ false, then $minlink(f) \neq nil$, $rootchanged(f) =$ false, and $subtree(f) = \{p\}$ for some p.
- COM-D: If $f \neq g$, then $subtree(f) \neq subtree(g)$.
- COM-E: If $subtree(f) = \{p\}$ for some p, then $minlink(f) \neq nil$.
- COM-F: If $|nodes(f)| = 1$, then $level(f) = 0$ and $core(f) = nil$; if $|nodes(f)| > 1$, then $level(f) > 0$ and $core(f) \in subtree(f)$.

Let P_{COM} be the conjunction of COM-A through COM-F.

In order to show that COM simulates HI, we define an abstraction mapping $\mathcal{M}_1 = (\mathcal{S}_1, \mathcal{A}_1)$ from COM to HI. Define the function $\mathcal{S}_1$ from $states(COM)$ to $states(HI)$ as follows. In conformance with our convention (cf. the beginning of Section 4.2), the values of $awake$ and $answered(l)$ (for all l) in $\mathcal{S}_1(s)$ are the same as in s. The value of FST in $\mathcal{S}_1(s)$ is the multiset $\{subtree(f) : f \in fragments\}$.

Define the function $\mathcal{A}_1$ as follows. Let s be a state of COM and π an action of COM enabled in s.

- If $\pi =$ *Start*(p), *InTree*(l), or *NotInTree*(l), then $\mathcal{A}_1(s,\pi) = \pi$.
- If $\pi =$ *ComputeMin*(f) or *ChangeRoot*(f), then $\mathcal{A}_1(s,\pi)$ is empty.
- If $\pi =$ *Merge*(f,g) or *Absorb*(f,g), then $\mathcal{A}_1(s,\pi) =$ *Combine*(F,F',e), where $F =$ *subtree*(f) in s, $F' =$ *subtree*(g) in s, and $e =$ *minedge*(g) in s.

The following predicate is true in every state of COM satisfying $(P_{HI} \circ \mathcal{S}_1) \wedge P_{COM}$. (I.e., it is deducible from P_{COM} and the HI predicates.)

- COM-G: The multiset $\{$*subtree*$(f) : f \in$ *fragments*$\}$ forms a partition of $V(G)$, and *fragment*(p) is well-defined.

Proof: Let s be a state of COM satisfying $(P_{HI} \circ \mathcal{S}_1) \wedge P_{COM}$. In $\mathcal{S}_1(s)$, $FST = \{$*subtree*$(f) : f \in$ *fragments*$\}$. By HI-B, FST forms a partition of $V(G)$. By COM-D, the multiset $\{$*subtree*$(f) : f \in$ *fragments*$\} = FST$, and thus it forms a partition of $V(G)$. Consequently, *fragment*(p) is well-defined. $\square$

Lemma 13: *COM simulates HI via $\mathcal{M}_1$, P_{COM}, and P_{HI}.*

Proof: By inspection, the types of COM, HI, $\mathcal{M}_1$ and P_{COM} are correct. By Theorem 12, P_{HI} is a predicate true in every reachable state of HI.

(1) Let s be in $start(COM)$. Obviously, P_{COM} is true in s, and $\mathcal{S}_1(s)$ is in $start(HI)$.

(2) Obviously, $\mathcal{A}_1(s,\pi)|ext(HI) = \pi|ext(COM)$ for any state s of A.

(3) Let (s',π,s) be a step of COM such that P_{HI} is true of $\mathcal{S}_1(s')$ and P_{COM} is true of s'. We consider each possible value of π.

i) π is Start(p), InTree(l), or NotInTree(l). $\mathcal{A}_1(s',\pi) = \pi$. Obviously, P_{COM} is true in s, and $\mathcal{S}_1(s')\pi\mathcal{S}_1(s)$ is an execution fragment of HI.

ii) π is ComputeMin(f) or ChangeRoot(f). $\mathcal{A}_1(s',\pi)$ is empty. Obviously, $\mathcal{S}_1(s') = \mathcal{S}_1(s)$. Obviously, COM-A, COM-B, COM-D and COM-F are true in s. By COM-C for *ComputeMin*(f) and by precondition for *ChangeRoot*(f), *awake* $=$ true in s', and also in s; thus, COM-C is true in s.

Obviously, COM-E is true in s for any fragment $f' \neq f$. If $\pi = \mathit{ComputeMin}(f)$, then $\mathit{minlink}(f) \neq \mathit{nil}$ in s, and COM-E is vacuously true in s for f. If $\pi = \mathit{ChangeRoot}(f)$, then by COM-B, $\mathit{minlink}(f) \neq \mathit{nil}$ in s' and also in s, so COM-E is vacuously true in s for f.

iii) π is Merge(f,g).

(3c) $\mathcal{A}_1(s', \pi) = \mathit{Combine}(F, F', e)$, where $F = \mathit{subtree}(f)$ in s', $F' = \mathit{subtree}(g)$ in s', and $e = \mathit{minedge}(g)$ in s', for some fragments f and g.

Claims about s':

1. $f \neq g$, by precondition.
2. $\mathit{rootchanged}(f) = \mathit{rootchanged}(g) =$ true, by precondition.
3. $\mathit{minedge}(f) = \mathit{minedge}(g)$, by precondition.
4. $\mathit{awake} =$ true, by Claim 2 and COM-C.
5. $\mathit{minedge}(f) \neq \mathit{nil}$ and $\mathit{minedge}(g) \neq \mathit{nil}$, by Claim 2 and COM-B
6. $\mathit{minlink}(f)$ is an external link of f, by COM-A and Claim 5.
7. $\mathit{minlink}(g)$ is the minimum-weight external link of g, by COM-A and Claim 5.

Let $F = \mathit{subtree}(f)$, $F' = \mathit{subtree}(g)$ and $e = \mathit{minedge}(g)$.

Claims about $\mathcal{S}_1(s')$: (All depend on the definition of $\mathcal{S}_1$.)

8. $\mathit{awake} =$ true, by Claim 4.
9. $F \neq F'$, by Claim 1 and COM-D.
10. e is an external edge of F, by Claims 3 and 6.
11. e is the minimum-weight external edge of F', by Claim 7.

By Claims 8 through 11, $\mathit{Combine}(F, F', e)$ is enabled in $\mathcal{S}_1(s')$. Obviously, its effects are mirrored in $\mathcal{S}_1(s)$.

(3a) ***More claims about s':***

12. $\mathit{level}(f) \geq 0$, by COM-F.
13. $\mathit{subtree}(f')$ and $\mathit{subtree}(g')$ are disjoint, for all $f' \neq g'$, by COM-G.

Claims about s:

14. $\mathit{subtree}(h) = \mathit{subtree}(f) \cup \mathit{subtree}(g) \cup \mathit{minedge}(f)$, by code.
15. $\mathit{core}(h) = \mathit{minedge}(f)$, by code.
16. $\mathit{level}(h) = \mathit{level}(f) + 1$, by code.

17. *minlink*$(h) = nil$, by code.
18. *rootchanged*$(h) =$ false, by code.
19. f and g are removed from *fragments*, by code.
20. *awake* = true, by Claim 4.
21. *subtree*(f') and *subtree*(g') are disjoint, for all $f' \neq g'$, by Claims 13, 14 and 19.
22. $|nodes(h)| > 1$, by Claim 14.
23. *level*$(h) > 1$, by Claims 12 and 16.
24. *core*$(h) \in$ *subtree*(h), by Claims 14 and 15.

COM-A is vacuously true for h by Claim 17. COM-B is vacuously true for h by Claim 18. COM-C is vacuously true by Claim 20. COM-D is true by Claim 21. COM-E is vacuously true for h by Claim 22. COM-F is true for h by Claims 22, 23 and 24.

iv) π is Absorb(f,g).

(3c) $\mathcal{A}_1(s', \pi) = Combine(F, F', e)$, where $F = subtree(f)$ in s', $F' = subtree(g)$ in s', and $e = minedge(g)$ in s', for some fragments f and g.

Claims about s':

1. *rootchanged*$(g) =$ true, by precondition.
2. *level*$(g) <$ *level*(f), by precondition.
3. *fragment*(*target*(*minlink*(g))) $= f$, by precondition.
4. $f \neq g$, by Claim 2.
5. *minlink*(g) is an external link of f, by Claims 3 and 4.
6. *minlink*$(g) \neq nil$, by Claim 3.
7. *minlink*(g) is the minimum-weight external link of g, by Claim 6 and COM-A.
8. *awake* = true, by Claim 1 and COM-C.

Let $F = subtree(f)$, $F' = subtree(g)$ and $e = minedge(g)$.

Claims about $\mathcal{S}_1(s')$: (All depend on the definition of $\mathcal{S}_1$.)

9. *awake* = true, by Claim 8.
10. $F \neq F'$, by Claim 4 and COM-D.
11. e is an external edge of F, by Claim 5.
12. e is the minimum-weight external edge of F', by Claim 7.

By Claims 9 through 12, $Combine(F, F', e)$ is enabled in $\mathcal{S}_1(s')$. Obviously, its effects are mirrored in $\mathcal{S}_1(s)$.

(3a) COM-A: If *minlink*$(f) = nil$ in s', then the same is true in s, and COM-A is vacuously true for f. Suppose *minlink*$(f) = l$ in s'. Let $f' = fragment(target(l))$.

More claims about s':

13. *level*$(f) \leq$ *level*(f'), by COM-A.
14. $f' \neq g$, by Claims 2 and 13.
15. *minedge*$(f) \neq$ *minedge*(g), by Claim 14.
16. *minlink*(f) is the minimum-weight external link of f, by COM-A.
17. If $e' \neq minedge(g)$ is an external edge of g, then $wt(e') > wt(minedge(f))$. Pf: $wt(e') > wt(minedge(g))$ by Claim 7, and $wt(minedge(g)) > wt(minedge(f))$ by Claims 5, 15 and 16.

Since *minlink*(f) is the same in s as in s', Claims 16 and 17 imply that in s, *minlink*(f) is the minimum-weight external link of f. The only fragment whose *level* changes in going from s' to s is g (since g disappears). Thus, Claim 14 implies that in s, *level*$(f) \leq$ *level*(f'). Finally, COM-A is true in s.

The next claims are used to verify COM-B through COM-F.

More claims about s':

18. *subtree*(f') and *subtree*(g') are disjoint, for all $f' \neq g'$, by COM-G.
19. *level*$(g) \geq 0$, by COM-F.
20. *level*$(f) > 0$, by Claims 2 and 19.
21. $|nodes(f)| > 1$, by Claim 20 and COM-F.
22. $core(f) \in subtree(f)$, by Claim 21 and COM-F.

Claims about s:

23. *awake* = true, by Claim 1.
24. *subtree*(f) in s is equal to *subtree*$(f) \cup$ *subtree*$(g) \cup$ *minedge*(g) in s', by code.
25. *subtree*(f') and *subtree*(g') are disjoint, for all $f' \neq g'$, by Claims 18 and 24.
26. $|nodes(f)| > 1$, by Claims 21 and 24.
27. *level*$(f) > 0$, by Claim 20.
28. $core(f) \in subtree(f)$, by Claims 22 and 24.

COM-B is unaffected. COM-C is vacuously true by Claim 23. COM-D is true by Claim 25. COM-E is vacuously true for f by Claim 26. COM-F is true for f by Claims 26, 27 and 28. □

Let $P'_{COM} = (P_{HI} \circ S_1) \wedge P_{COM}$.

Corollary 14: *P'_{COM} is true in every reachable state of COM.*

Proof: By Lemmas 1 and 13. □

4.2.2 GC Simulates COM

The GC automaton expands on the process of finding the minimum-weight external link of a fragment, by introducing for each fragment f a set *testset*(f) of nodes that are participating in the search. Once a node in f has found its minimum-weight external link, it is removed from *testset*(f). A new action, *TestNode*(p), is added, by which a node p atomically finds its minimum-weight external link — however, the fragment at the other end of the link cannot be at a lower level than p's fragment in order for this action to occur. The new variable *accmin*(f) (for "accumulated minlink") stores the link with the minimum weight over all links external to nodes of f no longer in *testset*(f). *ComputeMin*(f) cannot occur until *testset*(f) is empty. When an *Absorb*(f, g) action occurs, all the nodes formerly in g are added to *testset*(f) if and only if the target of *minlink*(g) is in *testset*(f). This version of the algorithm is still totally global in approach.

Define automaton GC (for "Global ComputeMin") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;
- *level*(f), a nonnegative integer;
- *minlink*(f), a link of G or *nil*;
- *rootchanged*(f), a Boolean;
- *testset*(f), a subset of $V(G)$; and
- *accmin*(f), a link of G or *nil*.

The state also contains Boolean variables, *answered*(l), one for each $l \in L(G)$, and Boolean variable *awake*.

In the start state of COM, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*(f) $= \{p\}$, *core*(f) $=$ *nil*, *level*(f) $=$

0, *minlink*(f) is the minimum-weight link adjacent to p, *rootchanged*(f) is false, *testset*(f) is empty, and *accmin*(f) is *nil*. Each *answered*(l) is false and *awake* is false.

Input actions:

```
• $Start(p)$, $p \in V(G)$
      Effects:
        $awake :=$ true
```

Output actions:

```
• $InTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
      Preconditions:
        $awake =$ true
        $(p,q) \in subtree(fragment(p))$ or $\langle p,q\rangle = minlink(fragment(p))$
        $answered(\langle p,q\rangle) =$ false
      Effects:
        $answered(\langle p,q\rangle) :=$ true

• $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
      Preconditions:
        $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$
        $answered(\langle p,q\rangle) =$ false
      Effects:
        $answered(\langle p,q\rangle) :=$ true
```

Internal actions:

```
• $TestNode(p)$, $p \in V(G)$
      Preconditions:
        — let $f = fragment(p)$ —
        $p \in testset(f)$
        if $\langle p,q\rangle$, the minimum-weight external link of $p$, exists
          then $level(f) \leq level(fragment(q))$
      Effects:
        $testset(f) := testset(f) - \{p\}$
        if $\langle p,q\rangle$, the minimum-weight external link of $p$, exists
                  and $wt(p,q) < wt(accmin(f))$
          then $accmin(f) := \langle p,q\rangle$
```

- $ComputeMin(f)$, $f \in fragments$

    Preconditions:

    $minlink(f) = nil$
    $accmin(f) \neq nil$
    $testset(f) = \emptyset$

    Effects:

    $minlink(f) := accmin(f)$
    $accmin(f) := nil$

- $ChangeRoot(f)$, $f \in fragments$

    Preconditions:

    $awake = \text{true}$
    $rootchanged(f) = \text{false}$
    $minlink(f) \neq nil$

    Effects:

    $rootchanged(f) := \text{true}$

- $Merge(f, g)$, $f, g \in fragments$

    Preconditions:

    $f \neq g$
    $rootchanged(f) = rootchanged(g) = \text{true}$
    $minedge(f) = minedge(g) \neq nil$

    Effects:

    add a new element h to $fragments$
    $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
    $core(h) := minedge(f)$
    $level(h) := level(f) + 1$
    $minlink(h) := nil$
    $rootchanged(h) := \text{false}$
    $testset(h) := nodes(h)$
    $accmin(h) := nil$
    delete f and g from $fragments$

- $Absorb(f, g)$, $f, g \in fragments$

    Preconditions:

    $rootchanged(g) = \text{true}$
    $level(g) < level(f)$
    — let $p = target(minlink(g))$ —
    $fragment(p) = f$

Effects:
  $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
  if $p \in testset(f)$ then $testset(f) := testset(f) \cup testset(g)$
  delete g from *fragments*

Define the following predicates on the states of GC. (All free variables are universally quantified.)

- GC-A: If $accmin(f) = \langle p,q \rangle$, then $\langle p,q \rangle$ is the minimum-weight external link of any node in $nodes(f) - testset(f)$, and $level(f) \leq level(fragment(q))$.
- GC-B: If there is an external link of f, if $minlink(f) = nil$, and if $testset(f) = \emptyset$, then $accmin(f) \neq nil$.
- GC-C: If $testset(f) \neq \emptyset$, then $minlink(f) = nil$.

Let $P_{GC} =$ GC-A $\wedge$ GC-B $\wedge$ GC-C.

In order to show that GC simulates COM, we define an abstraction mapping $\mathcal{M}_2 = (\mathcal{S}_2, \mathcal{A}_2)$ from GC to COM. Define the function $\mathcal{S}_2$ from $states(GC)$ to $states(COM)$ by simply ignoring the variables $accmin(f)$ and $testset(f)$ for all fragments f when going from a state of GC to a state of COM.

Define the function $\mathcal{A}_2$ as follows. Let s be a state of GC and π an action of GC enabled in s. If $\pi = TestNode(p)$, then $\mathcal{A}_2(s,\pi)$ is empty. Otherwise, $\mathcal{A}_2(s,\pi) = \pi$.

Recall that $P'_{COM} = (P_{HI} \circ \mathcal{S}_1) \wedge P_{COM}$. If $P'_{COM}(\mathcal{S}_2(s))$ is true, then the COM predicates are true in $\mathcal{S}_2(s)$, and the HI predicates are true in $\mathcal{S}_1(\mathcal{S}_2(s))$.

Lemma 15: *GC simulates COM via $\mathcal{M}_2$, P_{GC}, and P'_{COM}.*

Proof: By inspection, the types of GC, COM, $\mathcal{M}_2$, and P_{GC} are correct. By Corollary 14, P'_{COM} is a predicate true in every reachable state of COM.

(1) Let s be in $start(GC)$. Obviously, P_{GC} is true in s, and $\mathcal{S}_2(s)$ is in $start(COM)$.

(2) Obviously, $\mathcal{A}_2(s,\pi)|ext(COM) = \pi|ext(GC)$.

(3) Let (s',π,s) be a step of GC such that P'_{COM} is true of $\mathcal{S}_2(s')$ and P_{GC} is true of s'.

i) π is Start(p), InTree(l), NotInTree(l), or ChangeRoot(f). Obviously, $\mathcal{S}_2(s')\pi\mathcal{S}_2(s)$ is an execution fragment of COM, and P_{GC} is true in s.

ii) π is ComputeMin(f).

(3a) Obviously, P_{GC} is still true in s for any $f' \neq f$. GC-A is vacuously true for f in s, since $accmin(f)$ is set to nil. GC-B is vacuously true for f in s, since $minlink(f) \neq nil$. By COM-C, $awake$ = true in $\mathcal{S}_2(s')$ and thus in s'; the same is true in s, so GC-C(a) is true in s for f. GC-C(b) is vacuously true for f in s, since $testset(f) = \emptyset$.

(3c) $\mathcal{A}_2(s', \pi) = \pi$.

Claims about s':

1. $testset(f) = \emptyset$, by precondition.
2. $accmin(f) \neq nil$, by precondition.
3. $level(f) \leq level(fragment(target(accmin(f))))$, by Claim 2 and GC-A.
4. $accmin(f)$ is the minimum-weight external link of f, by Claim 2, GC-A, and Claim 1.
5. $level(f) \leq level(fragment(target(l)))$, where l is the minimum-weight external link of f, by Claims 3 and 4.

Using Claim 5, it is easy to see that $\mathcal{S}_2(s')\pi\mathcal{S}_2(s)$ is an execution fragment of COM.

iii) π is TestNode(p).

(3a) Obviously, P_{GC} is still true in s for any $f' \neq f$. Inspecting the code verifies that GC-A and GC-B are still true in s for f as well. By GC-C(b), $minlink(f) = nil$ in s'; GC-C is true for f in s because $minlink(f)$ is not changed.

(3b) $\mathcal{A}_2(s', \pi)$ is empty, and obviously $\mathcal{S}_2(s') = \mathcal{S}_2(s)$.

iv) π is Merge(f,g).

(3a) Obviously, P_{GC} is still true in s for any f' other than f and g. GC-A is vacuously true in s for h, since $accmin(h) = nil$. GC-B is vacuously true in s for h, since $testset(h) \neq \emptyset$. GC-C is true in s for h since $minlink(h) = nil$.

(3c) $\mathcal{A}_2(s', \pi) = \pi$. Obviously, $\mathcal{S}_2(s')\pi\mathcal{S}_2(s)$ is an execution fragment of COM.

v) π is Absorb(f,g).

(3a) Obviously, P_{GC} is still true in s for any f' other than f and g.

In going from s' to s, *testset*(f) is either empty in both or non-empty in both, *minlink*(f) remains the same, and the truth of the existence of an external link of f either stays true or goes from true to false. Thus GC-B and GC-C are true in s for f.

We now deal with GC-A. If *accmin*(f) = *nil* in s', then the same is true in s, so GC-A is vacuously true for f in s.

Assume *accmin*$(f) = \langle r,t\rangle$. Let *minlink*$(g) = \langle q,p\rangle$.

Claims about s':

1. *level*$(g) <$ *level*(f), by precondition.
2. *fragment*$(p) = f$, by precondition.
3. *level*$(f) \leq$ *level*(*fragment*(t)), by GC-A.
4. *fragment*$(t) \neq g$, by Claims 1 and 3.
5. $\langle q,p\rangle \neq \langle t,r\rangle$, by Claim 4 and COM-A.
6. $wt(q,p) < wt(l)$, for any $l \neq \langle q,p\rangle$ that is an external link of g, by COM-A.
7. If $p \notin$ *testset*(f), then $wt(r,t) < wt(q,p)$, by Claim 5 and GC-A.
8. If $p \notin$ *testset*(f), then $wt(r,t) < wt(l)$, for any l that is an external link of g, by Claims 6 and 7.

If $p \notin$ *testset*(f) in s', then any node $p' \in$ *nodes*(f) is not in *testset*(f) in s exactly if, in s', p' is either in *nodes*$(f)-$ *testset*(f) or in *nodes*(g). Claim 8 implies that in s, $\langle r,t\rangle$ is still the minimum-weight external link of any node in f that is not in *testset*(f).

If $p \in$ *testset*(f) in s', then any node $p' \in$ *nodes*(f) is not in *testset*(f) in s exactly if p' is in *nodes*$(f)-$ *testset*(f) in s'. Thus in s, $\langle r,t\rangle$ is still the minimum-weight external link of any node in f that is not in *testset*(f).

Since g is the only fragment whose *level* changes in going from s' to s, Claim 4 implies that *level*$(f) \leq$ *level*(*fragment*(t)) in s. Thus, since *accmin*$(f) = \langle r,t\rangle$ in s, GC-A is true in s for f.

(3c) $\mathcal{A}_2(s,\pi) = \pi$. Obviously $\mathcal{S}_2(s')\pi\mathcal{S}_2(s)$ is an execution fragment of *COM*. □

Let $P'_{GC} = (P'_{COM} \circ S_2) \wedge P_{GC}$.

Corollary 16: *P'_{GC} is true in every reachable state of GC.*

Proof: By Lemmas 1 and 15. □

4.2.3 TAR Simulates GC

This automaton expands on the method by which a node finds its local minimum-weight external link. Some local information is introduced in this version, in the form of node variables and messages. Three FIFO message queues are associated with each link $\langle p, q \rangle$: $tarqueue_p(\langle p, q \rangle)$, the outgoing queue local to p; $tarqueue_{pq}(\langle p, q \rangle)$, modelling the communication channel; and $tarqueue_q(\langle p, q \rangle)$, the incoming queue local to q. The action *ChannelSend*(l, m) transfers a message m from the outgoing local queue of link l to the communication channel of l; and the action *ChannelRecv*(l, m) transfers a message m from the communication channel of link l to the incoming local queue of l.

Each link l is classified by the variable *lstatus*(l) as branch, rejected, or unknown. Branch means the link will definitely be in the minimum spanning tree; rejected means it definitely will not be; and unknown means that the link's status is currently unknown. Initially, all the links are unknown.

The search for node p's minimum-weight external link is initiated by the action *SendTest*(p), which causes p to identify its minimum-weight unknown link as *testlink*(p), and to send a TEST message over its testlink together with information about the level and core (identity) of p's fragment. If the level of the recipient q's fragment is less than p's, the message is requeued at q, to be dealt with later (when q's level has increased sufficiently). Otherwise, a response is sent back. If the fragments are different, the response is an ACCEPT message, otherwise, it is a REJECT message. An optimization is that if q has already sent a TEST message over the same edge and is waiting for a response, and if p and q are in the same fragment, then q does not respond — the TEST message that q already sent will inform p that the edge (p, q) is not external.

When a REJECT message (or a TEST in the optimized case described above) is received, the recipient marks that link as rejected, if it is unknown. It is possible that the link is already marked as branch, in which case it should not be changed to rejected.

When a *ChangeRoot*(f) occurs, *minlink*(f) is marked as branch; when an *Absorb*(f, g) occurs, the reverse link of *minlink*(g) is marked as branch. As soon as a link l is classified as branch, the *InTree*(l) output action can occur; as soon as a link l is classified as rejected, the *NotInTree*(l) output action can occur.

The requeuing of a message is a delicate aspect of this (as well as the original) algorithm. When p receives a message that it is not yet ready to handle, it cannot simply block receiving any more messages on that link, but instead it must allow other messages to jump over that message, as the following example shows. Suppose p is in a fragment at level 3, q is in a fragment at level 4, p sends a TEST message to q with parameter 3, and before it is received, q sends a TEST message to p with parameter 4. When p receives q's TEST message, it is not ready to handle it. When q receives p's TEST message, it sends back an ACCEPT message. In order to prevent deadlock, p must be able to receive this ACCEPT message, even though it was sent after the TEST message. Thus, the correctness of the algorithm depends on a subtle interplay between FIFO behavior, and occasional, well-defined, exceptions to it.

The following scenario demonstrates the necessity of checking that *lstatus*(l) is unknown before changing it to rejected, when a TEST or REJECT is received. (The reason for the check, which also appears the full algorithm, is not explained in [GHS].) Suppose p is in fragment f with level 8 and core c, q is in fragment g with level 4 and core d, and $\langle q, p\rangle$ is the minimum-weight external link of g. First, q determines that $\langle q, p\rangle$ is its local minimum-weight external link. Then p sends a TEST$(8, c)$ message to p, which is requeued, since $8 > 4$. Eventually, *ComputeMin*(g) occurs, and *minlink*(g) is set equal to $\langle q, p\rangle$. Then *ChangeRoot*(g) occurs, and $\langle q, p\rangle$ is marked as branch. Then *Absorb*(f, g) occurs, and $\langle p, q\rangle$ is marked as branch. The next time that q tries to process p's TEST$(8, d)$ message, it succeeds, determines that $\langle q, p\rangle$ is not external, since d is the core of q's fragment, and sends REJECT to q. But q had better not change the classification of $\langle q, p\rangle$ from branch to rejected. Similarly, when p receives q's REJECT message, it had better not change the classification of $\langle p, q\rangle$ from branch to rejected.

Define automaton TAR (for "Test-Accept-Reject") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;

- *level*(f), a nonnegative integer;
- *minlink*(f), a link of G or *nil*;
- *rootchanged*(f), a Boolean; and
- *testset*(f), a subset of $V(G)$.

For each node p, there is a variable *testlink*(p), which is either a link of G or *nil*.

For each link $\langle p, q\rangle$, there are associated four variables:

- *lstatus*$(\langle p, q\rangle)$, which takes on the values "unknown", "branch" and "rejected";
- $tarqueue_p(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;
- $tarqueue_{pq}(\langle p, q\rangle)$, a FIFO queue of messages from p to q that are in the communication channel; and
- $tarqueue_q(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at q to be processed.

The set of possible messages M is $\{\text{TEST}(l, c) : l \geq 0, c \in E(G)\} \cup \{\text{ACCEPT}, \text{REJECT}\}$.

The state also contains Boolean variables, *answered*(l), one for each $l \in L(G)$, and Boolean variable *awake*.

In the start state of *TAR*, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*$(f) = \{p\}$, *core*$(f) = nil$, *level*$(f) = 0$, *minlink*(f) is the minimum-weight link adjacent to p, *rootchanged*(f) is false, and *testset*(f) is empty. For all p, *testlink*(p) is *nil*. For each link l, *lstatus*(l) = unknown. The message queues are empty. Each *answered*(l) is false and *awake* is false.

The derived variable *tarqueue*$(\langle p, q\rangle)$ is defined to be $tarqueue_p(\langle p, q\rangle) \,||\, tarqueue_{pq}(\langle p, q\rangle) \,||\, tarqueue_q(\langle p, q\rangle)$. [1]

The derived variable *accmin*(f) is defined as follows. If *minlink*$(f) \neq nil$, or if there is no external link of any $p \in nodes(f) - testset(f)$, then *accmin*$(f) = nil$.

[1] Given two FIFO queues q_1 and q_2, define $q_1||q_2$ to be the FIFO queue obtained by appending q_2 to the end of q_1. Obviously this operation is associative.

Otherwise, $accmin(f)$ is the minimum-weight external link of all $p \in nodes(f) - testset(f)$.

Input actions:

- $Start(p)$, $p \in V(G)$
  ```
  Effects:
    $awake$ := true
  ```

Output actions:

- $InTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
  ```
  Preconditions:
    $lstatus(\langle p,q\rangle) =$ branch
    $answered(\langle p,q\rangle) =$ false
  Effects:
    $answered(\langle p,q\rangle) :=$ true
  ```

- $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
  ```
  Preconditions:
    $lstatus(\langle p,q\rangle) =$ rejected
    $answered(\langle p,q\rangle) =$ false
  Effects:
    $answered(\langle p,q\rangle) :=$ true
  ```

Internal actions (and a procedure):

- $ChannelSend(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$
  ```
  Preconditions:
    $m$ at head of $tarqueue_p(\langle p,q\rangle)$
  Effects:
    dequeue($tarqueue_p(\langle p,q\rangle)$)
    enqueue($m$, $tarqueue_{pq}(\langle p,q\rangle)$)
  ```

- $ChannelRecv(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$
  ```
  Preconditions:
    $m$ at head of $tarqueue_{pq}(\langle p,q\rangle)$
  Effects:
    dequeue($tarqueue_{pq}(\langle p,q\rangle)$)
    enqueue($m$, $tarqueue_q(\langle p,q\rangle)$)
  ```

```
• $SendTest(p)$, $p \in V(G)$
      Preconditions:
        $p \in testset(fragment(p))$
        $testlink(p) = nil$
      Effects:
        execute procedure $Test(p)$

• Procedure $Test(p)$, $p \in V(G)$
      — let $f = fragment(p)$ —
      if $l$, the minimum-weight link of $p$ with $lstatus(l)$ = unknown, exists then [
        $testlink(p) := l$
        enqueue($\text{TEST}(level(f), core(f)), tarqueue_p(l)$) ]
      else [
        remove $p$ from $testset(f)$
        $testlink(p) := nil$ ]

• $ReceiveTest(\langle q,p\rangle, l, c)$, $\langle p,q\rangle \in L(G)$
      Preconditions:
        $\text{TEST}(l,c)$ at head of $tarqueue_p(\langle q,p\rangle)$
      Effects:
        dequeue($tarqueue_p(\langle q,p\rangle)$)
        if $l > level(fragment(p))$ then
          enqueue($\text{TEST}(l,c), tarqueue_p(\langle q,p\rangle)$)
        else
          if $c \neq core(fragment(p))$ then
            enqueue($\text{ACCEPT}, tarqueue_p(\langle p,q\rangle)$)
          else [
            if $lstatus(\langle p,q\rangle)$ = unknown then $lstatus(\langle p,q\rangle)$ := rejected
            if $testlink(p) \neq \langle p,q\rangle$ then
              enqueue($\text{REJECT}, tarqueue_p(\langle p,q\rangle)$)
            else execute procedure $Test(p)$ ]

• $ReceiveAccept(\langle q,p\rangle)$, $\langle q,p\rangle \in L(G)$
      Preconditions:
        ACCEPT at head of $tarqueue_p(\langle q,p\rangle)$
      Effects:
        dequeue($tarqueue_p(\langle q,p\rangle)$)
        $testlink(p) := nil$
        remove $p$ from $testset(fragment(p))$
```

```
• $ReceiveReject(\langle q,p\rangle)$, $\langle q,p\rangle \in L(G)$
    Preconditions:
        REJECT at head of $tarqueue_p(\langle q,p\rangle)$
    Effects:
        dequeue($tarqueue_p(\langle q,p\rangle)$)
        if $lstatus(\langle p,q\rangle)$ = unknown then $lstatus(\langle p,q\rangle)$ := rejected
        execute procedure $Test(p)$

• $ComputeMin(f)$, $f \in fragments$
    Preconditions:
        $minlink(f) = nil$
        $accmin(f) \neq nil$
        $testset(f) = \emptyset$
    Effects:
        $minlink(f) := accmin(f)$

• $ChangeRoot(f)$, $f \in fragments$
    Preconditions:
        $awake$ = true
        $rootchanged(f)$ = false
        $minlink(f) \neq nil$
    Effects:
        $rootchanged(f)$ := true
        $lstatus(minlink(f))$ := branch

• $Merge(f,g)$, $f,g \in fragments$
    Preconditions:
        $f \neq g$
        $rootchanged(f) = rootchanged(g)$ = true
        $minedge(f) = minedge(g)$
    Effects:
        add a new element $h$ to $fragments$
        $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
        $core(h) := minedge(f)$
        $level(h) := level(f) + 1$
        $minlink(h) := nil$
        $rootchanged(h)$ := false
        $testset(h) := nodes(h)$
        delete $f$ and $g$ from $fragments$
```

- $Absorb(f, g)$, $f, g \in fragments$

```
    Preconditions:
        $rootchanged(g) = \text{true}$
        $level(g) < level(f)$
        — let $\langle q,p \rangle = minlink(g)$ —
        $fragment(p) = f$
    Effects:
        $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
        if $p \in testset(f)$ then $testset(f) := testset(f) \cup nodes(g)$
        $lstatus(\langle p,q \rangle) := \text{branch}$
        delete $g$ from $fragments$
```

A message m is defined to be a ***protocol message*** for link $\langle p, q\rangle$ in a state if m is one of the following:
(a) a TEST message in $tarqueue(\langle p,q\rangle)$ with $lstatus(\langle p,q\rangle) \neq$ rejected.
(b) an ACCEPT message in $tarqueue(\langle q,p\rangle)$
(c) a REJECT message in $tarqueue(\langle q,p\rangle)$
(d) a TEST message in $tarqueue(\langle q,p\rangle)$ with $lstatus(\langle q,p\rangle) =$ rejected.
A protocol message for $\langle p, q\rangle$ can be considered a message that is actively helping p to discover whether $\langle p, q\rangle$ is external.

Define the following predicates on states of TAR. (All free variables are universally quantified.)

- TAR-A:

  (a) If $lstatus(\langle p,q\rangle) =$ branch, then either $(p,q) \in subtree(fragment(p))$ or $minlink\ (fragment(p)) = \langle p,q\rangle$.

  (b) If $(p,q) \in subtree(fragment(p))$, then $lstatus(\langle p,q\rangle) = lstatus(\langle q,p\rangle) =$ branch.

- TAR-B: If $lstatus(\langle p,q\rangle) =$ rejected, then $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$.

- TAR-C: If $testlink(p) \neq nil$, then

  (a) $testlink(p) = \langle p,q\rangle$ for some q;

  (b) $p \in testset(fragment(p))$;

  (c) there is exactly one protocol message for $\langle p,q\rangle$;

  (d) if $lstatus(\langle p,q\rangle) \neq$ branch, then $\langle p,q\rangle$ is the minimum-weight link of p with $lstatus$ unknown;

(e) if $lstatus(\langle p,q\rangle)$ = branch, then $lstatus(\langle q,p\rangle)$ = branch and $testlink(q) \neq \langle q,p\rangle$.

- TAR-D: If there is a protocol message for $\langle p,q\rangle$, then $testlink(p) = \langle p,q\rangle$.

- TAR-E: If $\text{TEST}(l,c)$ is in $tarqueue(\langle p,q\rangle)$ then
  (a) $\langle p,q\rangle \neq core(fragment(p))$;
  (b) if $lstatus(\langle p,q\rangle) \neq$ rejected, then $c = core(fragment(p))$ and $l = level(fragment(p))$; and
  (c) if $lstatus(\langle p,q\rangle)$ = rejected, then $c = core(fragment(q))$ and $l = level(fragment(q))$.

- TAR-F: If ACCEPT is in $tarqueue(\langle p,q\rangle)$, then $fragment(p) \neq fragment(q)$ and $level\ (fragment(p)) \geq level(fragment(q))$.

- TAR-G: If REJECT is in $tarqueue(\langle p,q\rangle)$, then $fragment(p) = fragment(q)$ and $lstatus\ (\langle p,q\rangle) \neq$ unknown.

- TAR-H: $rootchanged(f)$ is true if and only if $lstatus(minlink(f))$ = branch.

- TAR-I: If $p \notin testset(fragment(p))$, then either no $\langle p,q\rangle$ has $lstatus(\langle p,q\rangle)$ = unknown, or else there is an external link $\langle r,t\rangle$ of $fragment(p)$ with $level(fragment(p)) \leq level(fragment(t))$.

- TAR-J: If $awake$ = false, then $lstatus(\langle p,q\rangle)$ = unknown.

Let P_{TAR} be the conjunction of TAR-A through TAR-J.

In order to show that TAR simulates GC, we define an abstraction mapping $\mathcal{M}_3 = (\mathcal{S}_3, \mathcal{A}_3)$ from TAR to GC. Define the function $\mathcal{S}_3$ from $states(TAR)$ to $states(GC)$ by ignoring the message queues, and the *testlink* and *lstatus* variables. The derived variables *accmin* of TAR map to the (non-derived) variables *accmin* of GC. Define the function $\mathcal{A}_3$ as follows. Let s be a state of TAR and π an action of TAR enabled in s. The GC action $TestNode(p)$ is simulated in TAR when p receives the message that tells p either that this link is external or that p has no external links.

- If $\pi = ReceiveAccept(\langle q,p\rangle)$, then $\mathcal{A}_3(s,\pi) = TestNode(p)$.

- If $\pi = SendTest(p)$ or $ReceiveReject(\langle q,p\rangle)$, then $\mathcal{A}_3(s,\pi) = TestNode(p)$ if there is no link $\langle p,r\rangle$, $r \neq q$, with $lstatus(\langle p,r\rangle)$ = unknown in s; otherwise, $\mathcal{A}_3(s,\pi)$ is empty.

- If $\pi = \mathit{ReceiveTest}(\langle q,p\rangle, l, c)$, then $\mathcal{A}_3(s,\pi) = \mathit{TestNode}(p)$ if $l \leq \mathit{level}(\mathit{fragment}(p))$, $c = \mathit{core}(\mathit{fragment}(p))$, $\mathit{testlink}(p) = \langle p,q\rangle$, and there is no link $\langle p,r\rangle$, $r \neq q$, with $\mathit{lstatus}(\langle p,r\rangle) =$ unknown in s; otherwise, $\mathcal{A}_3(s,\pi)$ is empty.

- If $\pi = \mathit{ChannelSend}(\langle p,q\rangle, m)$ or $\mathit{ChannelRecv}(\langle p,q\rangle, m)$, then $\mathcal{A}_3(s,\pi)$ is empty.

- For all other values of π, $\mathcal{A}_3(s,\pi) = \pi$.

The following predicates are true in every state of TAR satisfying $(P'_{GC} \circ \mathcal{S}_3) \wedge P_{TAR}$. Recall that $P'_{GC} = (P'_{COM} \circ \mathcal{S}_2) \wedge P_{GC}$. If $P'_{GC}(\mathcal{S}_3(s))$ is true, then the GC predicates are true in $\mathcal{S}_3(s)$, the COM predicates are true in $\mathcal{S}_2(\mathcal{S}_3(s))$, and the HI predicates are true in $\mathcal{S}_1(\mathcal{S}_2(\mathcal{S}_3(s)))$. Thus, these predicates are derivable from P_{TAR}, together with the HI, COM and GC predicates.

- TAR-K: If $\mathit{testlink}(p) = \langle p,q\rangle$, then $\mathit{lstatus}(\langle p,q\rangle) \neq$ rejected.

Proof: By TAR-C(d) and TAR-C(e).

- TAR-L: If $\mathit{minlink}(f) = \mathit{nil}$ and l is an external link of f, then $\mathit{lstatus}(l) =$ unknown.

Proof: By TAR-A(a), if $\mathit{lstatus}(l) =$ branch, then l is internal. By TAR-B, if $\mathit{lstatus}(l) =$ rejected, then l is internal. □

- TAR-M: If $\text{TEST}(l,c)$ is in $\mathit{tarqueue}(\langle p,q\rangle)$, then $l \geq 1$ and $c \neq \mathit{nil}$.

Proof: Let $f = \mathit{fragment}(p)$ and $g = \mathit{fragment}(q)$.
1. $\text{TEST}(l,c)$ is in $\mathit{tarqueue}(\langle p,q\rangle)$, by assumption.

Case 1: $\mathit{lstatus}(\langle p,q\rangle) \neq$ rejected.
2. $\mathit{lstatus}(\langle p,q\rangle) \neq$ rejected, by assumption.
3. $c = \mathit{core}(f)$ and $l = \mathit{level}(f)$, by Claim 2 and TAR-E(b).
4. $\mathit{testlink}(p) = \langle p,q\rangle$, by Claims 1 and 2 and TAR-D.
5. $p \in \mathit{testset}(f)$, by Claim 4 and TAR-C(b).
6. $\mathit{minlink}(f) = \mathit{nil}$, by Claim 5 and GC-C.
7. $\mathit{subtree}(f) \neq \{p\}$, by Claim 6 and COM-E.
8. $\mathit{core}(f) \neq \mathit{nil}$ and $\mathit{level}(f) \neq 0$, by Claim 7 and COM-F.
9. $\mathit{level}(f) \geq 1$, by Claim 8 and COM-F.
10. $c \neq \mathit{nil}$ and $l \geq 1$, by Claims 3, 8 and 9.

Case 2: $\mathit{lstatus}(\langle p,q\rangle) =$ rejected.

11. $lstatus(\langle p,q\rangle)$ = rejected, by assumption.
12. $c = core(g)$ and $l = level(g)$, by Claim 11 and TAR-E(c).
13. $testlink(q) = \langle q,p\rangle$, by Claims 1 and 11 and TAR-D.
14. $q \in testset(g)$, by Claim 13 and TAR-C(b).
15. $minlink(g) = nil$, by Claim 14 and GC-C.
16. $subtree(g) \neq \{q\}$, by Claim 15 and COM-E.
17. $core(g) \neq nil$ and $level(g) \neq 0$, by Claim 16 and COM-F
18. $level(g) \geq 1$, by Claim 17 and COM-F.
19. $c \neq nil$ and $l \geq 1$, by Claims 12, 17 and 18. □

- TAR-N: If $\text{TEST}(l,c)$ is in $tarqueue(\langle q,p\rangle)$ and $c = core(fragment(p))$, then $fragment(p) = fragment(q)$.

Proof:
1. $\text{TEST}(l,c)$ is in $tarqueue(\langle q,p\rangle)$, by assumption.
2. $c = core(fragment(p))$, by assumption.
3. $c \neq nil$, by Claim 1 and TAR-M.
4. If $lstatus(\langle q,p\rangle) \neq$ rejected, then $c = core(fragment(q))$, by TAR-E(b).
5. If $lstatus(\langle q,p\rangle) \neq$ rejected, then $fragment(q) = fragment(p)$, by Claims 2, 3 and 4, and COM-F.
6. If $lstatus(\langle q,p\rangle)$ = rejected, then $fragment(q) = fragment(p)$, by TAR-B. □

- TAR-O: If $minlink(f) \neq nil$, then there is no protocol message for any link of any node in $nodes(f)$.

Proof:
1. $minlink(f) \neq nil$, by assumption.
2. $testset(f) = \emptyset$, by Claim 1 and GC-C.
3. $testlink(p) = nil$ for all $p \in nodes(f)$, by Claim 2 and TAR-C(b).
4. There is no protocol message for any link $\langle p,q\rangle$, $p \in nodes(f)$, by Claim 3 and TAR-D. □

- TAR-P: If $\text{TEST}(l,c)$ is in $tarqueue(\langle q,p\rangle)$, $c = core(fragment(p))$, $testlink(p) = \langle p,q\rangle$, and $lstatus(\langle q,p\rangle) \neq$ rejected, then a $\text{TEST}(l',c')$ message is in $tarqueue(\langle p,q\rangle)$ and $lstatus(\langle p,q\rangle)$ = unknown.

Proof:
1. $\text{TEST}(l,c)$ is in $tarqueue(\langle q,p\rangle)$, by assumption.
2. $c = core(fragment(p))$, by assumption.
3. $testlink(p) = \langle p,q\rangle$, by assumption.

4. $\mathit{lstatus}(\langle q,p\rangle) \neq$ rejected, by assumption.
5. $\mathit{fragment}(p) = \mathit{fragment}(q)$, by Claims 1 and 2 and TAR-N.
6. No ACCEPT message is in $\mathit{tarqueue}(\langle q,p\rangle)$, by Claim 5 and TAR-F.
7. The TEST(l,c) message in $\mathit{tarqueue}(\langle q,p\rangle)$ is a protocol message for $\langle q,p\rangle$, by Claim 4.
8. $\mathit{testlink}(q) = \langle q,p\rangle$, by Claim 7 and TAR-D.
9. $\mathit{lstatus}(\langle q,p\rangle) \neq$ branch, by Claims 3, 8 and TAR-C(e).
10. $\mathit{lstatus}(\langle q,p\rangle) =$ unknown, by Claims 4 and 9.
11. No REJECT message is in $\mathit{tarqueue}(\langle q,p\rangle)$, by Claim 10 and TAR-G.
12. There is exactly one protocol message for $\langle p,q\rangle$, by Claim 3 and TAR-C(c).
13. A TEST(l',c') message is in $\mathit{tarqueue}(\langle p,q\rangle)$ and $\mathit{lstatus}(\langle p,q\rangle) \neq$ rejected, by Claims 6, 7, 11 and 12.
14. $\mathit{lstatus}(\langle p,q\rangle) \neq$ branch, by Claims 3 and 8 and TAR-C(e).
15. $\mathit{lstatus}(\langle p,q\rangle) =$ unknown, by Claims 13 and 14.

Claims 13 and 15 give the result. □

Lemma 17: *TAR simulates GC via $\mathcal{M}_3$, P_{TAR}, and P'_{GC}.*

Proof: By inspection, the types of TAR, GC, $\mathcal{M}_3$, and P_{TAR} are correct. By Corollary 16, P'_{GC} is a predicate true in every reachable state of COM.

(1) Let s be in $start(TAR)$. Obviously, P_{TAR} is true in s, and $\mathcal{S}_3(s)$ is in $start(GC)$.

(2) Obviously, $\mathcal{A}_3(s,\pi)|ext(GC) = \pi|ext(TAR)$.

(3) Let (s',π,s) be a step of TAR such that P'_{GC} is true of $\mathcal{S}_3(s')$ and P_{TAR} is true of s'. Condition (3a) is only shown below for those predicates that are not obviously true in s.

i) π is ChannelSend($\langle$p,q$\rangle$,m) or ChannelRecv($\langle$p,q$\rangle$,m). $\mathcal{A}_3(s',\pi)$ is empty. (3a) and (3b) are obviously true.

ii) π is Start(p) or InTree(l) or NotInTree(l).

(3c) $\mathcal{A}_3(s',\pi) = \pi$. If $\pi = \mathit{InTree}(l)$, then by TAR-J and TAR-A(a), π is enabled in $\mathcal{S}_3(s')$. If $\pi = \mathit{NotInTree}(l)$, then by TAR-J and TAR-B, π is enabled in $\mathcal{S}_3(s')$. Thus, $\mathcal{S}_3(s')\pi\mathcal{S}_3(s)$ is an execution fragment of GC.

(3a) Obviously, P_{TAR} is still true in s.

iii) π is SendTest(p). Let $f = \mathit{fragment}(p)$ in s'.

Case 1: There is a link $\langle p,q\rangle$ with $\mathit{lstatus}(\langle p,q\rangle) =$ unknown in s'.

(3b) $\mathcal{A}_3(s',\pi)$ is empty. It is easy to see that $\mathcal{S}_3(s') = \mathcal{S}_3(s)$.

(3a) By TAR-D and precondition that $\mathit{testlink}(p) = \mathit{nil}$, there is no protocol message for any link of p in s'.

TAR-C(c): In s, there is exactly one protocol message for $\langle p,q\rangle$, namely the TEST message in $\mathit{tarqueue}(\langle p,q\rangle)$.

TAR-D: The TEST message added in s is a protocol message for $\langle p,q\rangle$, and is not a protocol message for any other link. By the code, $\mathit{testlink}(p) = \langle p,q\rangle$.

TAR-E(a): By TAR-A(b), $(p,q) \notin \mathit{subtree}(f)$. By COM-F, $(p,q) \neq \mathit{core}(f)$.

Case 2: There is no link $\langle p,q\rangle$ with $\mathit{lstatus}(\langle p,q\rangle) =$ unknown in s'.

(3c) $\mathcal{A}_3(s',\pi) = \mathit{TestNode}(p)$.

Claims about s':

1. $p \in \mathit{testset}(f)$, by precondition.
2. $\mathit{minlink}(f) = \mathit{nil}$, by Claim 1 and GC-C.
3. There is no external link of p, by Claim 2, TAR-L, and assumption.

By Claims 1 and 3, $\mathit{TestNode}(p)$ is enabled in $\mathcal{S}_3(s')$.

Claims about s:

4. $p \notin \mathit{testset}(f)$, by code.
5. There is no external link of p, by Claim 3 and code.
6. $\mathit{accmin}(f)$ does not change, by Claim 5.

By Claims 4, 5, and 6, the effects of $\mathit{TestNode}(p)$ are mirrored in $\mathcal{S}_3(s)$.

(3a) TAR-I: By assumption for Case 2, p has no unknown links in s', and the same is true in s.

iv) π is ReceiveTest($\langle$q,p$\rangle$,l,c). Let $f = \mathit{fragment}(p)$ in s'.

Case 1: $l \leq level(f)$, $c = core(f)$, $testlink(p) = \langle p, q\rangle$, and there is no link $\langle p, r\rangle$, $r \neq q$, with $lstatus(\langle p, r\rangle) =$ unknown in s'.

(3c) $\mathcal{A}_3(s', \pi) = TestNode(p)$.

Claims about s':

1. $c = core(f)$, by assumption.
2. $testlink(p) = \langle p, q\rangle$, by assumption.
3. There is no link $\langle p, r\rangle$, $r \neq q$, with $lstatus(\langle p, r\rangle) =$ unknown, by assumption.
4. $\text{TEST}(l, c)$ is in $tarqueue(\langle q, p\rangle)$, by preconditions.
5. $p \in testset(f)$, by Claim 2 and TAR-C(b).
6. $minlink(f) = nil$, by Claim 5 and GC-C.
7. No link $\langle p, r\rangle$, $r \neq q$, is external, by Claims 6 and 3 and TAR-L.
8. $\langle p, q\rangle$ is not external, by Claims 2, 3 and 4 and TAR-N.

By Claims 5, 7 and 8, $TestNode(p)$ is enabled in s'.

Claims about s:

9. $p \notin testset(f)$, by code.
10. There is no external link of p, by Claims 7 and 8 and code.
11. $accmin(f)$ does not change, by Claim 10.

By Claims 9, 10 and 11, the effects of $TestNode(p)$ are mirrored in s.

(3a) TAR-B: The only case of interest is when $lstatus(\langle p, q\rangle)$ changes from unknown in s' to rejected in s. By TAR-N, $f = fragment(q)$ in s' and the same is still true in s. By TAR-A(b), $(p, q) \notin subtree(f)$ in s', and the same is still true in s.

TAR-D:

Claims about s':

1. $\text{TEST}(l, c)$ is in $tarqueue(\langle q, p\rangle)$, by precondition.
2. $c = core(f)$, by assumption.
3. $testlink(p) = \langle p, q\rangle$, by assumption.
4. There is exactly one protocol message for $\langle p, q\rangle$, by Claim 3 and TAR-C(c).
5. There is no protocol message for any link $\langle p, r\rangle$, $r \neq q$, by Claim 3 and TAR-D.

Case A: $lstatus(\langle q, p\rangle) =$ rejected. The $\text{TEST}(l, c)$ message in $tarqueue(\langle q, p\rangle)$ is the protocol message for $\langle p, q\rangle$ in s'. Since it is removed in s, by Claims 4 and 5 there is no protocol message for any link of p in s. Concerning q: by TAR-K,

$testlink(q) \neq \langle q,p \rangle$; thus, the predicate is still true for q in s, even if $lstatus(\langle p,q \rangle)$ is changed to rejected.

Case B: $lstatus(\langle q,p \rangle) \neq$ rejected.

6. A $\text{TEST}(l', c')$ is in $tarqueue(\langle p,q \rangle)$ and $lstatus(\langle p,q \rangle) =$ unknown, by Claims 1, 2, 3, assumptions for Case B, and TAR-P.
7. $testlink(q) = \langle q,p \rangle$, by Claim 1, assumption for Case B and TAR-D.

In s, the $\text{TEST}(l', c')$ message in $tarqueue(\langle p,q \rangle)$, which exists by Claim 6, becomes a protocol message for $\langle q,p \rangle$, since $lstatus(\langle p,q \rangle)$ is changed to rejected. By Claim 7, $testlink(q)$ has the correct value. By Claims 4 and 5, the predicate is vacuously true for p in s.

TAR-E(c): The only case of interest is when $lstatus(\langle p,q \rangle)$ goes from unknown in s' to rejected in s, while there is a $\text{TEST}(l', c')$ message in $tarqueue(\langle p,q \rangle)$. By TAR-E(b), $c' = core(f)$ and $l' = level(f)$ in s'. By TAR-N, $fragment(q) = f$. Thus $c' = core(fragment(q))$ and $l' = level(fragment(q))$.

TAR-I: By the assumption for Case 1 and code, p has no unknown links in s.

TAR-J: The TEST message in $tarqueue(\langle q,p \rangle)$ is a protocol message for either $\langle p,q \rangle$ or $\langle q,p \rangle$. Without loss of generality, suppose for $\langle p,q \rangle$. By TAR-D, $testlink(p) = \langle p,q \rangle$, and by TAR-C(b), $p \in testset(f)$. Thus, by GC-C, $minlink(f) = nil$, and by COM-C $awake =$ true.

Case 2: $l > level(f)$, or $c \neq core(f)$, or $testlink(p) \neq \langle p,q \rangle$, or there is a link $\langle p,r \rangle$, $r \neq q$, with $lstatus(\langle p,r \rangle) =$ unknown in s'.

(3b) $\mathcal{A}_3(s', \pi)$ is empty. The only variables that are possibly changed are $lstatus(\langle p,q \rangle)$, $tarqueue$'s, and $testlink(p)$, none of which is reflected (directly) in the state of GC. Thus $accmin(f)$ does not change and $\mathcal{S}_3(s') = \mathcal{S}_3(s)$.

(3a) TAR-B: As in Case 1.

TAR-C(b): If $testlink(p) \neq nil$ in s, then by inspecting the code, the same is true in s'. So the predicate is true in s because it is true in s'.

TAR-C(c): If $l > level(f)$ in s', nothing affecting the predicate changes in going from s' to s. Suppose $l \leq level(f)$ in s'.

Claims about s':

1. TEST(l, c) is in *tarqueue*$(\langle q,p\rangle)$, by precondition.

Case A: $c \neq core(f)$.

2. *lstatus*$(\langle q,p\rangle) \neq$ rejected, by TAR-E(c).
3. The TEST(l, c) message in *tarqueue*$(\langle q,p\rangle)$ is a protocol message for $\langle q,p\rangle$, by Claim 2.

The ACCEPT message added in s is a protocol message for $\langle q,p\rangle$. There is no change that affects the truth of the predicate for p.

Case B: $c = core(f)$.

Case B.1: testlink$(p) \neq \langle p,q\rangle$.

4. There is no protocol message for $\langle p,q\rangle$, by TAR-D.
5. The TEST(l, c) message in *tarqueue*$(\langle q,p\rangle)$ is a protocol message for $\langle q,p\rangle$, by Claim 4.

The REJECT message added in s is a protocol message for $\langle q,p\rangle$. No change affects the truth of the predicate for p.

Case B.2: testlink$(p) = \langle p,q\rangle$.

6. There is a link $\langle p,r\rangle$, $r \neq q$, with *lstatus*$(\langle p,r\rangle) =$ unknown, by assumption for Case B.2.
7. There is no protocol message for $\langle p,r\rangle$, by Claim 6 and TAR-D.

Case B.2.1: lstatus$(\langle q,p\rangle) \neq$ rejected.

8. There is a TEST(l', c') message in *tarqueue*$(\langle p,q\rangle)$ and *lstatus*$(\langle p,q\rangle) =$ unknown, by assumptions for Case B.2.1 and TAR-P.
9. The TEST(l, c) message in *tarqueue*$(\langle q,p\rangle)$ is a protocol message for $\langle q,p\rangle$, by assumptions for Case B.2.1.

The TEST(l', c') message of Claim 8 becomes a protocol message for $\langle q,p\rangle$ in s, since *lstatus*$(\langle p,q\rangle)$ is changed to rejected. Concerning p: *testlink*$(p) = \langle p,r\rangle$ in s, and a TEST message is added to *tarqueue*$(\langle p,r\rangle)$ and is the sole protocol message for $\langle p,r\rangle$ by Claim 7.

Case B.2.2 lstatus$(\langle q,p\rangle) =$ rejected.

10. The TEST(l, c) message in *tarqueue*$(\langle q,p\rangle)$ is the protocol message for $\langle p, q\rangle$, by assumptions for Case B.2.2.
11. *testlink*$(q) \neq \langle q,p\rangle$, by assumption for Case B.2.2 and TAR-K.

The predicate is true for p in s because the TEST(l, c) message, which was the sole protocol message for $\langle p, q\rangle$ by Claim 10, is removed in s; *testlink*(p) is now $\langle p, r\rangle$, and $\langle p, r\rangle$ has exactly one protocol message, by inspecting the code. No change is made that affects the truth of the predicate for q, by Claim 11.

TAR-D: If $l > level(f)$ in s', nothing affecting the predicate changes in going from s' to s. Suppose $l \leq level(f)$ in s'.

Claims about s':

1. TEST(l, c) is in *tarqueue*$(\langle q,p\rangle)$, by precondition.

Case A: $c \neq core(f)$.

2. *lstatus*$(\langle q,p\rangle) \neq$ rejected, by assumption for Case A and TAR-E(c).
3. *testlink*$(q) = \langle q,p\rangle$, by Claims 1 and 2 and TAR-D.

Then *testlink*(q) is still $\langle q,p\rangle$ in s, and there is an ACCEPT message in *tarqueue*$(\langle p,q\rangle)$. No change affects the truth of the predicate for p.

Case B: $c = core(f)$.

Case B.1: testlink$(p) \neq \langle p,q\rangle$.

4. The TEST(l, c) message in *tarqueue*$(\langle q,p\rangle)$ is a protocol message for $\langle q,p\rangle$, by assumptions for Case B.1 and TAR-D.
5. *testlink*$(q) = \langle q,p\rangle$, by Claim 4 and TAR-D.

Then in s, there is a REJECT message in *tarqueue*$(\langle p,q\rangle)$ and *testlink*(q) is still $\langle q,p\rangle$. No change affects the truth of the predicate for p.

Case B.2: testlink$(p) = \langle p,q\rangle$.

6. There is a link $\langle p,r\rangle$, $r \neq q$, with *lstatus*$(\langle p,r\rangle) =$ unknown, by assumption for Case 2.
7. There is exactly one protocol message for $\langle p,q\rangle$, by TAR-C(c).

Case B.2.1: lstatus$(\langle q,p\rangle) =$ rejected.

8. $testlink(q) \neq \langle q, p\rangle$, by TAR-K.

No changes affect the truth of the predicate for q. For p: The TEST(l, c) message in $tarqueue(\langle q, p\rangle)$ is the protocol message for $\langle p, q\rangle$. It is removed in s. A TEST message is added to $tarqueue(\langle p, r\rangle)$ in s, where $lstatus(\langle p, r\rangle)$ = unknown, and $testlink(p) = \langle p, r\rangle$ by code.

Case B.2.2: $lstatus(\langle q, p\rangle) \neq$ rejected.

9. A TEST(l', c') message is in $tarqueue(\langle p, q\rangle)$ and $lstatus(\langle p, q\rangle)$ = unknown, by Claim 1, the assumption for Case B.2.2 and TAR-P.
10. $testlink(q) = \langle q, p\rangle$, by Claim 8 and TAR-D.

For q: In s, since $lstatus(\langle q, p\rangle)$ is changed to rejected, the TEST(l', c') message in $tarqueue(\langle p, q\rangle)$ (of Claim 9) becomes a protocol message for $\langle q, p\rangle$. This is OK by Claim 10.

For p: The TEST(l', c') message of Claim 9 is the protocol message for $\langle p, q\rangle$. The rest of the argument is as in Case B.2.1.

TAR-E: (a) Suppose a TEST message is added to $tarqueue(\langle p, r\rangle)$. As in $\pi = SendTest(p)$, Case 1. (c) As in Case 1.

TAR-F: The only case of interest is when an ACCEPT message is added to $tarqueue(\langle p, q\rangle)$ in s.

Claims about s':

1. TEST(l, c) is in $tarqueue(\langle q, p\rangle)$, by precondition.
2. $l \leq level(f)$, by assumption.
3. $c \neq core(f)$, by assumption.
4. $lstatus(\langle q, p\rangle) \neq$ rejected, by Claims 1 and 3 and TAR-E(c).
5. $c = core(fragment(q))$, by Claims 1, 4 and TAR-E(b).
6. $l = level(fragment(q))$, by Claims 1, 4 and TAR-E(b).
7. $core(f) \neq core(fragment(q))$, by Claims 3 and 5.
8. $level(f) \leq level(fragment(q))$, by Claims 2 and 6.

Claims 7 and 8 are still true in s.

TAR-G: The only case of interest is when a REJECT message is added to $tarqueue(\langle p, q\rangle)$.

Claims about s':

1. $\text{TEST}(l, c)$ is in $tarqueue(\langle q,p\rangle)$, by precondition.
2. $c = core(f)$, by assumption.
3. $testlink(p) \neq \langle p, q\rangle$, by assumption.
4. If $lstatus(\langle q,p\rangle) \neq$ rejected, then $c = core(fragment(q))$, by Claim 1 and TAR-E(b).
5. If $lstatus(\langle q,p\rangle) \neq$ rejected, then $f = fragment(q)$, by Claim 4 and COM-F.
6. If $lstatus(\langle q,p\rangle) =$ rejected, then $f = fragment(q)$, by TAR-B.
7. $f = fragment(q)$, by Claims 5 and 6.

Claim 7 is still true in s.

TAR-I: The only case of interest is when p is removed from $testset(f)$. But when that happens, there are no unknown links of p.

TAR-J: Suppose $lstatus(\langle p,q\rangle)$ is changed to rejected. As in Case 1.

v) π is ReceiveAccept($\langle$q,p$\rangle$). Let $f = fragment(p)$ in s'.

(3c) $\mathcal{A}_3(s', \pi) = TestNode(p)$.

Claims about s':

1. ACCEPT is in $tarqueue(\langle q,p\rangle)$, by precondition.
2. $fragment(q) \neq f$, by Claim 1 and TAR-F.
3. $level(f) \leq level(fragment(q))$, by Claim 1 and TAR-F.
4. $\langle p,q\rangle$ is an external link of f, by Claim 2.
5. $testlink(p) = \langle p,q\rangle$, by Claim 1 and TAR-D.
6. $p \in testset(f)$, by Claim 5 and TAR-C(b).
7. $minlink(f) = nil$, by Claim 6 and GC-C.
8. $lstatus(\langle p,q\rangle) \neq$ branch, by Claims 4 and 7 and TAR-L.
9. $\langle p,q\rangle$ is the minimum-weight link of p with $lstatus$ unknown, by Claims 5 and 8 and TAR-C(d).
10. $\langle p,q\rangle$ is the minimum-weight external link of p, by Claims 7 and 9 and TAR-L.

By Claims 6, 10, and 3, $TestNode(p)$ is enabled in s'.

Claims about s:

11. $p \notin testset(f)$, by code.
12. $\langle p,q\rangle$ is the minimum-weight external link of p, by Claim 10.

13. If $wt(p,q) < wt(accmin(f))$ in s', then $accmin(f) = \langle p,q \rangle$ in s, by Claims 11 and 12.

By Claims 11 and 13, the effects of *TestNode*(p) are mirrored in s.

(3a) TAR-D: In s', ACCEPT in $tarqueue(\langle q,p \rangle)$ is a protocol message for $\langle p,q \rangle$. By TAR-C(c) and TAR-D, it is the only protocol message for any link of p in s'. Thus in s, there is no protocol message for any link of p, and the predicate is vacuously true in s for p. No other node is affected.

TAR-I: By Claims 3 and 4, it is OK to remove p from $testset(f)$.

vi) π is ReceiveReject($\langle$q,p$\rangle$). Let $f = fragment(p)$ in s'.

Case 1: There is a link $\langle p,r \rangle$, $r \neq q$, with $lstatus(\langle p,r \rangle) =$ unknown.

(3b) $\mathcal{A}_3(s',\pi)$ is empty. Obviously $\mathcal{S}_3(s') = \mathcal{S}_3(s)$.

(3a) *Claims about s':*

1. REJECT is in $tarqueue(\langle q,p \rangle)$, by assumption.
2. The REJECT in $tarqueue(\langle q,p \rangle)$ is a protocol message for $\langle p,q \rangle$, by Claim 1.
3. $testlink(p) = \langle p,q \rangle$, by Claim 2 and TAR-D.
4. There is only one protocol message for $\langle p,q \rangle$, by Claim 3 and TAR-C(c).
5. There is no protocol message for any other link of p, by Claim 3 and TAR-D.
6. $p \in testset(f)$, by Claim 3 and TAR-C(b).

TAR-B: Suppose $lstatus(\langle p,q \rangle)$ goes from unknown in s' to rejected in s. By TAR-G, $f = fragment(q)$ in s'. By TAR-A(b), $(p,q) \notin subtree(f)$ in s'. Both facts are still true in s.

TAR-C(b): By Claim 6.

TAR-C(c): In s, $testlink(p) = \langle p,r \rangle$, and the TEST message is the sole protocol message for $\langle p,r \rangle$ by Claim 5.

TAR-D: In s, the REJECT message is removed and a TEST message is added to $tarqueue(\langle p,r \rangle)$ with $lstatus(\langle p,r \rangle) =$ unknown. So there is a protocol message for $\langle p,r \rangle$ and no other link of p by Claims 4 and 5. By code, $testlink(p) = \langle p,r \rangle$.

TAR-E(a): Suppose a TEST messge is added to some $tarqueue(\langle p,r\rangle)$. As in $\pi = SendTest(p)$, Case 1.

TAR-E(c): The only case of interest is when $lstatus(\langle p,q\rangle)$ goes from unknown in s' to rejected in s. But by Claims 2 and 4, there is no TEST message in $tarqueue(\langle p,q\rangle)$ in s' if $lstatus(\langle p,q\rangle) =$ unknown.

TAR-I: By Claim 6, the predicate is vacuously true.

TAR-J: Suppose $lstatus(\langle p,q\rangle)$ is changed from unknown to rejected. Similar to $\pi = ReceiveTest(\langle q,p\rangle, l, c)$, Case 1, with REJECT being the protocol message for $\langle p,q\rangle$.

Case 2: There is no link $\langle p,r\rangle$, $r \neq q$, with $lstatus(\langle p,r\rangle) =$ unknown.

(3c) $\mathcal{A}_3(s',\pi) = TestNode(p)$.

Claims about s':

1. REJECT is in $tarqueue(\langle q,p\rangle)$, by precondition.
2. $testlink(p) = \langle p,q\rangle$, by Claim 1 and TAR-D.
3. $p \in testset(f)$, by Claim 2 and TAR-C(b)
4. $minlink(f) = nil$, by Claim 3 and GC-C.
5. $fragment(q) = f$, by Claim 1 and TAR-G.
6. $\langle p,q\rangle$ is not external, by Claim 5.
7. There is no external link $\langle p,r\rangle$, $r \neq q$, of p, by Claim 4, TAR-L, and assumption for Case 2.

By Claims 3, 6 and 7, $TestNode(p)$ is enabled in s'.

Claims about s:

8. $p \notin testset(f)$, by code.
9. There is no external link of p, by Claims 6 and 7 and code.
10. $accmin(f)$ does not change, by Claim 9.

By Claims 8, 9 and 10, the effects of $TestNode(p)$ are mirrored in s.

(3a) TAR-B: Same as Case 1.

TAR-D: In s, *testlink*$(p) = nil$. We must show there is no protocol message for any link of p. In s', the REJECT message in *tarqueue*$(\langle q,p\rangle)$ is the sole protocol message for any link of p, as in Case 1. The REJECT message is removed in s and no protocol message is added.

TAR-E(c): As in Case 1.

TAR-I: By assumption for Case 2 and code, there are no unknown links of p in s.

TAR-J: As in Case 1.

vii) π is ComputeMin(f).

(3c) $\mathcal{A}_3(s',\pi) = \pi$. Since *accmin*$(f) = nil$ in s because *minlink*$(f) = nil$ in s, it is easy to see that π is enabled in $\mathcal{S}_3(s')$ and that its effects are mirrored in $\mathcal{S}_3(s)$.

(3a) TAR-H: By GC-A, *accmin*$(f) = l$ is an external link of f in s'. Since *minlink*$(f) = nil$ in s', *lstatus*$(l) \neq$ branch by TAR-A(a). Also, by COM-B, *rootchanged*$(f) =$ false in s'. Thus in s, *rootchanged*$(f) =$ false and *lstatus*(*minlink*$(f)) \neq$ branch.

viii) π is ChangeRoot(f).

(3c) $\mathcal{A}_3(s',\pi) = \pi$. It is easy to see that π is enabled in $\mathcal{S}_3(s')$ and that its effects are mirrored in $\mathcal{S}_3(s)$.

(3a) Only TAR-A(a), TAR-H and TAR-J are affected. Obviously TAR-A(a) and TAR-H are still true in s. For TAR-J: by precondition *awake* = true in s', and is still true in s.

ix) π is Merge(f,g).

(3c) $\mathcal{A}_3(s',\pi) = \pi$. After noting that *accmin*$(h) = nil$ in s because *testset*$(h) =$ *nodes*(h) in s, it is easy to see that π is enabled in $\mathcal{S}_3(s')$ and that its effects are mirrored in $\mathcal{S}_3(s)$.

(3a) TAR-A(b): The predicate is true for h by TAR-H.

TAR-B: The predicate is true for h by TAR-H.

TAR-C: By GC-C, no r in *nodes*(f) or *nodes*(g) is in *testset*(f) or *testset*(g) in s'. By TAR-C(b), *testlink*$(r) = nil$ for all such r. So the predicate is vacuously true in h.

TAR-E(a): By TAR-O, there is no TEST message in $tarqueue(\langle p,q\rangle)$ or in $tarqueue(\langle q,p\rangle)$, where $\langle p,q\rangle = minlink(f)$, in s'. Since $(p,q) = core(h)$ in s, done.

TAR-E(b): By TAR-O, there is no $\text{TEST}(l,c)$ message in $tarqueue(\langle p,q\rangle)$ with $lstatus(\langle p,q\rangle) \neq$ rejected in s', for any p in $nodes(f)$ or $nodes(g)$. Thus, the same is true in s for any p in $nodes(h)$, and the predicate is vacuously true in s for h.

TAR-E(c): If $\text{TEST}(l,c)$ is in $tarqueue(\langle p,q\rangle)$ and $lstatus(\langle p,q\rangle) =$ rejected in s', then it is a protocol message for $\langle q,p\rangle$ in s'. By TAR-O, $fragment(q)$ is neither f nor g in s'. So the predicate is still true in s.

TAR-F: If ACCEPT is in $tarqueue(\langle p,q\rangle)$ in s', it is a protocol message for $\langle q,p\rangle$ in s'. By TAR-O, $fragment(q)$ is neither f nor g in s'. If $fragment(p)$ is neither f nor g in s', then the predicate is still true in s. Without loss of generality, suppose $fragment(p) = f$ in s'. By TAR-F, $level(f) \geq level(fragment(q))$ in s'. Then $fragment(p) = h \neq fragment(q)$ in s, and $level(h)$ (in s) $>$ $level(f)$ (in s') $\geq$ $level(fragment(q))$ (in s' and s).

TAR-H: By code, $rootchanged(h) =$ false. Since $minlink(h) = nil$ by code, $lstatus\ (minlink(f)) \neq$ branch.

TAR-I: For nodes in h, the predicate is vacuously true since $testset(h) = nodes(h)$. For nodes not in h, the predicate is still true since the level of every node formerly in $nodes(f)$ or $nodes(g)$ is increased.

x) π is Absorb(f,g).

(3c) $\mathcal{A}_3(s',\pi) = \pi$. It is easy to see that π is enabled in $\mathcal{S}_3(s')$. Below we show that $accmin(f)$ is the same in s as in s', which together with inspecting the code, shows that the effects of π are mirrored in $\mathcal{S}_3(s)$.

Let $\langle q,p\rangle = minlink(g)$. If $p \in testset(f)$ in s', then every node in $nodes(g)$ in s' is added to $testset(f)$ in s. No change is made to any of the criteria for defining $accmin(f)$.

Suppose $p \notin testset(f)$ in s'. If $minlink(f) \neq nil$ in s', then the same is true in s, and $accmin(f) = nil$ in s' and s. Suppose $minlink(f) = nil$ in s'.

Claims about s':

1. $level(f) < level(g)$, by precondition.
2. $p \in nodes(f)$, by precondition.

3. $p \notin testset(f)$, by assumption.
4. $minlink(f) = nil$, by assumption.
5. $q \in nodes(g)$, by COM-A.
6. $f \neq g$, by Claim 1.
7. $accmin(f) = \langle r, t\rangle$, for some r and t, by Claims 2 through 6.
8. $fragment(t) \neq g$, by Claims 1 and 7 and GC-A.
9. $\langle r, t\rangle \neq \langle p, q\rangle$, by Claims 5 and 8.
10. $wt(r, t) < wt(p, q)$, by Claims 2, 3, 5, 6, 7, and 9 and GC-A.
11. $wt(p, q) \leq wt(u, v)$ for any external link $\langle u, v\rangle$ of g, by COM-A.
12. $wt(r, t) < wt(u, v)$ for any external link $\langle u, v\rangle$ of g, by Claims 10 and 11.

By Claims 7, 8 and 12, $accmin(f) = \langle r, t\rangle$ in s.

(3a) TAR-A(b): The predicate is true in s for f by TAR-H.

TAR-B: The predicate is true in s for f by TAR-H.

TAR-C(b): By GC-C, since $minlink(g) \neq nil$, $testset(g) = \emptyset$ in s'. By TAR-C(b), $testlink(p) = nil$ in s' for all $p \in nodes(g)$. There is no change for $p \in nodes(f)$ in s' in going from s' to s. Thus the predicate is true in s for f.

TAR-C(e): Suppose $\langle q, p\rangle = minlink(g)$ in s' and $lstatus(\langle p, q\rangle)$ becomes branch in s. By TAR-H, $lstatus(\langle q, p\rangle) =$ branch in s'. As in TAR-C(b), $testlink(q) \neq \langle q, p\rangle$, so the predicate is still true in s.

TAR-E(a): OK because $core(f)$ does not change.

TAR-E(b): Let $\langle q, p\rangle = minlink(g)$ in s'. If we can show $lstatus(\langle p, q\rangle) \neq$ rejected in s', we'd be done. If $lstatus(\langle p, q\rangle) =$ rejected in s', then $fragment(p) = fragment(q)$. This contradicts $level(g) < level(f)$, which implies that $g \neq f$.

TAR-E(c): Suppose TEST(l, c) is in $tarqueue(\langle p, q\rangle)$ and $lstatus(\langle p, q\rangle) =$ rejected in s', for some link $\langle p, q\rangle$ in $L(G)$. This is a protocol message for $\langle q, p\rangle$. By TAR-O, $fragment(q) \neq g$ in s'. Thus $fragment(q)$ is the same in s' and s, and $c = core(fragment(q))$ and $l = level(fragment(q))$ in s.

TAR-F: Suppose ACCEPT is in $tarqueue(\langle p, q\rangle)$ in s', for some link $\langle p, q\rangle$ in $L(G)$. This is a protocol message for $\langle q, p\rangle$. By TAR-O, $fragment(q) \neq g$ in s'. By TAR-F, $fragment(p) \neq fragment(q)$ in s'. By preconditions, $level(g) < level(f)$, so it cannot be the case that $fragment(p) = g$ and $fragment(q) = f$.

Suppose $fragment(p) = g$. Since $level(fragment(p))$ in s is greater than it is in s', and since $fragment(q) \neq f$ in s', the predicate is still true in s.

Suppose $fragment(q) = f$. Since $fragment(q)$ is the same in s as in s', and since $fragment(p) \neq g$ in s', the predicate is still true in s.

If $fragment(p) \neq g$ and $fragment(q) \neq f$ in s', the predicate is obviously still true in s.

TAR-G: Suppose REJECT is in $tarqueue(\langle p,q \rangle)$ in s', for some link $\langle p,q \rangle$ in $L(G)$. This is a protocol message for $\langle q,p \rangle$. By TAR-O, $fragment(q) \neq g$ in s'. By TAR-G, $fragment(p) \neq g$ in s', since otherwise $fragment(p) = fragment(q) = g$ in s'. So the predicate is still true in s.

TAR-H: Let $\langle q,p \rangle = minlink(g)$. Since $level(f) > level(g)$ by COM-A, $\langle p,q \rangle \neq minlink(g)$. So it is OK to set $lstatus(\langle p,q \rangle)$ to branch.

TAR-I: First note that if there is some node $r \in nodes(f) - testset(f)$ in s' with an unknown link, then by TAR-I there is an external link $\langle t,u \rangle$ of f, and $level(f) \leq level(fragment(u))$. Thus $fragment(u) \neq g$, so in s, the predicate is still true for nodes that were in $nodes(f)$ in s'.

To show that the predicate is true in s for nodes that were in $nodes(g)$ in s': we only need to consider the case when $p \notin testset(f)$ in s', i.e., when nodes formerly in $nodes(g)$ are not added to $testset(f)$. Since $level(f) > level(g)$, $minlink(f) \neq \langle p,q \rangle$, by COM-A. Thus, by TAR-A(a) and TAR-B, $lstatus(\langle p,q \rangle) =$ unknown, and the argument in the previous paragraph holds.

To show that the predicate is true in s for nodes that are not in either $nodes(f)$ or $nodes(g)$ in s', it is enough to note that the only relevant change is that the level of every node formerly in $nodes(g)$ is increased. □

Let $P'_{TAR} = (P'_{GC} \circ \mathcal{S}_3) \wedge P_{TAR}$.

Corollary 18: *P'_{TAR} is true in every reachable state of TAR.*

Proof: By Lemmas 1 and 17. □

4.2.4 DC Simulates GC

This automaton focuses on how the nodes of a fragment cooperate to find the minimum-weight external link of the fragment in a distributed fashion. The variable *minlink*(f) is now a derived variable, depending on variables local to each node, and the contents of message queues. There is no action *ComputeMin*(f). The two nodes adjacent to the core send out FIND messages over the core. These messages are propagated throughout the fragment. When a node p receives a FIND message, it changes the variable *dcstatus*(p) from unfind to find, relays FIND messages, and records the link from which the FIND was received as its *inbranch*(p). Then the node atomically finds its local minimum-weight external link using action *TestNode*(p) as in GC, and waits to receive REPORT(w) messages from all its "children" (the nodes to which it sent FIND). The variable *findcount*(p) records how many children have not yet reported. Then p takes the minimum over all the weights w reported by its children and the weight of its own local minimum-weight external link and sends that weight to its "parent" in a REPORT message, along *inbranch*(p); the weight and the link associated with this minimum are recorded as *bestwt*(p) and *bestlink*(p), and *dcstatus*(p) is changed back to unfind. When a node adjacent to the core has heard from all its children, it sends a REPORT over the core. This message is not processed by the recipient until its *dcstatus* is set back to unfind. When a node p adjacent to the core receives a REPORT(w) over the core with $w >$ *bestwt*(p), then *minlink*(f) becomes defined, and is the link found by following *bestlinks* from p.

The *ChangeRoot*(f) action is the same as in GC. When two fragments merge, a FIND message is added to one link of the new core. A new action, *AfterMerge*(p, q), adds a FIND message to the other link of the new core. When an *Absorb*(f, g) action occurs, a FIND message is directed toward the old g along the reverse link of *minlink*(g) if and only if the target of *minlink*(g) is in *testset*(f) and its *dcstatus* is find.

This algorithm (as well as the original one) correctly handles "leftover" REPORT messages. Recall that a REPORT message is sent in both directions over the core (p, q) of a fragment f. Suppose the root p receives its REPORT message first, and the other REPORT message, the "leftover" one, which is headed toward q, remains in the queue until after f merges or is absorbed. Since the queues are FIFO relative to REPORT and FIND messages, the state of q remains such that when the leftover REPORT message is received, the only change is the removal of the message.

Define automaton DC (for "Distributed ComputeMin") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;
- *level*(f), a nonnegative integer;
- *rootchanged*(f), a Boolean; and
- *testset*(f), a subset of $V(G)$.

For each node p, there are the following variables:

- *dcstatus*(p), either find or unfind;
- *findcount*(p), a nonnegative integer;
- *bestlink*(p), a link of G or *nil*;
- *bestwt*(p), a weight or ∞; and
- *inbranch*(p), a link of G or *nil*.

For each link $\langle p, q\rangle$, there are associated three variables:

- $dcqueue_p(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;
- $dcqueue_{pq}(\langle p, q\rangle)$, a FIFO queue of messages from p to q that are in the communication channel; and
- $dcqueue_q(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at q to be processed.

The set of possible messages M is $\{\text{REPORT}(w) : w \text{ a weight or } \infty\} \cup \{\text{FIND}\}$.

The state also contains Boolean variables, *answered*(l), one for each $l \in L(G)$, and Boolean variable *awake*.

In the start state of DC, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*(f) $= \{p\}$, *core*(f) $=$ *nil*, *level*(f) $= 0$, *rootchanged*(f) is false, and *testset*(f) is empty. For each p, *dcstatus*(p) $=$ unfind, *findcount*(p) $= 0$, *bestlink*(p) is the minimum-weight external link of p, *bestwt*(p) is the weight of *bestlink*(p), and *inbranch*$(p) = nil$. The message queues are empty. Each *answered*(l) is false and *awake* is false.

The derived variable *dcqueue*$(\langle p,q\rangle)$ is defined to be *dcqueue*$_q(\langle p,q\rangle)$ $||$ *dcqueue*$_{pq}(\langle p,q\rangle)$ $||$ *dcqueue*$_p(\langle p,q\rangle)$.

A REPORT(w) message is ***headed toward*** p if either it is in *dcqueue*$(\langle q,p\rangle)$ for some q, or it is in some *dcqueue*$(\langle q,r\rangle)$, where $q \in$ *subtree*(r) and $r \in$ *subtree*(p). A FIND message is ***headed toward*** p if it is in some *dcqueue*$(\langle q,r\rangle)$ and p is in *subtree*(r). A message is said to be *in subtree*(f) if it is in some *dcqueue*$(\langle q,p\rangle)$ and $p \in$ *nodes*(f).

Now *minlink*(f) is a derived variable, defined as follows. If *nodes*$(f) = \{p\}$, then *minlink*(f) is the minimum-weight external link of p. Suppose *nodes*(f) contains more than one node. If f has an external link, if *dcstatus*(p) = unfind for all $p \in$ *nodes*(f), if no FIND message is in *subtree*(f), and if no REPORT message is headed toward *mw-root*(f), then *minlink*(f) is the first external link reached by starting at *mw-root*(f) and following bestlinks; otherwise, *minlink*$(f) = nil$.

Also *accmin*(f) is a derived variable, defined as in TAR as follows. If *minlink*$(f) \neq nil$, or if there is no external link of any $p \in$ *nodes*(f) $-$ *testset*(f), then *accmin*$(f) = nil$. Otherwise, *accmin*(f) is the minimum-weight external link of all $p \in$ *nodes*(f) $-$ *testset*(f).

Note below that *ReceiveFind*$(\langle q,p\rangle)$ is only enabled if *AfterMerge*(p,q) is not enabled; without this precondition on *ReceiveFind*, p could receive the FIND before sending a FIND to q, and thus q's side of the subtree would not participate in the search.

Input actions:

- *Start*(p), $p \in V(G)$
  - Effects:
    - *awake* := true

Output actions:

- *InTree*$(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
  - Preconditions:
    - *awake* = true
    - $(p,q) \in$ *subtree*(*fragment*(p)) or $\langle p,q\rangle$ = *minlink*(*fragment*(p))
    - *answered*$(\langle p,q\rangle)$ = false

```
    Effects:
      $answered(\langle p,q\rangle) :=$ true
```

- $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$

```
    Preconditions:
      $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$
      $answered(\langle p,q\rangle) =$ false
    Effects:
      $answered(\langle p,q\rangle) :=$ true
```

Internal actions:

- $ChannelSend(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$

```
    Preconditions:
      $m$ at head of $dcqueue_p(\langle p,q\rangle)$
    Effects:
      dequeue($dcqueue_p(\langle p,q\rangle)$)
      enqueue($m$, $dcqueue_{pq}(\langle p,q\rangle)$)
```

- $ChannelRecv(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$

```
    Preconditions:
      $m$ at head of $dcqueue_{pq}(\langle p,q\rangle)$
    Effects:
      dequeue($dcqueue_{pq}(\langle p,q\rangle)$)
      enqueue($m$, $dcqueue_q(\langle p,q\rangle)$)
```

- $TestNode(p)$, $p \in V(G)$

```
    Preconditions:
      — let $f = fragment(p)$ —
      $p \in testset(f)$
      if $\langle p,q\rangle$, the minimum-weight external link of $p$, exists
        then $level(f) \leq level(fragment(q))$
      $dcstatus(p) =$ find
    Effects:
      $testset(f) := testset(f) - \{p\}$
      if $\langle p,q\rangle$, the minimum-weight external link of $p$, exists then
        if $wt(p,q) < bestwt(p)$ then [
          $bestlink(p) := \langle p,q\rangle$
          $bestwt(p) := wt(p,q)$ ]
      execute procedure $Report(p)$
```

```
• $ReceiveReport(\langle q,p\rangle, w)$, $\langle q,p\rangle \in L(G)$
    Preconditions:
      $\text{REPORT}(w)$ message at head of $dcqueue_p(\langle q,p\rangle)$
    Effects:
      dequeue($dcqueue_p(\langle q,p\rangle)$)
      if $\langle p,q\rangle \neq inbranch(p)$ then [
        $findcount(p) := findcount(p) - 1$
        if $w < bestwt(p)$ then [
          $bestwt(p) := w$
          $bestlink(p) := \langle p,q\rangle$ ]
        execute procedure $Report(p)$ ]
      else
        if $dcstatus(p) =$ find then enqueue($\text{REPORT}(w)$, $dcqueue_p(\langle q,p\rangle)$)

• $ReceiveFind(\langle q,p\rangle)$, $\langle q,p\rangle \in L(G)$
    Preconditions:
      FIND message at head of $dcqueue_p(\langle q,p\rangle)$
      $AfterMerge(p,q)$ not enabled
    Effects:
      dequeue($dcqueue_p(\langle q,p\rangle)$)
      $dcstatus(p) :=$ find
      $inbranch(p) := \langle p,q\rangle$
      $bestlink(p) := nil$
      $bestwt(p) := \infty$
      — let $S = \{\langle p,r\rangle : (p,r) \in subtree(fragment(p)), r \neq q\}$ —
      $findcount(p) := |S|$
      enqueue(FIND, $dcqueue_p(l)$) for all $l \in S$

• Procedure $Report(p)$, $p \in V(G)$
    if $findcount(p) = 0$ and $p \notin testset(fragment(p))$ then [
      $dcstatus(p) :=$ unfind
      enqueue($\text{REPORT}(bestwt(p))$, $dcqueue_p(inbranch(p))$) ]

• $ChangeRoot(f)$, $f \in fragments$
    Preconditions:
      $awake =$ true
      $rootchanged(f) =$ false
      $minlink(f) \neq nil$
    Effects:
        $rootchanged(f) :=$ true

• $Merge(f, g)$, $f, g \in fragments$
    Preconditions:
        $f \neq g$
        $rootchanged(f) = rootchanged(g) =$ true
        $minedge(f) = minedge(g)$
    Effects:
        add a new element $h$ to $fragments$
        $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
        $core(h) := minedge(f)$
        $level(h) := level(f) + 1$
        $rootchanged(h) :=$ false
        $testset(h) := nodes(h)$
        — let $\langle p, q\rangle = minlink(f)$ —
        enqueue(FIND, $dcqueue_p(\langle p, q\rangle)$)
        delete $f$ and $g$ from $fragments$

• $AfterMerge(p, q)$, $p, q \in V(G)$
    Preconditions:
        $(p, q) = core(fragment(p))$
        FIND message in $dcqueue(\langle q, p\rangle)$
        no FIND message in $dcqueue(\langle p, q\rangle)$
        $dcstatus(q) =$ unfind
        no REPORT message in $dcqueue(\langle q, p\rangle)$
    Effects:
        enqueue(FIND, $dcqueue_p(\langle p, q\rangle)$)

• $Absorb(f, g)$, $f, g \in fragments$
    Preconditions:
        $rootchanged(g) =$ true
        $level(g) < level(f)$
        — let $\langle q, p\rangle = minlink(g)$ —
        $fragment(p) = f$
    Effects:
        $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
        if $p \in testset(f)$ then [
            $testset(f) := testset(f) \cup nodes(g)$
            if $dcstatus(p) =$ find then [
        enqueue(FIND, $dcqueue_p(\langle p,q\rangle)$)
        $findcount(p) := findcount(p) + 1$ ] ]
    delete $g$ from $fragments$
```

Define the following predicates on *states*(DC), using these definitions.

A child q of p is *completed* if no node in $subtree(q)$ is in $testset(fragment(p))$, and no REPORT is headed toward p in $subtree(q)$ or in $dcqueue(\langle q,p\rangle)$. Node p is *up-to-date* if either $subtree(fragment(p)) = \{p\}$, or the following two conditions are met: (1) following *inbranches* from p leads along edges of $subtree(fragment(p))$ toward and over $core(f)$, and (2) if $p \in testset(fragment(p))$, then $dcstatus(p) =$ find. Given node p, define C_p to be the set $\{r :$ either $r = p$ and $p \notin testset(fragment(p))$, or r is in $subtree(q)$ for some completed child q of $p\}$.

All free variables are universally quantified, except that $f = fragment(p)$, in these predicates. (The fact that an old REPORT message, in a link that was formerly the core of a fragment, can remain even after that fragment has merged or been absorbed, complicated the statement of some of the predicates.)

- DC-A: If REPORT(w) is in $dcqueue(\langle q,p\rangle)$ and $inbranch(p) \neq \langle p,q\rangle$, then
  (a) if $(p,q) = core(f)$, then a FIND message is ahead of the REPORT in $dcqueue(\langle q,p\rangle)$;
  (b) $\langle q,p\rangle = inbranch(q)$;
  (c) $bestwt(q) = w$;
  (d) $dcstatus(q) =$ unfind;
  (e) every child of q is completed;
  (f) $q \notin testset(f)$; and
  (g) if $(p,q) \neq core(f)$, then $dcstatus(p) =$ find, and q is a child of p.

- DC-B: If REPORT(w) is in $dcqueue(\langle q,p\rangle)$ and $inbranch(p) = \langle p,q\rangle$, then
  (a) either $(p,q) = core(f)$ or p is a child of q; and
  (b) if $(p,q) \neq core(f)$, then $dcstatus(p) =$ unfind.

- DC-C: If REPORT(w) is in $dcqueue(\langle q,p\rangle)$ and $(p,q) = core(f)$, then
  (a) q is up-to-date;
  (b) $dcstatus(q) =$ unfind; and
  (c) $bestwt(q) = w$.

- DC-D: If FIND is in $dcqueue(\langle q,p\rangle)$, then
  (a) if $(p,q) \neq core(f)$ then p is a child of q and $dcstatus(q) =$ find;

(b) $dcstatus(p)$ = unfind; and
(c) every node in $subtree(p)$ is in $testset(f)$.

- DC-E: If $p \in testset(f)$, then a FIND message is headed toward p, or $dcstatus(p)$ = find, or $AfterMerge(q,r)$ is enabled, where $p \in subtree(r)$.

- DC-F: If $(p,q) = core(f)$ and $inbranch(q) \neq \langle q,p\rangle$, then either a FIND is in $dcqueue(\langle p,q\rangle)$, or $AfterMerge(p,q)$ is enabled.

- DC-G: If $AfterMerge(p,q)$ is enabled, then every node in $subtree(q)$ is in $testset(f)$.

- DC-H: If $dcstatus(p)$ = unfind, then
  (a) $dcstatus(q)$ = unfind for all $q \in subtree(p)$; and
  (b) $findcount(p) = 0$.

- DC-I: If $dcstatus(p)$ = find, then
  (a) p is up-to-date; and
  (b) either a REPORT message is in $subtree(p)$ headed toward p, or some $q \in subtree(p)$ is in $testset(f)$.

- DC-J: If $dcstatus(p)$ = find and $core(f) = (p,q)$, then a FIND message is in $dcqueue(\langle p,q\rangle)$, or $dcstatus(q)$ = find, or a REPORT message is in $dcqueue(\langle q,p\rangle)$.

- DC-K: If p is up-to-date, then
  (a) $findcount(p)$ is the number of children of p that are not completed;
  (b) if $bestlink(p) = nil$, then $bestwt(p) = \infty$, and there is no external link of any node in C_p.
  (c) if $bestlink(p) \neq nil$, then following $bestlinks$ from p leads along edges in $subtree(f)$ to the minimum-weight external link l of all nodes in C_p; $wt(l) = bestwt(p)$, and $level(fragment(target(l))) \geq level(f)$.

- DC-L: If $inbranch(p) \neq nil$, then $inbranch(p) = \langle p,q\rangle$ for some q, and $(p,q) \in subtree(f)$.

- DC-M: $findcount(p) \geq 0$.

- DC-N: If $mw\text{-}minnode(f)$ is not in $testset(f)$, then $mw\text{-}minnode(f)$ is up-to-date.

- DC-O: The only possible values of $dcqueue(\langle p,q\rangle)$ are empty, or FIND, or REPORT, or FIND followed by REPORT (only if $(p,q) = core(f)$), or REPORT followed by FIND (only if $(p,q) \neq core(f)$).

Let P_{DC} be the conjunction of DC-A through DC-O.

In order to show that DC simulates GC, we define an abstraction mapping $\mathcal{M}_4 = (\mathcal{S}_4, \mathcal{A}_4)$ from DC to GC.

Define the function $\mathcal{S}_4$ from $states(DC)$ to $states(GC)$ by ignoring the message queues, and the variables *dcstatus*, *findcount*, *bestlink*, *bestwt*, and *inbranch*. The derived variables *minlink* and *accmin* of DC map to the (non-derived) variables *minlink* and *accmin* of GC.

Define the function $\mathcal{A}_4$ as follows. Let s be a state of DC and π an action of DC enabled in s. The GC action $ComputeMin(f)$ is simulated in DC when a node adjacent to the core, having already heard from all its children, receives a REPORT message over the core with a weight larger than its own *bestwt*. Then the node knows that the minimum-weight external link of the fragment is on its own side of the subtree.

- Suppose $\pi = ReceiveReport(\langle q,p\rangle, w)$. If $(p,q) = core(f)$ and $dcstatus(p) =$ unfind and $w > bestwt(p)$, then $\mathcal{A}_4(s,\pi) = ComputeMin(fragment(p))$. Otherwise $\mathcal{A}_4(s,\pi)$ is empty.

- If $\pi = ChannelSend(\langle q,p\rangle, m)$, $ChannelRecv(\langle q,p\rangle, m)$, $ReceiveFind(\langle q,p\rangle)$ or $AfterMerge(p,q)$, then $\mathcal{A}_4(s,\pi)$ is empty.

- For all other values of π, $\mathcal{A}_4(s,\pi) = \pi$.

The following predicates are true in any state of DC satisfying $(P'_{GC} \circ \mathcal{S}_4) \wedge P_{DC}$. Recall that $P'_{GC} = (P'_{COM} \circ \mathcal{S}_2) \wedge P_{GC}$. If $P'_{GC}(\mathcal{S}_4(s))$ is true, then the GC predicates are true in $\mathcal{S}_4(s)$, the COM predicates are true in $\mathcal{S}_2(\mathcal{S}_4(s))$, and the HI predicates are true in $\mathcal{S}_1(\mathcal{S}_2(\mathcal{S}_4(s)))$. Thus, these predicates are deducible from P_{DC}, together with the GC, COM and HI predicates.

- DC-P: If REPORT(w) is at the head of $dcqueue(\langle q,p\rangle)$ and $(p,q) = core(f)$ and $dcstatus(p) =$ unfind, then

  (a) if $w < bestwt(p)$, then the minimum-weight external link l of f is closer to q than to p, and $wt(l) = w$;

  (b) if $w > bestwt(p)$, then the minimum-weight external link l of f is closer to p than to q, and $wt(l) = bestwt(p)$; and

  (c) if $w = bestwt(p)$, then $w = \infty$ and there is no external link of f.

Proof:

1. REPORT(w) is at head of *dcqueue*($\langle q,p\rangle$), by assumption.
2. *dcstatus*(p) = unfind, by assumption.
3. (p,q) = *core*(f), by assumption.
4. q is up-to-date, by Claims 1 and 3 and DC-C(a).
5. *dcstatus*(q) = unfind, by Claims 1 and 3 and DC-C(b).
6. w = *bestwt*(q), by Claims 1 and 3 and DC-C(c).
7. $q \notin$ *testset*(f), by Claims 4 and 5.
8. No FIND is in *dcqueue*($\langle q,p\rangle$), by Claims 1 and 3 and DC-O.
9. p is up-to-date, by Claims 2, 3, 4 and 8 and DC-T.
10. $p \notin$ *testset*(f), by Claims 2 and 9.
11. *findcount*(p) = 0, by Claim 2 and DC-H(b).
12. *findcount*(q) = 0, by Claim 5 and DC-H(b).
13. All children of p are completed, by Claims 9 and 11 and DC-K(a).
14. All children of q are completed, by Claims 4 and 12 and DC-K(a).
15. If *bestwt*(p) = ∞, then there is no external link of *subtree*(p), by Claims 9, 10 and 13 and DC-K(b) and (c).
16. If *bestwt*(p) $\neq \infty$, then following *bestlinks* from p leads to the minimum-weight external link l of *subtree*(p) and $wt(l)$ = *bestwt*(p), by Claims 9, 10 and 13, and DC-K(b) and (c).
17. If *bestwt*(q) = $w = \infty$, then there is no external link of *subtree*(q), by Claims 4, 6, 7 and 14 and DC-K(b) and (c).
18. If *bestwt*(q) = $w \neq \infty$, then following *bestlinks* from q leads to the minimum-weight external link l of *subtree*(q) and $wt(l) = w$, by Claims 4, 6, 7 and 14 and DC-K(b) and (c).

Claims 3 and 15 through 18 give the result, together with the fact that edge weights are distinct. □

- DC-Q: If a REPORT is at the head of *dcqueue*($\langle q,p\rangle$) and is not headed toward *mw-root*(f), then *inbranch*(p) = $\langle p,q\rangle$.

Proof: If (p,q) = *core*(f), then *inbranch*(p) = $\langle p,q\rangle$ by DC-A(a). Suppose $(p,q) \neq$ *core*(f), and, in contradiction, that *inbranch*(p) $\neq \langle p,q\rangle$. By DC-A(g), *dcstatus*(p) = find, and by DC-I(a) p is up-to-date, i.e., following *inbranches* from p leads toward and over *core*(f). Thus the REPORT in *dcqueue*($\langle q,p\rangle$is headed toward both endpoints of *core*(f), contradicting the hypothesis. □

- DC-R: If *dcstatus*(p) = find, then no REPORT is in *dcqueue*(*inbranch*(p)).

Proof: Let *inbranch*(p) = $\langle p,q\rangle$.

1. $dcstatus(p) =$ find, by assumption.
2. p is up-to-date, by Claim 1 and DC-I(a).
3. Following *inbranches* from p leads toward and over $core(f)$, by Claim 2.
4. Either $(p, q) = core(f)$, or $inbranch(q) \neq \langle q, p\rangle$, or no REPORT is in $dcqueue(\langle p, q\rangle)$, by Claim 3 and DC-B(b).
5. If $(p, q) = core(f)$, then no REPORT is in $dcqueue(\langle p, q\rangle)$, by Claim 1 and DC-C(b).
6. If $inbranch(q) \neq \langle q, p\rangle$, then no REPORT is in $dcqueue(\langle p, q\rangle)$, by Claim 1 and DC-A(d).
7. No REPORT is in $dcqueue(\langle p, q\rangle)$, by Claims 4, 5 and 6. □

- DC-S: At most one FIND message is headed toward p.

Proof: Suppose a FIND message is headed toward p.
1. A FIND is in $dcqueue(\langle q, r\rangle)$, by assumption.
2. $p \in subtree(r)$, by assumption.
3. $dcstatus(r) =$ unfind, by Claim 1 and DC-D(b).
4. $dcstatus(t) =$ unfind for all $t \in subtree(r)$, by Claim 3 and DC-H(a).
5. No FIND message is in $dcqueue(\langle t, u\rangle)$, for any $(t, u) \in subtree(r)$, by Claim 4 and DC-D(a).

If $(q, r) = core(f)$, Claim 5 proves the result. Suppose $(q, r) \neq core(f)$.

6. $(q, r) \neq core(f)$, by assumption.
7. $dcstatus(q) =$ find, by Claims 1 and 6 and DC-D(a).
8. $dcstatus(t) =$ find for all t between q and the endpoint of $core(f)$ closest to q, by Claim 7 and DC-H(a).
9. No FIND message is in $dcqueue(\langle t, u\rangle)$ for any (t, u) between $core(f)$ and q, by Claim 8 and DC-D(b).

Claim 9 completes the proof. □

- DC-T: If $(p, q) = core(f)$, no FIND is in $dcqueue(\langle p, q\rangle)$, p is up-to-date, and $dcstatus(q) =$ unfind, then q is up-to-date.

Proof:
1. $(p, q) = core(f)$, by assumption.
2. No FIND is in $dcqueue(\langle p, q\rangle)$, by assumption.
3. p is up-to-date, by assumption.
4. $dcstatus(q) =$ unfind, by assumption.

5. No FIND is headed toward q, by Claims 1 and 2 and DC-D(a).
6. No FIND is in $dcqueue(\langle q,p\rangle)$, by Claim 3 and DC-D(b) and (c).
7. $AfterMerge(p,q)$ is not enabled, by Claim 6.
8. $inbranch(q) = \langle q,p\rangle$, by Claims 5 and 7 and DC-F.
9. $q \notin testset(f)$, by Claims 4, 5 and 7 and DC-E.
10. q is up-to-date, by Claims 1, 8 and 9. □

Lemma 19: *DC simulates GC via $\mathcal{M}_4$, P_{DC}, and P'_{GC}.*

Proof: By inspection, the types of DC, GC, $\mathcal{M}_4$, and P_{DC} are correct. By Corollary 16, P'_{GC} is a predicate true in every reachable state of GC.

(1) Let s be in $start(DC)$. Obviously, P_{DC} is true in s, and $\mathcal{S}_4(s)$ is in $start(GC)$.

(2) Obviously, $\mathcal{A}_4(s,\pi)|ext(GC) = \pi|ext(DC)$.

(3) Let (s',π,s) be a step of DC such that P'_{GC} is true of $\mathcal{S}_4(s')$ and P_{DC} is true of s'. For (3a) we verify below only those DC predicates whose truth in s is not obvious.

i) π is Start(p), ChangeRoot(f), InTree(l), or NotInTree(l). $\mathcal{A}_4(s',\pi) = \pi$. Obviously $\mathcal{S}_4(s')\pi\mathcal{S}_4(s)$ is an execution fragment of GC and P_{DC} is true in s.

ii) π is ChannelSend(l,m) or ChannelRecv(l,m). $\mathcal{A}_4(s',\pi)$ is empty. Obviously $\mathcal{S}_4(s) = \mathcal{S}_4(s')$ and P_{DC} is true in s.

iii) π is TestNode(p). Let $f = fragment(p)$ in s'.

(3c) $\mathcal{A}_4(s',\pi) = \pi$. Obviously, π is enabled in $\mathcal{S}_4(s')$. To show the effects are mirrored in $\mathcal{S}_4(s)$, we must show that $accmin(f)$ is updated properly (which is obvious) and that $minlink(f)$ is unchanged. Since $p \in testset(f)$ in s', $minlink(f) = nil$ in s' by GC-C. If $accmin(f) \neq nil$, or if p has an external link in s', then $accmin(f) \neq nil$ in s, and $minlink(f)$ is still nil in s. If some $q \neq p$ is in $testset(f)$ in s', then by DC-E either a FIND is in $subtree(f)$ or $dcstatus(q) =$ find; since the same is true in s, $minlink(f)$ is still nil in s. Finally, if $accmin(f) = nil$, p has no external link, and p is the sole element of $testset(f)$ in s', then f has no external link in s' or in s, and $minlink(f)$ is still nil in s.

(3a) Two cases are considered. First we prove some facts true in both cases.

Claims about s':

1. *dcstatus*$(p) =$ find, by precondition.
2. $p \in$ *testset*(f), by precondition.
3. If $\langle p, u\rangle$, the minimum-weight external link of p, exists, then *level*$(f) \leq$ *level*(*fragment*(u)), by precondition.
4. p is up-to-date, by Claim 1 and DC-I(a).
5. No FIND is headed toward p, by Claim 1 and DC-D(c).
6. If $(p, r) =$ *core*(f), then no REPORT is in *dcqueue*$(\langle p, r\rangle)$, for any r, by Claim 1 and DC-C(b).
7. If a REPORT is in *dcqueue*$(\langle p, r\rangle)$, then *inbranch*$(r) = \langle r, p\rangle$, for any r, by Claim 1 and DC-A(d).
8. *AfterMerge*(r, t), where $p \in$ *subtree*(t), is not enabled, by Claim 1 and DC-H(a).
9. If *bestlink*$(p) = nil$, then *bestwt*$(p) = \infty$ and there is no external link of any node r, where r is in the subtree of any completed child of p, by Claims 2 and 4 and DC-K(b).
10. If *bestlink*$(p) \neq nil$, then following *bestlinks* from p leads to the minimum-weight external link l of all nodes r, where r is in the subtree of any completed child of p; $wt(l) =$ *bestwt*(p) and *level*$(f) \leq$ *level*(*fragment*(*target*(l))), by Claims 2 and 4 and DC-K(c).

Case 1: findcount$(p) \neq 0$ in s'.

More claims about s'*:*

11. *findcount*$(p) \neq 0$, by assumption.
12. *findcount*$(p) > 0$, by Claim 11 and DC-M.
13. Some child r of p is not completed, by Claims 4 and 12 and DC-K(a).
14. There is a child r of p such that either some node in *subtree*(r) is in *testset*(f), or a REPORT is in *subtree*(r) or *dcqueue*$(\langle r, p\rangle)$ headed toward p, by Claim 13.

DC-A(c): By Claim 7, changing *bestwt*(p) and removing p from *testset*(f) are OK.

DC-C: By Claim 6, changing *bestwt*(p) is OK.

DC-D(c): By Claim 5, removing p from *testset*(f) is OK.

DC-G: By Claim 8 and the fact that *dcstatus*(p) is still find in s, removing p from *testset*(f) is OK.

DC-I(b): By Claim 14, removing p from $testset(f)$ is OK.

DC-K: (b) By Claim 9 and code. (c) by Claims 3 and 10 and code.

DC-N: If p is $mw\text{-}minnode(f)$, then by Claim 4, removing p from $testset(p)$ is OK.

Case 2: $findcount(p) = 0$ in s'. Let $\langle p, q\rangle = inbranch(p)$.

More claims about s':

15. $findcount(p) = 0$, by assumption.
16. If $(p, q) = core(f)$ and $inbranch(q) \neq \langle q, p\rangle$, then a FIND is in $dcqueue(\langle p, q\rangle)$, by Claim 5 and DC-F.
17. All children of p are completed, by Claims 3 and 15 and DC-K(a).
18. If $(p, q) \neq core(f)$, then $dcstatus(q) =$ find, by Claim 1 and DC-H(a).
19. If REPORT is in $dcqueue(\langle q, p\rangle)$, then $(p, q) = core(f)$, by Claim 4 and DC-B(a).
20. No REPORT is in $dcqueue(\langle p, q\rangle)$, by Claim 1 and DC-R.
21. If FIND is in $dcqueue(\langle p, q\rangle)$, then $(p, q) = core(f)$, by Claim 4 and DC-D(a).
22. Every node $r \neq p$ in $subtree(p)$ has $dcstatus(r) =$ unfind, by Claims 1 and 17 and DC-I(b).
23. Every node $r \neq p$ in $subtree(p)$ has $findcount(r) = 0$ by Claim 22 and DC-H(b).

DC-A: By Claim 7 and the fact that $inbranch(p) = \langle p, q\rangle$, we need only consider the REPORT added to $dcqueue(\langle p, q\rangle)$. (a) by Claim 16. (b), (c) and (d) by code. (e) by Claim 17. (f) by code. (g) by Claims 4 and 18.

DC-B for REPORT added to $dcqueue(\langle p, q\rangle)$: If $inbranch(q) = \langle q, p\rangle$, then $(p, q) = core(f)$, by Claim 4.

DC-B for REPORT that might be in $dcqueue(\langle q, p\rangle)$: by Claim 19.

DC-C: By Claim 4, $inbranch(p)$ is the only relevant link; by Claim 20, the new message is the only REPORT in that queue. (a) by Claim 4. (b) and (c) by code.

DC-D(a) and (c): By Claim 5, it is OK to change $dcstatus(p)$ to unfind and remove p from $testset(f)$.

DC-E: The addition of a REPORT to $dcqueue(\langle p, q\rangle)$ in s cannot cause *After-Merge*(q, p) to go from enabled in s' to disabled in s, by Claim 1.

DC-F: Cf. DC-E.

DC-G: By Claim 8 and the addition of REPORT to $dcqueue(\langle p,q\rangle)$, removing p from $testset(f)$ is OK.

DC-H: (a) By Claim 22 and code. (b) By Claim 23.

DC-I(b): Suppose $r \neq q$ is some node such that $p \in subtree(r)$ and $dcstatus(r) =$ find in s'. By Claim 4, removing p from $testset(f)$ is compensated for by adding REPORT to $dcqueue(\langle p,q\rangle)$.

DC-J: By Claim 4, the only link of p that can be part of $core(f)$ is $\langle p,q\rangle$. If $(p,q) = core(f)$ and $dcstatus(q) =$ find, then the fact that $dcstatus(p)$ becomes unfind in s is compensated for by the addition of REPORT to $dcqueue(\langle p,q\rangle)$.

DC-K(b) and (c): As in Case 1.

DC-N: As in Case 1.

DC-O: By Claims 20, 21 and code.

iv) π is ReceiveReport($\langle$q,p$\rangle$,w). Let $f = fragment(p)$ in s'.

(3b)/(3c) *Case 1:* $(p,q) = core(f)$ and $dcstatus(p) =$ unfind and $w > bestwt(p)$ in s'. $\mathcal{A}_4(s',\pi) = ComputeMin(f)$.

Let $\langle r,t\rangle$ be the minimum-weight external link of f in s'. (Below we show it exists.)

Claims about s':

1. REPORT(w) is at the head of $dcqueue(\langle q,p\rangle)$, by precondition.
2. $(p,q) = core(f)$, by assumption.
3. $dcstatus(p) =$ unfind, by assumption.
4. $w > bestwt(p)$, by assumption.
5. No FIND is in $dcqueue(\langle q,p\rangle)$, by Claim 1 and DC-O.
6. q is up-to-date, by Claims 1 and 2 and DC-C(a).
7. p is up-to-date, by Claims 2, 3, 5 and 6 and DC-T.
8. $dcstatus(q) =$ unfind, by Claims 1 and 2 and DC-C(b).
9. $bestwt(q) = w$, by Claims 1 and 2 nad DC-C(c).
10. $p = mw\text{-}root(f)$ (so $\langle r,t\rangle$ exists), by Claims 1, 2, 3 and 4 and DC-P(b).
11. $minlink(f) = nil$, by Claims 1 and 10.

12. $\mathit{findcount}(p) = 0$, by Claim 3 and DC-H(b).
13. $\mathit{findcount}(q) = 0$, by Claim 8 and DC-H(b).
14. Every child of p is completed, by Claims 7 and 12 and DC-K(a).
15. Every child of q is completed, by Claims 6 and 13 and DC-K(a).
16. $p \notin \mathit{testset}(f)$, by Claims 3 and 7.
17. $q \notin \mathit{testset}(f)$, by Claims 6 and 8.
18. $\mathit{testset}(f) = \emptyset$, by Claims 14 through 17.
19. $\mathit{accmin}(f) = \langle r, t \rangle$, by Claims 11 and 18.

By Claims 11, 18 and 19, *ComputeMin*(f) is enabled in s'.

Now we must show that the effects of *ComputeMin*(f) are mirrored in s. All that must be shown is that $\mathit{minlink}(f)$ and $\mathit{accmin}(f)$ are updated properly.

More claims about s':

20. $\mathit{dcstatus}(u) =$ unfind, for all $u \in \mathit{subtree}(p)$, by Claim 3 and DC-H(a).
21. $\mathit{dcstatus}(u) =$ unfind, for all $u \in \mathit{subtree}(q)$, by Claim 8 and DC-H(a).
22. No REPORT is headed toward p in $\mathit{subtree}(p)$, by Claim 14.
23. No REPORT is headed toward q in $\mathit{subtree}(q)$, by Claim 15.
24. Only one REPORT is in $\mathit{subtree}(p)$, by DC-O.
25. No FIND is in $\mathit{subtree}(f)$, by Claim 18 and DC-D(c).
26. Following *bestlinks* from p leads to $\langle r, t \rangle$, by Claims 7, 10, 14 and 16 and DC-K(b) and (c).

By Claims 10 and 20 through 26, $\mathit{minlink}(f) = \langle r, t \rangle$ in s. By Claim 19, this is the correct value. Thus, $\mathit{accmin}(f) = \mathit{nil}$ in s.

Case 2: $(p, q) \neq \mathit{core}(f)$ or $\mathit{dcstatus}(p) =$ find or $w \leq \mathit{bestwt}(p)$ in s'. $\mathcal{A}_4(s', \pi)$ is empty. We just need to verify that $\mathit{minlink}(f)$ and $\mathit{accmin}(f)$ are unchanged in order to show that $\mathcal{S}_4(s') = \mathcal{S}_4(s)$.

Subcase 2a: $(p, q) \neq \mathit{core}(f)$ in s'.

Suppose $\langle p, q \rangle = \mathit{inbranch}(p)$ in s'. By DC-B(b), $\mathit{dcstatus}(p) =$ unfind, so the only effect is to remove the REPORT. By DC-B(a), $p \in \mathit{subtree}(q)$, so this REPORT message is not headed toward $\mathit{mw\text{-}root}(f)$ in s'. Thus $\mathit{minlink}(f)$ is unchanged, and $\mathit{accmin}(f)$ is also unchanged.

Suppose $\langle p, q \rangle \neq \mathit{inbranch}(p)$ in s'.

Claims about s':

1. REPORT(w) is at the head of *dcqueue*($\langle q,p\rangle$), by precondition.
2. $\langle p,q\rangle \neq$ *inbranch*(p), by assumption.
3. $(p,q) \neq$ *core*(f), by assumption.
4. *dcstatus*(p) = find, by Claims 1, 2 and 3 and DC-A(g).
5. p is up-to-date, by Claim 4 and DC-I(a).
6. Following *inbranches* from p leads toward and over *core*(f), by Claim 5.
7. A REPORT message is headed toward *mw-root*(f), by Claims 1 and 6.
8. *minlink*(f) = *nil*, by Claim 7.
9. If *core*(f) = (p,t) for some t, then FIND is in *dcqueue*($\langle p,t\rangle$), *dcstatus*(t) = find, or REPORT is in *dcqueue*($\langle t,p\rangle$), by Claim 4 and DC-J.

Claims about s:

10. *subtree*(f), *core*(f), *nodes*(f), and *testset*(f) do not change, by code.
11. REPORT is in *inbranch*(p), by code.
12. Following *inbranches* from p leads toward and over *core*(f), by Claims 6 and 10 and code.
13. If $p \neq$ *mw-root*(f), then REPORT is headed toward *mw-root*(f), by Claims 11 and 12.
14. If $p =$ *mw-root*(f), then FIND is in *dcqueue*($\langle p,t\rangle$), *dcstatus*(t) = find, or REPORT is in *dcqueue*($\langle t,p\rangle$), where $(p,t) =$ *core*(f), by Claim 9 and code.
15. *minlink*(f) = *nil*, by Claims 13 and 14.
16. *accmin*(f) does not change, by Claims 8, 10 and 15.

Claims 15 and 16 give the result.

Subcase 2b: $(p,q) =$ *core*(f) and *dcstatus*(p) = find in s'. Since REPORT(w) is at the head of *dcqueue*($\langle q,p\rangle$), DC-A(a) implies that *inbranch*(p) = $\langle p,q\rangle$. The only change is that the REPORT message is requeued. Obviously *minlink*(f) and *accmin*(f) are unchanged.

Subcase 2c: $(p,q) =$ *core*(f) and *dcstatus*(p) = unfind and $w \leq$ *bestwt*(p) in s'. As in Subcase 2b, *inbranch*(p) = $\langle p,q\rangle$. The only change is that the REPORT message is removed. If $w =$ *bestwt*(p), then by DC-P(c), there is no external link of f in s' or in s. Thus *minlink*(f) and *accmin*(f) are both *nil* in s' and s.

Suppose $w < bestwt(p)$. By DC-P(a), $q = mw\text{-}root(f)$. Thus the REPORT message in $dcqueue(\langle q,p\rangle)$ is not headed toward $mw\text{-}root(f)$ in s', and no criteria for $minlink(f)$, or $accmin(f)$ changes.

(3a) *Case 1:* $\langle p,q\rangle = inbranch(p)$ in s'.

Suppose $dcstatus(p) =$ find. By DC-D(b), no FIND is in $dcqueue(\langle q,p\rangle)$ in s', so by DC-O, $dcqueue(\langle q,p\rangle)$ contains just the one REPORT message in s'. Since the only effect is to requeue the message, the *DC* state is unchanged.

Suppose $dcstatus(p) =$ unfind. The only change is the removal of the REPORT message from $dcqueue(\langle q,p\rangle)$. By DC-B(a), either $(p,q) = core(f)$, or $p \in subtree(q)$ in s'. In both cases, the REPORT is not headed toward any node whose subtree it is in.

DC-I(b): By remark above.

DC-J: Even though REPORT is removed from $dcqueue(\langle q,p\rangle)$, $dcstatus(p) =$ unfind in s.

DC-K(a): By remark above, removing the REPORT does not affect the completeness of any node's child.

Case 2: $\langle p,q\rangle \neq inbranch(p)$. Let $\langle p,r\rangle = inbranch(p)$.

Claims about s':

1. REPORT(w) is at head of $dcqueue(\langle q,p\rangle)$, by precondition.
2. $\langle p,q\rangle \neq inbranch(p)$, by assumption.
3. $(p,q) \neq core(f)$, by Claims 1 and 2 and DC-A(a).
4. $\langle q,p\rangle = inbranch(q)$, by Claims 1 and 2 and DC-A(b).
5. $w = bestwt(q)$, by Claims 1 and 2 and DC-A(c).
6. $dcstatus(q) =$ unfind, by Claims 1 and 2 and DC-A(d).
7. Every child of q is completed, by Claims 1 and 2 and DC-A(e).
8. $q \notin testset(f)$, by Claims 1 and 2 and DC-A(f).
9. $dcstatus(p) =$ find, by Claim 3 and DC-A(g).
10. If REPORT is in $dcqueue(p,t)$, then $inbranch(t) = \langle t,p\rangle$, for any t, by Claim 9 and DC-A(d).

11. p is up-to-date, by Claim 9 and DC-I(a).
12. *inbranch*(p) leads toward and over *core*(f), by Claim 11.
13. q is an uncompleted child of p, by Claims 1, 2 and 12.
14. *findcount*$(p) \geq 1$, by Claims 11 and 13 and DC-K(a).
15. Only one REPORT is in *dcqueue*$(\langle q,p\rangle)$, by Claim 1 and DC-O.
16. q is up-to-date, by Claims 4, 8 and 12.
17. If REPORT is in *dcqueue*$(\langle p,t\rangle)$, then $(p,t) \neq$ *core*(f), for all t, by Claim 9 and DC-C(b).
18. If *bestwt*$(p) = \infty$, then there is no external link of p (if $p \notin$ *testset*(f)) or of any node in the subtree of any completed child of p, by Claim 11 and DC-F(b) and (c).
19. If *bestwt*$(p) \neq \infty$, then following *bestlinks* from p leads to the minimum-weight external link l of all nodes in C_p; $wt(l) =$ *bestwt*(p); and *level*$(f) \leq$ *level*(*fragment*(*target*(l))), by Claim 11 and DC-F(b) and (c).
20. If $w = \infty$, then there is no external link of *subtree*(q), by Claims 5, 7, 8 and 16 and DC-K(b) and (c).
21. If $w \neq \infty$, then following *bestlinks* from q leads to the minimum-weight external link l of *subtree*(q); $wt(l) = w$, and *level*$(f) \leq$ *level*(*fragment*(*target*(l))), by Claims 5, 7, 8 and 16 and DC-F(b) and (c).

Subcase 2a: $p \in$ *testset*(f) or *findcount*$(p) \neq 1$ in s'.

More claims about s'*:*

22. $p \in$ *testset*(f) or *findcount*$(p) \neq 1$, by assumption.
23. If *findcount*$(p) \neq 1$, then *findcount*$(p) > 1$, by Claim 14.
24. If *findcount*$(p) \neq 1$, then some child $t \neq q$ of p is not completed, by Claims 11 and 23 and DC-K(a).
25. If *findcount*$(p) = 1$, then $p \in$ *testset*(f), by Claim 22.

DC-A(c): by Claim 10, any change to *bestwt*(p) is OK.

DC-C: By Claim 17, changing *bestwt*(p) is OK.

DC-F: Cf. DC-G.

DC-G: Removing REPORT from *dcqueue*$(\langle q,p\rangle)$ does not cause *AfterMerge*(p,q) to become enabled, by Claim 3.

DC-I(b): Let t be some node such that $p \in$ *subtree*(t) and *dcstatus*$(t) =$ find in s'. By Claims 24 and 25, either a REPORT message is in *subtree*(p) headed toward

p (and hence toward t), or some node in *subtree*(p) (and hence in *subtree*(t)) is in *testset*(f).

DC-J: The removal of the REPORT message is OK by Claim 3.

DC-K(a): Since *findcount*(p) is decremented by 1, we just need to show that the number of uncompleted children of p decreases by 1: by Claim 1, q is not completed in s'. By Claims 7, 8 and 15 and code, q is completed in s.

DC-K(b) and (c): by Claims 18, 19, 20 and 21 and code.

DC-M: By Claim 14 and code.

Subcase 2b: $p \notin$ *testset*(f) and *findcount*(p) = 1.

26. $p \notin$ *testset*(f), by assumption.
27. *findcount*(p) = 1, by assumption.
28. No FIND is headed toward p, by Claim 9 and DC-D(b).
29. If $(p,r) =$ *core*(f) and *inbranch*(r) $\neq \langle r,p\rangle$,then FIND is in *dcqueue*($\langle p,r\rangle$), by Claim 28 and DC-F.
30. No REPORT is in *dcqueue*($\langle p,r\rangle$), by Claim 9 and DC-R.
31. Every child of p but q is completed, by Claims 11, 13, 27 and DC-K(a).
32. No FIND is in *dcqueue*($\langle p,t\rangle$), $t \neq r$, by Claims 7, 8 and 31 and DC-D(c).
33. If REPORT is in *dcqueue*($\langle r,p\rangle$), then $(p,r) =$ *core*(f), by Claim 9 and DC-B(a) and (b).
34. If $(p,r) \neq$ *core*(f), then *dcstatus*(r) = find, by Claims 9 and 12 and DC-H(a).
35. If FIND is in *dcqueue*($\langle p,r\rangle$), then $(p,r) =$ *core*(f), by Claim 12 and DC-D(a).

DC-A: By Claim 10 and the fact that *inbranch*(p) = $\langle p,r\rangle$), we need only consider the REPORT added to *dcqueue*($\langle p,r\rangle$). (a) by Claim 29. (b), (c) and (d) by code. (e) by Claim 31 for any child of p except q; by Claims 7, 8 and 15 and code for q. (f) by Claim 8. (g) by Claims 12 and 34.

DC-B for REPORT added to *dcqueue*($\langle p,r\rangle$): if *inbranch*(r) = $\langle r,p\rangle$, then by Claim 12, *core*(f) = (p,r).

DC-B for REPORT in *dcqueue*($\langle r,p\rangle$): By Claim 33, *core*(f) = (p,r).

DC-C: By Claim 12, *inbranch*(p) is the only relevant link; by Claim 30, the new message is the only REPORT message in its queue. (a) by Claim 11. (b) and (c) by code.

DC-D(a): By Claims 32 and 35, changing $dcstatus(p)$ to unfind is OK.

DC-E: The addition of the REPORT to $dcqueue(\langle p,r\rangle)$ in s cannot cause $AfterMerge(r,p)$ to go from enabled in s' to disabled in s, because $dcstatus(p) =$ find in s' by Claim 9.

DC-F: Cf. DC-E.

DC-H(a): By Claims 7 and 8, no node in $subtree(q)$ is in $testset(f)$. By Claim 31, no node in $subtree(t)$, for any child $t \neq q$ of p, is in $testset(f)$. By Claim 23, $p \notin testset(f)$.

DC-H(b): By Claim 27 and code.

DC-I(b): Let $t \neq p$ be such that $p \in subtree(t)$ and $dcstatus(t) =$ find in s'. By Claim 12, removing the REPORT from $dcqueue(\langle q,p\rangle)$ is compensated for by adding the REPORT to $dcqueue(\langle p,r\rangle)$.

DC-J: By Claim 12, the only link of p that can be part of $core(f)$ is $\langle p,r\rangle$. If $(p,r) = core(f)$ and $dcstatus(q) =$ find in s', then changing $dcstatus(p)$ to unfind in s is compensated for by adding the REPORT to $dcqueue(\langle p,r\rangle)$.

DC-K: As in Subcase 2a.

DC-M: Claim 27 and code.

DC-O: by Claim 30 and DC-O and code.

v) π is ReceiveFind(⟨q,p⟩). Let $f = fragment(p)$.

(3b) $\mathcal{A}_4(s',\pi)$ is empty. To show that $\mathcal{S}_4(s') = \mathcal{S}_4(s)$, we just need to show that $minlink(f)$ and $accmin(f)$ are unchanged. Because of the FIND message, $minlink(f) = nil$ in s', and $minlink(f) = nil$ in s since $dcstatus(p) =$ find. Since there is no change to $minlink(f)$, $nodes(f)$, $testset(f)$, or $subtree(f)$, $accmin(f)$ is unchanged.

(3a) *Claims about s':*

1. FIND is at head of $dcqueue(\langle q,p\rangle)$, by precondition.
2. $AfterMerge(p,q)$ is not enabled, by precondition.
3. If $(p,q) \neq core(f)$, then p is a child of q, by Claim 1 and DC-D(a).
4. If $(p,q) \neq core(f)$, then $dcstatus(q) =$ find, by Claim 1 and DC-D(a).

5. $dcstatus(p) =$ unfind, by Claim 1 and DC-D(b).
6. Every node in $subtree(p)$ is in $testset(f)$, by Claim 1 and DC-D(c).
7. No REPORT is in $dcqueue(\langle p,r\rangle)$ with $inbranch(r) \neq \langle r,p\rangle$, for all r, by Claim 6 and DC-A(f).
8. If REPORT is in $dcqueue(\langle p,r\rangle)$, then $(p,r) \neq core(f)$, for all r, by Claim 6 and DC-C.
9. If REPORT is in $dcqueue(\langle q,p\rangle)$, then $(p,q) = core(f)$, by Claim 1 and DC-O.
10. If $(r,p) \in subtree(f)$, $r \neq q$, then r is a child of p, by Claim 3.
11. No REPORT is in $dcqueue(\langle r,p\rangle)$, $r \neq q$, with $inbranch(p) \neq \langle p,r\rangle)$, by Claims 6 and 10 and DC-A(f).
12. No REPORT is in $dcqueue(\langle r,p\rangle)$, $r \neq q$, with $inbranch(p) = \langle p,r\rangle$, by Claim 10 and DC-B(a).
13. If $\langle p,r\rangle \in S$, then r is a child of p, by Claim 10.
14. $dcstatus(r) =$ unfind for all $r \in subtree(p)$, by Claim 5 and DC-H(a).
15. If $(p,q) \neq core(f)$, then $dcstatus(r) =$ find, for all r such that $q \in subtree(r)$, by Claim 4 and DC-H(a).
16. $dcqueue(\langle p,r\rangle)$ is either empty or contains only a REPORT for all r such that $\langle p,r\rangle \in S$, by Claims 5 and 13 and DC-D(a) and DC-O.
17. If $(p,q) \neq core(f)$, then following inbranches from q leads toward and over $core(f)$, by Claim 4 and DC-I(a).

DC-A(a): By Claim 7, we need not consider any REPORT in a link leaving p. By Claim 11 we need not consider any REPORT in a link coming into p, except for $\langle q,p\rangle$. Since $inbranch(p)$ is set to $\langle p,q\rangle$ in s, removing FIND from $dcqueue(\langle q,p\rangle)$ is OK.

DC-B: By Claim 9 and 12, changing $dcstatus(p)$ is OK.

DC-C: By Claim 8, changing $dcstatus(p)$ and $bestwt(p)$ is OK.

DC-D: (a) by Claim 13 and code. (b) by Claim 14. (c) by Claim 6.

DC-E: By Claim 12 and code (adding FIND messages and setting $dcstatus(p)$ to find), removing FIND from $dcqueue(\langle q,p\rangle)$ is OK.

DC-F: As argued for DC-I(a), the only possible link of p that is part of $core(f)$ is $\langle p,q\rangle$. Since code sets $inbranch(p)$ to $\langle p,q\rangle$, removing the FIND is OK.

DC-H(a): If $(p,q) = core(f)$, then changing $dcstatus(p)$ to find is OK. If $(p,q) \neq core(f)$, then Claim 15 implies that it is OK to change $dcstatus(p)$ to find.

DC-I: (a) If $(p,q) = core(f)$, then code gives the result, since $inbranch(p)$ is set to $\langle p,q\rangle$ and $dcstatus(p)$ is set to find. If $(p,q) \neq core(f)$, then Claim 17, the fact that p is a child of q by DC-D(a), and code give the result. (b) by Claim 6.

DC-J: By Claims 1 and 2.

DC-K: (a) $findcount(p) = |S| =$ number of children of p. None is complete, by Claim 6. (b) and (c) are true by code, since no children are complete.

DC-L: by code and Claim 3.

DC-M: by code.

DC-O: Removing the FIND from $dcqueue(\langle q,p\rangle)$ is OK. Adding FIND to $dcqueue(\langle p,r\rangle)$, $\langle p,r\rangle \in S$, is OK by Claim 16.

vi) π is Merge(f,g).

(3c) $\mathcal{A}_4(s',\pi) = \pi$. Obviously π is enabled in $\mathcal{S}_4(s')$. Effects are mirrored in $\mathcal{S}_4(s)$ if we can show $accmin(h) = minlink(h) = nil$ in s. Inspecting the code reveals that in s, a FIND message is in $subtree(h)$, so $minlink(h) = nil$, and $nodes(h) = testset(h)$, so $accmin(h) = nil$.

(3a) *Claims about s':*

1. $f \neq g$, by precondition.
2. $rootchanged(f) =$ true, by precondition.
3. $rootchanged(g) =$ true, by precondition.
4. $minedge(f) = minedge(g)$, by precondition.
5. $minlink(f) \neq nil$, by Claim 2 and COM-B.
   Let $\langle p,q\rangle = minlink(f)$.
6. $minlink(g) = \langle q,p\rangle$, by Claims 1, 4 and 5.
7. No REPORT is headed toward $root(f)$, by Claim 5.
8. No REPORT is headed toward $root(g)$, by Claim 6.
9. No FIND is in $subtree(f)$, by Claim 5.
10. No FIND is in $subtree(g)$, by Claim 6.
11. $dcstatus(r) =$ unfind for all $r \in nodes(f)$, by Claim 5.
12. $dcstatus(r) =$ unfind for all $r \in nodes(g)$, by Claim 6.
13. $\langle p,q\rangle$ is the minimum-weight external link of f, by Claim 5 and COM-A.
14. $\langle q,p\rangle$ is the minimum-weight external link of g, by Claim 6 and COM-A.
15. $testset(f) = \emptyset$, by Claim 5 and GC-C.

16. *testset*$(g) = \emptyset$, by Claim 6 and GC-C.
17. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then *inbranch*$(t) = \langle t,r\rangle$, for all $(r,t) \in$ *subtree*(f), by Claims 9 and 11 and DC-A(a) and (f).
18. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then *inbranch*$(t) = \langle t,r\rangle$, for all $(r,t) \in$ *subtree*(f), by Claims 10 and 12 and DC-A(a) and (f).
19. If REPORT is in *dcqueue*$(\langle r,t\rangle)$ and $(r,t) =$ *core*(f), then $r =$ *root*(f), by Claim 7.
20. If REPORT is in *dcqueue*$(\langle r,t\rangle)$ and $(r,t) =$ *core*(g), then $r =$ *root*(g), by Claim 8.
21. If REPORT is in *dcqueue*$(\langle r,t\rangle)$ and $(r,t) \neq$ *core*(f), then t is a child of r, for all $(r,t) \in$ *subtree*(f), by Claim 17 and DC-B(a).
22. If REPORT is in *dcqueue*$(\langle r,t\rangle)$ and $(r,t) \neq$ *core*(g), then t is a child of r, for all $(r,t) \in$ *subtree*(g), by Claim 18 and DC-B(a).
23. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then (r,t) is not on the path between *root*(f) and p, for all $(r,t) \in$ *subtree*(f), by Claims 5, 7, 13, 15 and 17 and DC-N.
24. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then (r,t) is not on the path between *root*(g) and q, for all $(r,t) \in$ *subtree*(g), by Claims 6, 8, 14, 16 and 18 and DC-N.
25. *dcqueue*$(\langle p,q\rangle)$ is empty, by Claim 13 and DC-A(g), DC-B(a) and DC-D(a).
26. *dcqueue*$(\langle q,p\rangle)$ is empty, by Claim 14 and DC-A(g), DC-B(a) and DC-D(a).
27. *findcount*$(r) = 0$ for all $r \in$ *nodes*(f), by Claim 11 and DC-H(b).
28. *findcount*$(r) = 0$ for all $r \in$ *nodes*(g), by Claim 12 and DC-H(b).

Claims about s:

29. *subtree*(h) is the old *subtree*(f) and *subtree*(g) and (p,q), by code.
30. *core*$(h) = (p,q)$, by code.
31. *testset*$(h) =$ *nodes*(h), by code.
32. *dcqueue*$(\langle p,q\rangle)$ contains only a FIND, by Claim 25 and code.
33. No FIND is in any other link of *subtree*(h), by Claims 9, 10 and 29.
34. *dcstatus*$(r) =$ unfind for all $r \in$ *nodes*(h), by Claims 11, 12 and 29.
35. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then *inbranch*$(t) = \langle t,r\rangle$, for all $(r,t) \in$ *subtree*(h), by Claims 17, 18, 25, 26 and 29.
36. If REPORT is in *dcqueue*$(\langle r,t\rangle)$, then t is a child of r, for all $(r,t) \in$ *subtree*(h), by Claims 21 through 26 and 28.
37. *AfterMerge*(q,p) is enabled, by Claims 30, 32, 33 and 34.
38. *dcqueue*$(\langle q,p\rangle)$ is empty, by Claim 26.
39. *findcount*$(r) = 0$ for all $r \in$ *nodes*(h), by Claims 27, 28 and 29.

DC-A: Vacuously true, by Claim 35.

DC-B: By Claims 34 and 36.

DC-C: By Claims 30, 32 and 38.

DC-D: The only FIND is in *dcqueue*($\langle p, q\rangle$), by Claims 32 and 33. (a) by Claim 30. (b) by Claim 34. (c) by Claim 31.

DC-E: By Claim 32 for *subtree*(q); by Claim 37 for *subtree*(p).

DC-F: By Claims 32 and 37.

DC-G: By Claim 31.

DC-H: (a) by Claim 34. (b): by Claim 39.

DC-I: Vacuously true by Claim 34.

DC-J: Vacuously true by Claim 34.

DC-K: By Claims 31 and 34, none is up-to-date.

DC-M: By Claim 39.

DC-N: Vacuously true by Claim 31.

DC-O: By Claim 30.

vii) π is AfterMerge(p,q). Let f = *fragment*(p).

(3b) $\mathcal{A}_4(s', \pi)$ is empty. We just need to show that *accmin*(f) and *minlink*(f) do not change. The FIND message(s) imply that *minlink*(f) = *nil* in both s' and s. Since there is no change to *minlink*(f), *nodes*(f), *testset*(f), or *subtree*(f), *accmin*(f) does not change.

(3a) *Claims about s':*

1. (p, q) = *core*(f), by precondition.
2. FIND is in *dcqueue*($\langle q, p\rangle$), by precondition.
3. No FIND is in *dcqueue*($\langle p, q\rangle$), by precondition.
4. *dcstatus*(q) = unfind, by precondition.
5. No REPORT is in *dcqueue*($\langle q, p\rangle$), by precondition.
6. Every node in *subtree*(q) is in *testset*(f), by Claims 1 through 5 and DC-G.
7. $p \in$ *testset*(f), by Claim 2 and DC-D(c).

8. No REPORT is in $dcqueue(\langle p,q\rangle)$, by Claim 7 and DC-C.
9. $dcqueue(\langle q,p\rangle)$ consists solely of a FIND, by Claims 2 and 5 and DC-O.
10. $dcqueue(\langle p,q\rangle)$ is empty, by Claims 3 and 8 and DC-O.
11. $(p,q) \in subtree(f)$, by Claim 1 and COM-F.

Claims about s:

12. $(p,q) = core(f)$, by Claim 1.
13. Every node in $subtree(q)$ is in $testset(f)$, by Claim 6.
14. $dcqueue(\langle q,p\rangle)$ consists solely of FIND, by Claim 9.
15. $dcqueue(\langle p,q\rangle)$ consists solely of FIND, by Claim 10 and code.
16. $dcstatus(q) =$ unfind, by Claim 4.
17. $AfterMerge(p,q)$ is not enabled, by Claim 15.
18. $AfterMerge(q,p)$ is not enabled, by Claim 14.

DC-D: (a) by Claim 12. (b) by Claim 16. (c) by Claim 13.

DC-E: By Claim 15 (FIND in $dcqueue(\langle p,q\rangle)$ replaces $AfterMerge(p,q)$ being enabled).

DC-F: By Claim 15 (FIND in $dcqueue(\langle p,q\rangle)$ replaces $AfterMerge(p,q)$ being enabled).

DC-G: vacuously true by Claims 17 and 18.

DC-O: By Claim 15.

viii) π is Absorb(f,g).

(3c) $\mathcal{A}_4(s',\pi) = \pi$. Obviously π is enabled in $\mathcal{S}_4(s')$. Effects are mirrored in $\mathcal{S}_4(s)$ if we can show that $accmin(f)$ and $minlink(f)$ do not change.

Case 1: $p \in testset(f)$ in s'. By GC-C, $minlink(f) = nil$ in s'. By inspecting the code, a FIND message is in $subtree(f)$ in s, so $minlink(f) = nil$ in s also.

Suppose $accmin(f) = nil$ in s'. Then there is no external link of any $q \in nodes(f) - testset(f)$ in s'. Since $testset(f)$ does not change and no formerly internal links become external, $accmin(f) = nil$ in s also.

Suppose $accmin(f) = \langle q,r\rangle$ in s'. By GC-A, $level(f) \leq level(fragment(r))$. So by precondition, $fragment(r) \neq g$. Since all of $nodes(g)$ is added to $testset(f)$, there is no change to $nodes(f) - testset(f)$. Thus $accmin(f)$ is unchanged.

Case 2: $p \notin testset(f)$ in s'.

Claims about s':

1. $rootchanged(g) =$ true, by precondition.
2. $level(g) < level(f)$, by precondition.
3. $minlink(g) = \langle q, p \rangle \neq nil$, by precondition.
4. $fragment(p) = f$, by precondition.
5. $dcstatus(r) =$ unfind for all $r \in nodes(g)$, by Claim 3.
6. No FIND message is in $subtree(g)$, by Claim 3.
7. No REPORT message is headed toward $mw\text{-}root(g)$, by Claim 3.
8. $root(g) = mw\text{-}root(g)$, by Claim 3 and COM-A.
9. $wt(l) > wt(q,p)$ for all external links l of g, by Claim 3 and COM-A.
10. If $minlink(f) = \langle r,t \rangle$, then $level(fragment(t)) \geq level(f)$, by COM-A.
11. If $minlink(f) = \langle r,t \rangle$, then $g \neq fragment(t)$, by Claims 2 and 10.
12. If $accmin(f) = \langle r,t \rangle$, then $level(fragment(t)) \geq level(f)$, by GC-A.
13. If $accmin(f) = \langle r,t \rangle$, then $g \neq fragment(t)$, by Claims 2 and 12.

If $minlink(f) = nil$ in s', then obviously it is still nil in s. Suppose $minlink(f) = \langle r,t \rangle$ in s'. By Claims 5, 6, 7, 8 and 11 (and code), $minlink(f) = \langle r,t \rangle$ in s as well.

If $accmin(f) = \langle r,t \rangle$ in s', then it is unchanged in s by Claims 9 and 13. Suppose $accmin(f) = nil$ in s'. If this is because $minlink(f) \neq nil$ in s', then, since we just showed that $minlink(f)$ does not change, $accmin(f)$ is still nil in s. Suppose $accmin(f) = nil$ not because $minlink(f) = nil$, but because no node in $nodes(f) - testset(f)$ has an external link. But by the assumption for this case, $p \notin testset(f)$, yet it is in $nodes(f)$ by Claim 4, and $\langle p, q \rangle$ is an external link of p by Claim 3 and COM-A.

(3a) We consider two cases. First we prove some facts true in both cases.

Claims about s':

1. $rootchanged(g) =$ true, by precondition.
2. $level(g) < level(f)$, by precondition.
3. $minlink(g) = \langle q, p \rangle$, by precondition.
4. $p \in nodes(f)$, by precondition.
5. No REPORT is headed toward $root(g)$, by Claim 3.
6. No FIND is in $subtree(g)$, by Claim 3.

7. $dcstatus(r) =$ unfind, for all $r \in nodes(g)$, by Claim 3.
8. $\langle q,p \rangle$ is the minimum-weight external link of g, by Claim 3 and COM-A.
9. $testset(g) = \emptyset$, by Claim 3 and GC-C.
10. q is up-to-date, by Claim 9 and DC-N.
11. Following *bestlinks* from q leads toward and over $core(g)$, by Claim 10.
12. If REPORT is in $dcqueue(\langle r,t \rangle)$, then $inbranch(t) = \langle t,r \rangle$, for all $(r,t) \in subtree(g)$, by Claims 6 and 7 and DC-A(a) and (f).
13. If REPORT is in $dcqueue(\langle r,t \rangle)$ and $(r,t) = core(f)$, then $r = root(g)$, for all $(r,t) \in subtree(g)$, by Claim 5.
14. If REPORT is in $dcqueue(\langle r,t \rangle)$ and $(r,t) \neq core(f)$, then t is a child of r, for all $(r,t) \in subtree(g)$, by Claim 9 and DC-B(a).
15. If REPORT is in $dcqueue(\langle r,t \rangle)$, then (r,t) is not on the path between $root(g)$ and q, for all $(r,t) \in subtree(g)$, by Claims 3, 5, 8, 9 and DC-N.
16. No REPORT is headed toward q, by Claims 5, 14 and 15.
17. $dcqueue(\langle p,q \rangle)$ and $dcqueue(\langle q,p \rangle)$ are empty, by Claim 8 and DC-A(g), DC-B(a) and DC-D(a).

Case 1: $p \notin testset(f)$.

More claims about s':

18. $p \notin testset(f)$, by assumption.
19. $AfterMerge(r,t)$, where $p \in subtree(t)$, is not enabled, by Claim 18 and DC-G.
20. No FIND is headed toward p, by Claim 18 and DC-C(a).

DC-A: By Claim 12, vacuously true for any REPORT in old g. For a REPORT that could be in some $dcqueue(\langle r,t \rangle)$ with $p \in subtree(t)$: (e) by Claims 16 and 17.

DC-B: By Claim 16, change in location of core for nodes formerly in g is OK.

DC-D(a): by Claim 6, change in location of core for nodes formerly in g is OK. By Claim 20, it is OK not to add $nodes(g)$ to $testset(f)$.

DC-G: By Claim 19, vacuously true.

DC-H(a): By Claim 7.

DC-K: Choose any up-to-date node r in $nodes(f)$ in s. By Claims 7 and 11 and code, no node that is in $nodes(g)$ in s' is up-to-date in s. Thus r is in $nodes(f)$ in s', and is up-to-date.

(a) If $r = p$, then *findcount*(p) is changed (incremented by 1) if and only if the number of children of p that are not completed is changed (increased by 1). If $r \neq p$, then neither *findcount*(r) nor the number of children of r that are not completed is changed.

(b) Suppose $bestlink(r) = nil$ in s. Then the same is true in s'. By DC-K(b), $bestwt(r) = \infty$ and there is no external link of C_r in s'. In going to s, there is no change to $bestwt(r)$, and no internal links become external.

(c) Suppose $bestlink(r) \neq nil$ in s. Then the same is true in s'. Let l be the minimum-weight external link of C_r in s'. By DC-K(c), following *bestlinks* from r leads to l, $wt(l) = bestwt(r)$, and $level(h) \geq level(f)$, where $h = fragment(target(l))$, in s'. By the precondition on $level(g)$, $h \neq g$ in s', and thus l is still external in s. If $p \notin C_r$ in s', then C_r is unchanged in s, and the predicate is still true. Suppose $p \in C_r$ in s'. By COM-A, $wt(p, q)$ is less than the weight of any other external link of g, and thus $wt(l)$ is less than the weight of any external link of g in s'. Thus adding all the nodes of g to C_r in going to s does not falsify the predicate.

DC-O: By Claim 6, the former $core(g)$ is OK.

DC-N: Let l be the minimum-weight external link of f in s'. If $l \neq \langle p, q\rangle$, then $wt(l) < wt(p, q)$, and by Claim 8, $wt(l) < wt(l')$ for any external link l' of g. Thus, in s, l is still the minimum-weight external link of s, and DC-N is true in s.

Now suppose $l = \langle p, q\rangle$. By DC-N and Claim 18, p is up-to-date. But by DC-K(b) and (c), $bestlink(p) = \langle p, q\rangle$ and $level(f) \leq level(g)$, wich contradicts Claim 2.

Case 2: $p \in testset(f)$.

More claims about s':

21. $p \in testset(f)$, by assumption.
22. For all $\langle r, t\rangle$ such that $p \in subtree(r)$ and $inbranch(t) = \langle t, r\rangle$, no REPORT is in $dcqueue(\langle r, t\rangle)$, by Claim 21 and DC-A(e).
23. A FIND is headed toward p, or $dcstatus(p) =$ find, or $AfterMerge(r, t)$ is enabled, where $p \in subtree(t)$, by Claim 21 and DC-E.

DC-A(e): by Claim 22, the addition of uncompleted child q to p is OK.

DC-B: As in Case 1.

DC-D: As in Case 1.

DC-E: By Claim 23.

DC-G: By code, since all of *nodes*(g) is added to *testset*(f).

DC-H: By Claim 7.

DC-K: As in Case 1.

DC-M: By code, since *findcount*(p) is incremented.

DC-N: By code, since all of *nodes*(g) is added to *testset*(f).

DC-O: By Claim 17 and code. □

Let $P'_{DC} = (P'_{GC} \circ S_4) \wedge P_{DC}$.

Corollary 20: *P'_{DC} is true in every reachable state of DC.*

Proof: By Lemmas 1 and 19. □

4.2.5 NOT Simulates COM

This automaton refines on *COM* by implementing the level and core of a fragment with local variables *nlevel*(p) and *nfrag*(p) for each node p in the fragment, and with NOTIFY messages. When two fragments merge, a NOTIFY message is sent over one link of the new core, carrying the level and core of the newly created fragment. The action *AfterMerge*(p, q) adds such a NOTIFY message to the other link of the new core. A *ComputeMin*(f) action cannot occur until the source of *minlink*(f) has the correct *nlevel*, and the target of *minlink*(f) has an *nlevel* at least as big as the source's. The preconditions for *Absorb*(f, g) now include the fact that the level of fragment g must be less than the *nlevel* of the target of *minlink*(g). When an *Absorb*(f, g) occurs, a NOTIFY message is sent to the old fragment g, over the reverse link of *minlink*(g), with the *nlevel* and *nfrag* of the target of *minlink*(g).

Define automaton *NOT* (for "Notify") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *minlink*(f), a link of G or *nil*; and
- *rootchanged*(f), a Boolean.

For each node p, there are associated two variables:

- *nlevel*(p), a nonnegative integer; and
- *nfrag*(p), an edge of G or *nil*.

For each link $\langle p, q\rangle$, there are associated three variables:

- $nqueue_p(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;
- $nqueue_{pq}(\langle p, q\rangle)$, a FIFO queue of messages from p to q that are in the communication channel; and
- $nqueue_q(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at q to be processed.

The set of possible messages M is $\{\text{NOTIFY}(l, c) : l \geq 0, c \in E(G)\}$. The state also contains Boolean variables, *answered*(l), one for each $l \in L(G)$, and Boolean variable *awake*.

In the start state of *NOT*, *fragments* has one element for each node in $V(G)$; for fragment f corresponding to node p, *subtree*$(f) = \{p\}$, *minlink*(f) is the minimum-weight link adjacent to p, and *rootchanged*(f) is false. For each node p, *nlevel*$(p) = 0$ and *nfrag*$(p) = nil$. The message queues are empty. Each *answered*(l) is false and *awake* is false.

We say that a message m is *in subtree*(f) if m is in some *nqueue*$(\langle q, p\rangle)$ and $p \in nodes(f)$. A NOTIFY message is *headed toward* p if it is in *nqueue*$(\langle q, r\rangle)$ and $p \in subtree(r)$. The following are derived variables:

- For link $\langle p, q\rangle$, *nqueue*$(\langle p, q\rangle)$ is defined to be $nqueue_q(\langle p, q\rangle) \,||\, nqueue_{pq}(\langle p, q\rangle) \,||\, nqueue_p(\langle p, q\rangle)$.
- For fragment f, *level*$(f) = \max\{l : nlevel(p) = l$ for $p \in nodes(f)$, or a $\text{NOTIFY}(l, c)$ message is in *subtree*(f) for some $c\}$.

- For fragment f, $core(f) = nfrag(p)$ if $nlevel(p) = level(f)$ for some $p \in nodes(f)$, and $core(f) = c$, if a NOTIFY$(level(f), c)$ message is in $subtree(f)$.

As for the *DC* action *ReceiveFind*, $ReceiveNotify(\langle q,p\rangle, l, c)$ is only enabled if $AfterMerge(p,q)$ is not enabled, in order to make sure that q's side of the subtree is notified of the new information.

Input actions:

```
• $Start(p)$, $p \in V(G)$
      Effects:
        $awake$ := true
```

Output actions:

```
• $InTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
      Preconditions:
        $awake = $ true
        $(p,q) \in subtree(fragment(p))$ or $\langle p,q\rangle = minlink(fragment(p))$
        $answered(\langle p,q\rangle) = $ false
      Effects:
        $answered(\langle p,q\rangle) :=$ true

• $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
      Preconditions:
        $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$
        $answered(\langle p,q\rangle) = $ false
      Effects:
        $answered(\langle p,q\rangle) :=$ true
```

Internal actions:

```
• $ChannelSend(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$
      Preconditions:
        $m$ at head of $nqueue_p(\langle p,q\rangle)$
      Effects:
        dequeue($nqueue_p(\langle p,q\rangle)$)
        enqueue($m$, $nqueue_{pq}(\langle p,q\rangle)$)

• $ChannelRecv(\langle p,q\rangle, m)$, $\langle p,q\rangle \in L(G)$, $m \in M$
      Preconditions:
```

```
      $m$ at head of $nqueue_{pq}(\langle p,q\rangle)$
    Effects:
      dequeue($nqueue_{pq}(\langle p,q\rangle)$)
      enqueue($m, nqueue_q(\langle p,q\rangle)$)

• $ReceiveNotify(\langle q,p\rangle, l, c)$, $\langle q,p\rangle \in L(G)$, $l \geq 0$, $c \in E(G)$
    Preconditions:
      NOTIFY$(l,c)$ at head of $nqueue_p(\langle q,p\rangle)$
      $AfterMerge(p,q)$ not enabled
    Effects:
      dequeue($nqueue_p(\langle q,p\rangle)$)
      $nlevel(p) := l$
      $nfrag(p) := c$
      — let $S = \{\langle p,r\rangle : (p,r) \in subtree(fragment(p)), r \neq q\}$ —
      enqueue(NOTIFY$(l,c), nqueue_p(k)$) for all $k \in S$

• $ComputeMin(f)$, $f \in fragments$
    Preconditions:
      $minlink(f) = nil$
      $\langle p,q\rangle$ is the minimum-weight external link of $f$
      $nlevel(p) = level(f)$
      $level(f) \leq nlevel(q)$
    Effects:
      $minlink(f) := l$

• $ChangeRoot(f)$, $f \in fragments$
    Preconditions:
      $awake$ = true
      $rootchanged(f)$ = false
      $minlink(f) \neq nil$
    Effects:
      $rootchanged(f)$ := true

• $Merge(f,g)$, $f,g \in fragments$
    Preconditions:
      $f \neq g$
      $rootchanged(f) = rootchanged(g)$ = true
      $minedge(f) = minedge(g)$
    Effects:
      add a new element $h$ to $fragments$
```

```
        $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
        $minlink(h) := nil$
        $rootchanged(h) :=$ false
        — let $(p, q) = minedge(f)$ —
        enqueue($\text{NOTIFY}(nlevel(p) + 1, (p, q)), nqueue_p(\langle p, q\rangle)$)
        delete $f$ and $g$ from $fragments$
```

- $AfterMerge(p,q)$, $p, q \in V(G)$

```
    Preconditions:
        $(p, q) = core(fragment(p))$
        $\text{NOTIFY}(nlevel(p) + 1, (p, q))$ message in $nqueue(\langle q, p\rangle)$
        no $\text{NOTIFY}(nlevel(p) + 1, (p, q))$ message in $nqueue(\langle p, q\rangle)$
        $nlevel(q) \neq nlevel(p) + 1$
    Effects:
        enqueue($\text{NOTIFY}(nlevel(p) + 1, (p, q)), nqueue_p(\langle p, q\rangle)$)
```

- $Absorb(f, g)$, $f, g \in fragments$

```
    Preconditions:
        $rootchanged(g) =$ true
        — let $\langle q, p\rangle = minlink(g)$ —
        $level(g) < nlevel(p)$
        $fragment(p) = f$
    Effects:
        $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
        enqueue($\text{NOTIFY}(nlevel(p), nfrag(p)), nqueue_p(\langle p, q\rangle)$)
        delete $g$ from $fragments$
```

Define the following predicates on states of NOT. (All free variables are universally quantified.)

- NOT-A: $core(f)$ is well-defined. (I.e., the set of all c such that a $\text{NOTIFY}(level(f), c)$ is in $subtree(f)$ or some $p \in nodes(f)$ has $nlevel(p) = level(f)$ and $nfrag(p) = c$, has exactly one element.)

- NOT-B: If $q \in subtree(p)$, then $nlevel(q) \leq nlevel(p)$.

- NOT-C: If $(p, q) = core(f)$, then $nlevel(p) \geq level(f) - 1$.

- NOT-D: If $minlink(f) = \langle p, q\rangle$, then $nlevel(p) = level(f) \leq nlevel(q)$.

- NOT-E: If $nfrag(p) = core(fragment(p))$, then $nlevel(p) = level(fragment(p))$.

- NOT-F: Either $nlevel(p) = 0$ and $nfrag(p) = nil$, or else $nlevel(p) > 0$ and $nfrag(p) \in subtree(fragment(p))$.

- NOT-G: If $nlevel(p) < level(fragment(p))$, then either a NOTIFY$(level\ (fragment(p)), core(fragment(p)))$ message is headed toward p, or else $AfterMerge\ (q, r)$ is enabled, where $p \in subtree(r)$.

- NOT-H: If a NOTIFY(l, c) message is in $nqueue(\langle q, p\rangle)$, then
  (a) $nlevel(p) < l$;
  (b) if $(p, q) \neq core(fragment(p))$, then $nlevel(q) \geq l$;
  (c) if $c = core(fragment(p)))$ then $l = level(fragment(p))$;
  (d) if NOTIFY(l', c') is ahead of the NOTIFY(l, c) in $nqueue(\langle q, p\rangle)$, then $l' < l$;
  (e) p is a child of q, or $(p, q) = core(fragment(p))$;
  (f) if $(p, q) = core(fragment(p))$, then $l = level(fragment(p))$;
  (g) $c \in subtree(fragment(p))$; and
  (h) $l > 0$.

Let P_{NOT} be the conjunction of NOT-A through NOT-H.

In order to show that NOT simulates COM, we define an abstraction mapping $\mathcal{M}_5 = (\mathcal{S}_5, \mathcal{A}_5)$ from NOT to COM. Define the function $\mathcal{S}_5$ from $states(NOT)$ to $states(COM)$ by simply ignoring the message queues, and mapping the derived variables $level(f)$ and $core(f)$ in the NOT state to the (non-derived) variables $level(f)$ and $core(f)$ in the COM state. Define the function $\mathcal{A}_5$ as follows. Let s be a state of NOT and π an action of NOT enabled in s.

- If $\pi = ChannelSend(k, m)$, $ChannelRecv(k, m)$, $ReceiveNotify(k, l, c)$, or $AfterMerge(p, q)$, then $\mathcal{A}_5(s, \pi)$ is empty.

- For all other values of π, $\mathcal{A}_5(s, \pi) = \pi$.

The following predicates are true in any state of NOT satisfying $(P'_{COM} \circ \mathcal{S}_5) \wedge P_{NOT}$. Recall that $P'_{COM} = (P_{S1} \circ \mathcal{S}_1) \wedge P_{COM}$. If $P'_{COM}(\mathcal{S}_5(s))$ is true, then the COM predicates are true in $\mathcal{S}_5(s)$, and the S1 predicates are true in $\mathcal{S}_1(\mathcal{S}_5(s))$. Thus, these predicates follow from P_{NOT}, together with the HI and COM predicates.

- NOT-I: If $p = minnode(f)$, then no NOTIFY message is headed toward p.

- NOT-J: For all p, at most one NOTIFY(l, c) message is headed toward p, for a fixed l.

Lemma 21: *NOT simulates COM via* $\mathcal{M}_5$, P_{NOT}, *and* P'_{COM}.

Proof: By inspection, the types of NOT, COM, $\mathcal{M}_5$, and P_{NOT} are correct. By Corollary 14, P'_{COM} is a predicate true in every reachable state of COM.

(1) Let s be in $start(NOT)$. Obviously P_{NOT} is true in s and $\mathcal{S}_5(s)$ is in $start(COM)$.

(2) Obviously, $\mathcal{A}_5(s,\pi)|ext(COM) = \pi|ext(NOT)$.

(3) Let (s',π,s) be a step of NOT such that P'_{COM} is true of $\mathcal{S}_5(s')$ and P_{NOT} is true of s'. Below, we only show (3a) for those predicates that are not obviously true in s.

i) π is Start(p), InTree(l), NotInTree(l), or ChangeRoot(f). $\mathcal{A}_5(s',\pi) = \pi$. Obviously, $\mathcal{S}_5(s')\pi\mathcal{S}_5(s)$ is an execution fragment of COM, and P_{NOT} is true in s.

ii) π is ChannelSend(l,m) or ChannelRecv(l,m). $\mathcal{A}_5(s',\pi)$ is empty. Obviously, $\mathcal{S}_5(s') = \mathcal{S}_5(s)$, and P_{NOT} is true in s.

iii) π is ReceiveNotify($\langle$q,p$\rangle$,l,c). Let $f = fragment(p)$.

(3b) $\mathcal{A}_5(s',\pi)$ is empty. To show that $\mathcal{S}_5(s) = \mathcal{S}_5(s')$, we only need to show that $level(f)$ and $core(f)$ don't change. By NOT-H(a), $nlevel(p) < l$ in s', and thus $nlevel(p) \neq level(f)$. So changing $nlevel(p)$ is OK. Also, since $nlevel(p)$ and $nfrag(p)$ are set to l and c, removing the NOTIFY(l,c) from $nqueue(\langle q,p\rangle)$ is OK.

(3a) NOT-A: By code.

NOT-B: By NOT-B, $nlevel(q) \leq nlevel(r)$ for all r such that $q \in subtree(r)$ in s'. By NOT-H(b), if $(p,q) \neq core(f)$, then $nlevel(q) \geq l$ in s'. Since $nlevel(p) = l$ in s, the predicate is true.

NOT-C: Since this predicate is true in s' and fact that $nlevel(p)$ increases.

NOT-D: As argued in (3b), $nlevel(p) < l \leq level(f)$. By NOT-D, $p \neq minnode(f)$ in s', or in s. Suppose $p = target(minlink(g))$ in s', for some g. Since $nlevel(p)$ increases in going from s' to s, the predicate is still true in s.

NOT-E: By NOT-H(c), $c = core(f)$ implies that $l = level(f)$ in s'. So in s, $c = nfrag(p) = core(f)$ implies that $l = nlevel(p) = level(f)$.

NOT-F: By NOT-H(g), $c \neq nil$, and by NOT-H(h), $l > 0$ in s'. Thus in s, $c = nfrag(p) \neq nil$ and $l = nlevel(p) \neq 0$.

NOT-G: The NOTIFY(l,c) message removed from $nqueue(\langle q,p\rangle)$ is replaced by the NOTIFY(l,c) messages added to $nqueue(\langle p,r\rangle)$, for all $\langle p,r\rangle \in S$.

NOT-H: Suppose NOTIFY(l,c) is added to $nqueue(p,r)$ in s. (I.e., $\langle p,r\rangle \in S$.)

Claims about s':

1. NOTIFY(l,c) is at head of $nqueue(\langle q,p\rangle)$, by precondition.
2. $p \in subtree(q)$ or $(p,q) = core(f)$, by Claim 1 and NOT-H(e).
3. $r \in subtree(p)$, by Claim 2 and definition of S.
4. $nlevel(r) \leq nlevel(p)$, by Claim 3 and NOT-B.
5. $nlevel(p) < l$, by Claim 1 and NOT-H(a).
6. If NOTIFY(l',c') is in $nqueue(\langle p,r\rangle)$, then $l' < l$, by Claims 3 and 5 and NOT-H(b).
7. $nlevel(r) < l$, by Claims 4 and 5.

(a) by Claim 7. (b) by Claim 3. (d) by Claim 7. (e) by Claim 3. (f) vacuously true by Claim 3. (c), (g) and (h) since the same is true for the NOTIFY(l,c) in $nqueue(\langle q,p\rangle)$ in s'.

iv) π is ComputeMin(f).

(3c) $\mathcal{A}_5(s',\pi) = \pi$. Obviously π is enabled in $\mathcal{S}_5(s')$, since by definition $nlevel(q) \leq level(fragment(q))$. The effects are obviously mirrored in $\mathcal{S}_5(s)$.

(3a) By the preconditions, NOT-D is true in s. No other predicate is affected.

v) π is Merge(f,g).

(3c) $\mathcal{A}_5(s',\pi) = \pi$. Obviously π is enabled in $\mathcal{S}_5(s')$. To show that its effects are mirrored in $\mathcal{S}_5(s)$, we show that $level(h)$ and $core(h)$ are correct. Let $minlink(f) = \langle p,q\rangle$ and $l = level(f)$ in s'.

Claims about s':

1. $minedge(f) = minedge(g)$, by precondition.
2. $level(g) = l$, by Claim 1 and COM-A.
3. $rootchanged(f) =$ true, by precondition.
4. $minlink(f) \neq nil$, by Claim 3 and COM-B.
5. $nlevel(p) = l$, by Claim 4 and NOT-D.
6. $nlevel(r) \leq l$ for all $r \in nodes(f)$, by definition of $level(f)$.
7. If NOTIFY(m,c) is in $subtree(f)$, then $m \leq l$, by definition of $level(f)$.

8. $rootchanged(g) =$ true, by precondition.
9. $minlink(g) \neq nil$, by Claim 8 and COM-B.
10. $nlevel(q) = l$, by Claims 2 and 9 and NOT-D.
11. $nlevel(r) \leq l$ for all $r \in nodes(g)$, by definition of $level(g)$.
12. If NOTIFY(m, c) is in $subtree(g)$, then $m \leq l$, by definition of $level(g)$.
13. $\langle p, q \rangle$ is an external link of f, by COM-A.
14. $nqueue(\langle p, q \rangle)$ and $nqueue(\langle q, p \rangle)$ are empty, by Claim 13 and NOT-H(e).

Claims about s:

15. $nlevel(r) < l + 1$, for all $r \in nodes(h)$, by Claims 6 and 11 and code.
16. The only NOTIFY message in $subtree(h)$ with level greater than l is the NOTIFY$(l + 1, (p, q))$ message added to $nqueue(\langle p, q \rangle)$, by Claims 7, 12 and 14 and code.
17. $level(h) = l + 1$, by Claims 15 and 16.
18. $core(h) = (p, q)$, by Claims 15 and 16.

Claims 17 and 18 give the result.

(3a) Only fragment h needs to be checked.

NOT-A: By Claims 15 and 16.

NOT-B: As argued in the proof of NOT-I, $nlevel(r) = l$ for all r on the path from $core(f)$ to p, and all r on the path from $core(g)$ to q. Since these are the only nodes affected by the change of core, the predicate is still true in s.

NOT-C: By Claims 5, 10 and 17.

NOT-D: vacuously true since $minlink(h) = nil$ by code.

NOT-E: By NOT-F and Claim 13, $nfrag(r) \neq (p, q)$ for all r in $nodes(f)$ or $nodes(g)$. So the predicate is vacuously true.

NOT-F: No relevant change.

NOT-G: If r is in $nodes(g)$ in s', the predicate is true in s because of Claims 17 and 18 and the NOTIFY$(l+1, (p, q))$ added to $nqueue(\langle p, q \rangle)$ in s. If r is in $nodes(f)$ in s', then $AfterMerge(q, p)$ is enabled in s, by code and Claims 5, 10, 14 and 18.

NOT-H for the NOTIFY$(l + 1, (p, q))$ added to $nqueue(\langle p, q \rangle)$: (a) $nlevel(q) < l+1$, by Claim 15. (b) By Claim 18. (c) By Claim 17. (d) Vacuously true by Claim

14. (e) By Claim 18. (f) By Claims 17 and 18. (g) By code. (h) By COM-F, $l \geq 0$, so $l + 1 > 0$.

NOT-H for any NOTIFY(l', c') message in *subtree*(f) in s' (similar argument for g): (a), (d), (g) and (h) No relevant change.

(b) Suppose the message is in a link of *core*$(f) = (r, t)$. Suppose $p \in$ *subtree*(t). By NOT-I, the message is not in *nqueue*$(\langle r, t\rangle)$. As argued in the proof of NOT-I, *nlevel*$(t) = l$. If the message is in *nqueue*$(\langle t, r\rangle)$, then, since $l' \leq l$, the predicate is true in s.

(c) By Claim 13 and NOT-H(g), $c' \neq (p, q)$, so the predicate is vacuously true in s.

(e) The only nodes for which the subtree relationship changes are those along the path from *core*(f) to p. By NOT-I, there is no NOTIFY message in this path.

(f) Vacuously true, by Claim 18.

vi) π is AfterMerge(p,q). Let $f =$ *fragment*(p).

(3b) $\mathcal{A}_5(s')$ is empty. Obviously $\mathcal{S}_5(s') = \mathcal{S}_5(s)$.

(3a) Let $l =$ *nlevel*$(p) + 1$ and $c = (p, q)$.

NOT-A: Obvious.

NOT-B, C, D, and E: No relevant changes.

NOT-G: The NOTIFY(l, c) message added to *nqueue*$(\langle p, q\rangle)$ in s compensates for the fact that *AfterMerge*(p, q) goes from enabled in s' to disabled in s.

NOT-H: Let $c = (p, q)$ and $l =$ *nlevel*$(p) + 1$. Consider the NOTIFY(l, c) added to *nqueue*$(\langle p, q\rangle)$.

1. $(p, q) =$ *core*(f), by precondition.
2. NOTIFY(l, c) is in *nqueue*$(\langle q, p\rangle)$, by precondition.
3. No NOTIFY(l, c) is in *nqueue*$(\langle p, q\rangle)$, by precondition.
4. *nlevel*$(q) \neq l$, by precondition.
5. $l =$ *level*(f), by Claims 1 and 2 and NOT-H(f).
6. *nlevel*$(q) < l$, by Claims 4 and 5.
7. If NOTIFY(l', c') is in *nqueue*$(\langle p, q\rangle)$, then $l' = l$, by Claims 1 and 5 and NOT-H(d).

8. If NOTIFY(l', c') is in *nqueue*$(\langle p, q\rangle)$, then $c' = c$, by Claim 7 and NOT-A.
9. No NOTIFY is in *nqueue*$(\langle p, q\rangle)$, by Claims 3, 7 and 8.
10. *nlevel*$(p) \geq 0$, by NOT-F.

(a) by Claim 6. (b) vacuously true, by Claim 1. (c) by Claim 5. (d) by Claim 9. (e) by Claim 1. (f) by Claim 5. (g) by Claim 1 and COM-F. (h) by Claim 10.

vii) π is Absorb(f,g).

(3c) $\mathcal{A}(s', \pi) = \pi$.

Claims about s':

1. *rootchanged*(g) = true, by precondition.
2. *level*$(g) <$ *nlevel*(p), by precondition.
3. *fragment*$(p) = f$, by precondition.
4. *nlevel*$(p) \leq$ *level*(f), by Claim 3 and definition of level.
5. *nlevel*$(r) \leq$ *level*(g), for all $r \in$ *nodes*(g), by definition of level.
6. If NOTIFY(l, c) is in *subtree*(g), then $l \leq$ *level*(g), by definition of level.
7. $\langle q, p\rangle$ is an external link of g, by COM-A.
8. *nqueue*$(\langle p, q\rangle)$ and *nqueue*$(\langle q, p\rangle)$ are empty, by Claim 7 and NOT-H(e).

By Claim 4, π is enabled in $\mathcal{S}_5(s')$. The effects of π are mirrored in $\mathcal{S}_5(s)$ if *core*(f) and *level*(f) are unchanged; by code and Claims 6, 7 and 8, they are unchanged.

(3a) Let $l =$ *nlevel*(p) and $c =$ *nfrag*(p) in s'.

More claims about s':

9. $f \neq g$, by Claims 7 and 3.
10. *level*$(f) > 0$, by Claims 2 and 3 and COM-F.
11. *core*$(f) \in$ *subtree*(f), by Claim 10 and COM-F.
12. *nfrag*$(r) \neq$ *core*(f), for all $r \in$ *nodes*(g), by Claim 11 and NOT-F.
13. *nlevel*$(q) \leq$ *level*(g), by definition.
14. *nfrag*$(p) \in$ *subtree*(f), by Claims 2 and 10 and NOT-F.

NOT-A: by code and Claims 6, 7 and 8.

NOT-B: Same argument as for *Merge*(f, g).

NOT-D: No relevant changes.

NOT-E: By Claim 12, vacuously true for nodes formerly in $nodes(g)$.

NOT-F: No relevant changes.

NOT-G: Suppose $nlevel(p) = level(f)$ in s'. By code, in s there is a NOTIFY$(level(f), c)$ message headed toward every node formerly in $nodes(g)$.

Suppose $nlevel(p) \neq level(f)$ in s'. By NOT-G, either a NOTIFY$(level(f), c)$ message is headed toward p in s', and thus is headed toward all nodes formerly in $nodes(g)$ in s, or $AfterMerge(r, t)$ is enabled in s' with $p \in subtree(t)$, and thus in s, $AfterMerge(r, t)$ is still enabled and every node formerly in $nodes(g)$ is in $subtree(t)$.

NOT-H for the NOTIFY(l, c) added to $nqueue(\langle p, q\rangle)$: (a) by Claims 2 and 12. (b) by code. (c) by NOT-E. (d) vacuously true by Claim 8. (e) q is a child of p, by Claim 11. (f) vacuously true, by Claim 11. (g) by Claim 14. (h) by Claims 2 and 10.

NOT-H for any NOTIFY(l', c') in $subtree(g)$ in s': (a), (d), (g) and (h): no relevant change. (b) and (e) same argument as for $Merge(f, g)$. (c) vacuously true, by Claim 11. (f) vacuously true, by code. □

Let $P'_{NOT} = (P_{COM} \circ S_5) \wedge P_{NOT}$.

Corollary 22: *P'_{NOT} is true in every reachable state of NOT.*

Proof: By Lemmas 1 and 21. □

4.2.6 CON Simulates COM

This automaton concentrates on what happens after *minlink*(f) is identified, until fragment f merges or is absorbed, i.e., the *ChangeRoot*(f, g), *Merge*(f, g) and *Absorb*(g, f) actions are broken down into a series of actions, involving message-passsing. The variable *rootchanged*(f) is now derived. As soon as *ComputeMin*(f) occurs, the node adjacent to the core closest to *minlink*(f) sends a CHANGEROOT message on its outgoing link that leads to *minlink*(f). A chain of such messages makes its way to the source of *minlink*(f), which then sends a CONNECT(*level*(f)) message over *minlink*(f). The presence of a CONNECT message in *minlink*(f) means that *rootchanged*(f) is true. Thus, the *ChangeRoot*(f) action is only needed for fragments f consisting of a single node. Two fragments can merge when they have the same *minedge* and a CONNECT message is in both its links; the result is that one of the CONNECT messages is removed. The action *AfterMerge*(p, q) removes the other CONNECT message from the new core. (A delicate point is that *ComputeMin*(f) cannot occur until the appropriate *AfterMerge*(p, q) has, in order to make sure old CONNECT messages are not hanging around.) *Absorb*(f, g) can occur if there is a CONNECT(l) message in *minlink*(g), and *minlink*(g) points to a fragment whose level is greater than l.

Define automaton *CON* (for "Connect") as follows.

The state consists of a set *fragments*. Each element f of the set is called a *fragment*, and has the following components:

- *subtree*(f), a subgraph of G;
- *core*(f), an edge of G or *nil*;
- *level*(f), a nonnegative integer; and
- *minlink*(f), a link of G or *nil*.

For each link $\langle p, q\rangle$, there are associated three variables:

- $cqueue_p(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;
- $cqueue_{pq}(\langle p, q\rangle)$, a FIFO queue of messages from p to q that are in the communication channel; and
- $cqueue_q(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at q to be processed.

The set of possible messages M is $\{\text{CONNECT}(l) : l \geq 0\} \cup \{\text{CHANGEROOT}\}$. The state also contains Boolean variables, $answered(l)$, one for each $l \in L(G)$, and Boolean variable $awake$.

In the start state of COM, $fragments$ has one element for each node in $V(G)$; for fragment f corresponding to node p, $subtree(f) = \{p\}$, $core(f) = nil$, $level(f) = 0$, and $minlink(f)$ is the minimum-weight link adjacent to p. The message queues are empty. Each $answered(l)$ is false and $awake$ is false.

The derived variable $cqueue(\langle p,q\rangle)$ is $cqueue_q(\langle p,q\rangle) \mid\mid cqueue_{pq}(\langle p,q\rangle) \mid\mid c$-$queue_p(\langle p,q\rangle)$. For each fragment f, we define the derived Boolean variable $rootchanged(f)$ to be true if and only if a CONNECT message is in $cqueue(\langle p,q\rangle)$, for some external link $\langle p,q\rangle$ of f. Derived variable $tominlink(p)$ is defined to be the link $\langle p,q\rangle$ such that (p,q) is on the path in $subtree(fragment(p))$ from p to $minnode(fragment(p))$.

Message m is defined to be *in* $subtree(f)$ if m is in $cqueue(\langle q,p\rangle)$ and $p \in nodes(f)$.

Input actions:

- $Start(p)$, $p \in V(G)$
  - Effects:
    - $awake :=$ true

Output actions:

- $InTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
  - Preconditions:
    - $awake =$ true
    - $(p,q) \in subtree(fragment(p))$ or $\langle p,q\rangle = minlink(fragment(p))$
    - $answered(\langle p,q\rangle) =$ false
  - Effects:
    - $answered(\langle p,q\rangle) :=$ true

- $NotInTree(\langle p,q\rangle)$, $\langle p,q\rangle \in L(G)$
  - Preconditions:
    - $fragment(p) = fragment(q)$ and $(p,q) \notin subtree(fragment(p))$
    - $answered(\langle p,q\rangle) =$ false
  - Effects:
    - $answered(\langle p,q\rangle) :=$ true

Internal actions:

- *ChannelSend*($\langle p,q\rangle, m$), $\langle p,q\rangle \in L(G)$, $m \in M$
  Preconditions:
    m at head of $cqueue_p(\langle p,q\rangle)$
  Effects:
    dequeue($cqueue_p(\langle p,q\rangle)$)
    enqueue(m, $cqueue_{pq}(\langle p,q\rangle)$)

- *ChannelRecv*($\langle p,q\rangle, m$), $\langle p,q\rangle \in L(G)$, $m \in M$
  Preconditions:
    m at head of $cqueue_{pq}(\langle p,q\rangle)$
  Effects:
    dequeue($cqueue_{pq}(\langle p,q\rangle)$)
    enqueue(m, $cqueue_q(\langle p,q\rangle)$)

- *ComputeMin*(f), $f \in fragments$
  Preconditions:
    $minlink(f) = nil$
    l is the minimum-weight external link of $subtree(f)$
    $level(f) \leq level(fragment(target(l)))$
    no CONNECT message is in $cqueue(k)$, for any internal link k of f
  Effects:
    $minlink(f) := l$
    — let $p = root(f)$ —
    if $p \neq minnode(f)$ then enqueue(CHANGEROOT,$cqueue_p(tominlink(p))$)
    else enqueue(CONNECT($level(f)$), $cqueue_p(minlink(f))$)

- *ReceiveChangeRoot*($\langle q,p\rangle$), $\langle q,p\rangle \in L(G)$
  Preconditions:
    CHANGEROOT at head of $cqueue_p(\langle q,p\rangle)$
  Effects:
    dequeue($cqueue_p(\langle q,p\rangle)$)
    — let $f = fragment(p)$ —
    if $p \neq minnode(f)$ then enqueue(CHANGEROOT,$cqueue_p(tominlink(p))$)
    else enqueue(CONNECT($level(f)$), $cqueue_p(minlink(f))$)

- *ChangeRoot*(f), $f \in fragments$
  Preconditions:
    $awake$ = true

$rootchanged(f) =$ false
$subtree(f) = \{p\}$
Effects:
enqueue(CONNECT(0), $cqueue_p(minlink(f))$)

- $Merge(f, g)$, $f, g \in fragments$
  Preconditions:
  CONNECT(l) in $cqueue(\langle p, q\rangle)$, $\langle p, q\rangle$ external link of f
  CONNECT(l) at head of $cqueue_p(\langle q, p\rangle)$, $\langle q, p\rangle$ external link of g
  Effects:
  dequeue($cqueue_p(\langle q, p\rangle)$)
  add a new element h to *fragments*
  $subtree(h) := subtree(f) \cup subtree(g) \cup minedge(f)$
  $core(h) := minedge(f)$
  $level(h) := level(f) + 1$
  $minlink(h) := nil$
  delete f and g from *fragments*

- $AfterMerge(p, q)$, $p, q \in V(G)$
  Preconditions:
  $fragment(p) = fragment(q)$
  CONNECT(l) at head of $cqueue_p(\langle q, p\rangle)$
  Effects:
  dequeue($cqueue_p(\langle q, p\rangle)$)

- $Absorb(f, g)$, $f, g \in fragments$
  Preconditions:
  — let $p = target(minlink(g))$ —
  CONNECT(l) at head of $cqueue_p(minlink(g))$
  $l < level(f)$
  $f = fragment(p)$
  Effects:
  dequeue($cqueue_p(minlink(g))$)
  $subtree(f) := subtree(f) \cup subtree(g) \cup minedge(g)$
  delete g from *fragments*

Define the following predicates on states of *CON*. (All free variables are universally quantified.)

- CON-A: If $awake =$ false, then $cqueue(\langle q, p\rangle)$ is empty.

- CON-B: If *rootchanged*(f) = false and *minlink*$(f) \neq$ *nil*, then either *subtree*(f) $= \{p\}$, or else *minnode*$(f) \neq$ *root*(f) and there is exactly one CHANGEROOT message in *subtree*(f).

- CON-C: If a CHANGEROOT message is in *cqueue*$(\langle q, p\rangle)$, then *minlink*$(f) \neq$ *nil*, *rootchanged*(f) = false, p is a child of q, and *minnode*$(f) \in$ *subtree*(p), where $f =$ *fragment*(p).

- CON-D: If a CONNECT(l) message is in *cqueue*(k), where k is an external link of f, then $k =$ *minlink*(f), $l =$ *level*(f), and only one CONNECT message is in *cqueue*(k).

- CON-E: If a CONNECT(l) message is in *cqueue*$(\langle p, q\rangle)$, where $\langle p, q\rangle$ is an internal link of f, then $(p, q) =$ *core*(f), $l <$ *level*(f), and only one CONNECT message is in *cqueue*$(\langle p, q\rangle)$.

- CON-F: If *minlink*$(f) \neq$ *nil*, then no CONNECT message is in *cqueue*(k), for any internal link k of f.

Let P_{CON} be the conjunction of CON-A through CON-F.

In order to show that CON simulates COM, we define an abstraction mapping $\mathcal{M}_6 = (\mathcal{S}_6, \mathcal{A}_6)$ from CON to COM.

Define the function $\mathcal{S}_6$ from *states*(CON) to *states*(COM) by simply ignoring the message queues, and mapping the derived variables *rootchanged*(f) in the CON state to the (non-derived) variables *rootchanged*(f) in the COM state.

Define the function $\mathcal{A}_6$ as follows. Let s be a state of CON and π an action of CON enabled in s. If the minimum-weight external link of f is adjacent to *core*(f), then *ComputeMin*(f) causes *ComputeMin*(f), immediately followed by *ChangeRoot*(f), to be simulated in COM. Otherwise, *ChangeRoot*(f) is simulated when the source of *minlink*(f) receives a CHANGEROOT message.

- If π = *ChannelSend*$(\langle p, q\rangle, m)$, *ChannelRecv*$(\langle p, q\rangle, m)$, or *AfterMerge*(p, q), then $\mathcal{A}_6(s, \pi)$ is empty.

- If π = *ComputeMin*(f) and *mw-root*(f) = *mw-minnode*(f) in s, then $\mathcal{A}_6(s, \pi)$ = *ComputeMin*(f) t *ChangeRoot*(f), where t is identical to $\mathcal{S}_6(s)$ except that *minlink*(f) equals the minimum-weight external link of f in t.

- If $\pi = ComputeMin(f)$ and $mw\text{-}root(f) \neq mw\text{-}minnode(f)$ in s, then $\mathcal{A}_6(s, \pi) = ComputeMin(f)$.

- If $\pi = ReceiveChangeRoot(\langle q,p \rangle)$ and $p = minnode(fragment(p))$ in s, then $\mathcal{A}_6(s, \pi) = ChangeRoot(fragment(p))$.

- If $\pi = ReceiveChangeRoot(\langle q,p \rangle)$ and $p \neq minnode(fragment(p))$ in s, then $\mathcal{A}_6(s, \pi)$ is empty.

- For all other values of π, $\mathcal{A}_6(s, \pi) = \pi$.

Recall that $P'_{COM} = (P_{HI} \circ \mathcal{S}_1) \wedge P_{COM}$. If $P'_{COM}(\mathcal{S}_6(s))$ is true, then the COM predicates are true in $\mathcal{S}_6(s)$, and the HI predicates are true in $\mathcal{S}_1(\mathcal{S}_6(s))$.

Lemma 23: *CON simulates COM via $\mathcal{M}_6$, P_{CON}, and P'_{COM}.*

Proof: By inspection, the types of CON, COM, $\mathcal{M}_6$, and P_{CON} are correct. By Corollary 14, P'_{COM} is a predicate true in every reachable state of COM.

(1) Let s be in $start(CON)$. Obviously P_{CON} is true in s and $\mathcal{S}_6(s)$ is in $start(COM)$.

(2) Obviously, $\mathcal{A}_6(s, \pi)|ext(COM) = \pi|ext(CON)$.

(3) Let (s', π, s) be a step of CON such that P'_{COM} is true of $\mathcal{S}_6(s')$ and P_{CON} is true of s'. Below we show (3a) only for those predicates that are not obviously true in s.

i) π is Start(p), InTree(l) or NotInTree(l). $\mathcal{A}_6(s', \pi) = \pi$. Obviously, $\mathcal{S}_6(s')\pi\mathcal{S}_6(s)$ is an execution fragment of COM, and P_{CON} is true in s.

ii) π is ChannelSend($\langle$q,p$\rangle$,m) or ChannelRecv($\langle$q,p$\rangle$,m). $\mathcal{A}_6(s', \pi)$ is empty. Obviously, $\mathcal{S}_6(s') = \mathcal{S}_6(s)$, and P_{CON} is true in s.

iii) π is ComputeMin(f).

Case 1: $mw\text{-}root(f) \neq mw\text{-}minnode(f)$ in s'.

(3b) $\mathcal{A}_6(s', \pi) = \pi$. Obviously $\mathcal{S}_6(s')\pi\mathcal{S}_6(s)$ is an execution fragment of COM.

(3a) *Claims about s':*

1. $minlink(f) = nil$, by precondition.

2. l is the minimum-weight external link of f, by precondition.
3. $level(f) \leq level(fragment(target(l)))$, by precondition.
4. No CONNECT message is in $cqueue(k)$, for any internal link k of f, by precondition.
5. $p = mw\text{-}root(f)$, by assumption.
6. $p \neq mw\text{-}minnode(f)$, by assumption.
7. $awake$ = true, by Claim 1 and COM-C.
8. No CHANGEROOT mesage is in $subtree(f)$, by Claim 1 and CON-C.
9. $mw\text{-}minnode(f) \in subtree(p)$, by Claim 5.
10. $rootchanged(f)$ = false, by Claim 1 and COM-B.

Claims about s:

11. $minlink(f) = l$, the minimum-weight external link of f, by Claim 2 and code.
12. $level(f) \leq level(fragment(target(l)))$, by Claim 3.
13. $p = root(f)$, by Claims 5 and 11.
14. $p \neq minnode(f)$, by Claims 6 and 11.
15. $awake$ = true, by Claim 7.
16. Exactly one CHANGEROOT message is in $subtree(f)$, by Claim 8 and code.
17. $minnode(f) \in subtree(p)$, by Claims 9 and 11.
18. $rootchanged(f)$ = false, by Claim 10.
19. No CONNECT message is in $cqueue(k)$, for any internal link k of f, by Claim 4.

CON-A is true by Claim 15. CON-B is true by Claims 13, 14, and 16. CON-C is true by definition of *tominlink*, Claims 17, 18 and 11. CON-D and CON-E are true since no relevant changes are made. CON-F is true by Claim 19.

Case 2: $mw\text{-}root(f) = mw\text{-}minnode(f)$ in s'.

(3b) $\mathcal{A}_6(s', \pi) = \pi\ t\ ChangeRoot(f)$, where t is identical to $\mathcal{S}_6(s')$ except that $minlink(f)$ equals the minimum-weight external link of f in t.

Claims about s':

1. $minlink(f) = nil$, by precondition.
2. l is the minimum-weight external link of f, by precondition.
3. $level(f) \leq level(fragment(target(l)))$, by precondition.
4. $awake$ = true, by Claim 1 and COM-C.
5. $rootchanged(f)$ = false, by Claim 1 and COM-B.

Claims about t:

6. *minlink*(f) is the minimum-weight external link of f, by definition of t.
7. *awake* = true, by Claim 4.
8. *rootchanged*(f) = false, by Claim 5.

Claims about s:

9. *minlink*(f) is the minimum-weight external link of f, by code.
10. A CONNECT message is in *cqueue*(*minlink*(f)), by code.
11. *rootchanged*(f) = true, by Claims 9 and 10.

By Claims 1, 2 and 3, π is enabled in $\mathcal{S}_6(s')$. By Claim 6 (and definition of t), the effects of π are mirrored in t. By Claims 6, 7, and 8, *ChangeRoot*(f) is enabled in t. By Claim 11 (and definition of t), the effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_6(s)$. Therefore, $\mathcal{S}_6(s')\pi\ t\ ChangeRoot(f)\mathcal{S}_6(s)$ is an execution fragment of *COM*.

(3a) *More claims about s':*

12. No CHANGEROOT message is in *subtree*(f), by Claim 1 and CON-C.
13. No CONNECT message is in any *cqueue*(k), where k is an external link of f, by Claim 1 and CON-D.
14. No CONNECT message is in any *cqueue*(k), where k is an internal link of f, by precondition.

More claims about s:

15. *awake* = true, by Claim 4.
16. No CHANGEROOT message is in *subtree*(f), by Claim 12.

CON-A is true by Claim 15. CON-B is true by Claim 11. CON-C is true by Claim 16. CON-D is true by Claims 9, 10, and 13 and code. CON-E is true because no relevant changes are made. CON-F is true by Claim 14.

iv) π is ReceiveChangeRoot($\langle$q,p$\rangle$). Let f = *fragment*(p).

Case 1: $p \neq$ minnode(f) in s'.

(3c) $\mathcal{A}_6(s',\pi)$ is empty. Below we show that *rootchanged*(f) is the same in s' and s, which implies that $\mathcal{S}_6(s) = \mathcal{S}_6(s')$.

Claims about s':

1. A CHANGEROOT message is in *cqueue*($\langle q,p\rangle$), by precondition.

2. $(p, q) \in \mathit{subtree}(f)$, by Claim 1 and CON-C.
3. $\mathit{rootchanged}(f) =$ false, by Claims 1 and 2 and CON-A.

Claims about s:

4. $\mathit{rootchanged}(f) =$ false, by Claim 1 and code.

Claims 2 and 4 give the result.

(3a) Let $\langle p, r\rangle = \mathit{tominlink}(p)$.

More claims about s′:

5. $\mathit{awake} =$ true, by Claim 1 and CON-A.
6. $\mathit{minlink}(f) \neq \mathit{nil}$, by Claims 1 and 2 and CON-C.
7. $\mathit{minnode}(f) \in \mathit{subtree}(p)$, by Claims 1 and 2 and CON-C.
8. There is exactly one CHANGEROOT message in $\mathit{subtree}(f)$, by Claims 2, 3 and 6 and CON-B.
9. r is a child of p and $\mathit{minnode}(f) \in \mathit{subtree}(r)$, by definition of $\mathit{tominlink}(p)$.

More claims about s:

10. $\mathit{awake} =$ true, by Claim 5.
11. There is exactly one CHANGEROOT message in $\mathit{subtree}(f)$, by Claim 8 and code.
12. r is a child of p, by Claim 9.
13. $\mathit{minlink}(f) \neq \mathit{nil}$, by Claim 6.
14. $(p, r) \neq \mathit{core}(f)$, by Claim 9.
15. $\mathit{minnode}(f) \in \mathit{subtree}(r)$, by Claims 7 and 9.

CON-A is true by Claim 10. CON-B is true by Claim 11 and assumption for Case 1. CON-C is true by Claims 4, 12, 13, 14 and 15. CON-D, CON-E and CON-F are true because no relevant changes are made.

Case 2: $p = \mathit{minnode}(f)$ in s'.

(3b) $\mathcal{A}_6(s', \pi) = \mathit{ChangeRoot}(f)$.

Claims about s′ :

1. A CHANGEROOT message is in $\mathit{cqueue}(\langle q, p\rangle)$, by precondition.

2. $p = minnode(f)$, by assumption.
3. *awake* = true, by Claim 1 and CON-A.
4. $minlink(f) \neq nil$, by Claim 1 and CON-C.
5. *rootchanged*(f) = false, by Claim 1 and CON-C.
6. *minlink*(f) is an external link of f, by Claim 4 and COM-A.

By Claims 3, 4 and 5, *ChangeRoot*(f) is enabled in $\mathcal{S}_6(s')$.

Claims about s:

7. A CONNECT message is in *cqueue*(*minlink*(f)), by code.
8. *minlink*(f) is an external link of f, by Claim 6.
9. *rootchanged*(f) = true, by Claims 7 and 8.

By Claim 9, the effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_6(s)$.

So $\mathcal{S}_6(s')$ *ChangeRoot*(f) $\mathcal{S}_6(s)$ is an execution fragment of *COM*.

(3a) *More claims about s':*

10. p is a child of q, by Claim 1 and CON-C.
11. Exactly one CHANGEROOT message is in *subtree*(f), by Claims 5, 4, 10 and CON-B.
12. No CONNECT message is in any *cqueue*(k), where k is an external link of f, by Claim 5.
13. No CONNECT message is in any *cqueue*(k), where k is an internal link of f, by Claim 4 and CON-F.

More claims about s:

14. *awake* = true, by Claim 3.
15. No CHANGEROOT message is in *subtree*(f), by Claims 1, 10 and 11 and code.
16. No CONNECT message is in any *cqueue*(k), where k is an internal link of f, by Claim 13.

CON-A is true by Claim 14. CON-B is true by Claim 9. CON-C is true by Claim 15. CON-D is true by Claims 7, 8, 12 and code. CON-E is true because no relevant changes are made. CON-F is true by Claim 16.

v) π is ChangeRoot(f).

(3b) $\mathcal{A}_6(s', \pi) = \pi$.

Claims about s':

1. *awake* = true, by precondition.
2. *rootchanged*(f) = false, by precondition.
3. *subtree*$(f) = \{p\}$, by precondition.
4. *minlink*$(f) \neq nil$, by Claim 3 and COM-E.
5. *minlink*(f) is an external link of f, by Claim 4 and COM-A.

Claims 1, 2 and 4 imply that π is enabled in $\mathcal{S}_6(s')$.

Claims about s:

6. *minlink*(f) is an external link of f, by Claim 5.
7. A CONNECT message is in *cqueue*(*minlink*(f)), by code.
8. *rootchanged*(f) = true, by Claims 6 and 7.

Claim 8 implies that the effects of π are mirrored in $\mathcal{S}_6(s)$.

So $\mathcal{S}_6(s')\pi\mathcal{S}_6(s)$ is an execution fragment of *COM*.

(3a) *More claims about s':*

9. No CHANGEROOT message is in *cqueue*$(\langle q,p \rangle)$, for any q, by Claim 3 and CON-C.
10. No CONNECT message is in any *cqueue*(k), where k is an external link of f, by Claim 2.
11. No CONNECT message is in any *cqueue*(k), where k is an internal link of f, by Claim 3.

More claims about s:

12. *awake* = true, by Claim 1 and code.
13. No CHANGEROOT message is in *cqueue*$(\langle q,p \rangle)$, for any q, by Claim 9.
14. No CONNECT message is in any *cqueue*(n), where n is an internal link of f, by Claim 11.

CON-A is true by Claim 12. CON-B is true by Claim 8. CON-C is true by Claim 13. CON-D is true by Claims 6, 7 and 10 and code. CON-E is true because no relevant changes are made. CON-F is true, by Claims 6 and 14.

vi) π is Merge(f,g).

(3b) $\mathcal{A}_6(s', \pi) = \pi$.

Claims about s':

1. A CONNECT(l) message is in $cqueue(\langle p, q\rangle)$, by precondition.
2. $\langle p, q\rangle$ is an external link of f, by precondition.
3. A CONNECT(l) message is in $cqueue(\langle q, p\rangle)$, by precondition.
4. $\langle q, p\rangle$ is an external link of g, by precondition.
5. $f \neq g$, by Claims 2 and 4.
6. $rootchanged(f)$ = true, by Claims 1 and 2.
7. $rootchanged(g)$ = true, by Claims 3 and 4.
8. $\langle p, q\rangle = minlink(f)$, by Claims 1 and 2 and CON-D.
9. $\langle q, p\rangle = minlink(g)$, by Claims 3 and 4 and CON-D.
10. $minedge(f) = minedge(g)$, by Claims 8 and 9.
11. If $k \neq minlink(f)$ is an external link of f, then no CONNECT message is in $cqueue(k)$, by CON-D.
12. If $k \neq minlink(g)$ is an external link of g, then no CONNECT message is in $cqueue(k)$, by CON-D.

By Claims 5, 6, 7 and 10, π is enabled in $\mathcal{S}_6(s')$. By Claims 11 and 12 and definition of h, $rootchanged(h)$ = false in s, so the effects of π are mirrored in $\mathcal{S}_6(s)$. Thus, $\mathcal{S}_6(s')\pi\mathcal{S}_6(s)$ is an execution fragment of *COM*.

(3a) *More claims about s':*

13. $awake$ = true, by Claim 1 and COM-A.
14. No CHANGEROOT message is in $subtree(f)$, by Claim 6 and CON-C.
15. No CHANGEROOT message is in $subtree(g)$, by Claim 7 and CON-C.
16. No CONNECT message is in $cqueue(k)$, for any internal link k of f, by Claim 8 and CON-F.
17. No CONNECT message is in $cqueue(k)$, for any internal link k of g, by Claim 9 and CON-F.
18. Exactly one CONNECT message is in $cqueue(\langle p, q\rangle)$, by Claims 1 and 2 and CON-D.
19. Exactly one CONNECT message is in $cqueue(\langle q, p\rangle)$, by Claims 3 and 4 and CON-D.
20. $l = level(f)$, by Claims 1 and 2 and CON-D.

Claims about s:

21. *awake* = true, by Claim 13 and code.
22. *minlink*(h) = *nil*, by code.
23. No CHANGEROOT message is in *subtree*(h), by Claims 14 and 15 and code.
24. No CONNECT message is in *cqueue*(k), for any external link k of h, by Claims 11 and 12 and code.
25. Exactly one CONNECT message is in *cqueue*$(\langle p,q\rangle)$ and (p,q) = *core*(h), by Claim 18 and code.
26. $l <$ *level*(h), by Claim 20 and code.
27. No CONNECT message is in *cqueue*$(\langle q,p\rangle)$, by Claim 19 and code.
28. No CONNECT message is in any non-core internal link of h, by Claims 16 and 17 and code.

CON-A is true by Claim 21. CON-B is true by Claim 22. CON-C is true by Claim 23. CON-D is true by Claim 24. CON-E is true by Claims 25, 26, 27 and 28. CON-F is true by Claim 22.

vii) π is AfterMerge(p,q). $\mathcal{A}_6(s',\pi)$ is empty. Obviously, $\mathcal{S}_6(s) = \mathcal{S}_6(s')$, and P_{CON} is true in s.

viii) π is Absorb(f,g).

(3b) $\mathcal{A}_6(s',\pi) = \pi$.

Claims about s':

1. $\langle q,p\rangle$ = *minlink*(g), by assumption.
2. A CONNECT(l) message is in *cqueue*(*minlink*(g)), by precondition.
3. $l <$ *level*(f), by precondition.
4. f = *fragment*(p), by precondition.
5. *minlink*(g) is an external link of g, by Claim 1 and COM-A.
6. *rootchanged*(g) = true, by Claims 2 and 5.
7. l = *level*(g), by Claim 2 and CON-D.
8. *level*$(g) <$ *level*(f), by Claims 7 and 3.
9. If a CONNECT message is in *cqueue*$(\langle p,q\rangle)$, then $\langle p,q\rangle$ = *minlink*(f), by Claims 4 and 5 and CON-D.
10. If a CONNECT message is in *cqueue*$(\langle p,q\rangle)$, then *level*$(f) \leq$ *level*(g), by Claim 9 and COM-A.
11. No CONNECT message is in *cqueue*$(\langle p,q\rangle)$, by Claims 8 and 10.
12. No CONNECT message is in *cqueue*(k), for any external link $k \neq$ *minlink*(g) of g, by CON-D.

By Claims 6, 8, 4 and 1, π is enabled in $\mathcal{S}_6(s')$. By Claims 11 and 12, *rootchanged*(f) remains unchanged, and the effects of π are mirrored in $\mathcal{S}_6(s)$. Thus, $\mathcal{S}_6(s')\pi\mathcal{S}_6(s)$ is an execution fragment of COM.

(3a) *More claims about s':*

13. *awake* = true, by Claim 2 and CON-A.
14. $l \geq 0$, by COM-F.
15. $level(f) > 0$, by Claims 7, 8 and 14.
16. $|nodes(f)| > 1$, by Claim 15 and COM-F.
17. No CHANGEROOT message is in *subtree*(g), by Claim 6 and CON-C.
18. No CONNECT message is in *cqueue*(k), where k is an internal link of g, by Claim 1 and CON-F.

Claim about s:

19. *awake* = true, by Claim 12 and code.

CON-A is true by Claim 19. CON-B is true since by Claims 16 and 17 no relevant changes are made. CON-C is true since by Claim 11, 12 and 17 no relevant changes are made. CON-D is true since by Claim 12 no relevant changes are made. CON-E is true since by Claims 11 and 18 no relevant changes are made. CON-F is true by Claim 18 and code. □

Let $P'_{CON} = (P'_{COM} \circ \mathcal{S}_6) \wedge P_{CON}$.

Corollary 24: *P'_{CON} is true in every reachable state of CON.*

Proof: By Lemmas 1 and 23. □

4.2.7 GHS Simultaneously Simulates TAR, DC, NOT and CON

This automaton is a fully distributed version of the original algorithm of [GHS]. (We have made some slight changes, which are discussed below.) The functions of TAR, DC, NOT and CON are united into one. All variables that are derived in one of these automata are also derived (in the same way) in GHS. In addition, there are the following derived variables. The variable *dcstatus*(p) of DC is refined by the variable *nstatus*(p), and has values sleeping, find, and found; initially, it is sleeping. The *awake* variable is now derived, and is true if and only if at least one node is not sleeping. The fragments are also derived, as follows. A subgraph of G is defined to have node set $V(G)$ and edge set equal to all edges of G, at least one of whose links is classified as branch and has no CONNECT message in it. A fragment is associated with each connected component of this graph. Also, *testset*(f) is defined to be all nodes p such that either *testlink*$(p) \neq nil$, or a FIND message is headed toward p (or will be soon).

The bulk of the arguing done at this stage is showing that the derived variables (*subtree, level, core, minlink, testset, rootchanged*) have the proper values in the state mappings. In addition, a substantial argument is needed to show that the implementation of *level* and *core* by local variables interacts correctly with the test-accept-reject protocol. (See in particular the definition of the TAR action mapping for *ReceiveTest*, and the case for *ReceiveTest* in Lemma 25.) It would be ideal to do this argument in NOT, where the rest of the argument that *core* and *level* are implemented correctly is done, but reorganizing the lattice to allow this consolidation caused graver violations of modularity.

The messages sent in this automaton are all those sent in TAR, DC, NOT and CON, except that NOTIFY messages are replaced by INITIATE messages, which have a parameter that is either find or found, and FIND messages are replaced by INITIATE messages with the parameter equal to find.

Some minor changes were made to the algorithm as presented in [GHS]. First, our version initializes all variables to convenient values. (This change makes it easier to state the predicates.) Second, provision is made for the output actions *InTree*(l) and *NotInTree*(l). Third, when node p receives an INITIATE message, variables *inbranch*(p), *bestlink*(p) and *bestwt*(p) are only changed if the parameter of the INITIATE message is find. This change does not affect the performance or correctness of the algorithm. The values of these variables will not be relevant until p subsequently receives an INITIATE-find message, yet the receipt of this message will cause these variables to be reset. The advantage of the change is that it greatly simplifies the state mapping from GHS to DC.

Our version of the algorithm is slightly more general than that in [GHS]. There, each node p has a single queue for incoming messages, whereas in our description, p has a separate queue of incoming messages for each of its neighbors. A node p in our algorithm could happen to process messages in the order, taken over all the neighbors, in which they arrive (modulo the requeueing), which would be consistent with the original algorithm. But p could also handle the messages in some other order (although, of course, still in order for each individual link). Thus, the set of executions of our version is a proper superset of the set of executions of the original.

A small optimization to the original algorithm was also found. (It does not affect the worst-case performance.) When a CONNECT message is received by p under circumstances that cause fragment g to be absorbed into fragment f, an INITIATE message with parameter find is only sent if $\mathit{testlink}(p) \neq \mathit{nil}$ in our version, instead of whenever $\mathit{nstatus}(p) =$ find as in the original. As a result of this change, if $\mathit{nstatus}(p) =$ find and $\mathit{testlink}(p) = \mathit{nil}$, p need not wait for the entire (former) fragment g to find its new minimum-weight external link before p can report to its parent, since this link can only have a larger weight than the minimum-weight external link of p already found.

The automaton GHS is the result of composing an automaton $\mathit{Node}(p)$, for all $p \in V(G)$, and $\mathit{Link}(l)$, for all $l \in L(G)$, and then hiding actions appropriately to fit the $MST(G)$ problem specification.

First we describe the automaton $\mathit{Node}(p)$, for $p \in V(G)$. The state has the following components:

- $\mathit{nstatus}(p)$, either sleeping, find, or found;
- $\mathit{nfrag}(p)$, an edge of G or nil;
- $\mathit{nlevel}(p)$, a nonnegative integer;
- $\mathit{bestlink}(p)$, a link of G or nil;
- $\mathit{bestwt}(p)$, a weight or ∞;
- $\mathit{testlink}(p)$, a link of G or nil;
- $\mathit{inbranch}(p)$, a link of G or nil; and
- $\mathit{findcount}(p)$, a nonnegative integer.

For each link $\langle p, q\rangle \in L_p(G)$, there are the following variables:

- $\mathit{lstatus}(\langle p, q\rangle)$, either unknown, branch or rejected;
- $\mathit{queue}_p(\langle p, q\rangle)$, a FIFO queue of messages from p to q waiting at p to be sent;

- $queue_p(\langle q,p\rangle)$, a FIFO queue of messages from q to p waiting at p to be processed; and

- $answered(\langle p,q\rangle)$, a Boolean.

The set of possible messages M is $\{\text{CONNECT}(l) : l \geq 0\} \cup \{\text{INITIATE}(l, c, st) : l \geq 0, c \in E(G), st \text{ is find or found}\} \cup \{\text{TEST}(l, c) : l \geq 0, c \in E(G)\} \cup \{\text{REPORT}(w) : w \text{ is a weight or } \infty\} \cup \{\text{ACCEPT}, \text{REJECT}, \text{CHANGEROOT}\}$.

In the start state of $Node(p)$, $nstatus(p) =$ sleeping, $nfrag(p) = nil$, $nlevel(p) = 0$, $bestlink(p)$ is arbitrary, $bestwt(p)$ is arbitrary, $testlink(p) = nil$, $inbranch(p)$ is arbitrary, $findcount(p) = 0$, $lstatus(l) =$ unknown for all $l \in L_p(G)$, $answered(l) =$ false for all $l \in L_p(G)$, and both queues are empty.

Now we describe the actions of $Node(p)$.

Input actions:

```
• Start(p)
      Effects:
         if $nstatus(p)$ = sleeping then execute procedure $WakeUp(p)$

• ChannelRecv(l), $l \in L_p(G)$, $m \in M$
      Effects:
         enqueue($m$, $queue_p(l)$)
```

Output actions:

```
• InTree(l), $l \in L_p(G)$
      Preconditions:
         $answered(l)$ = false
         $lstatus(l)$ = branch
      Effects:
         $answered(l)$ := true

• NotInTree(l), $l \in L_p(G)$
      Preconditions:
         $answered(l)$ = false
         $lstatus(l)$ = rejected
      Effects:
         $answered(l)$ := true
```

```
• $ChannelSend(l, m)$, $l \in L_p(G)$, $m \in M$
      Preconditions:
        $m$ at head of $queue_p(l)$
      Effects:
        dequeue($queue_p(l)$)
```

Internal actions:

```
• $ReceiveConnect(\langle q,p\rangle, l)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        CONNECT($l$) at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
        if $nstatus(p)$ = sleeping then execute procedure $WakeUp(p)$
        if $l < nlevel(p)$ then [
          $lstatus(\langle p,q\rangle)$ := branch
          if $testlink(p) \neq nil$, then [
            enqueue(INITIATE($nlevel(p)$, $nfrag(p)$,find), $queue_p(\langle p,q\rangle)$)
            $findcount(p) := findcount(p) + 1$ ]
          else enqueue(INITIATE($nlevel(p)$, $nfrag(p)$,found), $queue_p(\langle p,q\rangle)$) ]
        else
          if $lstatus(\langle p,q\rangle)$ = unknown then enqueue(CONNECT($l$), $queue_p(\langle q,p\rangle)$)
          else enqueue(INITIATE($nlevel(p)+1$, $\langle p,q\rangle$, find), $queue_p(\langle p,q\rangle)$)

• $ReceiveInitiate(\langle q,p\rangle, l, c, st)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        INITIATE($l, c, st$) at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
        $nlevel(p) := l$
        $nfrag(p) := c$
        $nstatus(p) := st$
        — let $S = \{\langle p,r\rangle : lstatus(\langle p,r\rangle) = \text{branch}, r \neq q\}$ —
        enqueue(INITIATE($l, c, st$), $queue_p(k)$) for all $k \in S$
        if $st$ = find then [
          $inbranch(p) := \langle p,q\rangle$
          $bestlink(p) := nil$
          $bestwt(p) := \infty$
          execute procedure $Test(p)$
```

```
            $findcount(p) := |S|$ ]

• $ReceiveTest(\langle q,p\rangle, l, c)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        TEST$(l,c)$ at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
        if $nstatus(p)$ = sleeping then execute procedure $WakeUp(p)$
        if $l > nlevel(p)$ then enqueue(TEST$(l,c)$, $queue_p(\langle q,p\rangle)$)
        else
          if $c \neq nfrag(p)$ then enqueue(ACCEPT, $queue_p(\langle p,q\rangle)$)
          else [
            if $lstatus(\langle p,q\rangle)$ = unknown then $lstatus(\langle p,q\rangle)$ := rejected
            if $testlink(p) \neq \langle p,q\rangle$ then enqueue(REJECT, $queue_p(\langle p,q\rangle)$)
            else execute procedure $Test(p)$ ]

• $ReceiveAccept(\langle q,p\rangle)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        ACCEPT at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
        $testlink(p) := nil$
        if $wt(p,q) < bestwt(p)$ then [
          $bestlink(p) := \langle p,q\rangle$
          $bestwt(p) := wt(p,q)$ ]
        execute procedure $Report(p)$

• $ReceiveReject(\langle q,p\rangle)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        REJECT at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
        if $lstatus(\langle p,q\rangle)$ = unknown then $lstatus(\langle p,q\rangle)$ := rejected
        execute procedure $Test(p)$

• $ReceiveReport(\langle q,p\rangle, w)$, $\langle p,q\rangle \in L_p(G)$
      Preconditions:
        REPORT$(w)$ at head of $queue_p(\langle q,p\rangle)$
      Effects:
        dequeue($queue_p(\langle q,p\rangle)$)
```

```
    if $\langle p,q\rangle \neq inbranch(p)$ then [
      $findcount(p) := findcount(p) - 1$
      if $w < bestwt(p)$ then [
        $bestwt(p) := w$
        $bestlink(p) := \langle p,q\rangle$ ]
      execute procedure $Report(p)$ ]
    else
      if $nstatus(p) =$ find then $enqueue(\text{REPORT}(w), queue_p(\langle q,p\rangle))$
      else if $w > bestwt(p)$ then execute procedure $ChangeRoot(p)$
```

- $ReceiveChangeRoot(\langle q,p\rangle)$, $\langle p,q\rangle \in L_p(G)$

```
  Preconditions:
    CHANGEROOT at head of $queue_p(\langle q,p\rangle)$
  Effects:
    dequeue($queue_p(\langle q,p\rangle)$)
    execute procedure $ChangeRoot(p)$
```

Procedures

- $WakeUp(p)$

```
  — let $\langle p,q\rangle$ be the minimum-weight link of $p$ —
  $lstatus(\langle p,q\rangle) :=$ branch
  $nstatus(p) :=$ found
  enqueue(CONNECT(0), $queue_p(\langle p,q\rangle)$)
```

- $Test(p)$

```
  if $l$, the minimum-weight link of $p$ with $lstatus(l) =$ unknown, exists then [
    $testlink(p) := l$
    enqueue(TEST($nlevel(p), nfrag(p)$), $queue_p(l)$) ]
  else [
    $testlink(p) := nil$
    execute procedure $Report(p)$ ]
```

- $Report(p)$

```
  if $findcount(p) = 0$ and $testlink(p) = nil$ then [
    $nstatus(p) :=$ found
    enqueue(REPORT($bestwt(p)$), $queue_p(inbranch(p))$) ]
```

- $ChangeRoot(p)$

```
  if $lstatus(bestlink(p)) =$ branch then
```

```
        enqueue(CHANGEROOT, $queue_p(bestlink(p))$)
    else [
        enqueue(CONNECT($nlevel(p)$), $queue_p(bestlink(p))$)
        $lstatus(bestlink(p)) :=$ branch ]
```

Now we describe the automaton $Link(\langle p,q\rangle)$, for each $\langle p,q\rangle \in L(G)$.

The state consists of the single variable $queue_{pq}(\langle p,q\rangle)$, a FIFO queue of messages. The set of messages, M, is the same as for $Node(p)$. The queue is empty in the start state.

Input Actions:

- $ChannelSend(\langle p,q\rangle, m)$, $m \in M$
  Effects:
  enqueue(m, $queue_{pq}(\langle p,q\rangle)$)

Output Actions:

- $ChannelRecv(\langle p,q\rangle, m)$, $m \in M$
  Preconditions:
  m at head of $queue_{pq}(\langle p,q\rangle)$
  Effects:
  dequeue($queue_{pq}(\langle p,q\rangle)$)

Now we can define the automaton that models the entire network. Define the automaton GHS to be the result of composing the automata $Node(p)$, for all $p \in V(G)$, and $Link(l)$, for all $l \in L(G)$, and then hiding all actions except for $Start(p)$, $p \in V(G)$, $InTree(l)$ and $NotInTree(l)$, $l \in L(G)$.

Given a FIFO queue q and a set M, define $q|M$ to be the FIFO queue obtained from q by deleting all elements of q that are not in M.

Derived Variables:

- $queue(\langle p,q\rangle)$ is $queue_p(\langle p,q\rangle) \parallel queue_{pq}(\langle p,q\rangle) \parallel queue_q(\langle p,q\rangle)$.
- $tarqueue_p(\langle p,q\rangle)$ is $queue_p(\langle p,q\rangle)|M_{TAR}$, where M_{TAR} is the set of all possible messages in TAR; similarly for $tarqueue_{pq}(\langle p,q\rangle)$ and $tarqueue_q(\langle p,q\rangle)$. Similar

definitions are made for the *dcqueue*'s, *nqueue*'s, and *cqueue*'s, except that for the *dcqueue*'s, each INITIATE(l, c,find) message is replaced with a FIND message, and for the *nqueue*'s, each INITIATE($l, c, *$) message is replaced with a NOTIFY(l, c) message.

- *awake* is false if and only if $nstatus(p) =$ sleeping for all $p \in V(G)$.
- For all $p \in V(G)$, $dcstatus(p) =$ unfind if $nstatus(p) =$ sleeping or found, and $dcstatus(p) =$ find if $nstatus(p) =$ find.
- MSF is the subgraph of G whose nodes are $V(G)$, and whose edges are all edges (p, q) of G such that either (1) $lstatus(\langle p, q\rangle) =$ branch and no CONNECT message is in $queue(\langle p, q\rangle)$, or (2) $lstatus(\langle q, p\rangle) =$ branch and no CONNECT message is in $queue(\langle p, q\rangle)$.
- *fragments* is a set of elements, called *fragments*, one for each connected component of MSF.

  Each fragment f has the following components:

- $subtree(f)$, the corresponding connected component of MSF;
- $level(f)$, defined as in NOT;
- $core(f)$, defined as in NOT;
- $testset(f)$, the set of all $p \in nodes(f)$ such that one of the following is true: (1) a FIND message is headed toward p, (2) $testlink(p) \neq nil$, or (3) a CONNECT message is in $queue(\langle q, r\rangle)$, where $(q, r) = core(f)$ and $p \in subtree(q)$;
- $minlink(f)$, defined as in DC;
- $rootchanged(f)$, defined as in CON; and
- $accmin(f)$, defined as in TAR and DC.

Define the following predicates on $states(GHS)$. (All free variables are universally quantified.)

- GHS-A: If $nstatus(p) =$ sleeping, then
  (a) there is a fragment f such that $subtree(f) = \{p\}$,
  (b) $queue(\langle p, q\rangle)$ is empty for all q, and
  (c) $lstatus(\langle p, q\rangle) =$ unknown for all q.

- GHS-B: If CONNECT(l) is in $queue(\langle q,p\rangle)$, $lstatus(\langle p,q\rangle) \neq$ unknown, and no CONNECT is in $queue(\langle p,q\rangle)$, then
  (a) the state of $queue(\langle q,p\rangle)$ is CONNECT(l) followed by INITIATE($l+1,(p,q)$, find);
  (b) $queue(\langle p,q\rangle)$ is empty;
  (c) $nstatus(q) \neq$ find; and
  (d) $nlevel(p) = nlevel(q) = l$.

- GHS-C: If a CONNECT message is in $queue(l)$, then no FIND message precedes the CONNECT in $queue(l)$, and no TEST or REJECT message is in $queue(l)$.

- GHS-D: If INITIATE(l, c,find) is in $subtree(f)$, then $l = level(f)$.

- GHS-E: If INITIATE(l, c, st) is in $queue(\langle p,q\rangle)$ and $(p,q) = core(fragment(p))$, then $st =$ find.

- GHS-F: If TEST(l, c) is in $queue(\langle q,p\rangle)$, then $nlevel(q) \geq l$.

- GHS-G: If ACCEPT is in $queue(\langle q,p\rangle)$, then $nlevel(p) \leq nlevel(q)$.

- GHS-H: If $testlink(p) \neq nil$, then $nstatus(p) =$ find.

- GHS-I: If p is up-to-date, then $nlevel(p) = level(fragment(p))$.

- GHS-J: If p is up-to-date, $p \notin testset(fragment(p))$, and $\langle p,q\rangle$ is the minimum-weight external link of p, then $nlevel(p) \leq nlevel(q)$.

- GHS-K: If $subtree(f) = \{p\}$ and $nstatus(p) \neq$ sleeping, then $rootchanged(f) =$ true.

Let P_{GHS} be the conjunction of GHS-A through GHS-K.

We now define $\mathcal{M}_x = (\mathcal{S}_x, \mathcal{A}_x)$, an abstraction mapping from GHS to x, for $x = TAR$, DC, NOT and CON. $\mathcal{S}_x$ should be obvious for all x, given the above derived functions. We now define $\mathcal{A}_x(s,\pi)$ for all x, states s of GHS, and actions π of GHS enabled in s.

- $\pi = InTree(l)$ or $NotInTree(l)$. $\mathcal{A}_x(s,\pi) = \pi$ for all x.

- $\pi = Start(p)$. Let $f = fragment(p)$.

Case 1: $nstatus(p) =$ sleeping in s. For all x, $\mathcal{A}_x(s,\pi) = Start(p)\ t_x\ ChangeRoot(f)$, where t_x is the same as $\mathcal{S}_x(s)$ except that $awake =$ true in t_x.

Case 2: $nstatus(p) \neq$ sleeping in s. $\mathcal{A}_x(s,\pi) = \pi$ for all x.

- $\pi = ChannelRecv(k,m)$. For all x, $\mathcal{A}_x(s,\pi)$ is empty, with the following exceptions: If $m = \text{CONNECT}(l)$ or CHANGEROOT, then $\mathcal{A}_{CON}(s,\pi) = \pi$. If $m = \text{INITIATE}(l,c,st)$, then $\mathcal{A}_{NOT}(s,\pi) = ChannelRecv(k, \text{NOTIFY}(l,c))$, and if $st =$ find, then $\mathcal{A}_{DC}(s,\pi) = ChannelRecv(k, \text{FIND})$. If $m =$ TEST, ACCEPT or REJECT, then $\mathcal{A}_{TAR}(s,\pi) = \pi$. If $m = \text{REPORT}(w)$, then $\mathcal{A}_{DC}(s,\pi) = \pi$.

- $\pi = ChannelSend(k,m)$. Analogous to $ChannelRecv(k,m)$.

- $\pi = ReceiveConnect(\langle q,p\rangle, l)$. Let $f = fragment(p)$ and $g = fragment(q)$. (Later we will show that the following four cases are exhaustive.)

Case 1: $nstatus(p) =$ sleeping in s. If $\langle p,q\rangle$ is not the minimum-weight external link of p in s, then $\mathcal{A}_x(s,\pi) = ChangeRoot(f)$ for all x. If $\langle p,q\rangle$ is the minimum-weight external link of p in s, then, for all x, $\mathcal{A}_x(s,\pi) = ChangeRoot(f)$ t_x $Merge(f,g)$, where t_x is the state of x resulting from applying $ChangeRoot(f)$ to $\mathcal{S}_x(s)$.

Case 2: $nstatus(p) \neq$ sleeping, $l = nlevel(p)$, and no CONNECT message is in $queue(\langle p,q\rangle)$ in s. If $lstatus(\langle p,q\rangle) =$ unknown in s, then $\mathcal{A}_x(s,\pi)$ is empty for all x. If $lstatus(\langle p,q\rangle) \neq$ unknown in s, then $\mathcal{A}_{TAR}(s,\pi)$ is empty, and $\mathcal{A}_x(s,\pi) = AfterMerge(p,q)$ for all other x.

Case 3: $nstatus(p) \neq$ sleeping, $l = nlevel(p)$, and a CONNECT message is in $queue(\langle p,q\rangle)$ in s. $\mathcal{A}_x(s,\pi) = Merge(f,g)$ for all x.

Case 4: $nstatus(p) \neq$ sleeping, and $l < nlevel(p)$ in s. $\mathcal{A}_x(s,\pi) = Absorb(f,g)$ for all x.

- $\pi = ReceiveInitiate(\langle q,p\rangle, l, c, st)$.

$\mathcal{A}_{TAR}(s,\pi) = SendTest(p)$ if $st =$ find, and is empty otherwise.

If $st \neq$ find, then $\mathcal{A}_{DC}(s,\pi)$ is empty; if $st =$ find and there is a link $\langle p,r\rangle$ such that $lstatus(\langle p,r\rangle) =$ unknown in s, then $\mathcal{A}_{DC}(s,\pi) = ReceiveFind(\langle q,p\rangle)$; if $st =$ find and there is no link $\langle p,r\rangle$ such that $lstatus(\langle p,r\rangle) =$ unknown in s, then $\mathcal{A}_{DC}(s,\pi) = ReceiveFind(\langle q,p\rangle)$ t $TestNode(p)$, where t is the state of DC resulting from applying $ReceiveFind(\langle q,p\rangle)$ to $\mathcal{S}_{DC}(s)$.

$\mathcal{A}_{NOT}(s,\pi) = ReceiveNotify(\langle q,p\rangle, l, c)$.

$\mathcal{A}_{CON}(s,\pi)$ is empty.

- $\pi = \mathit{ReceiveTest}(\langle q,p\rangle, l, c)$. Let $f = \mathit{fragment}(p)$.

Case 1: $\mathit{nstatus}(p) =$ sleeping in s.

$\mathcal{A}_{TAR}(s,\pi) = \mathit{ChangeRoot}(f)\ t\ \pi$, where t is the same as $\mathcal{S}_{TAR}(s)$ except that $\mathit{rootchanged}(f) =$ true and $\mathit{lstatus}(\mathit{minlink}(f)) =$ branch in t.

$\mathcal{A}_x(s,\pi) = \mathit{ChangeRoot}(f)$ for all other x.

Case 2: $\mathit{nstatus}(p) \neq$ sleeping in s.

$\mathcal{A}_{TAR}(s,\pi) = \pi$ if $l \leq \mathit{nlevel}(p)$ or $\mathit{nlevel}(p) = \mathit{level}(f)$ in s, and is empty otherwise.

$\mathcal{A}_{DC}(s,\pi) = \mathit{TestNode}(p)$ if $l \leq \mathit{nlevel}(p)$, $c = \mathit{nfrag}(p)$, $\mathit{testlink}(p) = \langle p,q\rangle$, and $\mathit{lstatus}(\langle p,r\rangle) \neq$ unknown for all $r \neq q$, in s, and is empty otherwise.

$\mathcal{A}_x(s,\pi)$ is empty for all other x.

- $\pi = \mathit{ReceiveAccept}(\langle q,p\rangle)$.

$\mathcal{A}_{TAR}(s,\pi) = \pi$.

$\mathcal{A}_{DC}(s,\pi) = \mathit{TestNode}(p)$.

$\mathcal{A}_x(s,\pi)$ is empty for all other x.

- $\pi = \mathit{ReceiveReject}(\langle q,p\rangle)$.

$\mathcal{A}_{TAR}(s,\pi) = \pi$.

$\mathcal{A}_{DC}(s,\pi) = \mathit{TestNode}(p)$ if there is no $r \neq q$ such that $\mathit{lstatus}(\langle p,r\rangle) =$ unknown in s, and is empty otherwise.

$\mathcal{A}_x(s,\pi)$ is empty for all other x.

- $\pi = \mathit{ReceiveReport}(\langle q,p\rangle, w)$. Let $f = \mathit{fragment}(p)$.

Case 1: $(p,q) = \mathit{core}(f)$, $\mathit{nstatus}(p) \neq$ find, $w > \mathit{bestwt}(p)$, and $\mathit{lstatus}$ $(\mathit{bestlink}(p)) =$ branch in s.

$\mathcal{A}_{DC}(s,\pi) = \pi$.

$\mathcal{A}_x(s,\pi) = \mathit{ComputeMin}(f)$ for all other x.

Case 2: $(p,q) = \mathit{core}(f)$, $\mathit{nstatus}(p) \neq$ find, $w > \mathit{bestwt}(p)$, and $\mathit{lstatus}(\mathit{bestlink}(p)) \neq$ branch in s.

$\mathcal{A}_{DC}(s,\pi) = \pi\ t_{DC}\ \mathit{ChangeRoot}(f)$, where t_{DC} is the state of DC resulting from applying π to $\mathcal{S}_{DC}(s)$.

$\mathcal{A}_{CON}(s,\pi) = \mathit{ComputeMin}(f)$.

$\mathcal{A}_x(s,\pi) = \mathit{ComputeMin}(f)\ t_x\ \mathit{ChangeRoot}(f)$ for all other x, where t_x is the state of x resulting from applying $\mathit{ComputeMin}(f)$ to $\mathcal{S}_x(s)$.

Case 3: $(p,q) \neq \mathit{core}(f)$ or $\mathit{nstatus}(p) =$ find or $w \leq \mathit{bestwt}(p)$ in s.

$\mathcal{A}_{DC}(s,\pi) = \pi$.

$\mathcal{A}_x(s,\pi)$ is empty for all other x.

- $\pi = \mathit{ReceiveChangeRoot}(\langle q,p\rangle)$. Let $f = \mathit{fragment}(p)$.

$\mathcal{A}_{CON}(s,\pi) = \pi$.

For all other x, $\mathcal{A}_x(s,\pi) = \mathit{ChangeRoot}(f)$ if $\mathit{lstatus}(\mathit{bestlink}(p)) \neq$ branch in s, and is empty otherwise.

For the rest of this chapter, let I be the set of names $\{TAR, DC, NOT, CON\}$. The following predicates are true in any state of GHS satisfying $\bigwedge_{x\in I}(P'_x \circ \mathcal{S}_x) \wedge P_{GHS}$. I.e., they are derivable from P_{GHS}, together with the TAR, DC, NOT, CON, GC, COM and HI predicates.

- GHS-L: If $\mathit{AfterMerge}(p,q)$ is enabled for DC or NOT, then a CONNECT message is at the head of $\mathit{queue}(\langle q,p\rangle)$.

Proof: First we show the predicate for DC. Let $f = \mathit{fragment}(p)$.
1. $(p,q) = \mathit{core}(f)$, by precondition.
2. FIND is in $\mathit{dcqueue}(\langle q,p\rangle)$, by precondition.
3. No FIND is in $\mathit{dcqueue}(\langle p,q\rangle)$, by precondition.
4. $\mathit{dcstatus}(q) =$ unfind, by precondition.
5. No REPORT is in $\mathit{dcqueue}(\langle q,p\rangle)$, by precondition.
6. $q \in \mathit{testset}(f)$, by Claims 1 through 5 and DC-G.
7. $\mathit{testlink}(p) = \mathit{nil}$, by Claim 4 and GHS-H.

8. A CONNECT is in $queue(\langle q,p\rangle)$, by Claims 1, 3, 6 and 7.
9. $(p,q) \in subtree(f)$, by Claim 1 and COM-F.
10. No INITIATE($*,*$,found) is in $queue(\langle q,p\rangle)$, by Claim 1 and GHS-E.
11. No CHANGEROOT is in $queue(\langle q,p\rangle)$, by Claim 1.
12. No ACCEPT is in $queue(\langle q,p\rangle)$, by Claim 9 and TAR-F.
13. CONNECT precedes any FIND, TEST, or REJECT in $queue(\langle q,p\rangle)$, by Claim GHS-C.

Claims 5, 8, 10, 11, 12 and 13 give the result.

For NOT, we show that if *AfterMerge*(p,q) for NOT is enabled, then *AfterMerge*(p,q) for DC is enabled.

1. $(p,q) = core(f)$, by precondition.
2. NOTIFY$(nlevel(p)+1,(p,q))$ is in $nqueue(\langle q,p\rangle)$, by precondition.
3. No NOTIFY$(nlevel(p)+1,(p,q))$ is in $nqueue(\langle p,q\rangle)$, by precondition.
4. $nlevel(q) \neq nlevel(p)+1$, by precondition.
5. INITIATE$(nlevel(p)+1,(p,q)$,find) is in $queue(\langle q,p\rangle)$, by Claims 1 and 2 and GHS-E.
6. $nlevel(p)+1 = level(f)$, by Claim 5 and GHS-D.
7. No INITIATE($*,*$,find) is in $queue(\langle p,q\rangle)$, by Claims 3 and 6 and GHS-D.
8. q is not up-to-date, by Claims 4 and 6 and GHS-I.
9. $dcstatus(q) \neq$ find, by Claim 8 and DC-I(a).
10. No REPORT is in $queue(\langle q,p\rangle)$, by Claims 1 and 8 and DC-C(a).

By Claims 1, 5, 7, 9 and 10, *AfterMerge*(p,q) for DC is enabled. □

- GHS-M: If $testlink(p) \neq nil$ or $findcount(p) > 0$, then no FIND message is headed toward p, and no CONNECT message is in $queue(\langle q,r\rangle)$, where $(q,r) = core(fragment(p))$ and $p \in subtree(q)$.

Proof:
1. $testlink(p) \neq nil$ or $findcount(p) > 0$, by assumption.
2. $nstatus(p) =$ find, by Claim 1 and either GHS-H or DC-H(b).
3. $dcstatus(t) =$ find for all t between q and p inclusive, by Claim 2 and DC-H(a).
4. No FIND message is headed toward p, by Claim 4 and DC-D(b).
5. No CONNECT is in $queue(\langle q,r\rangle)$, or $lstatus(\langle r,q\rangle) =$ unknown, or CONNECT is in $queue(\langle r,q\rangle)$, by Claim 3 and GHS-B(c).
6. $(q,r) \in subtree(fragment(p))$, by COM-F.
7. $lstatus(\langle r,q\rangle) \neq$ unknown, by Claim 6 and TAR-A(b).

8. If CONNECT is in $queue(\langle r,q\rangle)$ then no CONNECT is in $queue(\langle q,r\rangle)$, by Claim 6.
9. If no CONNECT is in $queue(\langle r,q\rangle)$ then no CONNECT is in $queue(\langle q,r\rangle)$, by Claims 5 and 7.

Claims 4, 8 and 9 give the result. □

Lemma 25: *GHS simultaneously simulates the set of automata* $\{TAR, DC, NOT, CON\}$ *via* $\{\mathcal{M}_x : x \in I\}$, P_{GHS}, *and* $\{P'_x : x \in I\}$.

Proof: By inspection, the types are correct. By Corollaries 18, 20, 22 and 24, P'_x is a predicate true in every reachable state of x, for all x.

(1) Let s be in $start(GHS)$. Obviously P_{GHS} is true in s and $\mathcal{S}_x(s)$ is in $start(x)$ for all x.

(2) Obviously, $\mathcal{A}_x(s,\pi)|ext(x) = \pi|ext(GHS)$ for all x.

(3) Let (s',π,s) be a step of GHS such that $\bigwedge_{x\in I} P'_x(\mathcal{S}_x(s'))$ and $P_{GHS}(s')$ are true. By Corollaries 18, 20, 22 and 24, we can assume the HI, COM, GC, TAR, DC, NOT and CON predicates are true in s', as well as the GHS predicates. Below, we show (3a), that P_{GHS} is true in s (only for those predicates whose truth in s is not obvious), and either (3b) or (3c), as appropriate, that the step simulations for TAR, DC, NOT, and CON are correct.

i) π **is InTree(⟨p,q⟩).** Let $f = fragment(p)$ in s'.

(3a) Obviously, P_{GHS} is true in s.

(3b)/(3c) $\mathcal{A}_x(s',\pi) = \pi$ for all x.

Claims about s':

1. $answered(\langle p,q\rangle) =$ false, by precondition.
2. $lstatus(\langle p,q\rangle) =$ branch, by precondition.
3. $nstatus(p) \neq$ sleeping, by Claim 2 and GHS-A(c).
4. $awake =$ true, by Claim 3.
5. $(p,q) \in subtree(f)$ or $\langle p,q\rangle = minlink(f)$, by Claim 2 and TAR-A(a).

π is enabled in $\mathcal{S}_x(s')$ by Claims 1 and 2 for $x = TAR$, and by Claims 1, 4 and 5 for all other x. Obviously, its effects are mirrored in $\mathcal{S}_x(s)$ for all x.

ii) π **is NotInTree(⟨p,q⟩).** Let $f = fragment(p)$ in s'.

(3a) Obviously, P_{GHS} is true in s.

(3b)/(3c) $\mathcal{A}_x(s, \pi) = \pi$ for all x.

Claims about s':

1. $answered(\langle p, q\rangle)$ = false, by precondition.
2. $lstatus(\langle p, q\rangle)$ = rejected, by precondition.
3. $nstatus(p) \neq$ sleeping, by Claim 2 and GHS-A(c).
4. $awake$ = true, by Claim 3.
5. $fragment(p) = fragment(q)$ and $(p, q) \neq subtree(f)$, by Claim 2 and TAR-B.

π is enabled in $\mathcal{S}_x(s')$ by Claims 1 and 2 for $x = TAR$, and by Claims 1, 4 and 5 for all other x. Obviously its effects are mirrored in $\mathcal{S}_x(s)$ for all x.

iii) π is Start(p). Let $f = fragment(p)$.

Case 1: $nstatus(p) \neq$ sleeping in s'. $\mathcal{A}_x(s', \pi) = \pi$ for all x. Obviously $\mathcal{S}_x(s')\pi\mathcal{S}_x(s)$ is an execution fragment of x for all x, and P_{GHS} is true in s.

Case 2: $nstatus(p)$ = sleeping in s'.

(3b)/(3c) For all x, $\mathcal{A}_x(s', \pi) = \pi\ t_x\ ChangeRoot(f)$, where t_x is the same as $\mathcal{S}_x(s')$ except that $awake$ = true in t_x. For all x, we must show that π is enabled in $\mathcal{S}_x(s')$ (which is true because π is an input action), that its effects are mirrored in t_x (which is true by definition of t_x), that $ChangeRoot(f)$ is enabled in t_x, and that its effects are mirrored in $\mathcal{S}_x(s)$.

Let l be the minimum-weight external link of p. (It exists by GHS-A(a) and the assumption that $|V(G)| > 1$.)

Claims about s':

1. $nstatus(p)$ = sleeping, by assumption.
2. $subtree(f) = \{p\}$, by Claim 1 and GHS-A.
3. $minlink(f) = l$, by Claim 2 and definition.
4. $lstatus(\langle p, q\rangle)$ = unknown, for all q, by Claim 1 and GHS-A(c).
5. $rootchanged(f)$ = false, by Claim 4 and TAR-H.

Claims about t_x, for all x:

6. $awake$ = true, by definition.

7. *subtree*$(f) = \{p\}$, by Claim 2.
8. *rootchanged*$(f) =$ false, by Claim 5.
9. *minlink*$(f) = l$, by Claim 3.

ChangeRoot(f) is enabled in t_{CON} by Claims 6, 7 and 8. For all other x, *ChangeRoot*(f) is enabled in t_x by Claims 6, 8 and 9.

Claims about s:

10. CONNECT(0) is in *queue*(l), by code.
11. *lstatus*$(l) =$ branch, by code.
12. *rootchanged*$(f) =$ true, by Claims 10 and 11 and choice of l.

For most of the other derived variables, it is obvious that they are the same in s' and s. Although *nstatus*(p) changes, *dcstatus*(p) remains unchanged. Even though *lstatus*(l) changes to branch, MSF does not change, since a CONNECT message is in *queue*(l).

For $x = TAR$, the effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_x(s)$ by Claims 11 and 12. For $x = CON$, the effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_x(s)$ by Claim 10. For all other x, the effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_x(s)$ by Claim 12.

(3a) *More Claims about s':*

13. *lstatus*$(\langle q,p\rangle) \neq$ rejected, for all q, by Claim 2 and TAR-B.
14. If *lstatus*$(\langle q,p\rangle) =$ branch, then a CONNECT is in *queue*$(\langle q,p\rangle)$, for all q, by Claim 2.
15. *testset*$(f) = \emptyset$, by Claim 3 and GC-C.
16. *testlink*$(p) = nil$, by Claim 15.
17. *queue*(l) is empty, by Claim 1 and GHS-A(b).

GHS-A is vacuously true since *nstatus*$(p) =$ found in s.

GHS-B: vacuously true for CONNECT added to *queue*(l) by Claims 13 and 14; vacuously true for any CONNECT already in *queue*$(reverse(l))$ by Claim 10; vacuously true for any CONNECT already in *queue*$(\langle q,p\rangle)$, for any q such that $\langle p,q\rangle \neq l$, by Claim 4.

GHS-C is true by Claim 17 and code.

GHS-H is vacuously true by Claim 16.

No change affects the others.

iv) π is ChannelRecv(k,m) or ChannelSend(k,m). Obviously $P_{GHS}(s)$ is true, and the step simulations are correct.

v) π is ReceiveConnect($\langle$q,p$\rangle$,l). Let $f = \mathit{fragment}(p)$, and $g = \mathit{fragment}(q)$ in s'. We consider four cases. We now show that they are exhaustive, i.e., that $l > \mathit{nlevel}(p)$ is impossible. First, suppose $\langle q,p\rangle$ is an external link of g. By CON-D, $l = \mathit{level}(g)$ and $\langle q,p\rangle = \mathit{minlink}(g)$. By NOT-D, $\mathit{level}(g) \leq \mathit{nlevel}(p)$. Second, suppose $\langle q,p\rangle$ is an internal link of $g = f$. By CON-E, $(p,q) = \mathit{core}(f)$, and $l < \mathit{level}(f)$. But by NOT-C, $\mathit{nlevel}(p) \geq \mathit{level}(f) - 1$.

Case 1: nstatus(p) = sleeping. This case is divided into two subcases. First we prove some claims true in both subcases. Let k be the minimum-weight external link of p.

Claims about s':

1. CONNECT(l) is at head of $\mathit{queue}_p(\langle q,p\rangle)$, by precondition.
2. $\mathit{nstatus}(p)$ = sleeping, by assumption.
3. $\mathit{subtree}(f) = \{p\}$, by Claim 2 and GHS-A.
4. $\mathit{rootchanged}(f)$ = false, by Claim 2, GHS-A(c) and TAR-H.
5. $\mathit{minlink}(f) = k$, by Claim 3 and definition.
6. awake = true, by Claim 1 and CON-A.
7. No FIND is in $\mathit{queue}(\langle q,p\rangle)$, by Claim 3 and DC-D(a).
8. $f \neq g$, by Claim 3.
9. $\langle q,p\rangle$ is an external link of g, by Claim 8.
10. $\mathit{minlink}(g) = \langle q,p\rangle$, by Claims 1 and 9 and CON-D
11. $\mathit{level}(g) \leq \mathit{level}(f)$, by Claim 10 and COM-A.
12. $l = \mathit{level}(g)$, by Claims 1 and 9 and CON-D.
13. $\mathit{level}(f) = 0$, by Claim 3 and COM-F.
14. $l \leq 0$, by Claims 11, 12 and 13.
15. $l = 0$, by Claim 14 and COM-F.
16. $\mathit{nlevel}(p) = 0$, by Claims 3 and 13.

Subcase 1a: $\langle p,q\rangle \neq k$. By Claim 2 and GHS-A(c), $\mathit{lstatus}(\langle p,q\rangle)$ = unknown in s', and the same is true in s. This fact, together with Claims 15 and 16, shows that the only change is that the CONNECT(l) message is requeued.

(3a) P_{GHS} can be shown to be true in s by an argument very similar to that for $\pi = \mathit{Start}(p)$, Case 2, since the only change is that the CONNECT(l) message is requeued. Claim 7 verifies that GHS-C is true in s.

(3b)/(3c) For all x, $\mathcal{A}_x(s', \pi) = \mathit{ChangeRoot}(f)$. For $x = CON$, $\mathit{ChangeRoot}(f)$ is enabled in $\mathcal{S}_x(s')$ by Claims 6, 4 and 3; for all other x, it is enabled by Claims 6, 4 and 5.

Claims about s:

17. $\mathit{lstatus}(k) =$ branch, by code.
18. CONNECT(0) is added to the end of $\mathit{queue}(k)$, by code.
19. $\mathit{rootchanged}(f) =$ true, by Claims 17 and 18 and choice of k.

For most of the other derived variables, it is obvious that they are the same in s' and s. Although $\mathit{nstatus}(p)$ changes, $\mathit{dcstatus}(p)$ remains unchanged. Even though $\mathit{lstatus}(k)$ changes to branch, MSF does not change, since a CONNECT message is in $\mathit{queue}(k)$.

The effects of $\mathit{ChangeRoot}(f)$ are mirrored in $\mathcal{S}_x(s)$ by Claims 17 and 19 for $x = TAR$, by Claim 18 for $x = CON$, and by Claim 19 for all other x.

Subcase 1b: $\langle p, q \rangle = k$.

(3b)/(3c) For all x, $\mathcal{A}_x(s', \pi) = \mathit{ChangeRoot}(f)\ t_x\ \mathit{Merge}(f, g)$, where t_x is the result of applying $\mathit{ChangeRoot}(f)$ to $\mathcal{S}_x(s')$. $\mathit{ChangeRoot}(f)$ is enabled in $\mathcal{S}_x(s')$ by Claims 6, 4 and 3 for $x = CON$, and by Claims 6, 4 and 5 for all other x. Its effects are obviously mirrored in t_x.

More claims about s':

20. $k = \langle p, q \rangle$, by assumption.
21. $\langle p, q \rangle$ is an external link of f, by Claim 8.
22. $\mathit{rootchanged}(g) =$ true, by Claim 1 and Claim 9.
23. Only one CONNECT message is in $\mathit{queue}(\langle q, p \rangle)$, by Claims 1 and 9 and CON-D.
24. $\mathit{lstatus}(\langle q, p \rangle) =$ branch, by Claims 10 and 22 and TAR-H.
25. $\mathit{level}(g) = 0$, by Claims 12 and 15.
26. $\mathit{subtree}(g) = \{q\}$, by Claim 25 and COM-F.
27. $\mathit{nlevel}(q) = 0$, by Claims 25 and 26.

28. No INITIATE message is in *queue*($\langle p,q\rangle$) or *queue*($\langle q,p\rangle$), by Claims 9 and 21 and NOT-H(e).
29. No CONNECT message is in *queue*($\langle p,r\rangle$) for any $r \neq q$, by Claims 3 and 20 and CON-D.
30. No CONNECT message is in *queue*($\langle q,r\rangle$) for any $r \neq p$, by Claims 10 and 26 and CON-D.

Claims about t_x:

31. $f \neq g$, by Claim 8.
32. *rootchanged*(f) = true, by definition of t_x.
33. *rootchanged*(g) = true, by Claim 22.
34. *minedge*(f) = *minedge*(g) = (p,q), by Claims 5, 10 and 20.
35. If $x = CON$, then CONNECT(0) is in *cqueue*($\langle p,q\rangle$), by definition of t_x.
36. If $x = CON$, then CONNECT(0) is at the head of *cqueue*($\langle q,p\rangle$), by Claims 1 and 15.

Merge(f,g) is enabled in t_x by Claims 34, 35 and 36 for $x = CON$, and by Claims 31, 32, 33 and 34 for all other x.

As we shall shortly show, MSF has changed — the connected components corresponding to f and g have combined. Let h be the fragment corresponding to this new connected component.

Claims about s:

37. No CONNECT is in *queue*($\langle q,p\rangle$), by Claim 23 and code.
38. *lstatus*($\langle q,p\rangle$) = branch, by Claim 24 and code.
39. $(p,q) \in MSF$, by Claims 37 and 38.
40. *subtree*(h) is nodes p and q and the edge between them, by Claims 3, 26 and 39.
41. INITIATE(1, (p,q),find) is in *queue*($\langle p,q\rangle$), by code.
42. *level*(h) = 1, by Claims 16, 27, 28, 40 and 41.
43. *core*(h) = (p,q), by Claims 16, 27, 28, 40 and 41.
44. CONNECT(0) is in *queue*($\langle p,q\rangle$), by code.
45. *testset*(h) = $\{p,q\}$, by Claims 41 and 44.
46. *minlink*(h) = *nil*, by Claim 45.
47. *rootchanged*(h) = false, by Claims 29, 30 and 40.
48. f and g are no longer in *fragments*, by Claims 3, 26, 40 and 43.

The effects of *Merge*(f,g) are mirrored in $S_x(s)$ by Claims 40, 42, 43, 45, 46, 47 and 48 for $x = TAR$; by Claims 40, 41, 42, 43, 45, 47 and 48 for $x = DC$; by

Claims 40, 41, 46, 47 and 48 for $x = NOT$; and by Claims 40, 42, 43, 46 and 48 for $x = CON$.

(3a) GHS-A: vacuously true for p by code. By Claim 1 and GHS-A(c), $nstatus(q) \neq$ sleeping in s'; since the same is true in s, changing q's subtree does not invalidate GHS-A(a).

GHS-B: Obviously, the only situation affected is the CONNECT added to $queue(\langle p, q\rangle)$.

(a) $queue(\langle p, q\rangle)$ has the correct contents in s because of the code and the fact that $queue(\langle p, q\rangle)$ is empty in s' by Claim 2 and GHS-A(b).

(b) To show that $queue(\langle q, p\rangle)$ is empty in s, we must show that it contains only the CONNECT in s'. By Claim 1 and GHS-C, there is no TEST or REJECT in $queue(\langle q, p\rangle)$. By Claim 2 and GHS-H, $testlink(p) = nil$; thus, by TAR-D, no ACCEPT is in $queue(\langle q, p\rangle)$. By Claim 3, DC-A(g) and DC-B(a), there is no REPORT in $queue(\langle q, p\rangle)$. By Claim 3 and NOT-H(e), there is no NOTIFY in $queue(\langle q, p\rangle)$. By Claim 3 and CON-C, there is no CHANGEROOT in $queue(\langle q, p\rangle)$. By Claim 1, CON-D and CON-E, there is only one CONNECT in $queue(\langle q, p\rangle)$.

(c) $nstatus(p) \neq$ find in s by code.

(d) By Claims 16 and 27, $nlevel(p) = nlevel(q) = 0$.

GHS-C: No FIND is in $queue(\langle p, q\rangle)$ in s' by Claim 3 and DC-D(a). No REJECT is in $queue(\langle p, q\rangle)$ in s' by Claim 3 and TAR-G. No TEST(l, c), for any l and c, is in $queue(\langle p, q\rangle)$ in s', because by Claims 25 and 13 and TAR-E(b) and TAR-E(c), $l = 0$; yet by TAR-M, $l \geq 1$.

GHS-D: By Claim 42.

GHS-E: By code for the INITIATE added to $queue(\langle p, q\rangle)$. By Claim 28, this is the only relevant message affected.

GHS-H is true in s since $nstatus(p)$ goes from sleeping to found, and $testlink(p)$ is unchanged.

GHS-I: By Claim 45, p and q are both in $testset(h)$ in s. We now show that $nstatus(p) \neq$ find and $nstatus(q) \neq$ find. Then by Claim 40, no node in $subtree(h)$ is up-to-date, so the predicate is vacuously true (for h). By code, $dcstatus(p)$ = found. By Claim 10 and GC-C, $testset(g) = \emptyset$ in s'; by Claim 26, no REPORT message is in $subtree(g)$ in s'. Thus, by DC-I(b), $dcstatus(q) \neq$ find in s'.

GHS-J: vacuously true by Claims 40 and 45 for p and q. No relevant change for any other node.

No change affects the rest.

Case 2: $nstatus(p) \neq$ sleeping, $l = nlevel(p)$, and no CONNECT message is in $queue(\langle p,q\rangle)$ in s'.

Subcase 2a: $lstatus(\langle p,q\rangle)$ = unknown in s'. The only change in going from s' to s is that the CONNECT message is requeued.

(3a) The only GHS predicates affected are GHS-B(a) and GHS-C. By TAR-A(b), $(p,q) \neq subtree(f)$. Thus, by DC-D(a), no FIND is in $queue(\langle q,p\rangle)$ in s', and the predicates are still true in s.

(3b)/(3c) $\mathcal{A}_x(s',\pi)$ is empty for all x. We now show that $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all x, by showing that $cqueue(\langle q,p\rangle)$ contains only the one CONNECT message in s'. By TAR-A(b), (p,q) is not in MSF. Thus, by CON-C, no CHANGEROOT is in $cqueue(\langle q,p\rangle)$. By CON-D and CON-E, only one CONNECT message is in $cqueue(\langle q,p\rangle)$.

Subcase 2b: $lstatus(\langle p,q\rangle) \neq$ unknown in s'.

(3b)/(3c) $\mathcal{A}_{TAR}(s',\pi)$ is empty, and $\mathcal{A}_x(s',\pi) = AfterMerge(p,q)$ for all other x.

Claims about s':

1. CONNECT is at head of $queue_p(\langle q,p\rangle)$, by precondition.
2. $nstatus(p) \neq$ sleeping, by assumption.
3. $nlevel(p) = l$, by assumption.
4. No CONNECT is in $queue(\langle p,q\rangle)$, by assumption.
5. $lstatus(\langle p,q\rangle) \neq$ unknown, by assumption.
6. If $lstatus(\langle p,q\rangle)$ = rejected, then $fragment(p) = fragment(q)$, by TAR-B.

7. If $lstatus(\langle p,q\rangle) =$ branch, then $(p,q) \in subtree(f)$, by Claim 4 and definition of MSF.
8. $\langle p,q\rangle$ is an internal link of f, by Claims 5, 6 and 7.
9. $(p,q) = core(f)$, by Claims 1 and 8 and CON-E.
10. INITIATE($nlevel(p)+1,(p,q)$,find) is in $queue(\langle q,p\rangle)$, by Claims 1, 3, 4 and 5 and GHS-B(a).
11. No INITIATE($nlevel(p)+1,(p,q),*$) is in $queue(\langle p,q\rangle)$, by Claims 1, 3, 4 and 5 and GHS-B(b).
12. $dcstatus(q) \neq$ find, by Claims 1, 4 and 5 and GHS-B(c).
13. No REPORT is in $queue(\langle q,p\rangle)$, by Claims 1, 4 and 5 and GHS-B(a).
14. $nlevel(q) = l$, by Claims 1, 4 and 5 and GHS-B(d).

AfterMerge(p,q) is enabled in $\mathcal{S}_x(s')$ by Claims 9, 10, 11, 12 and 13 for $x = DC$; by Claims 3, 9, 10, 11 and 14 for $x = NOT$; and by Claims 1 and 9 for $x = CON$.

Claims about s:

15. CONNECT(l) is dequeued from $queue_p(\langle q,p\rangle)$, by code.
16. FIND is in $queue(\langle p,q\rangle)$, by code.
17. INITIATE($nlevel(p)+1,(p,q)$,find) is in $queue(\langle p,q\rangle)$, by code.

The only derived variables that are not obviously unchanged are $testset(f)$, $level(f)$ and $core(f)$. Claims 15 and 16 show that $testset(f)$ is unchanged. Claims 10 and 17 show that $level(f)$ and $core(f)$ are unchanged.

The effects of *AfterMerge*(p,q) are mirrored in $\mathcal{S}_x(s)$ by Claim 16 for $x = DC$; by Claim 17 for $x = NOT$; and by Claim 15 for $x = CON$. It is easy to see that $\mathcal{S}_{TAR}(s') = \mathcal{S}_{TAR}(s)$.

(3a) GHS-A: By Claim 2, adding a message to a queue of p does not invalidate GHS-A(b).

GHS-B: By Claim 8 and CON-E, there is only one CONNECT message in $queue(\langle q,p\rangle)$ in s'. Since it is removed in s, the predicate is vacuously true for a CONNECT in $queue(\langle q,p\rangle)$. By Claim 4, the predicate is vacuously true for a CONNECT in $queue(\langle p,q\rangle)$.

GHS-C: By Claim 4, vacuously true for $queue(\langle p,q\rangle)$.

GHS-D: By Claim 10 and GHS-D, $nlevel(p) + 1 = level(f)$. This together with Claim 9 gives the result.

GHS-E is true by code.

No change affects the rest.

Case 3: $nstatus(p)$ $\neq$ sleeping, $l = nlevel(p)$, and a CONNECT message is in $queue(\langle p, q\rangle)$ in s'.

(3b)/(3c) $\mathcal{A}_x(s', \pi) = Merge(f, g)$ for all x.

Claims about s':

1. CONNECT(l) is at head of $queue(\langle q, p\rangle)$, by precondition.
2. $l = nlevel(p)$, by assumption.
3. CONNECT(m) is in $queue(\langle p, q\rangle)$, by assumption.
4. $\langle p, q\rangle$ is an external link of p, by Claims 1 and 3.
5. $\langle q, p\rangle$ is an external link of q, by Claims 1 and 3.
6. $f \neq g$, by Claim 4.
7. $rootchanged(f) =$ true, by Claims 1 and 4.
8. $rootchanged(g) =$ true, by Claims 3 and 5.
9. $\langle q, p\rangle = minlink(g)$, by Claims 1 and 5 and CON-D.
10. $\langle p, q\rangle = minlink(f)$, by Claims 3 and 4 and CON-D.
11. $minedge(f) = minedge(g)$, by Claims 9 and 10.
12. $m = level(f)$, by Claims 3 and 4 and CON-D.
13. $nlevel(p) = level(f)$, by Claim 10 and NOT-D.
14. $m = l$, by Claims 2, 12 and 13.

$Merge(f, g)$ is enabled in $\mathcal{S}_{CON}(s')$ by Claims 1, 3, 4, 5 and 14, and for all other x by Claims 6, 7, 8 and 11.

15. Only one CONNECT message is in $queue(\langle q, p\rangle)$, by Claim 1 and CON-D.
16. $lstatus(\langle q, p\rangle) =$ branch, by Claims 8 and 9 and TAR-H.
17. $lstatus(\langle p, q\rangle) =$ branch, by Claims 7 and 10 and TAR-H.
18. $level(g) = l$, by Claims 1 and 5 and CON-D.
19. If INITIATE($l', c, *$) is in $subtree(f)$, then $l' \leq l$, by Claims 12 and 14.
20. If INITIATE($l', c, *$) is in $subtree(g)$, then $l' \leq l$, by Claim 18.
21. $nlevel(r) \leq l$ for all $r \in nodes(f)$, by Claims 12 and 14.

22. $nlevel(r) \leq l$ for all $r \in nodes(g)$, by Claim 18.
23. No INITIATE message is in $queue(\langle q,p\rangle)$ or $queue(\langle p,q\rangle)$, by Claims 4 and 5 and NOT-H(e).
24. No CONNECT is in $queue(\langle r,t\rangle)$, where $r \in nodes(f)$, and $\langle r,t\rangle \neq \langle p,q\rangle$, by Claim 10 and CON-D and CON-F.
25. No CONNECT is in $queue(\langle r,t\rangle)$, where $r \in nodes(g)$ and $\langle r,t\rangle \neq \langle q,p\rangle$, by Claim 9 and CON-D and CON-F.
26. $(p,q) \neq core(f)$, by Claim 4 and COM-F.
27. $(p,q) \neq core(g)$, by Claim 5 and COM-F.

As we shall shortly show, MSF has changed — the connected components corresponding to f and g have combined. Let h be the fragment corresponding to this new connected component.

Claims about s:

28. No CONNECT is in $queue(\langle q,p\rangle)$, by Claim 15 and code.
29. $lstatus(\langle q,p\rangle) =$ branch, by Claim 16.
30. $(p,q) \in MSF$, by Claims 28 and 29.
31. $subtree(h)$ is the union of the old $subtree(f)$ and $subtree(g)$ and (p,q), by Claim 30.
32. INITIATE$(l+1,(p,q),$find$)$ is in $queue(\langle p,q\rangle)$, by Claim 2 and 17 and code.
33. if INITIATE$(l',c,*)$ is in $subtree(h)$, then $l' \leq l+1$, by Claims 19, 20, 23, 31 and 32.
34. $nlevel(r) \leq l$ for all $r \in nodes(h)$, by Claims 21, 22 and 31.
35. $level(h) = l+1$, by Claims 33 and 34.
36. $core(h) = (p,q)$, by Claims 19, 20, 23, 31, 32, and 34.
37. CONNECT(l) is in $queue(\langle p,q\rangle)$, by Claims 3 and 14
38. $testset(h) = nodes(h)$, by Claims 31, 32 and 37.
39. $minlink(h) = nil$, by Claim 38.
40. $rootchanged(h) =$ false, by Claims 24, 25 and 31.
41. f and g are no longer in *fragments*, by Claims 26, 27, 31 and 36.

The effects of $Merge(f,g)$ are mirrored in $S_x(s)$ by Claims 31, 35, 36, 38, 39, 40 and 41 for TAR; by Claims 31, 35, 36, 38, 40 and 41 for DC; by Claims 31, 39, 40 and 41 for NOT; and by Claims 28, 31, 35, 36, 39, and 41 for CON.

(3a) GHS-A: Vacuously true for p by assumption. Vacuously true for q by Claim 1 and GHS-A(b).

GHS-B: Obviously, the only situation affected is the CONNECT in $queue(\langle p,q\rangle)$.

(a) We must show that in s', $queue(\langle p,q\rangle)$ consists only of a CONNECT(l) message. (The code adds the appropriate INITIATE message.) By Claim 3 and GHS-C, no TEST or REJECT is in $queue(\langle p,q\rangle)$. By Claim 4, DC-A(g) and DC-B(a), no REPORT is in $queue(\langle p,q\rangle)$. By Claim 23, no NOTIFY is in $queue(\langle p,q\rangle)$. By Claim 4 and CON-C, no CHANGEROOT is in $queue(\langle p,q\rangle)$. By Claims 3 and 14, a CONNECT(l) message is in $queue(\langle p,q\rangle)$, and by CON-E and CON-F, it is the only CONNECT message in that queue.

(b) A very similar argument to that in (a) shows that in s', $queue(\langle q,p\rangle)$ consists only of a CONNECT(l) message. (Since it is removed in s, the queue is then empty.)

(c) If $|nodes(f)| > 1$, then $dcstatus(p) \neq$ find by Claim 10. Suppose $subtree(f) = \{p\}$. Obviously, no REPORT message is headed toward p in s'. By Claim 10 and GC-C, $testset(f) = \emptyset$ in s'. Thus, by DC-I(b), $dcstatus(p) \neq$ find in s'. In both cases, $nstatus(p)$ does not change in s.

(d) $nlevel(p) = l$ in s' by assumption. $nlevel(q) = l$ in s' by Claims 9 and 18 and NOT-D. These values are unchanged in s.

GHS-C: By the same argument as in GHS-B(a), adding the INITIATE message is OK.

GHS-D: by Claim 35.

GHS-E: By code, for the INITIATE added. By Claim 23, there are no leftover INITIATE messages affected by the change of core.

GHS-I: We show no $r \in nodes(h)$ in s is up-to-date. By Claim 38, r is in $testset(h)$. By the same argument as in GHS-B(c), $dcstatus(r) \neq$ find.

GHS-J: Vacuously true by Claim 38.

No change affects the rest.

Case 4: $nstatus(p) \neq$ sleeping, and $l < nlevel(p)$ in s'.

(3b)/(3c) $\mathcal{A}_x(s', \pi) = Absorb(f, g)$ for all x.

Claims about s':

1. CONNECT(l) is at head of $queue(\langle q,p\rangle)$, by precondition.
2. $l < nlevel(p)$, by assumption.
3. $lstatus(\langle p,q\rangle) =$ unknown, or a CONNECT is in $queue(\langle p,q\rangle)$, by Claims 1 and 2 and GHS-B(d).
4. $\langle q,p\rangle$ is an external link of g, by Claims 1 and 3.
5. $minlink(g) = \langle q,p\rangle$, by Claims 1 and 4 and CON-D.
6. $l = level(g)$, by Claims 1 and 4 and CON-D.
7. $rootchanged(g) =$ true, by Claims 1 and 4.
8. $nlevel(p) \le level(f)$, by definition of $level(f)$.
9. $level(g) < level(f)$, by Claims 2, 6 and 8.
10. $lstatus(\langle q,p\rangle) =$ branch, by Claims 5 and 7 and TAR-H.
11. If INITIATE$(l', c, *)$ is in $subtree(g)$, then $l' < level(f)$, by Claims 6 and 9.
12. If INITIATE$(l', c, *)$ is in $subtree(f)$, then $l' \le level(f)$, by definition of $level(f)$.
13. $nlevel(r) < level(f)$, for all $r \in nodes(g)$, by Claims 6 and 9.
14. $nlevel(r) \le level(f)$, for all $r \in nodes(f)$, by definition of $level(f)$.
15. No INITIATE message is in $queue(\langle q,p\rangle)$ or $queue(\langle p,q\rangle)$, by Claim 4 and NOT-H(e).
16. No CONNECT message is in $queue(\langle r,t\rangle)$, where $r \in nodes(g)$, $\langle r,t\rangle \ne \langle q,p\rangle$, by Claim 5 and CON-D and CON-F.
17. $f \ne g$, by Claim 4.
18. $l \ge 0$, by Claim 6 and COM-F.
19. $level(f) > 0$, by Claims 18 and 9.
20. $core(f) \ne nil$, by Claim 19 and COM-F.
21. $core(f) \in subtree(f)$, by Claim 20 and COM-F.
22. If $subtree(g) = \{q\}$, then $core(g) = nil$, by COM-F.
23. if $subtree(g) \ne \{q\}$, then $core(g) \in subtree(g)$, by COM-F.
24. Only one CONNECT message is in $queue(\langle q,p\rangle)$, by Claims 1 and 4 and CON-D.
25. $testset(g) = \emptyset$, by Claim 5 and GC-C.
26. $testlink(r) = nil$, for all $r \in nodes(g)$, by Claim 25.
27. If $testlink(p) \ne nil$, then $p \in testset(f)$, by definition.
28. If $testlink(p) \ne nil$, then $nstatus(p) =$ find, by GHS-H.
29. If $nstatus(p) =$ find, then no FIND message is headed toward p, by DC-D(b) and DC-H(a).
30. $lstatus(\langle r,t\rangle) \ne$ unknown, where $(r,t) = core(f)$, by Claim 21 and TAR-A(b).
31. If CONNECT is in $queue(\langle r,t\rangle)$, then no CONNECT is in $queue(\langle t,r\rangle)$, where $(r,t) = core(f)$, by Claim 21.
32. If $nstatus(p) =$ find and $p \in subtree(r)$, then $nstatus(r) =$ find, for all r, by DC-H(a).

33. If $nstatus(p) =$ find, then no CONNECT is in $queue(\langle r,t\rangle)$, where $(r,t) = core(f)$ and $p \in subtree(r)$, by Claims 30, 31 and 32 and GHS-B(c).
34. If $nstatus(p) =$ find and $p \in testset(f)$, then $testlink(p) \neq nil$, by Claims 29 and 33.

$Absorb(f,g)$ is enabled in $S_x(s')$ by Claims 7, 9 and 5 for TAR and DC; by Claims 7, 6 and 2, and 5 for NOT; and by Claims 1, 6 and 9, and 5 for CON.

As we shall shortly show, MSF has changed — the connected components corresponding to f and g have combined. Let h be the fragment corresponding to this new connected component. We shall show that $h = f$, i.e., that the core of h in s is non-nil, and is the same as the core of f in s'.

Claims about s:

35. No CONNECT message is in $queue(\langle q,p\rangle)$, by Claim 24 and code.
36. $lstatus(\langle q,p\rangle) =$ branch, by Claim 10.
37. $(p,q) \in MSF$, by Claims 35 and 36.
38. $subtree(h)$ is the union of the old $subtree(f)$ and $subtree(g)$ and (p,q), by Claim 37.
39. INITIATE$(nlevel(p), nfrag(p), nstatus(p))$ is in $queue(\langle p,q\rangle)$, by code.
40. $level(h) =$ old $level(f)$, by Claims 11, 12, 13, 14, 15 and 38.
41. $core(h) =$ old $core(f)$, by Claims 11, 12, 13, 14, 15 and 38.
42. $h = f$, by Claim 41.
43. $g \notin fragments$, by Claims 38 and 41.
44. NOTIFY$(nlevel(p), nfrag(p))$ is added to $queue_p(\langle p,q\rangle)$, by code.

First, we discuss how $testset(f)$ changes. If $p \in testset(f)$ in s' because of a FIND or CONNECT message, then every node in $nodes(g)$ in s' is in $testset(f)$ in s because of the same FIND or CONNECT message. If $p \in testset(f)$ in s' because $testlink(p) \neq nil$, then a FIND message is added to $queue(\langle p,q\rangle)$ in s, causing every node formerly in $nodes(g)$ to be in $testset(f)$. If p is not in $testset(f)$ in s', then no FIND message is headed toward p, and no CONNECT message is in $queue(\langle r,t\rangle)$, with $p \in subtree(r)$; thus, Claim 25 implies that in s, no node formerly in $nodes(g)$ is in $testset(f)$.

By the previous paragraph, and inspection, the effects of $Absorb(f,g)$ are mirrored in $S_x(s)$ by Claims 36, 38, 42 and 43 for $x = TAR$; by Claims 27, 28, 34, 38, 42 and 43 for $x = DC$; by Claims 38, 42, 43 and 44 for $x = NOT$; and by Claims 35, 38, 42 and 43 for $x = CON$.

(3a) GHS-A is vacuously true in s by assumption that $nstatus(p) \neq$ sleeping in s'.

GHS-B: vacuously true for a CONNECT in $queue(\langle q,p \rangle)$ by Claim 35. By Claim 4 and CON-D, if CONNECT is in $queue(\langle p,q \rangle)$, then $minlink(f) = \langle p,q \rangle$. But by Claim 9 and COM-A, this cannot be. Thus the predicate is vacuously true for a CONNECT in $queue(\langle p,q \rangle)$.

GHS-D: Suppose $nstatus(p) =$ find in s'. By DC-I(a), p is up-to-date, and by GHS-I, $nlevel(p) = level(f)$.

GHS-E: Vacuously true by Claims 4, 21 and 41.

GHS-I: As argued in GHS-J, no node formerly in $nodes(g)$ is up-to-date in s. No change affects nodes formerly in $nodes(f)$.

GHS-J: Let r be any node in $nodes(f)$ in s'. If r is up-to-date, $r \notin testset(f)$, and $\langle r,t \rangle$ is the minimum-weight external link of r, then $nlevel(r) \leq nlevel(t)$ by GHS-J. By Claim 9, $fragment(t) \neq g$. Thus in s, $\langle r,t \rangle$ is still external. By DC-L, $inbranch(r)$ is in $subtree(g)$ (or nil) for all $r \in nodes(g)$ in s'. By Claim 21, $core(f) \in subtree(f)$ in s', and by Claim 41, $core(f)$ is unchanged in s. Thus following $inbranches$ in s from any r formerly in $nodes(g)$ does not lead to $core(f)$, so no r formerly in $nodes(g)$ is up-to-date in s.

No change affects the rest.

vi) π is ReceiveInitiate($\langle$q,p$\rangle$,l,c,st). Let $f = fragment(p)$.

(3b)/(3c) *Case 1:* $st =$ find. $\mathcal{A}_{TAR}(s',\pi) = SendTest(p)$.

If there is a link $\langle p,r \rangle$ such that $lstatus(\langle p,r \rangle) =$ unknown in s', then $\mathcal{A}_{DC}(s',\pi) = ReceiveFind(\langle q,p \rangle)$; otherwise $\mathcal{A}_{DC}(s',\pi) = ReceiveFind(\langle q,p \rangle)\ t\ TestNode(p)$, where t is the state resulting from applying $ReceiveFind(\langle q,p \rangle)$ to $\mathcal{S}_{DC}(s')$.

$\mathcal{A}_{NOT}(s',\pi) = ReceiveNotify(\langle q,p \rangle, l, c)$.

$\mathcal{A}_{CON}(s',\pi)$ is empty.

Claims about s':

1. INITIATE(l, c,find) is at the head of $queue_p(\langle q,p \rangle)$, by precondition.

2. $(p,q) \in \mathit{subtree}(f)$, by Claim 1 and DC-D(a).
3. $\mathit{minlink}(f) = \mathit{nil}$, by Claims 1 and 2.
4. If $\mathit{lstatus}(\langle p,r\rangle) =$ rejected then $\mathit{fragment}(p) = \mathit{fragment}(r)$, for all r, by TAR-B.
5. If $\mathit{lstatus}(\langle p,r\rangle) =$ branch, then $(p,r) \in \mathit{subtree}(f)$, for all r, by Claim 3 and TAR-A(a).
6. If $(p,r) \in \mathit{subtree}(f)$, then $\mathit{lstatus}(\langle p,r\rangle) =$ branch for all r, by TAR-A(b).
7. If $|S| = 0$ and no $\mathit{lstatus}(\langle p,r\rangle)$ is unknown, then $p \neq \mathit{mw\text{-}root}(f)$, by definition of *mw-root* and Claims 4, 5 and 6.
8. $p \in \mathit{testset}(f)$, by Claims 1 and 2.
9. $\mathit{dcstatus}(p) =$ unfind, by Claim 1 and DC-D(b).
10. $\mathit{testlink}(p) = \mathit{nil}$, by Claim 9 and GHS-H.
11. $l = \mathit{level}(f)$, by Claims 1 and 2 and GHS-D.
12. $c = \mathit{core}(f)$, by Claims 1 and 11 and NOT-A.
13. No other FIND message is headed toward p, by Claims 1 and 2 and DC-S.
14. $\mathit{core}(f) \neq \mathit{nil}$, by Claim 2 and COM-F.

Let $(r,t) = \mathit{core}(f)$.

15. $(r,t) \in \mathit{subtree}(f)$, by Claim 14 and COM-F.

Let p be in $\mathit{subtree}(r)$.

16. If $(p,q) \neq (r,t)$ then $\mathit{dcstatus}(q) =$ find, by Claim 1 and DC-D(a).
17. If $(p,q) \neq (r,t)$ then $\mathit{dcstatus}(r) =$ find, by Claim 16 and DC-H(a).
18. If $(p,q) \neq (r,t)$ then either no CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, or $\mathit{lstatus}(\langle t,r\rangle) =$ unknown, or a CONNECT is in $\mathit{queue}(\langle t,r\rangle)$, by Claim 17 and GHS-B(c).
19. If $(p,q) = (r,t)$ then either no CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, or $\mathit{lstatus}(\langle t,r\rangle) =$ unknown, or a CONNECT is in $\mathit{queue}(\langle t,r\rangle)$, by Claim 1 and GHS-B(b).
20. Either no CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, or $\mathit{lstatus}(\langle t,r\rangle) =$ unknown, or a CONNECT is in $\mathit{queue}(\langle t,r\rangle)$, by Claims 18 and 19.
21. $\mathit{lstatus}(\langle t,r\rangle) \neq$ unknown, by Claim 15 and TAR-A(b).
22. If CONNECT is in $\mathit{queue}(\langle t,r\rangle)$ then no CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, by Claim 15.
23. If no CONNECT is in $\mathit{queue}(\langle t,r\rangle)$ then no CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, by Claims 20, 21 and 22.
24. No CONNECT is in $\mathit{queue}(\langle r,t\rangle)$, by Claims 22 and 23.
25. If $(p,q) \neq (r,t)$ then $\mathit{AfterMerge}(p,q)$ is not enabled (for DC or NOT), since $(r,t) = \mathit{core}(f)$.
26. If $(p,q) = (r,t)$ then $\mathit{AfterMerge}(p,q)$ is not enabled (for DC or NOT), by Claim 24 and GHS-L.

27. If there is no unknown link of p, then there is no external link of p, by Claims 4 and 5.
28. If $(p, q) \neq (r,)$, then q is up-to-date, by Claim 16 and DC-I(a).

SendTest(p) is enabled in $\mathcal{S}_{TAR}(s')$ by Claims 8 and 10. *ReceiveFind*($\langle q, p\rangle$) is enabled in $\mathcal{S}_{DC}(s')$ by Claims 1, 25 and 26. *ReceiveNotify*($\langle q, p\rangle, l, c$) is enabled in $\mathcal{S}_{NOT}(s')$ by Claims 1, 25 and 26.

Claims about t: (only defined when there are no unknown links of p in s')

29. $p \in testset(f)$, by Claim 8.
30. There is no external link of p, by Claim 27.
31. $dcstatus(p) =$ find, by definition of t.

TestNode(p) is enabled in t by Claims 29, 30 and 31.

Claims about s:

32. $level(f) = l$, by Claim 11 and code.
33. $core(f) = c$, by Claim 12 and code.
34. No FIND message is headed toward p, by Claim 13 and code.
35. No CONNECT is in $queue(\langle t, r\rangle)$, by Claim 24 and code.
36. There is no unknown link of p (in s') if and only if $testlink(p) = nil$ (in s), by Claim 10 and code.
37. There is no unknown link of p (in s') if and only if $p \notin testset(f)$ (in s), by Claims 34, 35 and 36.
38. If $|S| > 0$ (in s') then a FIND message is in $subtree(f)$, by Claim 5 and code.
39. If $|S| = 0$ and there is no unknown link of p (in s'), then $p \neq mw\text{-}root(f)$ (in s), by Claim 7 and code.
40. If $|S| = 0$ and there is no unknown link of p (in s'), then either a REPORT message is headed toward $mw\text{-}root(f)$, or there is no external link of f (in s), by Claims 28 and 39 and code.
41. If there is an unknown link of p (in s'), then $nstatus(p) =$ find (in s), by code.
42. $minlink(f) = nil$, by Claims 38, 40 and 41.

The changes (or lack of changes) to the remaining derived variables are obvious.

The effects of *SendTest*(p) are mirrored in $\mathcal{S}_{TAR}(s)$ by Claims 11, 12, and 37 for the changes, and Claims 32, 33, 3 and 42 for the lack of changes. If there is an unknown link of p in s', then the effects of *ReceiveFind*($\langle q, p\rangle$) are mirrored in $\mathcal{S}_{DC}(s)$ by Claims 5, 6, 36 and 37 for changes, and Claims 3, 11, 12, 32, 33, 37 and

42 for lack of changes. If there is no unknown link of p in s', then the effects of *ReceiveFind*$(\langle q,p\rangle)$ followed by *TestNode*(p) are mirrored in $\mathcal{S}_{DC}(s)$ by Claims 5, 6, 36 and 37 for changes, and Claims 3, 11, 12, 32, 33 and 42 for lack of changes. The effects of *ReceiveNotify*$(\langle q,p\rangle, l, c)$ are mirrored in $\mathcal{S}_{NOT}(s)$ by Claims 3, 4 and 42. $\mathcal{S}_{CON}(s') = \mathcal{S}_{CON}(s)$ by Claims 3, 11, 12, 32, 33, and 42.

Case 2: $st \neq$ find.

$\mathcal{A}_{NOT}(s',\pi) =$ *ReceiveNotify*$(\langle q,p\rangle, l, c)$. $\mathcal{A}_x(s',\pi)$ is empty for all other x.

Claims about s':

1. INITIATE$(l, c,$found$)$ is at the head of $queue_p(\langle q,p\rangle)$, by precondition.
2. $(p,q) \in subtree(f)$, by Claim 1 and NOT-H(e).
3. $nlevel(p) < l$, by Claim 1 and NOT-H(a).
4. $nlevel(p) < level(f)$, by Claims 1, 2 and 3.
5. $p \neq minnode(f)$, by Claims 1 and 2 and NOT-I.
6. If $lstatus(\langle p,r\rangle) =$ branch, then $(p,r) \in subtree(f)$, for all $r \neq q$, by Claim 5 and TAR-A(a).
7. If $(p,r) \in subtree(f)$, then $lstatus(\langle p,r\rangle) =$ branch, for all $r \neq q$, by TAR-A(b).
8. p is not up-to-date, by Claim 4 and GHS-I.
9. $nstatus(p) \neq$ find, by Claim 8 and DC-I(a).
10. $(p,q) \neq core(f)$, by Claim 1 and GHS-E.
11. *AfterMerge*(p,q) for NOT is not enabled, by Claim 10.

By Claim 9, $dcstatus(p) =$ unfind in both s' and s, and thus $minlink(f)$ is unchanged. The changes, or lack of changes, to the remaining derived variables are obvious.

By Claims 1 and 11, *ReceiveNotify*$(\langle q,p\rangle, l, c)$ is enabled in $\mathcal{S}_{NOT}(s')$. Its effects are mirrored in $\mathcal{S}_{NOT}(s)$ by Claims 6 and 7.

It is easy to see that $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all other x.

(3a) GHS-A: By DC-D(a), $(p,q) \in subtree(f)$. So by GHS-A(a), $nstatus(p) \neq$ sleeping in s'. Since the same is true in s, the predicate is vacuously true.

GHS-B: Vacuously true for a CONNECT in $queue(\langle q,p\rangle)$ by GHS-B(a) and the fact that INITIATE is first in the queue. Vacuously true for a CONNECT in $queue(\langle p,q\rangle)$ by GHS-B(b) and the presence of INITIATE in $queue(\langle q,p\rangle)$. The only other situation to consider is the addition of an INITIATE message to $queue(\langle p,r\rangle)$, $r \neq q$, with $lstatus(\langle p,r\rangle) =$ branch. As shown in (b)/(c), $(p,r) \in subtree(f)$. By NOT-H(e), either $(p,q) = core(f)$ or p is a child of q, so $(p,r) \neq core(f)$. Thus by CON-E, no CONNECT is in $queue(\langle p,r\rangle)$, or in $queue(\langle r,p\rangle)$.

GHS-C: Adding a FIND message does not falsify the predicate. Suppose a TEST message is added to $queue(\langle p,r\rangle)$. Then in s', $st =$ find.

Case 1: $\langle p,r\rangle$ is an internal link of f. By TAR-A(b), $(p,r) \neq subtree(f)$. By COM-F, $(p,r) \neq core(f)$. By CON-E, no CONNECT is in $queue(\langle p,r\rangle)$.

Case 2: $\langle p,r\rangle$ is an external link of f. Since there is a FIND message in $subtree(f)$ in s', $minlink(f) = nil$. By CON-D, no CONNECT is in $queue(\langle p,r\rangle)$.

GHS-D: Since it is true for the INITIATE in $queue(\langle q,p\rangle)$ in s', it is true for any INITIATE added in s.

GHS-E: As shown in GHS-B, $(p,r) \neq core(f)$.

GHS-F: By NOT-H(a), $nlevel(p)$ increases, so the predicate is still true for any leftover TEST messages. The predicate is true by code for the TEST message added.

GHS-G: *Case 1:* An ACCEPT is in $queue(\langle p,r\rangle)$. By NOT-H(a), $nlevel(p)$ increases, so the predicate is still true.

Case 2: An ACCEPT is in $queue(\langle r,p\rangle)$. By TAR-D, $testlink(p) = \langle p,r\rangle$. By GHS-H, $nstatus(p) =$ find. But by Claim 9 (for both Case 1 and Case 2 of (3b)/(3c)), $nstatus(p) \neq$ find. So there is no ACCEPT in $queue(\langle r,p\rangle)$, and the predicate is vacuously true.

GHS-H is true by code.

GHS-I: *Case 1:* $st =$ find. By code $nlevel(p) = l$, and by Claim 32 in Case 1 of (3b)/(3c), $l = level(f)$.

Case 2: $st \neq$ found. By NOT-H(a), $nlevel(p) < l$. Thus $nlevel(p) < level(f)$, so by GHS-I, p is not up-to-date in s'. Since all *inbranches* remain the same in s and $nstatus(p) \neq$ find in s, p is still not up-to-date.

GHS-J: *Case 1:* $st =$ find. By Claim 37 in Case 1 of (3b)/(3c), $p \notin testset(f)$ in s if and only if there is no external link of p, so the predicate is vacuously true.

Case 2: $st \neq$ find. As in GHS-I, Case 2, p is not up-to-date, so the predicate is vacuously true.

vii) π is ReceiveTest($\langle$q,p$\rangle$,l,c). Let $f = fragment(p)$.

Case 1: $nstatus(p) =$ sleeping in s'.

(3b)/(3c) $\mathcal{A}_{TAR}(s', \pi) = ChangeRoot(f)\ t\ \pi$, where t is the same as $\mathcal{S}_{TAR}(s')$ except that $rootchanged(f) =$ true and $lstatus(minlink(f)) =$ branch in t.

$\mathcal{A}_x(s', \pi) = ChangeRoot(f)$ for all other x.

Claims about s':

1. TEST(l, c) is at the head of $queue_p(\langle q, p\rangle)$, by precondition.
2. $nstatus(p) =$ sleeping, by assumption.
3. $subtree(f) = \{p\}$, by Claim 2 and GHS-A.
4. $minlink(f) \neq nil$, by Claim 3 and definition.
5. $rootchanged(f) =$ false, by Claim 2, GHS-A(c) and TAR-H.
6. $level(f) = 0$, by Claim 3 and COM-F.
7. $nlevel(p) = 0$, by Claims 3 and 6.
8. $l \geq 1$, by TAR-M.
9. $l > nlevel(p)$, by Claims 7 and 8.
10. $l > level(f)$, by Claims 6 and 8.
11. $awake =$ true, by Claim 1 and GHS-A(b).

Claims about s:

12. The TEST message is requeued, by Claim 9.
13. $lstatus(minlink(f)) =$ branch, by code.
14. CONNECT(0) is in $queue(minlink(f))$, by code.
15. $minlink(f)$ does not change (i.e., is still external), by Claims 13 and 14.
16. $rootchanged(f) =$ true, by Claims 14 and 15.

$ChangeRoot(f)$ is enabled in $\mathcal{S}_x(s')$ by Claims 11, 3 and 5 for $x = CON$, and by Claims 11, 4 and 5 for all other x.

TAR: Effects of $ChangeRoot(f)$ are mirrored in t by its definition. π is enabled in t by definition. Its effects are mirrored in $\mathcal{S}_{TAR}(s)$ by Claim 12.

For all other x, the effects of *ChangeRoot*(f) are mirrored in $S_x(s)$ by Claim 16 for DC and NOT, and by Claim 14 for CON.

(3a) P_{GHS} is true in s by essentially the same argument as in $\pi = Start(p)$, Case 2.

Case 2: nstatus(p) $\neq$ sleeping in s'.

(3b)/(3c) $\mathcal{A}_{TAR}(s', \pi) = \pi$ if $l \leq nlevel(p)$ or $nlevel(p) = level(f)$ in s', and is empty otherwise.

$\mathcal{A}_{DC}(s', \pi) = TestNode(p)$ if $l \leq nlevel(p)$, $c = nfrag(p)$, $testlink(p) = \langle p, q \rangle$ and $lstatus(\langle p, r \rangle) \neq$ unknown for all $r \neq q$, in s', and is empty otherwise.

$\mathcal{A}_x(s', \pi)$ is empty for all other x.

First we discuss what happens to *testset*(f) and *minlink*(f).

We show *testset*(f) is unchanged, except that p is removed from *testset*(f) if and only if $l \leq nlevel(p)$, $c = nfrag(p)$, $testlink(p) = \langle p, q \rangle$, and there is no link $\langle p, r \rangle$, $r \neq q$, with $lstatus(\langle p, r \rangle) =$ unknown. If *testlink*(p) does not change from non-nil to *nil* (or vice versa), then obviously *testset*(f) is unchanged. The only place *testlink*(p) is changed in this way is in procedure *Test*(p), exactly if there are no more unknown links of p; *Test*(p) is executed if and only if $l \leq nlevel(p)$, $c = nfrag(p)$, and $testlink(p) = \langle p, q \rangle$ in s'. Suppose *testlink*(p) is changed from non-nil to *nil*. Since $testlink(p) \neq nil$ in s', GHS-M implies that no FIND message is headed toward p, and no CONNECT message is in $queue(\langle r, t \rangle)$, where $(r, t) = core(f)$ and $p \in subtree(r)$. Thus in s, since $testlink(p) = nil$, p is not in *testset*(f).

Now we show that *minlink*(f) does not change. If *dcstatus*(p) does not change, and no REPORT message is added to any queue, then obviously *minlink*(f) does not change. Suppose *dcstatus*(p) changes, and a REPORT message is added to a queue (in procedure *Report*(p)). Then $l \leq nlevel(p)$, $c = nfrag(p)$, $testlink(p) = \langle p, q \rangle$, there are no more unknown links of p (so *testlink*(p) is set to *nil*), and $findcount(p) = 0$.

Claims about s':

1. $testlink(p) = \langle p, q \rangle$, by assumption.

2. $nstatus(p) =$ find, by Claim 1 and GHS-H.
3. $minlink(f) = nil$, by Claim 2.
4. If $(p, r) = core(f)$, then a FIND message is in $queue(\langle p, r\rangle)$, or $dcstatus(r) =$ find, or a REPORT message is in $queue(\langle r, p\rangle)$, by Claim 2 and DC-J.
5. p is up-to-date, by Claim 2 and DC-I(a).

Claims about s:

6. If $p \neq mw\text{-}root(f)$, then either there is no external link of f, or a REPORT is headed toward $mw\text{-}root(f)$, by Claim 5 and code.
7. If $p = mw\text{-}root(f)$, then either a FIND is in $queue(\langle p, r\rangle)$, or $dcstatus(r) =$ find, or a REPORT is in $queue(\langle r, p\rangle)$, where $core(f) = (p, r)$, by Claim 4 and code.
8. $minlink(f) = nil$, by Claims 6 and 7.

Claims 3 and 8 give the result.

TAR: First, suppose $l > nlevel(p)$ and $nlevel(p) \neq level(f)$.

Claims about s′:

1. $l > nlevel(p)$, by assumption.
2. $nlevel(p) \neq level(f)$, by assumption.
3. p is not up-to-date, by Claim 2 and GHS-I.
4. $nstatus(p) \neq$ find, by Claim 3 and DC-I(a).
5. $testlink(p) = nil$, by Claim 4 and GHS-H.
6. There is no protocol message for $\langle p, q\rangle$, by Claim 5 and TAR-D.
7. The TEST message in $queue(\langle q, p\rangle)$ is a protocol message for $\langle q, p\rangle$, by Claim 6.
8. $testlink(q) = \langle q, p\rangle$, by Claim 7 and TAR-D.
9. There is exactly one protocol message for $\langle q, p\rangle$, by Claim 8 and TAR-C(c).
10. There is only one TEST message in $tarqueue(\langle q, p\rangle)$, by Claim 9.

By Claims 6 and 10, the TEST is the only *TAR* message in $tarqueue(\langle q, p\rangle)$. Since the TEST message is requeued in *GHS*, $tarqueue(\langle q, p\rangle)$ is unchanged. By earlier remarks about $testset(f)$ and $minlink(f)$, and by inspection, the other derived variables (for *TAR*) are unchanged. Thus, $\mathcal{S}_{TAR}(s') = \mathcal{S}_{TAR}(s)$,

Second, suppose $l > level(p)$ and $nlevel(p) = level(f)$. Then the TEST message is requeued in *GHS* and in *TAR*. By earlier remarks about $testlink(f)$ and $minlink(f)$, and by inspection, $\mathcal{S}_{TAR}(s')\pi\mathcal{S}_{TAR}(s)$ is an execution fragment of *TAR*.

Third, suppose $l \leq nlevel(p)$. Let $g = fragment(q)$.

Claims about s':

1. $\textsc{test}(l, c)$ is at the head of $queue_p(\langle q, p\rangle)$, by precondition.
2. $l \leq nlevel(p)$, by assumption.
3. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $c = core(g)$ and $l = level(g)$, by Claim 1 and TAR-E(b).
4. If $lstatus(\langle q, p\rangle) =$ rejected, then $c = core(f)$ and $l = level(f)$, by Claim 1 and TAR-E(c).
5. $c \neq nil$, by Claim 1 and TAR-M.

Next we show that $c = core(f)$ if and only if $c = nfrag(p)$. First, suppose $c = core(f)$.

6. $c = core(f)$, by assumption.
7. If $lstatus(\langle q, p\rangle) =$ rejected, then $nlevel(p) = level(f)$, by Claims 2 and 4 and definition of $level(f)$.
8. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $core(g) = core(f)$, by Claims 3 and 6.
9. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $c \in subtree(g)$ and $c \in subtree(f)$, by Claims 5, 6 and 8 and COM-F.
10. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $f = g$, by Claim 9 and COM-G.
11. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $l = level(f)$, by Claims 3 and 10.
12. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $nlevel(p) = level(f)$, by Claims 2 and 11 and definition of $level(f)$.
13. $nlevel(p) = level(f)$, by Claims 8 and 12.
14. $nfrag(p) = core(f)$, by Claim 13 and NOT-A.
15. $nfrag(p) = c$, by Claims 6 and 14.

Now suppose $c = nfrag(p)$.

16. $c = nfrag(p)$, by assumption.
17. $c \in subtree(f)$, by Claims 5 and 16 and NOT-F.
18. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $c \in subtree(g)$, by Claims 5 and 3 and COM-F.
19. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $f = g$, by Claims 17 and 18 and COM-G.
20. If $lstatus(\langle q, p\rangle) \neq$ rejected, then $c = core(f)$, by Claims 3 and 19.
21. $c = core(f)$, by Claims 4 and 20.

π is enabled in $\mathcal{S}_{TAR}(s')$ by Claim 1. We now verify that the effects are mirrored in $\mathcal{S}_{TAR}(s)$. By the above argument, $c \neq frag(p)$ if and only if $c \neq core(f)$. Thus, the body of *ReceiveTest* for *TAR* is simulated correctly. Consider procedure $Test(p)$. If it is executed, then $c = nfrag(p)$ in s'. By Claim 21, $nfrag(p) = core(f)$, and by

NOT-E, $nlevel(p) = level(f)$. Thus the TEST messages sent in procedure $Test(p)$ in GHS correspond to those sent in TAR. By the discussion at the beginning of Case 2, $testset(f)$ is updated correctly, and $minlink(f)$ is unchanged. The changes or lack of changes to the other derived variables are obvious.

DC: First, suppose $l \leq nlevel(p)$, $c = nfrag(p)$, $testlink(p) = \langle p, q\rangle$, and $lstatus(\langle p, r\rangle) \neq$ unknown for all $r \neq q$, in s'.

Claims about s':

1. $\text{TEST}(l, c)$ is at the head of $queue_p(\langle q, p\rangle)$, by precondition.
2. $l \leq nlevel(p)$, by assumption.
3. $c = nfrag(p)$, by assumption.
4. $testlink(p) = \langle p, q\rangle$, by assumption.
5. $lstatus(\langle p, r\rangle) \neq$ unknown, for all $r \neq q$, by assumption.
6. $p \in testset(f)$, by Claim 4 and TAR-C(b).
7. $minlink(f) = nil$, by Claim 6 and GC-C.
8. If $lstatus(\langle p, r\rangle) =$ branch, then $(p, r) \in subtree(f)$, for all $r \neq q$, by Claim 7 and TAR-A(a).
9. If $lstatus(\langle p, q\rangle) =$ rejected, then $fragment(r) = f$, for all $r \neq q$, by TAR-B.
10. $c = core(f)$, by Claims 1, 2 and 3 and the argument just given for TAR.
11. $fragment(q) = f$, by Claims 1 and 10 and TAR-N.
12. There is no external link of p, by Claims 8, 9, 11 and 5.
13. $nstatus(p) =$ find, by Claim 4 and GHS-H.

$TestNode(p)$ is enabled in $\mathcal{S}_{DC}(s')$ by Claims 6, 12 and 13. Its effects are mirrored in $\mathcal{S}_{DC}(s)$ by the earlier discussion about $testset(f)$ and $minlink(f)$ and by Claim 12. (The disposition of the rest of the derived variables should be obvious.)

Now suppose $l > nlevel(p)$ or $c \neq nfrag(p)$ or $testlink(p) \neq \langle p, q\rangle$ or there is a link $\langle p, r\rangle$ with $lstatus(\langle p, r\rangle) =$ unknown and $r \neq q$. Then $\mathcal{S}_{DC}(s') = \mathcal{S}_{DC}(s)$ by inspection and earlier discussion of $testset(f)$ and $minlink(f)$.

NOT and CON: We want to show$\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for $x = NOT$ and CON. The only derived variables for these two that are not obviously unchanged are $minlink(f)$ and $rootchanged(f)$. (Because of the presence of the TEST message in $queue(\langle q, p\rangle$, GHS-A(b) implies that $awake =$ true in s', so changes to $nstatus(p)$ do not change $awake$.) Since we already showed $minlink(f)$ is unchanged, it is obvious that $rootchanged(f)$ is unchanged.

(3a) GHS-A is vacuously true by the assumption that $nstatus(p) \neq$ sleeping.

GHS-B: First we show that if the hypotheses of this predicate are false for a link in s', then they are still false in s. The only way they could go from false to true is by $lstatus(\langle p,q \rangle)$ going from unknown to rejected. But since TEST is in $queue(\langle q,p \rangle)$ in s', by GHS-C no CONNECT is in $queue(\langle q,p \rangle)$ in s', or in s.

Now we show that the state changes do not invalidate (a) through (d) for a link, assuming that the hypotheses are true for that link in s'.

Case A: TEST is requeued. No change affects the predicate.

Case B: ACCEPT or REJECT is added to $queue(\langle p,q \rangle)$. We already showed that no CONNECT is in $queue(\langle q,p \rangle)$. Because of the TEST in $queue(\langle q,p \rangle)$, the preconditions of the predicate are not true for a CONNECT in $queue(\langle p,q \rangle)$ in s'.

Case C: TEST is added to some $queue(\langle p,r \rangle)$. Since $lstatus(\langle p,r \rangle) =$ unknown, the preconditions are not true in s' for a CONNECT in $queue(\langle r,p \rangle)$. Since the TEST is added, $testlink(p) = \langle p,q \rangle$ in s'. By GHS-H, $nstatus(p) =$ find in s'. So by GHS-B(c), the preconditions are not true in s' for a CONNECT in $queue(\langle p,r \rangle)$.

Case D: REPORT is added to $queue(inbranch(p))$. Let $\langle p,r \rangle = inbranch(p)$ in s'. As in Case 3, the predicate is vacuously true for a CONNECT in $queue(\langle p,r \rangle)$. As in Case 3, $nstatus(p) =$ find in s', so p is up-to-date by DC-I(a). By GHS-I, $nlevel(p) = level(f)$. Since by DC-L, $(p,r) \in subtree(f)$, there cannot be an INITIATE$(nlevel(p)+1,*,*)$ message in $queue(\langle r,p \rangle)$. By GHS-B(a), the preconditions are not true for a CONNECT in $queue(\langle r,p \rangle)$.

GHS-H: By code.

GHS-J: If p is removed from $testset(f)$, then as in Claim 12 of (3b)/(3c) for *DC*, there is no external link of p.

GHS-C: *Case 1:* REJECT is added to $queue(\langle p,q \rangle)$. Then $l \leq nlevel(p)$, $c = nfrag(p)$, and $testlink(p) \neq \langle p,q \rangle)$ in s'. As argued in Lemma 17, verifying (3c) of Case 1 for $\pi = ReceiveTest$, $\langle p,q \rangle$ is an internal link of f. By TAR-E(a), $(p,q) \neq core(f)$, so by CON-E, no CONNECT is in $queue(\langle p,q \rangle)$.

Case 2: TEST is added to $queue(\langle p,r \rangle)$. Then in s', $l \leq nlevel(p)$, $c = nfrag(p)$, $testlink(p) = \langle p,q \rangle$, and $lstatus(\langle p,r \rangle) =$ unknown.

Case 2a: $\langle p,r\rangle$ is an internal link of f. By TAR-A(b), $(p,r) \notin \mathit{subtree}(f)$. By COM-F, $(p,r) \neq \mathit{core}(f)$. By CON-E, no CONNECT is in $\mathit{queue}(\langle p,r\rangle)$.

Case 2b: $\langle p,r\rangle$ is an external link of f. By GHS-H, $\mathit{nstatus}(p) =$ find. Thus $\mathit{minlink}(f) = \mathit{nil}$. By CON-D, no CONNECT is in $\mathit{queue}(\langle p,r\rangle)$.

GHS-G: Suppose ACCEPT is added to $\mathit{queue}(\langle p,q\rangle)$. Then $l \leq \mathit{nlevel}(p)$ in s'. As argued in Lemma 17, verifying TAR-F for $\pi = \mathit{ReceiveTest}$, $l = \mathit{level}(\mathit{fragment}(q))$. By GHS-F, $l \leq \mathit{nlevel}(q)$. So $l = \mathit{nlevel}(q)$.

No changes affect the rest.

viii) π is ReceiveAccept($\langle$q,p$\rangle$). Let $f = \mathit{fragment}(p)$.

(3b)/(3c) $\mathcal{A}_{TAR}(s',\pi) = \pi$. $\mathcal{A}_{DC}(s',\pi) = \mathit{TestNode}(p)$. $\mathcal{A}_x(s',\pi)$ is empty for all other x.

An argument similar to that used in $\pi = \mathit{ReceiveTest}(\langle q,p\rangle, l, c)$, Case 2, shows that $\mathit{minlink}(f)$ is unchanged.

TAR: Claims about s':

1. ACCEPT is at the head of $\mathit{queue}_p(\langle q,p\rangle)$, by precondition.
2. There is a protocol message for $\langle p,q\rangle$, by Claim 1.
3. $\mathit{testlink}(p) = \langle p,q\rangle$, by Claim 2 and TAR-D.
4. No FIND message is headed toward p, by Claim 3 and GHS-M.
5. No CONNECT message is in $\mathit{queue}(\langle r,t\rangle)$, where $(r,t) = \mathit{core}(f)$ and $p \in \mathit{subtree}(r)$, by Claim 3 and GHS-M.

Claims about s:

6. $\mathit{testlink}(p) = \mathit{nil}$, by code.
7. No FIND message is headed toward p, by Claim 4.
8. No CONNECT message is in $\mathit{queue}(\langle r,t\rangle)$, where $(r,t) = \mathit{core}(f)$ and $p \in \mathit{subtree}(r)$, by Claim 5 and code.
9. $p \notin \mathit{testset}(f)$, by Claims 6, 7 and 8.

π is enabled in $\mathcal{S}_{TAR}(s')$ by Claim 1; its effects are mirrored in $\mathcal{S}_{TAR}(s)$ by Claims 6 and 9, and discussion of $\mathit{minlink}(f)$. (The disposition of the remaining derived variables should be obvious.)

DC: More Claims about s':

10. $p \in \textit{testset}(f)$, by Claim 3.
11. $\textit{minlink}(f) = \textit{nil}$, by Claim 10.
12. $\textit{fragment}(q) \neq f$, by Claim 1 and TAR-F.
13. $\textit{level}(f) \leq \textit{level}(\textit{fragment}(q))$, by Claim 1 and TAR-F.
14. $\textit{lstatus}(\langle p, q\rangle) \neq$ branch, by Claims 11 and 12 and TAR-A(a).
15. $\langle p, q\rangle$ is the minimum-weight external link of p with *lstatus* unknown, by Claims 3 and 14 and TAR-C(d).
16. If $\textit{lstatus}(\langle p, r\rangle) =$ rejected, then $\langle p, r\rangle$ is not external, for all r, by TAR-B.
17. If $\textit{lstatus}(\langle p, r\rangle) =$ branch, then $\langle p, r\rangle$ is not external, for all r, by Claim 11 and TAR-A(a).
18. If $\langle p, r\rangle$ is external, then $\textit{lstatus}(\langle p, r\rangle) =$ unknown, for all r, by Claims 16 and 17.
19. $\langle p, q\rangle$ is the minimum-weight external link of p, by Claims 15 and 18.
20. $\textit{nstatus}(p) =$ find, by Claim 3 and GHS-H.

TestNode(p) is enabled in $\mathcal{S}_{DC}(s')$ by Claims 10, 19 and 13, and 20. Its effects are mirrored in $\mathcal{S}_{DC}(s)$ by Claims 9, 19 and 6.

NOT and *CON*: It is easy to verify that $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for $x = NOT$ and *CON*.

(3a) GHS-A: By Claim 20, vacuously true in s.

GHS-B: Suppose a REPORT message is added to $\textit{queue}(\langle p, r\rangle)$ in s. Let $\langle p, r\rangle = \textit{inbranch}(p)$. By Claim 20 and DC-I(a), p is up-to-date in s'. By GHS-I, $\textit{nlevel}(p) = \textit{level}(f)$. By DC-L, $(p, r) \in \textit{subtree}(f)$, so no INITIATE$(\textit{nlevel}(p) + 1, *, *)$ can be in $\textit{queue}(\langle p, r\rangle)$ or $\textit{queue}(\langle r, p\rangle)$. By GHS-B(a), the preconditions for a CONNECT in $\textit{queue}(\langle p, r\rangle)$ or $\textit{queue}(\langle r, p\rangle)$ are not true in s', or in s.

GHS-H: By code, $\textit{testlink}(p) = \textit{nil}$.

GHS-J: By Claim 19 and GHS-G.

No changes affect the rest.

ix) π is ReceiveReject($\langle$q,p$\rangle$). Let $f = \textit{fragment}(p)$.

(3b)/(3c) $\mathcal{A}_{TAR}(s', \pi) = \pi$.

$\mathcal{A}_{DC}(s', \pi) = TestNode(p)$ if there is no $r \neq q$ such that $lstatus(\langle p, r\rangle) =$ unknown in s, and is empty otherwise.

$\mathcal{A}_x(s', \pi)$ is empty for all other x.

An argument similar to that in $\pi = ReceiveTest(\langle q, p\rangle, l, c)$, Case 2, shows that $minlink(f)$ is unchanged.

TAR: *Claims about s':*

1. REJECT is at the head of $queue_p(\langle q, p\rangle)$, by precondition.
2. There is a protocol message for $\langle p, q\rangle$, by Claim 1.
3. $testlink(p) = \langle p, q\rangle$, by Claim 2 and TAR-D.
4. No FIND message is headed toward p, by Claim 3 and GHS-M.
5. No CONNECT message is in $queue(\langle r, t\rangle)$, where $(r, t) = core(f)$ and $p \in subtree(r)$, by Claim 3 and GHS-M.
6. $nstatus(p) =$ find, by Claim 3 and GHS-H.
7. $nlevel(p) = level(f)$, by Claim 6, DC-I(a) and GHS-I.
8. $nfrag(p) = core(f)$, by Claim 7 and NOT-A.

Claims about s:

9. If there is no link $\langle p, r\rangle$ with $lstatus(\langle p, r\rangle) =$ unknown (in s'), then $testlink(p) = nil$ (in s), by code.
10. No FIND message is headed toward p, by Claim 4.
11. No CONNECT message is in $queue(\langle r, t\rangle)$, by Claim 5.
12. If there is no link $\langle p, r\rangle$ with $lstatus(\langle p, r\rangle) =$ unknown (in s'), then $p \notin testset(f)$ (in s), by Claims 9, 10 and 11.

π is enabled in $\mathcal{S}_{TAR}(s')$ by Claim 1. Its effects are mirrored in $\mathcal{S}_{TAR}(s)$ by Claims 9, 12, 7 and 8, and earlier discussion of $minlink(f)$.

DC: If there is a link $\langle p, r\rangle$ such that $lstatus(\langle p, r\rangle) =$ unknown and $r \neq q$, then it is easy to check that $\mathcal{S}_{DC}(s') = \mathcal{S}_{DC}(s)$. Suppose there is no unknown link (other than $\langle p, q\rangle$).

More claims about s':

13. $lstatus(\langle p, r\rangle) \neq$ unknown, for all $r \neq q$, by assumption.
14. $minlink(f) = nil$, by Claim 6.
15. If $lstatus(\langle p, r\rangle) =$ branch, then $(p, r) \in subtree(f)$, for all $r \neq q$, by Claim 14 and TAR-A(a).

16. If $lstatus(\langle p,r\rangle) =$ rejected, then $fragment(r) = f$, for all $r \neq q$, by TAR-B.
17. $fragment(q) = f$, by Claim 1 and TAR-G.
18. There are no external links of p, by Claims 13, 15, 16 and 17.
19. $p \in testset(f)$, by Claim 3 and TAR-C(b).

$TestNode(p)$ is enabled in $\mathcal{S}_{DC}(s')$ by Claims 19, 18 and 6. Its effects are mirrored in $\mathcal{S}_{DC}(s)$ by Claims 9 and 12.

NOT and *CON*: It is easy to show that $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for $x = NOT$ and CON.

(3a) GHS-A: Vacuously true by Claim 6.

GHS-B: Either a TEST or a REPORT message is added. The argument is very similar to that in $\pi = ReceiveTest(\langle q,p\rangle, l, c)$, Case 2 of (a).

GHS-C: Only affected if a TEST is added. The argument is very similar to that in $\pi = ReceiveTest(\langle q,p\rangle, l, c)$, Case 2 of (a).

GHS-H: The argument is very similar to that in $\pi = ReceiveTest(\langle q,p\rangle, l, c)$, Case 2 of (a).

GHS-I: Suppose p is removed from $testset(f)$. By Claim 12, this only happens when there are no more unknown links. By Claim 18, p has no external links if there are no more unknown links.

No changes affect the rest.

x) π is ReceiveReport(⟨q,p⟩,w). Let $f = fragment(p)$.

(3b)/(3c) *Case 1:* $(p,q) = core(f)$, $nstatus(p) \neq$ find and $w > bestwt(p)$ in s'. This case is divided into two subcases; first we prove some claims true in both subcases. Let $\langle r,t\rangle$ be the minimum-weight external link of f in s'. (Below, we show it exists.)

Claims about s':

1. REPORT(w) is at the head of $queue(\langle q,p\rangle)$, by assumption.
2. $(p,q) = core(f)$, by assumption.
3. $nstatus(p) \neq$ find, by assumption.

4. $w > bestwt(p)$, by assumption.
5. $ReceiveReport(\langle q,p\rangle, w)$ is enabled in $\mathcal{S}_{DC}(s')$, by Claim 1.
6. $ComputeMin(f)$ (for GC) is enabled in $\mathcal{S}_4(\mathcal{S}_{DC}(s'))$, by Claims 2, 3, 4 and 5 and argument in proof of Lemma 19, Case 1 of verifying (3c) for $\pi = ReceiveReport$.
7. $minlink(f) = nil$, by Claim 6.
8. $accmin(f) \neq nil$, by Claim 6.
9. $testset(f) = \emptyset$, by Claim 6.
10. $ComputeMin(f)$ (for COM) is enabled in $\mathcal{S}_2(\mathcal{S}_4(\mathcal{S}_{DC}(s')))$, by Claim 6 and argument in proof of Lemma 15, verifying (3c) for $\pi = ComputeMin$.
11. $level(f) \leq level(fragment(t))$, by Claim 10.
12. $accmin(f) = \langle r,t\rangle$, by Claims 8 and 9 and GC-A.
13. r is up-to-date, by Claim 9, DC-N, and choice of $\langle r,t\rangle$.
14. $nlevel(r) = level(f)$, by Claim 13 and GHS-I.
15. $nlevel(f) \leq nlevel(t)$, by Claims 9 and 13 and GHS-J.
16. No CONNECT message is in either queue of $core(f)$, by Claim 9.
17. No CONNECT message is in any internal queue of f, by Claim 16 and CON-E.
18. $inbranch(p) = \langle p,q\rangle$, by Claims 1 and 2 and DC-A(a).
19. p is up-to-date, by Claims 2, 9 and 18.
20. $findcount(p) = 0$, by Claim 3 and DC-H(b).
21. All children of p are completed, by Claims 19 and 20 and DC-K(a).
22. $r \in subtree(p)$, by Claims 1, 2, 3 and 4 and DC-P(b).
23. Following *bestlinks* from p leads along edges of $subtree(f)$ to $\langle r,t\rangle$, by Claims 9, 19, 21 and 22, choice of $\langle r,t\rangle$, and DC-K(b) and (c).

The following remarks apply to both Subcase 1a and Subcase 1b: $ComputeMin(f)$ is enabled in $\mathcal{S}_x(s')$ by Claims 7, 8 and 9 for $x = TAR$; by Claims 7, 14 and 15 (and definition of $\langle r,t\rangle$) for $x = NOT$; and by Claims 7, 11 and 17 for $x = CON$. π is obviously enabled in $\mathcal{S}_{DC}(s')$.

Subcase 1a: $lstatus(bestlink(p)) =$ branch. $\mathcal{A}_{DC}(s',\pi) = \pi$. $\mathcal{A}_x(s',\pi) = ComputeMin(f)$ for all other x.

More Claims about s':

24. $lstatus(bestlink(p)) =$ branch, by assumption.
25. $bestlink(p) \in subtree(f)$, by Claims 7 and 24 and TAR-A(a).
26. $p \neq r = mw\text{-}minnode(f)$, by Claims 23 and 25.

Claims about s:

27. The effects of π are reflected in $\mathcal{S}_{DC}(s)$, by code.
28. The effects of *ComputeMin*(f) are reflected in $\mathcal{S}_4(\mathcal{S}_{DC}(s))$, by Claim 27 and argument in proof of Lemma 19, Case 1 of verifying (3c) for $\pi =$ *ReceiveReport.*
29. *minlink*$(f) = \langle r, t\rangle$, by Claims 28 and 12.
30. Following *bestlinks* from p leads to $\langle r, t\rangle$, by Claim 23.
31. *tominlink*$(p) =$ *bestlink*(p), by Claims 30 and 24.
32. $p \neq$ *minnode*(f), by Claims 29 and 24.
33. $p =$ *root*(f), by Claims 2, 22 and 29.

By Claims 3, 4 and 17, procedure *ChangeRoot*(p) is executed in *GHS*. The effects of *ComputeMin*(f) are reflected in $\mathcal{S}_x(s)$ by Claims 29 and 12 for $x = TAR$; by Claim 29 and choice of $\langle r, t\rangle$ for $x = NOT$; and by Claims 29, 31, 32, 33 and choice of $\langle r, t\rangle$ for $x = CON$. The effects of π are reflected in $\mathcal{S}_{DC}(s)$ by Claim 27.

Subcase 1b: lstatus(*bestlink*$(p)) \neq$ branch.

$\mathcal{A}_{DC}(s', \pi) = \pi\ t_{DC}$ *ChangeRoot*(f), where t_{DC} is the result of applying π to $\mathcal{S}_{DC}(s')$.

$\mathcal{A}_{CON}(s', \pi) =$ *ComputeMin*(f).

For all other x, $\mathcal{A}_x(s', \pi) =$ *ComputeMin*$(f)\ t_x$ *ChangeRoot*(f), where t_x is the result of applying *ComputeMin*(f) to $\mathcal{S}_x(s')$.

More claims about s':

34. *lstatus*(*bestlink*$(p)) \neq$ branch.
35. *bestlink*$(p) = \langle r, t\rangle$, by Claims 23, 34 and 7 and TAR-A(b).
36. $p = r =$ *mw-minnode*(f), by Claim 35.
37. *nstatus*$(q) \neq$ sleeping, by Claim 1 and GHS-A.
38. *awake* $=$ true, by Claim 37.
39. *rootchanged*$(f) =$ false, by Claim 7 and COM-B.

Claims about t_x, $x \neq CON$:

40. If $x = TAR$, then *minlink*$(f) = \langle r, t\rangle$, by Claim 12.
41. If $x = NOT$, then *minlink*$(f) = \langle r, t\rangle$, by choice of $\langle r, t\rangle$.
42. If $x = DC$, then *minlink*$(f) = \langle r, t\rangle$, by Claims 6 and 12 and argument in proof of Lemma 15, verifying (3c) for $\pi =$ *ComputeMin.*

43. *awake* = true, by Claim 38.
44. *rootchanged*(f) = false, by Claim 39.

The effects of π are mirrored in t_{DC} and of *ComputeMin*(f) in t_{TAR} and t_{NOT} by definition. *ChangeRoot*(f) is enabled in t_x by Claims 40, 43 and 44 for $x = TAR$; by Claims 41, 43 and 44 for $x = NOT$; and by Claims 42, 43 and 44 for $x = DC$.

Claims about s:

45. *minlink*$(f) = \langle r, t\rangle$, by argument in proof of Lemma 19, Case 1 of verifying (3c) for $\pi =$ *ReceiveReport*.
46. *lstatus*(*bestlink*(p)) = branch, by code.
47. *lstatus*(*minlink*(p)) = branch, by Claims 35 and 45.
48. CONNECT is added to *queue*(*bestlink*(p)), by code.
49. *rootchanged*(f) = true, by Claims 45 and 48.

The effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_x(s)$ by Claims 47 and 49 for $x = TAR$; by Claim 49 for $x = DC$ and NOT. The effects of *ComputeMin*(f) are mirrored in $\mathcal{S}_{CON}(s)$ by Claims 36, 14 and 45.

Case 2: $(p,q) \neq$ *core*(f) or *nstatus*(p) = find or $w \leq$ *bestwt*(p) in s'.

$\mathcal{A}_{DC}(s', \pi) = \pi$. $\mathcal{A}_x(s', \pi)$ is empty for all other x.

Subcase 2a: $(p,q) \neq$ *core*(f) in s'. Suppose $\langle p,q\rangle =$ *inbranch*(p) in s'. By DC-B(b), *dcstatus*(p) = unfind. Thus, the only effect is to remove the REPORT message. Thus $\mathcal{S}_{DC}(s')\pi\mathcal{S}_{DC}(s)$ is an execution fragment of DC. As proved in Lemma 19, Case 2a of verifying (3b) for $\pi =$ *ReceiveReport*, *minlink*(f) is unchanged. Thus $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all $x \neq DC$.

Now suppose $\langle p, q\rangle \neq$ *inbranch*(p).

Claims about s':

1. REPORT is at head of *queue*$(\langle q,p\rangle)$, by precondition.
2. $(p,q) \neq$ *core*(f), by assumption.
3. $\langle p,q\rangle \neq$ *inbranch*(p), by assumption.
4. *dcstatus*(p) = find, by Claims 1, 2 and 3 and DC-A(g).
5. p is up-to-date, by Claim 4 and DC-I(a).
6. q is a child of p, by Claims 3 and 5.

7. *findcount*$(p) > 0$, by Claims 1, 5 and 6 and DC-K(a).
8. No FIND message is headed toward p, by Claim 7 and GHS-M.
9. No CONNECT is in *queue*$(\langle r,t \rangle)$, where (r,t) = *core*(f) and $p \in$ *subtree*(r), by Claim 7 and GHS-M.
10. $p \in$ *testset*(f) if and only if *testlink*$(p) \neq$ *nil*, by Claims 8 and 9.

Obviously, π is enabled in $\mathcal{S}_{DC}(s')$. By Claim 10 and inspection, the effects of π are mirrored in $\mathcal{S}_{DC}(s)$. Since the proof of Lemma 19, Case 2a of verifying (3b) for π = *ReceiveReport*, shows *minlink*(f) is unchanged, $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all $x \neq DC$.

Subcase 2b: (p,q) = *core*(f) and *nstatus*(p) = find in s'. Since REPORT(w) is at the head of *queue*$(\langle q,p \rangle)$, DC-A(a) implies that *inbranch*$(p) = \langle p,q \rangle$. Thus, the only change is that the REPORT message is requeued. Obviously $\mathcal{S}_{DC}(s')\pi\mathcal{S}_{DC}(s)$ is an execution fragment of DC, and $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all $x \neq DC$.

Subcase 2c: (p,q) = *core*(f), *nstatus*(p) = find and $w \leq$ *bestwt*(p) in s'. As in Subcase 2b, *inbranch*$(p) = \langle p,q \rangle$. The only change is that the REPORT message is removed. Thus $\mathcal{S}_{DC}(s')\pi\mathcal{S}_{DC}(s)$ is an execution fragment of DC. As proved in Lemma 19, Case 2c of verifying (3b) for π = *ReceiveReport*, *minlink*(f) is unchanged in s. Thus $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all $x \neq DC$.

(3a) *Case 1: inbranch*$(p) \neq \langle p,q \rangle$.

GHS-A: By DC-A(a), $(p,q) \neq$ *core*(f). By DC-A(g), *dcstatus*(p) = find. The predicate is vacuously true.

GHS-B: Only the addition of a REPORT message affects this predicate. The argument is very similar to that in π = *ReceiveTest*$(\langle q,p \rangle, l, c)$, Case 2, of (3a).

GHS-H: By code (in procedure *Report*(p)).

No change affects the rest.

Case 2: inbranch$(p) = \langle p,q \rangle$. If *nstatus*$(p)$ = find or $w \leq$ *bestwt*(p), then no change affects any predicate. Suppose *nstatus*$(p) \neq$ find and $w >$ *bestwt*(p).

GHS-A: By DC-B(a), $subtree(p) \neq \{p\}$. By GHS-A(a), $nstatus(p) \neq$ sleeping, so the predicate is vacuously true.

GHS-B: Let $\langle p,r \rangle = bestlink(p)$ in s'. If $lstatus(\langle p,r \rangle) =$ branch, then no change affects this predicate. Suppose $lstatus(\langle p,r \rangle) \neq$ branch. As shown in (3b)/(3c), Claim 35 of Case 1b, $bestlink(p)$ is the minimum-weight external link of f. Thus $lstatus(\langle r,p \rangle) \neq$ rejected by TAR-B, and if $lstatus(\langle r,p \rangle) =$ branch, then there is a CONNECT in $queue(\langle r,p \rangle)$. So the predicate is vacuously true for the CONNECT added to $queue(\langle p,r \rangle)$. If there is a leftover CONNECT in $queue(\langle r,p \rangle)$, then the predicate is vacuously true because of the new CONNECT in $queue(\langle p,r \rangle)$.

GHS-C: Let $\langle p,r \rangle = bestlink(p)$ in s'. Since $bestlink(p)$ is external (as shown in (3b)/(3c)), no REJECT is in $queue(\langle p,r \rangle)$ by TAR-G. Also since it is external, $lstatus(\langle p,r \rangle) \neq$ rejected by TAR-B. Suppose a TEST is in $queue(\langle p,r \rangle)$. By TAR-D, $testlink(p) = \langle p,r \rangle$, and by GHS-H, $nstatus(p) =$ find, which contradicts the assumption for this case. Also since the link is external, no FIND is in $queue(\langle p,r \rangle)$ by DC-D(a).

No change affects the rest.

xi) π is ReceiveChangeRoot($\langle$q,p$\rangle$).

(3b)/(3c) There are two cases. First we prove some facts true in both cases.

Claims about s':

1. CHANGEROOT is at the head of $queue(\langle q,p \rangle)$, by precondition.
2. $minlink(f) \neq nil$, by Claim 1 and CON-C.
3. $rootchanged(f) =$ false, by Claim 1 and CON-C.
4. $p \in subtree(q)$, by Claim 1 and CON-C.
5. $minnode(f) \in subtree(p)$, by Claim 1 and CON-C.
6. $nlevel(minnode(f)) = level(f)$, by NOT-D.
7. $testset(f) = \emptyset$, by Claim 2 and GC-C
8. $minlink(f)$ is the minimum-weight external link of f, by Claim 2 and COM-A.
9. $minnode(f)$ is up-to-date, by Claims 7 and 8 and DC-N.
10. p is up-to-date, by Claims 5, 7 and 9.
11. No REPORT message is headed toward $mw\text{-}root(f)$, by Claim 2.
12. No REPORT message is headed toward p, by Claims 4 and 11.
13. $dcstatus(p) =$ unfind, by Claims 7 and 12 and DC-I(b).
14. $findcount(p) = 0$, by Claim 13 and DC-H(b).

15. All children of p are completed, by Claims 10 and 14 and DC-K(a).
16. Following *bestlinks* from p leads along edges in *subtree*(f) to the minimum-weight external link of *subtree*(p), by Claims 7, 10 and 15 and DC-K(b) and (c).

Case 1: *lstatus*(*bestlink*(p)) $\neq$ branch in s'.

$\mathcal{A}_{CON}(s', \pi) = \pi$. $\mathcal{A}_x(s', \pi) =$ *ChangeRoot*(f) for all other x.

More claims about s':

17. *lstatus*(*bestlink*(p)) $\neq$ branch, by assumption.
18. *bestlink*(p) is not in *subtree*(f), by Claim 17 and TAR-A(b).
19. *bestlink*(p) $=$ *minlink*(f), by Claims 5, 8, 16 and 18.
20. *nstatus*(q) $\neq$ sleeping, by Claim 1 and GHS-A(b).
21. *awake* $=$ true, by Claim 20.

Claims about s:

22. *lstatus*(*bestlink*(p)) $=$ branch, by code.
23. CONNECT is in *queue*(*bestlink*(p)), by code.
24. MSF does not change, Claims 22 and 23.
25. *bestlink*(p) $=$ *minlink*(f), by Claims 19 and 24.
26. *rootchanged*(f) $=$ true, by Claims 23 and 25.

ChangeRoot(f) is enabled in $\mathcal{S}_x(s')$ by Claims 2, 3 and 21, for all $x \neq CON$. The effects of *ChangeRoot*(f) are mirrored in $\mathcal{S}_x(s)$ by Claims 22, 25 and 26 for $x = TAR$; and by Claim 26 for $x = DC$ and NOT. π is enabled in $\mathcal{S}_{CON}(s')$ by Claim 1; its effects are mirrored in $\mathcal{S}_{CON}(s)$ by Claims 6 and 19.

Case 2: *lstatus*(*bestlink*(p)) $=$ branch in s'.

$\mathcal{A}_{CON}(s', \pi) = \pi$. $\mathcal{A}_x(s', \pi)$ is empty for all other x.

More Claims about s':

27. *lstatus*(*bestlink*(p)) $=$ branch, by assumption.
28. *lstatus*(*minlink*(f)) $\neq$ branch, by Claim 3 and TAR-H.
29. *bestlink*(p) is in *subtree*(f), by Claims 27 and 28 and TAR-A(a).

30. $p \neq minnode(f)$, by Claims 16 and 29.
31. $bestlink(p) = tominlink(f)$, by Claims 8, 16 and 29.
32. $nlevel(p) = level(f)$, by Claim 10 and GHS-I.

Obviously, all derived (and non-derived) variables are unchanged, except *cqueues*. Thus, $\mathcal{S}_x(s') = \mathcal{S}_x(s)$ for all $x \neq CON$. π is enabled in $\mathcal{S}_{CON}(s')$ by Claim 1; its effects are mirrored in $\mathcal{S}_x(s)$ by Claims 30, 31 and 32.

(3a) GHS-A: By CON-C, $(p,q) \in subtree(f)$. By GHS-A(a), $nstatus(p) \neq$ sleeping in s', so the predicate is vacuously true in s.

GHS-B: Essentially the same argument as in $\pi = ReceiveReport(\langle q,p\rangle, w)$, Case 2 of (3a).

GHS-C: Essentially the same argument as in $\pi = ReceiveReport(\langle q,p\rangle, w)$, Case 2 of (3a).

No change affects the rest. □

Let $P'_{GHS} = \bigwedge_{x \in I}(P'_x \circ \mathcal{S}_x) \wedge P_{GHS}$.

Corollary 26: *P'_{GHS} is true in every reachable state of GHS.*

Proof: By Lemmas 1 and 25. □

4.3 Liveness

We show a path in the lattice along which liveness properties are preserved. The path is HI, COM, GC, TAR, GHS. In showing the edge from GHS to TAR, it is useful to know some liveness relationships between GC and DC, and between COM and CON.

The reason for considering liveness relationships in other parts of the lattice is to take advantage of the more abstract forms of the algorithm. For instance, the essence of showing that the GHS algorithm will take steps leading to the simulation of *ComputeMin*(f) in TAR is the same as showing that DC takes steps leading to the simulation of *ComputeMin*(f) in GC. (These steps are the convergecast of REPORT messages back to the core.) DC is not cluttered with variables and actions that are not relevant to this argument, unlike GHS. Thus, we make the argument for DC to GC, and then apply Lemma 7 for the GHS to TAR situation.

For the same reason, we show that the progression of CHANGEROOT messages in CON leads to the simulation of *ChangeRoot*(f) in COM, and that the movement of CONNECT messages over links in CON leads to *Absorb* and *Merge* in COM, and then apply Lemma 7.

4.3.1 COM is Equitable for HI

The main idea here is to show that as long as there exist two distinct subgraphs, progress is made; the heart of the argument is showing that some fragment at the lowest level can always take a step. This requires a global argument that considers all the fragments.

Lemma 27: *COM is equitable for HI via $\mathcal{M}_1$.*

Proof: By Corollary 14, $(P_{HI} \circ \mathcal{S}_1) \wedge P_{COM}$ is true in every reachable state of P_{COM}. Thus, in the sequel we will use the HI and COM predicates.

For each locally-controlled action φ of HI, we must show that COM is equitable for φ via $\mathcal{M}_1$.

i) φ **is Start(p) or NotInTree(l).** Since φ is enabled in $\mathcal{S}_1(s)$ if and only if it is also enabled in s, and since $\mathcal{A}_1(s,\varphi)$ includes φ, for any state s, Lemma 5 shows that COM is equitable for φ via $\mathcal{M}_1$.

ii) φ **is Combine(F,F',e).** We show COM is progressive for φ via $\mathcal{M}_1$; Lemma 6 implies COM is equitable for φ via $\mathcal{M}_1$.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of COM and internal actions ψ of COM enabled in s. For reachable state s, let $v_\varphi(s) = (x, y, z)$, where x is the number of fragments in s, y is the number of fragments f with $rootchanged(f)$ = false in s, and z is the number of fragments f with $minlink(f)$ $= nil$ in s. (Two triples are compared lexicographically.)

(1) Let s be a reachable state of COM in E_φ. We now demonstrate that some action ψ is enabled in s with $(s, \psi) \in \Psi_\varphi$.

Claims:

1. $awake$ = true in $\mathcal{S}_1(s)$, by precondition.
2. $F \neq F'$ in $\mathcal{S}_1(s)$, by precondition.
3. $awake$ = true in s, by Claim 1 and definition of $\mathcal{S}_1$.
4. There exist f and g in *fragments* such that $subtree(f) = F$ and $subtree(g) = F'$ in s, by Claim 2 and definition of $\mathcal{S}_1$.
5. $f \neq g$ in s, by Claims 2 and 4.

Let $l = \min\{level(f') : f' \in fragments\}$ in s. (By Claim 4, *fragments* is not empty in s, so l is defined.) Let $L = \{f' \in fragments : level(f') = l\}$.

Case 1: There exists $f' \in L$ with $minlink(f') = nil$. Let $\psi = ComputeMin(f')$. We now show ψ is enabled in s. By Claim 5, the minimum-weight external link $\langle p, q\rangle$ of f' exists. By choice of l, $level(f') \leq level(fragment(q))$. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2: For all $f' \in L$, $minlink(f') \neq nil$.

Case 2.1: There exists $f' \in L$ with $rootchanged(f')$ = false. Let $\psi = ChangeRoot(f')$. ψ is enabled in s by Claim 3 and the assumption for Case 2. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2.2: For all $f' \in L$, $rootchanged(f')$ = true.

Case 2.2.1: There exists fragment $g' \in L$ with $level(f') > l$, where $f' = fragment(target(minlink(g')))$. (By COM-G, f' is uniquely defined.) Let $\psi = Absorb(f', g')$. Obviously ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

Case 2.2.2: There is no fragment $g' \in L$ such that $level(f') > l$, where $f' = fragment(target(minlink(g')))$. Pick any fragment f_1 such that $level(f_1) = l$. For $i > 1$, define f_i to be $fragment(target(minlink(f_{i-1})))$.

More claims about s':

6. f_i is uniquely defined, for all $i \geq 1$. *Proof:* If $i = 1$, by definition. Suppose it is true for $i - 1 \geq 1$. Then it is true for i by COM-G, since *minlink*(f_i) is well-defined and non-nil.
7. *minlink*(f_i) is the minimum-weight external link of f_i, for all $i \geq 1$, by COM-A.
8. $f_i \neq f_{i-1}$, for all $i > 1$, by Claims 6 and 7 and definition of f_i.
9. If *minedge*$(f_i) \neq$ *minedge*(f_{i-1}) for some $i > 1$, then f_{i+1} is not among $f_1, \ldots, f_i$, by Claims 7 and 8, and since the edge-weights are totally ordered.
10. There are only a finite number of fragments, by COM-D and the fact that $V(G)$ is finite.

By Claims 9 and 10, there is an $i > 1$ such that *minedge*$(f_i) =$ *minedge*(f_{i-1}). Let $\psi = Merge(f_i, f_{i-1})$. Obviously ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

---

(2) Consider a step (s', π, s) of COM, where s' is reachable and in E_φ, $(s', \pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) $v_\varphi(s) \leq v_\varphi(s')$, because there is no action of COM that increases the number of fragments; only a *Merge* action increases the number of fragments with *minlink* equal to *nil* or *rootchanged* equal to false, and it simultaneously causes the number of fragments to decrease.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. Then $v_\varphi(s) < v_\varphi(s')$, since *Absorb* and *Merge* decrease the number of fragments, *ComputeMin* maintains the number of fragments and the number of fragments with *rootchanged* = false and decreases the number with *minlink* = *nil*, and *ChangeRoot* maintains the number of fragments and decreases the number with *rootchanged* = false.

(c) Suppose $(s', \pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s', \psi) \in \Psi_\varphi$. Then ψ is still enabled in s, since the only possible values of π are *Start*(p), *InTree*(l) and *NotInTree*(l), none of which disables ψ. By definition, $(s, \psi) \in \Psi_\varphi$.

iii) φ is InTree($\langle$p,q$\rangle$). We show COM is progressive for φ via $\mathcal{M}_1$; Lemma 6 implies that COM is equitable for φ via $\mathcal{M}_1$.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of COM and actions ψ of COM enabled in s such that ψ is either an internal action or is φ.

For reachable state s, let $v_\varphi(s) = v_{Combine(F,F',e)}(s)$.

(1) Let s be a reachable state of COM in E_φ. We now demonstrate that some action ψ is enabled in s with $(s, \psi) \in \Psi_\varphi$.

If $(p,q) \in F$ for some F in $\mathcal{S}_1(s)$, then $(p,q) \in \mathit{subtree}(\mathit{fragment}(p))$ in s. Let $\psi = \mathit{InTree}(\langle p,q \rangle)$.

Suppose $\langle p,q \rangle$ is the minimum-weight external link of some F in $\mathcal{S}_1(s)$. Then there is more than one fragment. Essentially the same argument as in $\varphi = \mathit{Combine}(F,F',e)$ shows that some $\mathit{Absorb}(f',g')$, or $\mathit{Merge}(f_i,f_{i+1})$, or $\mathit{ChangeRoot}(f')$, or $\mathit{ComputeMin}(f')$ is enabled in s.

(2) As in $\varphi = \mathit{Combine}(F,F',e)$, after noting that $\pi \neq \mathit{InTree}(\langle p,q \rangle)$. □

4.3.2 GC is Equitable for COM

The main part of the proof is showing that eventually every node is removed from *testset*(f), so that eventually *ComputeMin*(f) can occur. As in Section 4.3.1, a global argument is required, because a node might have to wait for many other fragments to merge or absorb until the level of the fragment at the other end of p's local minimum-weight external link is high enough.

Lemma 28: *GC is equitable for COM via* $\mathcal{M}_2$.

Proof: By Corollary 16, $(P'_{COM} \circ \mathcal{S}_2) \wedge P_{GC}$ is true in every reachable state of GC. Thus, in the sequel we will use the HI, COM, and GC predicates.

For each locally-controlled action φ of COM, we must show that GC is equitable for φ via $\mathcal{M}_2$.

i) φ is not ComputeMin(f) for any f. Since φ is enabled in s if and only if φ is enabled in $\mathcal{S}_2(s)$, and since $\mathcal{A}_2(s,\varphi)$ includes φ, for all s, Lemma 5 shows that GC is equitable for φ via $\mathcal{M}_2$.

ii) φ is ComputeMin(f). We show GC is progressive for φ via $\mathcal{M}_2$; Lemma 6 implies that GC is equitable for φ via $\mathcal{M}_2$.

Let Ψ_φ be the set of all pairs (s,π) of reachable states s of GC and internal actions π of GC enabled in s. For reachable state s, let $v_\varphi(s)$ be a quadruple with the following components:

1. the number of fragments;
2. the number of fragments with *rootchanged* = false;

3. the number of fragments with *minlink* $= nil$; and
4. the sum of the number of nodes in each fragment's *testset*.

(1) Let s be a reachable state of GC in E_φ. So *ComputeMin*(f) is enabled in $\mathcal{S}_2(s)$. We now show that some ψ is enabled in s with $(s, \psi) \in \Psi_\varphi$.

Let $\mathcal{G}$ be the directed graph defined as follows. There is one vertex of $\mathcal{G}$ for each element of *fragments* in s. We now specify the directed edges of $\mathcal{G}$. Let v and w be two vertices of $\mathcal{G}$, corresponding to fragments f' and g'. There is a directed edge from v to w in $\mathcal{G}$ if and only if there is a node p in *testset*(f') whose minimum-weight external link is $\langle p, q \rangle$, *fragment*$(q) = g'$, and *level*$(f') >$ *level*(g'). We will call fragment f' a *sink* if its corresponding vertex in $\mathcal{G}$ is a sink. (It should be obvious that there is at least one sink.)

Case 1: There is a sink f' such that *testset*$(f') \neq \emptyset$. Let $\psi =$ *TestNode*(p) for some $p \in$ *testset*(f'). Since f' is a sink, ψ is enabled in s. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2: For all sinks f', *testset*$(f') = \emptyset$.

Case 2.1: There is a sink f' such that *minlink*$(f') = nil$. Let $\psi =$ *ComputeMin*(f'). Since *ComputeMin*(f) is enabled in $\mathcal{S}_2(s)$, there are at least two fragments, so there is an external link of f'. By GC-B, *accmin*$(f') \neq nil$. Thus ψ is enabled in s. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2.2: For all sinks f', *minlink*$(f') \neq nil$.

Case 2.2.1: There is a sink f' such that *rootchanged*$(f') =$ false. Let $\psi =$ *ChangeRoot*(f'). Since *ComputeMin*(f) is enabled in $\mathcal{S}_2(s)$, *minlink*$(f) = nil$. By COM-C then, *awake* $=$ true. Thus ψ is enabled in s. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2.2.2: For all sinks f', *rootchanged*$(f') =$ true. By COM-A, the following two cases are exhaustive.

Case 2.2.2.1: There is a sink f' such that *level*$(g') >$ *level*(f'), where $g' =$ *fragment* (*target*(*minlink*(f'))). Let $\psi =$ *Absorb*(g', f'). Since f' is a sink, ψ is enabled in s. Obviously $(s, \psi) \in \Psi_\varphi$.

Case 2.2.2.2: For all sinks f', *level*$(g') =$ *level*(f'), where $g' =$ *fragment*(*target* (*minlink*(f'))). Let $m = \min\{$*level*$(f') : f'$ is a sink$\}$. Let f' be a sink with *level*$(f') = m$, and let $g' =$ *fragment*(*target*(*minlink*(f'))). If g' is not a sink, then from the vertex in $\mathcal{G}$ corresponding to g' a sink is reachable (along the directed edges) whose corresponding fragment is a sink with level less than m, contradicting our choice of m. Thus g' is a sink. Since the edge weights are totally ordered, by COM-A there are two sinks f' and g' at level m such that $minedge(f') = minedge(g')$. Let $\psi = Merge(f', g')$. Obviously ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

---

(2) Consider step (s', π, s) of GC, where s' is reachable and in E_φ, $(s', \pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) Obviously the external actions of GC do not change v_φ. This fact, together with (b) below, shows that $v_\varphi(s) \leq v_\varphi(s')$.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. If $\pi = TestNode(p)$, then component 4 of v_φ decreases and the rest stay the same. If $\pi = ComputeMin(f')$, then component 3 of v_φ decreases and the rest stay the same. If $\pi = ChangeRoot(f')$, then component 2 of v_φ decreases and the rest stay the same. If $\pi = Merge(f', g')$ or $Absorb(f', g')$, then component 1 of v_φ decreases.

(c) Suppose $(s', \pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s', \psi) \in \Psi_\varphi$. Since the only choice for π is an external action of GC, obviously ψ is enabled in s and $(s, \psi) \in \Psi_\varphi$.
□

4.3.3 TAR is Equitable for GC

The substantial argument here is that a node p's local test-accept-reject protocol eventually finishes, thus simulating $TestNode(p)$ in GC. Again, we need a global argument: to show that the recipient of p' TEST message eventually responds to it, we must show that the level of the recipient's fragment eventually is large enough. This proof is where the state component of the set Ψ in the definition of progressive is used. The receipt of a TEST message will generally make progress, but if it is requeued and the state is unchanged, no function on states can decrease; thus, we exclude such a state-action pair from Ψ.

Lemma 29: *TAR is equitable for GC via* $\mathcal{M}_3$.

Proof: By Corollary 18, $(P'_{GC} \circ \mathcal{S}_3) \wedge P_{TAR}$ is true in every reachable state of TAR. Thus, in the sequel we will use the HI, COM, GC, and TAR predicates.

For each locally-controlled action φ of GC, we must show that TAR is equitable for φ via $\mathcal{M}_3$.

i) φ is not TestNode(p) for any p, or InTree(l) or NotInTree(l) for any l. Since φ is enabled in s if and only if φ is enabled in $\mathcal{S}_3(s)$, and since $\mathcal{A}_3(s,\varphi)$ includes φ, for all s, Lemma 5 implies that TAR is equitable for φ via $\mathcal{M}_3$.

ii) φ is TestNode(p). We show TAR is progressive for φ via $\mathcal{M}_3$; Lemma 6 implies that TAR is equitable for φ via $\mathcal{M}_3$. In the worst case, we have to wait for the levels to have the correct relationship. This requires a "global" argument.

Let Ψ_φ be the set of all pairs (s,π) of reachable states s of TAR and internal actions π of TAR enabled in s, such that if $\pi = \mathit{ReceiveTest}(\langle q,r\rangle, l, c)$, then in s either $\mathit{level}(\mathit{fragment}(r)) \geq l$, or there is more than one message in $\mathit{tarqueue}_r(\langle q,r\rangle)$.

For reachable state s, let $v_\varphi(s)$ be a 10-tuple of:

1. the number of fragments in s,
2. the number of fragments f with $\mathit{rootchanged}(f) =$ false in s,
3. the number of fragments f with $\mathit{minlink}(f) = \mathit{nil}$ in s,
4. the number of nodes q such that $q \in \mathit{testset}(\mathit{fragment}(q))$ in s,
5. the number of links l such that either $\mathit{lstatus}(l) =$ unknown, or else $\mathit{lstatus}(l) =$ branch and there is a protocol message for l, in s,
6. the number of links l such that no ACCEPT or REJECT message is in $\mathit{tarqueue}(l)$ in s,
7. the number of links l such that no TEST message is in $\mathit{tarqueue}(l)$ in s,
8. the number of messages in $\mathit{tarqueue}_q(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$, in s,
9. the number of messages in $\mathit{tarqueue}_{qr}(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$, in s,
10. the number of messages in $\mathit{tarqueue}_r(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$, that are behind a TEST message in s.

(1) Let s be a reachable state of TAR in E_φ. We show that there exists an action ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

Let $l = \min\{\mathit{level}(f) : f \in \mathit{fragments}\}$.

Case 1: All fragments f at level l have $\mathit{rootchanged}(f) =$ true. Then some $\mathit{Absorb}(f,g)$ or $\mathit{Merge}(f,g)$ is enabled in s, as argued in Lemma 27, Case 2.2.1 for $\varphi = \mathit{Combine}$. Let ψ be one of these enabled actions.

Case 2: $\mathit{level}(f) = l$ and $\mathit{rootchanged}(f) \neq$ true, for some $f \in \mathit{fragments}$.

Claims about s:

1. $p \in \mathit{testset}(\mathit{fragment}(q))$, by precondition of φ.

2. *awake* = true, by Claim 1 and GC-C and COM-C.

Case 2.1: minlink$(f) \neq$ *nil.* Let ψ = *ChangeRoot*(f). By Claim 2 and assumption for Case 2.1, ψ is enabled in s.

Case 2.2: minlink$(f) =$ *nil.*

Case 2.2.1: testset$(f) = \emptyset$.

3. Either there is no external link of f, or *accmin*$(f) \neq$ *nil*, by GC-B and assumption for Case 2.2.1.
4. *fragment*$(p) \neq f$, by Claim 1 and assumption for Case 2.2.1.
5. *accmin*$(f) \neq$ *nil*, by Claims 3 and 4.

Let ψ = *ComputeMin*(f). It is enabled in s by Claim 5 and assumption for Case 2.2.1.

Case 2.2.2: testset$(f) \neq \emptyset$. Let q be some element of *testset*(f).

Case 2.2.2.1: testlink$(q) =$ *nil.* Let ψ = *SendTest*(q). It is enabled in s by assumptions for Case 2.2.2.1.

Case 2.2.2.2: testlink$(q) \neq$ *nil.* By TAR-C(a), *testlink*$(q) = \langle q, r\rangle$, for some r. There is a protocol message for $\langle q, r\rangle$, by TAR-C(c). So there is some message at the head of at least one of the six queues comprising *tarqueue*$(\langle q, r\rangle)$ and *tarqueue*$(\langle r, q\rangle)$. At least one of the following is enabled in s: *ReceiveTest*(k, l', c'), *ReceiveAccept*(k), *ReceiveReject*(k), *ChannelSend*(k, m), and *ChannelRecv*(k, m), where k is either $\langle q, r\rangle$ or $\langle r, q\rangle$, and $m \in M$.

Suppose in contradiction that there is no ψ enabled in s such that $(s, \psi) \in \Psi_\varphi$. That is, by definition of Ψ_φ, the only message in *tarqueue*$(\langle q, r\rangle)$ (if any) is a TEST(l', c') in *tarqueue*$_r(\langle q, r\rangle)$ with $l' >$ *level*(*fragment*(r)); and the only message in *tarqueue*$(\langle r, q\rangle)$ (if any) is a TEST(l'', c'') in *tarqueue*$_q(\langle r, q\rangle)$ with $l'' >$ *fragment*(q)).

Suppose the protocol message for $\langle q, r\rangle$ is a TEST(l', c') in *tarqueue*$(\langle q, r\rangle)$, with *lstatus*$(\langle q, r\rangle) \neq$ rejected. By TAR-E(b), l' = *level*(*fragment*(q)). Since *fragment*$(q) = f$, $l' = l$ by choice of f. But $l' >$ *level*(*fragment*(r)), by definition of Ψ_φ, which contradicts the definition of l.

Suppose the protocol message for $\langle q, r\rangle$ is a TEST(l'', c'') in *tarqueue*$(\langle r, q\rangle)$, with *lstatus*$(\langle r, q\rangle)$ = rejected. By TAR-E(c), l'' = *level*(*fragment*(q)). But by definition of Ψ_φ, $l'' >$ *level*(*fragment*(q)).

(2) Let (s', π, s) be a step of TAR, where s' is reachable and is in E_φ, $(s', \pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) If $(s', \pi) \notin \Psi_\varphi$, then π is either *InTree(l)*, *NotInTree(l)*, or *Start(p)*, or else π is $ReceiveTest(\langle q, r\rangle, l, c)$ and in s, $l > level(fragment(r))$ and there is only one message in $tarqueue_r(\langle q, r\rangle)$. In all cases, no component of v_φ is changed, so $v_\varphi(s) = v_\varphi(s')$.

Part (b) below finishes the proof that $v_\varphi(s) \leq v_\varphi(s')$.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. We show $v_\varphi(s) < v_\varphi(s')$.

- Suppose $\pi = ChannelSend(l, m)$. Component 8 of v_φ decreases and components 1 through 7 do not change.
- Suppose $\pi = ChannelRecv(l, m)$. Component 9 of v_φ decreases and components 1 through 8 do not change.
- Suppose $\pi = SendTest(q)$. Let $\langle q, r\rangle$ be the minimum-weight link of q with *lstatus* unknown in s'. By precondition, $testlink(q) = nil$ in s'. By TAR-D, there is no protocol message for $\langle q, r\rangle$ in s', so there is no TEST message in $tarqueue(\langle q, r\rangle)$ in s'. One is added in s. Thus component 7 of v_φ decreases and components 1 through 6 do not change. If there is no link of q with *lstatus* unknown, then q is removed from $testset(fragment(q))$. Thus component 4 of v_φ decreases and components 1 through 3 do not change.
- Suppose $\pi = ReceiveTest(\langle q, r\rangle, l, c)$ and in s' either $l \leq level(fragment(r))$ or there is more than one message in $tarqueue_r(\langle q, r\rangle)$.

Case 1: $l \leq level(fragment(r))$ and either $c \neq core(fragment(r))$ or $testlink(r) \neq \langle r, q\rangle$ in s'.

Claims about s':

1. TEST(l, c) message is in $tarqueue(\langle q, r\rangle)$, by precondition.
2. $c \neq core(fragment(r))$ or $testlink(r) \neq \langle r, q\rangle$, by assumption.
3. If $c \neq core(fragment(r))$, then $lstatus(\langle q, r\rangle) \neq$ rejected, by TAR-E(c).
4. If $testlink(r) \neq \langle r, q\rangle$, then there is no protocol message for $\langle r, q\rangle$, by TAR-D.
5. If $testlink(r) \neq \langle r, q\rangle$, then $lstatus(\langle q, r\rangle) \neq$ rejected, by Claim 4 and definition.

6. The TEST(l, c) message in *tarqueue*$(\langle q, r\rangle)$ is a protocol message for $\langle q, r\rangle$, by Claims 2, 3 and 5.
7. *testlink*$(q) = \langle q, r\rangle$, by Claim 6 and TAR-D.
8. There is no ACCEPT or REJECT message in *tarqueue*$(\langle r, q\rangle)$, by Claims 6 and 7 and TAR-C(c).

If *lstatus*$(\langle q, r\rangle)$ is changed from unknown to rejected, then component 5 of v_φ decreases and components 1 through 4 are unchanged. Otherwise, an ACCEPT or REJECT message is added to *tarqueue*$(\langle r, q\rangle)$ in s, causing component 6 of v_φ to decrease by Claim 8, while components 1 through 5 stay the same.

Case 2: $l \leq$ *level*(*fragment*(r)) and $c =$ *core*(*fragment*(r)) and *testlink*$(r) = \langle r, q\rangle$ in s'.

Claims about s':

1. TEST(l, c) is in *tarqueue*$(\langle q, r\rangle)$, by precondition.
2. $c =$ *core*(*fragment*(r)), by assumption.
3. *testlink*$(r) = \langle r, q\rangle$, by assumption.

Case 2.1: There is no link $\langle r, t\rangle$, $t \neq q$, with *lstatus* unknown in s'. Then q is removed from *testset*(*fragment*(q)) in s, causing component 4 of v_φ to decrease while components 1 through 3 do not change.

Case 2.2: There is a link $\langle r, t\rangle$, $t \neq q$, with *lstatus*$(\langle r, t\rangle) =$ unknown in s'.

4. *lstatus*$(\langle r, q\rangle) \neq$ rejected, by Claim 3 and TAR-K.

By Claim 4, Cases 2.2.1 and 2.2.2 are exhaustive.

Case 2.2.1: *lstatus*$(\langle r, q\rangle) =$ unknown in s'. It is changed to rejected in s, causing component 5 of v_φ to decrease and components 1 through 4 to stay the same.

Case 2.2.2: *lstatus*$(\langle r, q\rangle) =$ branch.

Case 2.2.2.1: The TEST(l, c) message in *tarqueue*$(\langle q, r\rangle)$ is a protocol message for $\langle r, q\rangle$.

5. The TEST(l, c) message in *tarqueue*$(\langle q, r\rangle)$ is the only protocol message for $\langle r, q\rangle$, by TAR-C(c).

Since the only protocol message for $\langle r,q\rangle$ is removed in s, component 5 of v_φ decreases and components 1 through 4 stay the same.

Case 2.2.2.2: The $\textsc{test}(l,c)$ message in *tarqueue*$(\langle q,r\rangle)$ is not a protocol message for $\langle r,q\rangle$.

6. *lstatus*$(\langle q,r\rangle) \neq$ rejected, by assumptions for Case 2.2.2.2.
7. There is a $\textsc{test}(l',c')$ message in *tarqueue*$(\langle r,q\rangle)$ and *lstatus*$(\langle r,q\rangle) =$ unknown, by Claims 1, 2, 3, 6 and TAR-P.

But Claim 7 contradicts the assumption for Case 2.2.2.

Case 3: $l >$ *level*(*fragment*(r)) and there is more than one message in *tarqueue*$_r(\langle q,r\rangle)$ in s'. All $\textsc{test}$ messages in *tarqueue*$_r(\langle q,r\rangle)$ are protocol messages for the same link, either $\langle q,r\rangle$ or $\langle r,q\rangle$. Since by TAR-D and TAR-C(c) there is never more than one protocol message for any link, this $\textsc{test}(l,c)$ message is the only one. The $\textsc{test}(l,c)$ message is put at the end of *tarqueue*$_r(\langle q,r\rangle)$ in s, decreasing component 10 and not changing components 1 through 9.

- Suppose $\pi =$ *ReceiveAccept*$(\langle q,r\rangle)$. Since r is removed from *testset*(*fragment*(r)), component 4 of v_φ decreases while components 1 through 3 stay the same.

- Suppose $\pi =$ *ReceiveReject*$(\langle q,r\rangle)$. If there are no more unknown links, then r is removed from *testset*(*fragment*(r)), decreasing component 4 of v_φ and not changing components 1 through 3. Suppose there is another unknown link.

Claims about s':

1. $\textsc{reject}$ is in *tarqueue*$(\langle q,r\rangle)$, by precondition.
2. There is a link $\langle r,t\rangle$, $t \neq q$, with *lstatus*$(\langle r,t\rangle) =$ unknown, by assumption.
3. *testlink*$(r) = \langle r,q\rangle$, by Claim 1 and TAR-D.
4. The $\textsc{reject}$ in *tarqueue*$(\langle q,r\rangle)$ is the only protocol message for $\langle q,r\rangle$, by Claim 3 and TAR-C(c).
5. *lstatus*$(\langle r,q\rangle) \neq$ rejected, by Claim 3 and TAR-K.

By Claim 5, *lstatus*$(\langle r,q\rangle) \neq$ rejected. If *lstatus*$(\langle r,q\rangle) =$ unknown in s', it is changed to rejected in s. If *lstatus*$(\langle r,q\rangle) =$ branch in s', then it stays branch in s, but there are no more protocol messages for $\langle r,q\rangle$ in s, by Claim 4. Thus component 5 of v_φ decreases while components 1 through 4 stay the same.

- Suppose $\pi = ComputeMin(f)$. Component 3 of v_φ decreases and components 1 and 2 are unchanged.
- Suppose $\pi = ChangeRoot(f)$. Component 2 of v_φ decreases and component 1 is unchanged.
- Suppose $\pi = Merge(f,g)$ or $Absorb(f,g)$. Component 1 of v_φ decreases.

(c) Suppose $(s',\pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s',\psi) \in \Psi_\varphi$. Then ψ is still enabled in s and $(s,\psi) \in \Psi_\varphi$, since the only possibilities are: $\pi = InTree(l)$, $NotInTree(l)$, or $Start(p)$, or else $\pi = ReceiveTest(\langle q,r\rangle,l,c)$ and in s', $l > level(fragment(r))$ and there is only one message in $tarqueue_r(\langle q,r\rangle)$.

iii) φ is InTree($\langle$p,q$\rangle$). We show TAR is progressive for φ via $\mathcal{M}_3$; Lemma 6 implies that TAR is equitable for φ via $\mathcal{M}_3$. We simply show that if $\langle p,q\rangle = minlink(f)$, but $lstatus(\langle p,q\rangle)$ is not yet branch, then eventually $ChangeRoot(f)$ will occur.

Let Ψ_φ be all pairs (s,ψ) of reachable states s and actions ψ enabled in s such that one of the following is true: (Let $f = fragment(p)$ in s.)

- $\psi = InTree(\langle p,q\rangle)$, or
- $\langle p,q\rangle = minlink(f)$ in s, and $\psi = ChangeRoot(f)$.

For reachable state s, let $v_\varphi(s)$ be 1 if $\langle p,q\rangle = minlink(f)$ and $ChangeRoot(f)$ is enabled in s, and 0 otherwise.

(1) Let s be a reachable state of TAR in E_φ. We show that there exists an action ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$. Let $f = fragment(p)$ in s.

Claims about s:

1. $awake$ = true, by precondition of φ.
2. $(p,q) \in subtree(f)$ or $\langle p,q\rangle = minlink(f)$, by precondition of φ.
3. $answered(\langle p,q\rangle)$ = false, by precondition of φ.
4. $lstatus(\langle p,q\rangle) \neq$ rejected, by Claim 2 and TAR-B.

By Claim 4, the following two cases are exhaustive.

Case 1: lstatus$(\langle p,q\rangle)$ = branch. Let $\psi = \mathit{InTree}(\langle p,q\rangle)$. It is enabled in s by Claims 1 and 3 and assumption for this case, and $(s,\psi) \in \Psi_\varphi$.

Case 2: lstatus$(\langle p,q\rangle)$ = unknown.

5. *minlink*$(f) = \langle p,q\rangle$, by Claim 2 and TAR-A(a).
6. *rootchanged*(f) = false, by Claim 5 and TAR-H.

Let $\psi = \mathit{ChangeRoot}(f)$. It is enabled in s by Claims 1, 5 and 6, and $(s,\psi) \in \Psi_\varphi$.

---

(2) Let (s',π,s) be a step of *TAR*, where s' is reachable and is in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) Suppose $(s',\pi) \notin \Psi_\varphi$. We show that no possibility for π can affect whether or not *ChangeRoot*(f) is enabled, i.e., $v_\varphi(s) = v_\varphi(s')$. This together with (b) below shows that $v_\varphi(s) \leq v_\varphi(s')$.

Case 1: ChangeRoot(f) is enabled in s'. No action sets *awake* to false. No action (other than *ChangeRoot*(f)) sets *rootchanged*(f) to false. No action sets *minlink*(f) to *nil*. f remains in *fragments* because π is not *Absorb*(g,f), *Merge*(f,g) or *Merge*(g,f), for any g, since *rootchanged*(f) = false.

Case 2: rootchanged(f) is not enabled in s'. By precondition of φ, *awake* is true in s'. If *rootchanged*(f) = true in s', then the same is true in s, because the only action that sets it to false is the *Merge* that created f. If *minlink*$(f) = \mathit{nil}$ in s', then $\langle p,q\rangle \neq \mathit{minlink}(f)$, so even if *minlink*$(f)$ becomes nonnil (by *ComputeMin*(f)), v_φ remains 0.

(b) Suppose $(s',\pi) \in \Psi_\varphi$. Since $(s',\pi) \notin X_\varphi$, $\pi \neq \mathit{InTree}(\langle p,q\rangle)$. Thus *minlink*$(f) = \langle p,q\rangle$ in s' and $\pi = \mathit{ChangeRoot}(f)$. Obviously v_φ goes from 1 to 0.

(c) Suppose $(s',\pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s',\psi) \in \Psi_\varphi$. The same argument as in (2a), Case 1, applies.

---

iv) φ is NotInTree($\langle$p,q$\rangle$). We show that *TAR* is progressive for φ via $\mathcal{M}_3$; Lemma 6 implies that *TAR* is equitable for φ via $\mathcal{M}_3$. The goal is to show that if $q \in nodes(fragment(p))$ and $(p,q) \notin subtree(fragment(p))$, then eventually $lstatus(\langle p,q\rangle) =$ rejected. This requires a global argument, as for *TestNode*(p), because it could be that some unknown link will never be tested until only one fragment remains.

Let Ψ_φ be $\Psi_{TestNode(p)} \cup \{(s, NotInTree(\langle p,q\rangle)) : s$ reachable, $NotInTree(\langle p,q\rangle)$ enabled in $s\}$.

Let $v_\varphi(s) = v_{TestNode(p)}(s)$ for all reachable states s.

Let v_φ be the same as for *TestNode*(p).

(1) Let s be a reachable state of *TAR* in E_φ. We show that there exists an action ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

$lstatus(\langle p,q\rangle) \neq$ branch, by TAR-A(a). If $lstatus(\langle p,q\rangle) =$ rejected, then let $\psi = NotInTree(\langle p,q\rangle)$.

Suppose $lstatus(\langle p,q\rangle) =$ unknown in s. The rest of the argument is just like that for *TestNode*(p), except for the following cases.

Case 2.1: *ChangeRoot*(f) is enabled in s because $awake =$ true by the precondition of φ.

Case 2.2.1: We show that *ComputeMin*(f) is enabled in s by showing that there are at least two fragments, as follows. If there is only one fragment, then $f = fragment(p)$, and $p \notin testset(f)$ (since we assume $testset(f) = \emptyset$). But since we also assume $lstatus(\langle p,q\rangle) =$ unknown, TAR-I gives as contradiction. Thus, there is an external link of f, and by GC-B, $accmin(f') \neq nil$.

(2) Like *TestNode*(p), after noting that π cannot be $NotInTree(\langle p,q\rangle)$. □

4.3.4 DC is Progressive for an Action of GC

The main idea is to show that REPORT messages converge on the core. This argument is local to one fragment.

Lemma 30: *DC is progressive for ComputeMin(f) via* $\mathcal{M}_4$.

Proof: By Corollary 20, $(P'_{GC} \circ S_4) \wedge P_{DC}$ is true in every reachable state of DC. Thus, in the sequel we will use the HI, COM, GC and DC predicates.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of DC and actions ψ of DC such that in s, a REPORT(w) is in some *dcqueue*$(\langle q,p\rangle)$ and either q is a child of p, or else *dcstatus*(p) = unfind and p = *mw-root*(f); and $\psi \in \{$*ChannelSend*$(\langle q,p\rangle$, REPORT$(w))$, *ChannelRecv*$(\langle q,p\rangle$, REPORT$(w))$, *ReceiveReport*$(\langle q,p\rangle, w)\}$.

For reachable state s, let $v_\varphi(s)$ be a quadruple with the following components:

1. The number of nodes $p \in$ *nodes*(f) with *dcstatus*(p) = find.
2. The number of REPORT messages in *dcqueue*$_q(\langle q,p\rangle)$, for all $(p,q) \in$ *subtree*(f) such that either q is a child of p or else p = *mw-root*(f) and *dcstatus*(p) = unfind.
3. The number of REPORT messages in *dcqueue*$_{qp}(\langle q,p\rangle)$ for all $(p,q) \in$ *subtree*(f) such that either q is a child of p or else p = *mw-root*(f) and *dcstatus*(p) = unfind.
4. The number of REPORT messages in *dcqueue*$_p(\langle q,p\rangle)$ for all $(p,q) \in$ *subtree*(f) such that either q is a child of p or else p = *mw-root*(f) and *dcstatus*(p) = unfind.

(1) Let s be a reachable state of DC in E_φ. We show that there exists an action ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

Claims about s:

1. *minlink*(f) = *nil*, by precondition.
2. *accmin*$(f) \neq$ *nil*, by precondition.
3. *testset*$(f) = \emptyset$, by precondition.
4. There is an external link of f, by Claim 2 and GC-A.
5. No FIND message is in *subtree*(f), by Claim 3 and DC-D(c).
6. If *dcstatus*(p) = find, then a REPORT message is in *subtree*(p) headed toward p, for any $p \in$ *nodes*(f), by Claim 3 and DC-I(b).

Suppose a REPORT(w) is in some *dcqueue*$(\langle q,p\rangle)$ and q is a child of p. By DC-B(a), *inbranch*$(p) \neq \langle p,q\rangle$. Obviously, $(p,q) \neq$ *core*(f), so by DC-A(g), *dcstatus*(p) = find. By Claim 5 and DC-O, the REPORT(w) is the only message in *dcqueue*$(\langle q,p\rangle)$. If it is in *dcqueue*$_q(\langle q,p\rangle)$, let ψ = *ChannelSend*$(\langle q,p\rangle$, REPORT$(w))$; if it is in *dcqueue*$_{qp}(\langle q,p\rangle)$, let ψ = *ChannelRecv*$(\langle q,p\rangle$, REPORT$(w))$; if it is in *dcqueue*$_p(\langle q,p\rangle)$, let ψ = *ReceiveReport*(w). Obviously, ψ is enabled in s, and $(s,\psi) \in \Psi_\varphi$.

Suppose no REPORT is in any *dcqueue*$(\langle q,p\rangle)$ with q a child of p. By Claim 6, *dcstatus*(p) = unfind for all $p \in$ *nodes*(f). Then by Claims 1, 4 and 5, a REPORT(w) is in $dcqueue(\langle q,p\rangle)$, where $(p,q) = core(f)$ and $p = mw\text{-}root(f)$. By Claim 5 and DC-O, the REPORT(w) is the only message in $dcqueue(\langle q,p\rangle)$. If it is in $dcqueue_q(\langle q,p\rangle)$, let $\psi = ChannelSend(\langle q,p\rangle, \text{REPORT}(w))$; if it is in $dcqueue_{qp}(\langle q,p\rangle)$, let $\psi = ChannelRecv(\langle q,p\rangle, \text{REPORT}(w))$; if it is in $dcqueue_p(\langle q,p\rangle)$, let $\psi = ReceiveReport(w)$. Obviously, ψ is enabled in s, and $(s,\psi) \in \Psi_\varphi$.

(2) Let (s', π, s) be a step of DC, where s' is reachable and is in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$. We note the following claims about s'.

1. $testset(f) = \emptyset$, by precondition.
2. $minlink(f) = nil$, by precondition.
3. No FIND is in $subtree(f)$, by Claim 1 and DC-D(c).

(a) To show $v_\varphi(s) \leq v_\varphi(s')$, we show that $v_\varphi(s) = v_\varphi(s')$ if $(s',\pi) \notin \Psi_\varphi$; this together with part (b) below gives the result. Suppose $(s',\pi) \notin \Psi_\varphi$.

TestNode(p) is not enabled, for $p \in nodes(f)$, by Claim 1. *ChangeRoot*(f), *Merge*(f,g), *Merge*(g,f), and *Absorb*(g,f) are not enabled, for $g \in fragments$, by Claim 2. *ReceiveFind*($\langle p,q\rangle$), *AfterMerge*(p,q), *ChannelSend*($\langle p,q\rangle$, FIND), and *ChannelRecv*($\langle p,q\rangle$, FIND) are not enabled, for $p \in nodes(f)$, by Claim 3. Thus π is none of the above actions.

If $\pi = ChannelSend(\langle q,p\rangle, \text{REPORT}(w))$ or $ChannelRecv(\langle q,p\rangle, \text{REPORT}(w))$, for $(q,p) \in subtree(f)$, then v_φ is unchanged, since $(s',\pi) \notin \Psi_\varphi$.

Suppose $\pi = ReceiveReport(\langle q,p\rangle, w)$.

Case 1: p is a child of q. By DC-A(a), $inbranch(p) = \langle p,q\rangle$. By DC-B(b), $dcstatus(p) =$ unfind. So the only change is the removal of the message. Since p is a child of q, $p \neq mw\text{-}root(f)$, so v_φ is unchanged.

Case 2: $(p,q) = core(f)$ and $p \neq mw\text{-}root(f)$. By DC-A(a), $inbranch(p) = \langle p,q\rangle$. The only effect is that either the message is requeued (if $dcstatus(p) =$ find), or the message is removed (if $dcstatus(p) =$ unfind); in both cases, v_φ is unchanged.

Case 3: $(p,q) = core(f)$, $p = mw\text{-}root(f)$, and $dcstatus(p) =$ find. The only effect is that the message is requeued, so v_φ is unchanged.

Suppose $\pi = Merge(g,h)$. By precondition, $minlink(g) = minlink(h) \neq nil$ in s'. So $f \neq g$ and $f \neq h$. Obviously v_φ is unchanged.

Suppose $\pi = Absorb(g,h)$. By precondition, $minlink(h) \neq nil$ in s', so $f \neq h$ by Claim 2. If $f \neq g$, then obviously v_φ is unchanged. Suppose $f = g$. As in the proof of condition (3a) in Lemma 19 for *viii)* $\pi = Absorb$, Case 2, no REPORT message is headed toward $minnode(h)$ and $dcstatus(r) =$ unfind for all $r \in nodes(h)$ in s'. Thus v_φ does not change.

The remaining actions (not mentioned above) obviously do not affect v_φ.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. We show $v_\varphi(s) < v_\varphi(s')$. If $\psi = ChannelSend(l,m)$, component 2 of v_φ decreases and component 1 is unchanged. If $\psi = Channel\text{-}Recv(l,m)$, component 3 of v_φ decreases and components 1 and 2 are unchanged.

Suppose $\psi = ReceiveReport(\langle q,p\rangle, w)$.

Case 1: q is a child of p. By DC-B(a), $inbranch(p) \neq \langle p,q\rangle$. By DC-A(g), $dcstatus(p) =$ find. If $findcount(p) = 1$ in s', then component 1 of v_φ decreases. Otherwise, component 4 decreases and components 1 through 3 are unchanged.

Case 2: q is not a child of p, $p = mw\text{-}root(f)$, and $dcstatus(p) =$ unfind. So $(p,q) = core(f)$. By DC-P, $w > bestwt(p)$. But this contradicts $(s', \pi) \notin X_\varphi$.

(c) Suppose $(s', \pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s', \psi) \in \Psi_\varphi$. We show that ψ is still enabled in s and $(s, \psi) \in \Psi_\varphi$. Since the queues are FIFO, there is no way to disable ψ.

It remains to show that (s, ψ) is still in Ψ_φ.

One possible way (s, ψ) could no longer be in Ψ_φ is if the position of $mw\text{-}root(f)$ changes, i.e., if π is $Merge(f,g)$, $Merge(g,f)$, $Absorb(f,g)$, or $Absorb(g,f)$, for some fragment g. But by Claim 2, $minlink(f) = nil$. Thus π cannot be $Merge(f,g)$, $Merge(g,f)$, or $Absorb(g,f)$. Suppose $\pi = Absorb(f,g)$. Let $core(f) = (p,q)$, $p = mw\text{-}root(f)$, and q be the endpoint of $core(f)$ closest to $target(minlink(g))$ in s'. The minimum-weight external link of f has smaller weight than $minlink(g)$, which by COM-A is the minimum-weight external link of g. Thus $mw\text{-}root(f)$ does not change after $Absorb(f,g)$.

Another way is if the position of $core(f)$ changes. This only happens if π is $Merge(f,g)$, $Merge(g,f)$ or $Absorb(g,f)$, which we showed is impossible.

The third way is if $dcstatus(p)$ changes from unfind to find, where $p = mw\text{-}root(f)$. This only happens if $\pi = ReceiveFind(\langle q,p\rangle)$ for some q. But by Claim 3, no FIND is in $subtree(f)$, and by DC-D(a), no FIND can be in an external link. □

4.3.5 CON is Progressive for Some Actions of COM

To show that CON is progressive for *Merge* and *Absorb*, we just show that the CONNECT message on the *minlink* makes it across. For *ChangeRoot*, we show that the chain of CHANGEROOT messages eventually reaches the *minnode*. These arguments are all local to one fragment.

Lemma 31: *CON is progressive for $Merge(f,g)$, $Absorb(f,g)$ and $ChangeRoot(f)$ via $\mathcal{M}_6$.*

Proof: By Corollary 24, $(P'_{COM} \circ \mathcal{S}_6) \wedge P_{CON}$ is true in every reachable state of CON. Thus, in the sequel we will use the HI, COM, and CON predicates.

i) φ is Merge(f,g). Let $(p,q) = minedge(f)$. Let Ψ_φ be the set of all pairs (s,ψ) of reachable states s of CON and actions ψ of CON enabled in s, such that $\psi \in \{ChannelSend(\langle q,p\rangle, \text{CONNECT}(l)),\ ChannelRecv(\langle q,p\rangle, \text{CONNECT}(l)),\ Merge(f,g)\}$.

For reachable state s of CON, let $v_\varphi(s) = (x,y)$, where x is the number of messages in $cqueue_q(\langle q,p\rangle)$ in s, and y is the number of messages in $cqueue_{qp}(\langle q,p\rangle)$ in s.

(1) Suppose s is a reachable state of CON in E_φ. We show that there is a ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

Claims about s:

1. $f \neq g$, by precondition.
2. $minedge(f) = minedge(g) = (p,q)$, by precondition.
3. $rootchanged(f) =$ true, by precondition.
4. $rootchanged(g) =$ true, by precondition.
5. A CONNECT(l) message is in $cqueue(k)$, for some external link k of f, by Claim 3.
6. A CONNECT(l) message is in $cqueue(\langle p,q\rangle)$, by Claims 2, 5 and CON-D.
7. A CONNECT(m) message is in $cqueue(k)$, for some external link k of g, by Claim 4.
8. A CONNECT(m) message is in $cqueue(\langle q,p\rangle)$, by Claims 2, 6 and CON-D.
9. $l = level(f)$, by Claim 5 and CON-D.
10. $m = level(g)$, by Claim 7 and CON-D.
11. $level(f) \leq level(g)$, by Claim 2 and COM-A.
12. $level(g) \leq level(f)$, by Claim 2 and COM-A.

13. *level*$(f) =$ *level*(g), by Claims 11 and 12.
14. $l = m$, by Claims 9, 10 and 13.
15. No CHANGEROOT message is in *cqueue*$(\langle q,p\rangle)$, by Claim 1 and CON-C.
16. Exactly one CONNECT message is in *cqueue*$(\langle q,p\rangle)$, by Claims 7, 8 and CON-D.

If CONNECT(l) is in $cqueue_q(\langle q,p\rangle)$, then let $\psi = ChannelSend(\langle q,p\rangle, \text{CONNECT}(l))$. If CONNECT$(l)$ is in $cqueue_{qp}(\langle q,p\rangle)$, then let $\psi = ChannelRecv(\langle q,p\rangle, \text{CONNECT}(l))$. If CONNECT$(l)$ is in $cqueue_p(\langle q,p\rangle)$, then let $\psi = Merge(f,g)$. It is easy to see in all cases that ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$.

(2) Suppose (s',π,s) is a step of *CON*, s' is reachable and in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) The only actions that can increase v_φ are *ComputeMin*(g), and *ChangeRoot*(g). (Even though *ChannelSend*$(\langle q,p\rangle, m)$ would increase y, it would simultaneously decrease x.) By Claim 2, *ComputeMin*(g) is not enabled in s'. By Claim 4, *ChangeRoot*(g) is not enabled in s'.

(b) Suppose $(s',\pi) \in \Psi_\varphi$. Since $(s',\pi) \notin X_\varphi$, $\pi \neq Merge(f,g)$. Obviously, the other two choices for ψ decrease v_φ.

(c) Suppose $(s',\pi) \notin \Psi_\varphi$, ψ is enabled in s' and $(s',\psi) \in \Psi_\varphi$. We show ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$. If $\psi =$ *ChannelSend* or *ChannelRecv*, then it can only be disabled by occurring. If $\psi = Merge(f,g)$, then since $s \in E_\varphi$, ψ is still enabled in s (by the argument in part (1)). In all cases, $(s,\psi) \in \Psi_\varphi$.

ii) φ is Absorb(f,g). Let $\langle q,p\rangle =$ *minlink*(g). Let Ψ_φ be the set of all pairs (s,ψ) of reachable states s of *CON* and actions ψ of *CON* enabled in s, such that $\psi \in \{ChannelSend(\langle q,p\rangle, \text{CONNECT}(l)),\ ChannelRecv(\langle q,p\rangle, \text{CONNECT}(l)),\ Absorb(f,g)\}$.

For reachable state s of *CON*, let $v_\varphi(s) = (x,y)$, where x is the number of messages in $cqueue_q(\langle q,p\rangle)$ in s, and y is the number of messages in $cqueue_{qp}(\langle q,p\rangle)$ in s.

(1) Suppose s is a reachable state of *CON* in E_φ. We show that there is a ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

Claims about s:

1. *level*$(g) <$ *level*(f), by precondition.

2. $\langle q,p\rangle = minlink(g)$, by assumption.
3. $f = fragment(p)$, by precondition.
4. $rootchanged(g) =$ true, by precondition.
5. A CONNECT(l) message is in $cqueue(k)$, where k is an external link of g, by Claim 4.
6. A CONNECT(l) message is in $cqueue(\langle q,p\rangle)$, by Claims 2, 5 and CON-D.
7. No CHANGEROOT message is in $cqueue(\langle q,p\rangle)$, by Claims 5 and 6 and CON-C.

If CONNECT(l) is in $cqueue_q(\langle q,p\rangle)$, then let $\psi = ChannelSend(\langle q,p\rangle, \text{CONNECT}(l))$. If CONNECT($l$) is in $cqueue_{qp}(\langle q,p\rangle)$, then let $\psi = ChannelRecv(\langle q,p\rangle, \text{CONNECT}(l))$. If CONNECT($l$) is in $cqueue_p(\langle q,p\rangle)$, then let $\psi = Absorb(f,g)$. In all cases, it is easy to see that ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$.

(2) Suppose (s',π,s) is a step of CON, s' is reachable and in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) The only actions that can increase v_φ are $ComputeMin(g)$, and $ChangeRoot(g)$. (Even though $ChannelSend(\langle q,p\rangle,m)$ would increase y, it would simultaneously decrease x.) By Claim 2, $ComputeMin(g)$ is not enabled in s'. By Claim 4, $ChangeRoot(g)$ is not enabled in s'.

(b) Suppose $(s',\pi) \in \Psi_\varphi$. Since $(s',\pi) \notin X_\varphi$, $\pi \neq Absorb(f,g)$. Obviously, the other two choices for ψ decrease v_φ.

(c) Suppose $(s',\pi) \notin \Psi_\varphi$, ψ is enabled in s' and $(s',\psi) \in \Psi_\varphi$. We show ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$. If $\psi = ChannelSend$ or $ChannelRecv$, then it can only be disabled by occurring. If $\psi = Absorb(f,g)$, then since $s \in E_\varphi$, ψ is still enabled in s (by the argument in part (1)). In all cases, $(s,\psi) \in \Psi_\varphi$ by definition.

iii) φ is ChangeRoot(f). Let Ψ_φ be the set of all pairs (s,ψ) of reachable states s of CON and actions ψ of CON enabled in s, such that $\psi \in \{ReceiveChangeRoot(\langle q,p\rangle), ChannelSend(\langle q,p\rangle, \text{CHANGEROOT}), ChannelRecv(\langle q,p\rangle, \text{CHANGEROOT}) : p \in nodes(f)\} \cup \{ChangeRoot(f)\}$.

For reachable state s of CON, let $v_\varphi(s)$ be a triple defined as follows. If there is no CHANGEROOT message in $subtree(f)$ in s, then $v_\varphi(s)$ is $(0,0,0)$. Suppose, in s, there is a CHANGEROOT message in $cqueue(\langle q,p\rangle)$, where $p \in nodes(f)$. Then $v_\varphi(s)$ is:

1. the number of nodes in the path in $subtree(f)$ from p to $minnode(f)$ in s (counting the endpoints p and $minnode(f)$);

2. the number of CHANGEROOT messages in $cqueue_r(\langle r,t\rangle)$, for all $t \in nodes(f)$ in s; and
3. the number of CHANGEROOT messages in $cqueue_{rt}(\langle r,t\rangle)$, for all $t \in nodes(f)$ in s.

(By CON-B and CON-C, there is only one CHANGEROOT message in $subtree(f)$. By COM-G, HI-A and HI-B, there is a unique path in $subtree(f)$ from p to $minnode(f)$. Thus, $v_\varphi(s)$ is well-defined.)

(1) We show that if s is a reachable state of CON in E_φ, then there is a ψ enabled in s such that $(s,\psi) \in \Psi_\varphi$.

Claims about s:

1. $rootchanged(f)$ = false, by precondition of φ.
2. $minlink(f) \neq nil$, by precondition of φ.

If $|nodes(f)| = 1$ (i.e., $subtree(f) = \{p\}$, for some p), then let ψ = *ChangeRoot*(f). Obviously, ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$. Now suppose $|nodes(f)| > 1$.

3. $minnode(f) \neq root(f)$, by Claims 1 and 2 and CON-B.
4. Exactly one CHANGEROOT message is in $cqueue(\langle q,p\rangle)$, for some $(p,q) \in subtree(f)$, by Claims 1 and 2 and CON-B.
5. $(q,p) \neq core(f)$, by Claim 4 and CON-C.
6. No CONNECT message is in $cqueue(\langle q,p\rangle)$, by Claim 5 and CON-E.

If the CHANGEROOT message is in $cqueue_q(\langle q,p\rangle)$, then let ψ = *ChannelSend*$(\langle q,p\rangle$, CHANGEROOT$)$. If the CHANGEROOT message is in $cqueue_{qp}(\langle q,p\rangle)$, then let ψ = *ChannelRecv*$(\langle q,p\rangle$, CHANGEROOT$)$. If the CHANGEROOT message is in $cqueue_p(\langle q,p\rangle)$, then let ψ = *ReceiveChangeRoot*$(\langle q,p\rangle)$. In all three cases, ψ is enabled in s because of Claims 4 and 6. By definition, $(s,\psi) \in \Psi_\varphi$.

(2) Suppose (s',π,s) is a step of CON such that s' is reachable and in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) We show that if $(s',\pi) \notin \Psi_\varphi$, then $v_\varphi(s) = v_\varphi(s')$. Together with (b) below, it implies that $v_\varphi(s) \leq v_\varphi(s')$.

Since $minlink(f) \neq nil$ in s', $\pi \neq ComputeMin(f)$. Since $rootchanged(f)$ = false in s', $\pi \neq Merge(f,g)$, $Merge(g,f)$, or $Absorb(g,f)$ for any g.

Suppose $\pi = \mathit{Absorb}(f, g)$. First we show that $\mathit{minnode}(f)$ is unchanged. By COM-A, $\mathit{level}(h) \geq \mathit{level}(f)$, where $h = \mathit{fragment}(\mathit{target}(\mathit{minlink}(f)))$; by precondition of $\mathit{Absorb}(f, g)$, $h \neq g$, and thus $wt(\mathit{minlink}(f)) < wt(\mathit{minlink}(g))$. Also by COM-A, $\mathit{minlink}(g)$ is the minimum-weight external link of g. Thus $\mathit{minlink}(f)$ does not change. Second, we show that no CHANGEROOT message is in $\mathit{subtree}(g)$. By precondition of $\mathit{Absorb}(f, g)$, $\mathit{rootchanged}(g)$ = true. Then by CON-C, no CHANGEROOT message is in $\mathit{subtree}(g)$.

No other value of π, such that $(s', \pi) \notin \Psi_\varphi$, affects v_φ.

(b) Suppose $(s', \pi) \in \Psi_\varphi$. We show $v_\varphi(s) < v_\varphi(s')$.

If $\pi = \mathit{ChannelSend}(\langle q, p\rangle, \text{CHANGEROOT})$, then the second component of v_φ decreases while the first remains the same. If $\pi = \mathit{ChannelRecv}(\langle q, p\rangle, \text{CHANGEROOT})$, then the third component of v_φ decreases while the first two remain the same.

Suppose $\pi = \mathit{ReceiveChangeRoot}(\langle q, p\rangle)$. By CON-C and CON-B there is exactly one CHANGEROOT message in $\mathit{subtree}(f)$. Since $(s, \pi) \notin X_\varphi$, $p \neq \mathit{minnode}(f)$. Thus, the first component of $v_\varphi(s')$ is at least 1. The first component of v_φ decreases by 1 in s, by definition of $\mathit{tominlink}(p)$. Thus $v_\varphi(s) < v_\varphi(s')$.

(c) Suppose $(s', \pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s', \psi) \in \Psi_\varphi$. We show ψ is enabled in s, and $(s, \psi) \in \Psi_\varphi$.

Suppose $\psi = \mathit{ChangeRoot}(f)$.

Claims about s':

1. $\mathit{rootchanged}(f)$ = false, by precondition of ψ.
2. $\mathit{minlink}(f) \neq \mathit{nil}$, by precondition of ψ.
3. $\mathit{subtree}(f) = \{p\}$, by precondition of ψ.
4. No CHANGEROOT message is in $\mathit{cqueue}(\langle q, p\rangle)$ for any q, by Claim 3 and CON-C.
5. $\mathit{ComputeMin}(f)$ is not enabled, by Claim 2.
6. $\mathit{Merge}(f, g)$, $\mathit{Merge}(g, f)$, and $\mathit{Absorb}(g, f)$ are not enabled for any g, by Claim 1.
7. $\mathit{ReceiveChangeRoot}(\langle q, p\rangle)$ is not enabled for any q, by Claim 4.

By Claims 5, 6 and 7, π is no action that can disable ψ; hence, ψ is enabled in s. By definition, $(s, \psi) \in \Psi_\varphi$.

Suppose ψ = *ReceiveChangeRoot*($\langle q,p\rangle$), *ChannelSend*($\langle q,p\rangle$, CHANGEROOT), or *ChannelRecv*($\langle q,p\rangle$, CHANGEROOT). The only action that can disable ψ is ψ itself. Thus, ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$.

4.3.6 GHS is Equitable for TAR

The interesting arguments are for showing GHS is equitable for *SendTest*(p), and for *ChangeRoot*(f) when *subtree*(f) is a singleton node. For *SendTest*(p), we show that an INITIATE-find message eventually reaches p. The big effort is for the *ChangeRoot*(f). We must show that eventually every node will be awakened, either by a *Start* action, or by the receipt of a CONNECT or TEST message. This requires a global argument about the entire graph. This is another place in which the state component of Ψ in the definition of progressive is needed, since it is possible for a message to be requeued, leaving the state unchanged.

Lemma 32: *GHS is equitable for TAR via* $\mathcal{M}_{TAR}$.

Proof: We show that GHS is equitable for each locally-controlled action φ of TAR via $\mathcal{M}_{TAR}$. First, a point of notation: let *Receive*($\langle q,p\rangle, m$) be a synonym for *ReceiveConnect*($\langle q,p\rangle, l$) if m = CONNECT(l), a synonym for *Receive-Initiate*($\langle q,p\rangle, l, c, st$) if m = INITIATE(l, c, st), etc.

By Corollary 26, P'_{GHS} is true in every reachable state of GHS. Thus, in the sequel we will use the HI, COM, GC, TAR, DC, NOT, CON and GHS predicates.

i) φ **is InTree(l) or NotInTree(l).** By Lemma 5, we are done.

ii) φ **is ChannelSend($\langle$q,p$\rangle$,m).** We show that GHS is progressive for φ via $\mathcal{M}_{TAR}$. Lemma 6 gives the result.

Let Ψ_φ be the set of all pairs (s,ψ) of reachable states s of GHS and actions ψ of GHS enabled in s such that m' is the message at the head of $queue_q(\langle q,p\rangle)$ in s, and ψ = *ChannelSend*($\langle q,p\rangle, m'$).

For reachable state s, let $v_\varphi(s)$ be the number of messages in $queue_q(\langle q,p\rangle)$ ahead of the message at the head of $tarqueue_q(\langle q,p\rangle)$.

Verifying the progressive conditions is straightforward.

iii) φ **is ChannelRecv($\langle$q,p$\rangle$,m).** We show that GHS is progressive for φ via $\mathcal{M}_{TAR}$. Lemma 6 gives the result.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of GHS and actions ψ of GHS enabled in s such that m' is the message at the head of $queue_{qp}(\langle q,p\rangle)$ in s, and $\psi = ChannelRecv(\langle q,p\rangle, m')$.

For reachable state s, let $v_\varphi(s)$ be the number of messages in $queue_{qp}(\langle q,p\rangle)$ ahead of the message at the head of $tarqueue_{qp}(\langle q,p\rangle)$.

Verifying the progressive conditions is straightforward.

iv) φ is ReceiveTest(⟨q,p⟩,l,c), ReceiveAccept(⟨q,p⟩), or Receive-Reject(⟨q,p⟩). We show that GHS is progressive for φ via $\mathcal{M}_{TAR}$. Lemma 6 gives the result.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of GHS and actions ψ of GHS enabled in s such that m' is the message at the head of $queue_p(\langle q,p\rangle)$ in s, and $\psi = Receive(\langle q,p\rangle, m)$.

For reachable state s, let $v_\varphi(s)$ be the number of messages in $queue_p(\langle q,p\rangle)$ ahead of the message at the head of $tarqueue_p(\langle q,p\rangle)$.

Verifying the progressive conditions is straightforward.

v) φ is SendTest(p). We show that GHS is progressive for φ via $\mathcal{M}_{TAR}$. Lemma 6 gives the result.

Let Ψ_φ be the set of all pairs (s, π) of reachable states s of GHS and actions ψ of GHS enabled in s such that one of the following is true: (Let $f = fragment(p)$.)

- CONNECT(l) is in $queue(\langle q,r\rangle)$, where $(q,r) = core(f)$ and $p \in subtree(q)$, m is any message in $queue(\langle q,r\rangle)$ that is not behind the CONNECT(l) in s, and $\psi \in \{ChannelSend(\langle q,r\rangle, m),\ ChannelRecv(\langle q,r\rangle, m),\ Receive(\langle q,r\rangle, m)\}$.

- An INITIATE(l, c,find) message in $queue(\langle t,u\rangle)$ is headed toward p and m is any message in $queue(\langle t,u\rangle)$ that is not behind the INITIATE(l, c,find) in s, and $\psi \in \{ChannelSend(\langle t,u\rangle, m),\ ChannelRecv(\langle t,u\rangle, m),\ Receive(\langle t,u\rangle, m)\}$.

For reachable state s, $v_\varphi(s)$ is a 7-tuple with the following components.

If no CONNECT is in $queue(\langle q,r\rangle)$, where $(q,r) = core(f)$ and $p \in subtree(q)$ in s, then components 1 through 3 are 0. Suppose otherwise. By CON-D and CON-E, there is only one CONNECT message in $queue(\langle q,r\rangle)$.

1. The number of messages in $queue_q(\langle q,r\rangle)$ that are not behind the CONNECT.
2. The number of messages in $queue_{qr}(\langle q,r\rangle)$ that are not behind the CONNECT.
3. The number of messages in $queue_r(\langle q,r\rangle)$ that are not behind the CONNECT.

If no INITIATE(l, c,find) is headed toward p, then components 4 through 6 are 0. By DC-S, there is at most one such message. Suppose such a message is in $queue(\langle t,u\rangle)$.

4. The number of nodes on the path in $subtree(f)$ from u to p, including the endpoints.
5. The number of messages in $queue_t(\langle t,u\rangle)$ that are not behind the INITIATE(l, c, find).
6. The number of messages in $queue_{tu}(\langle t,u\rangle)$ that are not behind the INITIATE(l, c, find).
7. The number of messages in $queue_u(\langle t,u\rangle)$ that are not behind the INITIATE(l, c, find).

(1) Let s be a reachable state of GHS in E_φ. Thus, $p \in testset(f)$ and $testlink(p) = nil$. By the definition of $testset(f)$, either a FIND message is headed toward p in some $queue(\langle q,r\rangle)$, or a CONNECT message is in $queue(\langle q,r\rangle)$, where $(q,r) = core(f)$ and $p \in subtree(q)$. In either case, let m be the message at the head of $queue(\langle t,u\rangle)$. Let ψ be $ChannelSend(\langle q,r\rangle, m)$ if m is in $queue_q(\langle q,r\rangle)$; let ψ be $ChannelRecv(\langle q,r\rangle, m)$ if m is in $queue_{qr}(\langle q,r\rangle)$; let ψ be $Receive(\langle q,r\rangle, m)$ if m is in $queue_r(\langle q,r\rangle)$. Obviously, ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$.

(2) Let (s', π, s) be a step of GHS, s' be reachable and in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) We show that if $(s',\pi) \notin \Psi_\varphi$, then $v_\varphi(s') = v_\varphi(s)$; together with (b) below, this is enough. We consider all the ways that v_φ could change.

Can a CONNECT be added to $queue(\langle q,r\rangle)$, with $(q,r) = core(f)$ by π? By COM-F, $(p,q) \in subtree(f)$, so by TAR-A(b), $lstatus(\langle q,r\rangle) =$ branch. Yet by inspecting the code, we see that CONNECT is only added to a queue if its *lstatus* is not branch, or if the source node is sleeping, in which case GHS-A(c) implies that the *lstatus* is not branch.

Since we've assumed $(s',\pi) \notin \Psi_\varphi$, no CONNECT can be removed from the relevant queue.

For a given fragment f, $core(f)$ never changes.

Can the identity of *fragment*(p) change? Since $p \in testset(f)$ by the precondition of φ, $minlink(f) = nil$ in s' by GC-C. Thus no $Absorb(g, f)$, $Merge(f, g)$ or $Merge(g, f)$ is enabled in s'.

The number of messages in the same queue as the relevant CONNECT message but not behind it cannot change, because the queues are FIFO (and $(s', \pi) \notin \Psi_\varphi$).

Can a relevant INITIATE message be added? The only way it can is if either a CONNECT message in $queue(\langle q, r \rangle)$ with $(q, r) = core(f)$ and $p \in subtree(q)$ is received, or if the same INITIATE message headed toward p is received. Since $(s', \pi) \notin \Psi_\varphi$, π is neither of these actions.

Can the path from u to p change, where an INITIATE$(l, c,$find$)$ is in $queue(\langle t, u \rangle)$ headed toward p? By definition of headed toward and HI-A and HI-B, there is a unique path from u to p in s'. Since HI-A and HI-B are also true in s and since the minimum spanning tree is unique (by Lemma 10), the same unique path from u to p exists in s.

The number of messages in the same queue as the relevant INITIATE message but not behind it cannot change, because the queues are FIFO (and $(s', \pi) \notin \Psi_\varphi$).

(b) It is easy to check that $v_\varphi(s) < v_\varphi(s')$ if $(s', \pi) \in \Psi_\varphi$.

(c) No action ψ such that $(s', \psi) \in \Psi_\varphi$ can become disabled in s without occurring, since the queues are FIFO.

vi) φ is ComputeMin(f). We show that the hypotheses of Lemma 7 are satisfied to get the result.

Let $A = GHS$, $B = TAR$, $C = DC$, $D = GC$, and $\rho = ComputeMin(f)$ in the hypotheses of Lemma 7.

(1) If e is an execution of GHS, then by Lemmas 1 and 25, $\mathcal{M}_{DC}(e)$ is an execution of DC.

(2) Let s be a reachable state of TAR. If φ is enabled in $\mathcal{S}_{TAR}(s)$, then as argued in Section 4.2.3 (TAR to GC), φ is enabled in $\mathcal{S}_3(\mathcal{S}_{TAR}(s))$. By the way the $\mathcal{S}$'s are defined, $\mathcal{S}_3(\mathcal{S}_{TAR}(s)) = \mathcal{S}_4(\mathcal{S}_{DC}(s))$, so $\rho = \varphi$ is enabled in $\mathcal{S}_4(\mathcal{S}_{DC}(s))$.

(3) Suppose (s', π, s) is a step of GHS and s' is reachable. If φ is not in $\mathcal{A}_{TAR}(s', \pi)$, then ρ is not in $\mathcal{M}_4(\mathcal{M}_{DC}(s'\pi s))$ by inspection.

(4) DC is progressive for ρ via $\mathcal{M}_4$, using Ψ_ρ and v_ρ, by Lemma 30.

(5) Let ψ be such that $(t, \psi) \in \Psi_\rho$ for some t. Possible values of ψ are *ChannelSend*(l, REPORT(w)), *ChannelRecv*(l, REPORT(w)), and *ReceiveReport*(l, w). Essentially the same arguments as in *ii*), *iii*) and *iv*) show that GHS is progressive for ψ.

vii) φ is ChangeRoot(f) and subtree(f) is not {p} for any p. We show that the hypotheses of Lemma 7 are satisfied to get the result.

Let $A = GHS$, $B = TAR$, $C = CON$, $D = COM$, and $\rho =$ *ChangeRoot*(f) in the hypotheses of Lemma 7.

(1) If e is an execution of GHS, then by Lemmas 1 and 25, $\mathcal{M}_{CON}(e)$ is an execution of DC.

(2) Let s be a reachable state of TAR. Suppose φ is enabled in $\mathcal{S}_{TAR}(s)$. As argued in Section 4.2.3 (TAR to GC), φ is enabled in $\mathcal{S}_3(\mathcal{S}_{TAR}(s))$. As argued in Section 4.2.2 (GC to COM), φ is enabled in $\mathcal{S}_2(\mathcal{S}_3(\mathcal{S}_{TAR}(s)))$. By the way the $\mathcal{S}$'s are defined, $\mathcal{S}_2(\mathcal{S}_3(\mathcal{S}_{TAR}(s))) = \mathcal{S}_6(\mathcal{S}_{CON}(s))$, so $\rho = \varphi$ is enabled in $\mathcal{S}_6(\mathcal{S}_{CON}(s))$.

(3) Suppose (s', π, s) is a step of GHS and s' is reachable. If φ is not in $\mathcal{A}_{TAR}(s', \pi)$, then ρ is not in $\mathcal{M}_6(\mathcal{M}_{CON}(s'\pi s))$ by inspection.

(4) CON is progressive for ρ via $\mathcal{M}_6$, using Ψ_ρ and v_ρ, by Lemma 31.

(5) Let ψ be such that $(t, \psi) \in \Psi_\rho$ for some t. Possible values of ψ are *ChannelSend*(l, CHANGEROOT), *ChannelRecv*(l, CHANGEROOT), and *Receive-ChangeRoot*(l). Essentially the same arguments as in *ii*), *iii*) and *iv*) show that GHS is progressive for ψ.

viii) φ is ChangeRoot(f), subtree(f) is {p} for some p. We show that GHS is progressive for φ via $\mathcal{M}_{TAR}$. Lemma 6 gives the result.

Let Ψ_φ be the set of all pairs (s, ψ) of reachable states s of GHS and internal actions ψ of GHS enabled in s such that none of the following is true:

- $\psi =$ *ReceiveConnect*($\langle q, r\rangle, l$) for some q, r and l, and in s, *nstatus*(r) $\neq$ sleeping, $l \geq$ *nlevel*(r), *lstatus*($\langle r, q\rangle$) $=$ unknown, and only one message is in *queue*$_r$($\langle q, r\rangle$).
- $\psi =$ *ReceiveTest*($\langle q, r\rangle, l, c$) for some q, r, l and c, and in s, *nstatus*(r) $\neq$ sleeping, $l >$ *nlevel*(r), and only one message is in *queue*$_r$($\langle q, r\rangle$).

- $\psi = \mathit{ReceiveReport}(\langle q,r\rangle, w)$ for some q, r and w, and in s, $\mathit{inbranch}(r) = \langle q,r\rangle$, $\mathit{nstatus}(r) =$ find, and only one message is in $\mathit{queue}_r(\langle q,r\rangle)$.

For reachable state s, let $v_\varphi(s)$ be the following tuple:

1. The number of fragments in s.
2. The number of fragments g with $\mathit{rootchanged}(g) =$ false in s.
3. The number of fragments g with $\mathit{minlink}(g) = \mathit{nil}$ in s.
4. The number of nodes $q \in V(G)$ such that $q \in \mathit{testset}(\mathit{fragment}(q))$.
5. The summation over all $q \in V(G)$ of $\mathit{level}(\mathit{fragment}(q)) - \mathit{nlevel}(q)$.
6. The summation over all $q \in V(G)$ of $\mathit{findcount}(q)$.
7. The number of links $\langle q,r\rangle$ such that either $\mathit{lstatus}(\langle q,r\rangle) =$ unknown, or else $\mathit{lstatus}(\langle q,r\rangle) =$ branch and there is a protocol message for $\langle q,r\rangle$.
8. The number of links $\langle q,r\rangle$ such that no ACCEPT or REJECT is in $\mathit{queue}(\langle q,r\rangle)$.
9. The summation over all fragments g such that a CHANGEROOT is in some $\mathit{queue}(\langle q,r\rangle)$ of $\mathit{subtree}(g)$ of the number of nodes in the path in $\mathit{subtree}(g)$ from r to $\mathit{minnode}(g)$.
10. The number of fragments g such that $\mathit{AfterMerge}(q,r)$ for DC is enabled for some $q \in \mathit{nodes}(g)$.
11. The number of messages in $\mathit{queue}_q(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$.
12. The number of messages in $\mathit{queue}_{qr}(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$.
13. The number of messages in $\mathit{queue}_r(\langle q,r\rangle)$, for all $\langle q,r\rangle \in L(G)$.
14. The number of messages in $\mathit{queue}_r(\langle q,r\rangle)$ that are behind a CONNECT or TEST, for all $\langle q,r\rangle \in L(G)$.

(1) Let s be a reachable state of GHS in E_φ. We now demonstrate that some action ψ is enabled in s with $(s,\psi) \in \Psi_\varphi$.

By preconditions of φ, $\mathit{awake} =$ true, $\mathit{minlink}(f) \neq \mathit{nil}$ and $\mathit{rootchanged}(f) =$ false in s. By GHS-K, $\mathit{nstatus}(p) =$ true in s. But since $\mathit{awake} =$ true, there is some node q such that $\mathit{nstatus}(q) \neq$ sleeping. Thus A, the set of all fragments g such that $\mathit{nstatus}(q) \neq$ sleeping for some $q \in \mathit{nodes}(g)$, is non-empty. Let l be the minimum level of all fragments in A, and let $A_l = \{g \in A : \mathit{level}(g) = l\}$.

The strategy is to use a case analysis as follows. For each case, we show that there is some $\mathit{queue}(\langle q,r\rangle)$ with some message m in it in s. Let ψ be chosen as follows. If some message m' is at the head of $\mathit{queue}_q(\langle q,r\rangle)$, let $\psi = \mathit{ChannelSend}(\langle q,r\rangle, m')$. If no message is in $\mathit{queue}_q(\langle q,r\rangle)$ and some message m' is at the head of $\mathit{queue}_{qr}(\langle q,r\rangle)$, let $\psi = \mathit{ChannelSend}(\langle q,r\rangle, m')$. If no message is in $\mathit{queue}_q(\langle q,r\rangle)$ or $\mathit{queue}_{qr}(\langle q,r\rangle)$, then at least one message, namely m, is in $queue_r(\langle q,r\rangle)$; let $\psi = Receive(\langle q,p\rangle, m')$, where m' is the message at the head of $queue_r(\langle q,r\rangle)$.

For each choice, ψ is obviously enabled in s. There are two methods to verify that $(s,\psi) \in \Psi_\varphi$. Method 1 is to show that m is not CONNECT, TEST or REPORT. Then, if $\psi = Receive(\langle q,r\rangle, m')$ and m' is CONNECT, TEST or REPORT, there is more than one message in $queue_r(\langle q,r\rangle)$. Method 2 is to show that some variable in s has a value such that even if $\psi = Receive(\langle q,r\rangle, m')$, where m' is CONNECT, TEST or REPORT, we have that $(s,\psi) \in \Psi_\varphi$.

Case 1: There is a fragment $g \in A_l$ with $testset(g) \neq \emptyset$. Let q be some element of $testset(g)$. By definition of $testset(g)$, Cases 1.1, 1.2 and 1.3 are exhaustive.

Case 1.1: A CONNECT(l) message is in $queue(r,t)$, where $(r,t) = core(g)$ and $q \in subtree(r)$ in s. We use Method 2. By COM-F, $(r,t) \in subtree(g)$, so by TAR-A(b), $lstatus(\langle t,r\rangle) =$ branch.

Case 1.2: An INITIATE(l, c,find) message is in some $queue(\langle r,t\rangle)$ headed toward q in s. By Method 1, we are done.

Case 1.3: $testlink(q) \neq nil$ in s. By TAR-C(a), $testlink(q) = \langle q,r\rangle$ for some r. By TAR-C(c), there is a protocol message for $\langle q,r\rangle$.

Case 1.3.1: The protocol message is an ACCEPT or REJECT in $queue(\langle r,q\rangle)$. By Method 1, we are done.

Case 1.3.2: The protocol message is TEST(l', c) in $queue(\langle q,r\rangle)$. Thus $lstatus$ $(\langle q,r\rangle) \neq$ rejected. By TAR-E(b), $l' = l$. If $nstatus(r) =$ sleeping or $l \leq nlevel(r)$, we are done, by Method 2. Suppose $nstatus(r) \neq$ sleeping and $l > nlevel(r)$. By definition of A_l, $l \leq level(fragment(r))$, and thus $nlevel(r) < level(fragment(r))$. By NOT-G, either a NOTIFY($level(fragment(r))$) message is in some $queue(\langle t,u\rangle)$ headed toward r, in which case we are done by Method 1, or $AfterMerge(t,u)$ is enabled for NOT, with $r \in subtree(u)$. In the latter case, by GHS-L, a CONNECT is at the head of $queue(\langle u,t\rangle)$; the same argument as in Case 1.1 gives the result.

Case 2: $testset(g) = \emptyset$ for all $g \in A_l$.

Case 2.1: There is a fragment g in A_l with $minlink(g) = nil$. Since $g \neq f$ and G is connected, there is an external link of g. Since $testset(g) = \emptyset$, by DC-D(c) no FIND message is in $subtree(g)$.

Suppose $dcstatus(q) =$ unfind for all $q \in nodes(g)$. By definition of $minlink(g)$, a REPORT message is in some $queue(\langle q,r\rangle)$ headed toward $mw\text{-}root(g)$. We are done by Method 2.

Suppose $dcstatus(q) =$ find for some $q \in nodes(g)$. By DC-I(b), since $testset(g) = \emptyset$, a REPORT message is in some $queue(\langle r,t\rangle)$ in $subtree(q)$ headed toward q. By DC-B(a), $inbranch(t) \neq \langle t,r\rangle$. We are done by Method 2.

Case 2.2: $minlink(g) \neq nil$ for all $g \in A_l$.

Case 2.2.1: There is a fragment g in A_l with $rootchanged(g) =$ false. By GHS-K, if $subtree(g) = \{q\}$ for some q, then $nstatus(q) =$ sleeping. By definition of A_l, $subtree(g) \neq \{q\}$ for any q. By CON-B, a CHANGEROOT message is in some $queue(\langle q,r\rangle)$ in $subtree(g)$. We are done by Method 1.

Case 2.2.2: $rootchanged(g) =$ true for all $g \in A_l$. By CON-D, a CONNECT message is in $queue(minlink(g))$ for all $g \in A_l$.

Case 2.2.2.1: There is a fragment g in A_l with $minlink(g) = \langle q,r\rangle$ and $level(fragment(r)) > l$.

If $nlevel(r) > l$, we are done by Method 2. Suppose $nlevel(r) \leq l$. Essentially the same argument as in Case 1.3(b) gives the result.

Case 2.2.2.2: For all fragments g in A_l, $level(fragment(target(minlink(g)))) \leq l$. By COM-A, $level(fragment(target(minlink(g)))) = l$ for all $g \in A_l$.

Case 2.2.2.2.1: There is a fragment g in A_l such that $minlink(g) = \langle q,r\rangle$, and $fragment(r) \notin A_l$. By definition of A_l, $nstatus(r) =$ sleeping, and we are done be Method 2.

Case 2.2.2.2.2: For all fragments g in A_l, $fragment(target(minlink(g))) \in A_l$. As argued in Lemma 27, Case 2.2.2 of verifying (1) for $\varphi = Combine$, there are two fragments g and h in A_l such that $minedge(g) = minedge(h) = (q,r)$. By TAR-H, $lstatus(\langle r,q\rangle) = lstatus(\langle q,r\rangle) =$ branch. By Method 2, we are done.

---

(2) Let (s',π,s) be a step of GHS, where s' is reachable and in E_φ, $(s',\pi) \notin X_\varphi$, and $s \in E_\varphi$.

(a) We show that if $(s', \pi) \notin \Psi_\varphi$, then $v_\varphi(s) = v_\varphi(s')$; together with part (b) below, this gives the result. Ψ_φ is defined to include all the state-action pairs that change the state. Thus, if $(s', \pi) \notin \Psi_\varphi$, then $s = s'$, and obviously $v_\varphi(s) = v_\varphi(s')$.

(b) Suppose $(s, \pi) \in \Psi_\varphi$. The breakdown of cases in this argument is essentially the same as in the proof of the safety step simulations in Lemma 25. The notation "Component 12" in a case means that component 12 of v_φ decreases in going from s' to s, and components 1 through 11 are unchanged.

- $\pi = \mathit{ChannelSend}(\langle q,r\rangle, m)$. Component 11.
- $\pi = \mathit{ChannelRecv}(\langle q,r\rangle, m)$. Component 12.
- $\pi = \mathit{ReceiveConnect}(\langle q,r\rangle, l)$.

Case 1: $\mathit{nstatus}(r)$ = sleeping in s'. If $\langle q,r\rangle$ is not the minimum-weight external link of r, then: component 2. Otherwise, component 1.

Case 2: $\mathit{nstatus}(r) \neq$ sleeping, $l = \mathit{nlevel}(r)$ and no CONNECT is in $\mathit{queue}(\langle r,q\rangle)$ in s'.

Suppose $\mathit{lstatus}(\langle r,q\rangle)$ = unknown. Since $(s', \pi) \in \Psi_\varphi$, another message is in $\mathit{queue}(\langle q,r\rangle)$. By CON-D, CON-E and GHS-C, the other message is not a CONNECT or TEST. Component 14.

Suppose $\mathit{lstatus}(\langle r,q\rangle) \neq$ unknown. Since DC simulates $\mathit{AfterMerge}(r,q)$, neither $\mathit{AfterMerge}(r,q)$ nor $\mathit{AfterMerge}(q,r)$ is enabled in s. Component 10.

Case 3: $\mathit{nstatus}(r) \neq$ sleeping, $l = \mathit{nlevel}(r)$, and CONNECT is in $\mathit{queue}(\langle r,q\rangle)$ in s'. Component 1.

Case 4: $\mathit{nstatus}(r) \neq$ sleeping and $l < \mathit{nlevel}(r)$ in s'. Component 1.

- $\pi = \mathit{ReceiveInitiate}(\langle q,r\rangle, l, c, st)$. By NOT-H(a), $l > \mathit{nlevel}(r)$. Component 5.
- $\pi = \mathit{ReceiveTest}(\langle q,r\rangle, l, c)$. Let $g = \mathit{fragment}(r)$.

Case 1: $\mathit{nstatus}(r)$ = sleeping in s'. Component 2.

Case 2: $\mathit{nstatus}(r) \neq$ sleeping in s'.

Case 2.1: $l \leq level(g)$, and either $c \neq core(g)$ or $testlink(r) \neq \langle r,q\rangle$ in s'. If an ACCEPT is added, then component 8. If a REJECT is added, then either component 7 or component 8.

Case 2.2: $l \leq level(g)$, $c = core(g)$, and $testlink(r) = \langle r,q\rangle$ in s'. If there is no link $\langle r,t\rangle$, $t \neq q$, with $lstatus(\langle r,t\rangle) =$ unknown, then component 4. If there is such a link, then component 7.

Case 2.3: $l > level(g)$ in s'. Since $(s,\pi) \in \Psi_\varphi$, there is another message in $queue_r(\langle q,r\rangle)$. By TAR-C(c) and GHS-C, the other message is not CONNECT or TEST. Component 14.

- $\pi = ReceiveAccept(langleq,r\rangle)$. Component 4.
- $\pi = ReceiveReject(\langle q,r\rangle)$. If there is no link $\langle r,t\rangle$, $t \neq q$, with $lstatus(\langle r,t\rangle) =$ unknown, then component 4. If there is such a link, then component 7.
- $\pi = ReceiveReport(\langle q,r\rangle, w)$.

Case 1: $(q,r) = core(g)$, $nstatus(r) \neq$ find and $w > bestwt(r)$ in s'. If $lstatus(bestlink(r)) =$ branch, then component 3. Otherwise, component 2.

Case 2a: $(q,r) \neq core(g)$ in s'. If $inbranch(r) = \langle r,q\rangle$, then component 13. Otherwise, component 6.

Case 2b: $(q,r) = core(g)$ and $nstatus(r) =$ find in s'. The only change is that the REPORT message is requeued. We show that there is no other message in $queue(\langle q,r\rangle)$, and thus $(s',\pi) \notin \Psi_\varphi$. First note that by COM-F, $(q,r) \in subtree(g)$. By GHS-B, no CONNECT is in the queue. By DC-O, no INITIATE($*,*$,found) is in the queue. By GHS-E, no INITIATE($*,*$,find) is in the queue. By TAR-E(a), no TEST or REJECT is in the queue. By DC-O, no other REPORT is in the queue. By TAR-F, no ACCEPT is in the queue. By CON-C, no CHANGEROOT is in the queue.

Case 2c: $(q,r) = core()$, $nstatus(r) =$ unfind, and $w \leq bestwt(p)$. Component 13.

- $\pi = ReceiveChangeRoot(\langle q,r\rangle)$. If $lstatus(bestlink(r)) \neq$ branch, then component 2. Otherwise, component 9.

(c) Suppose $(s',\pi) \notin \Psi_\varphi$, ψ is enabled in s', and $(s',\psi) \in \Psi_\varphi$. Since $(s',\pi) \notin \Psi_\varphi$, $s = s'$. Obviously, ψ is enabled in s and $(s,\psi) \in \Psi_\varphi$.

ix) φ is Merge(f,g). We use Lemma 7. The same argument as in *vii*), with $\rho = Merge(f,g)$ and (3) as below, gives the result.

(3) Let ψ be such that $(t,\psi) \in \Psi_\rho$ for some t. Possible values of ψ are $ChannelSend(k,$ CONNECT$(l))$, $ChannelRecv(k,$ CONNECT$(l))$, and $Merge(f,g)$. Essentially the same arguments as in *ii*), *iii*) and *iv*) show that GHS is progressive for ψ.

x) φ is Absorb(f,g). We use Lemma 7. The same argument as in *vii*), with $\rho = Absorb(f,g)$ and (3) as below, gives the result.

(3) Let ψ be such that $(t,\psi) \in \Psi_\rho$ for some t. Possible values of ψ are $ChannelSend(k,$ CONNECT$(l))$, $ChannelRecv(k,$ CONNECT$(l))$, and $Absorb(f,g)$. Essentially the same arguments as in *ii*), *iii*) and *iv*) show that GHS is progressive for ψ. □

4.4 Satisfaction

Theorem 33: *GHS solves $MST(G)$.*

Proof: By Theorem 12, HI solves $MST(G)$. By Lemmas 13 and 27 and Theorem 8, COM satisfies HI. By Lemmas 15 and 28 and Theorem 8, GC satisfies COM. By Lemmas 17 and 29 and Theorem 8, TAR satisfies GC. By Lemmas 25 and 32 and Theorem 9, GHS satisfies TAR. Thus, since "satisfies" and "solves" are defined using subsets of schedules, GHS solves $MST(G)$. □

6 Conclusion

This chapter summarizes the results of the thesis and suggests some avenues for further research.

1. Summary

Chapter 2 of this thesis studied the transaction commit problem in a realistic timing model. A randomized algorithm was presented, and proved correct. The algorithm is nonblocking, as long as less than half the processors fail, and it never produces inconsistent decisions. The expected time of the algorithm, as measured in asynchronous rounds, is constant. We proved a lower bound, showing that the fault-tolerance of our algorithm is optimal. We also showed that no algorithm can cause processors to terminate in a bounded expected number of their own steps, even if processors are synchronous.

In Chapter 3, we showed that a system with asynchronous processors, and asynchronous but reliable communication, can simulate a system with synchronous processors, and asynchronous but reliable communication, in the presence of various types of processor faults. The definition of simulation is such that the impossibility result of [FLP] now implies the impossibility result of [DDS].

Chapter 4 consisted of a description of a modular drinking philosophers algorithm (based on the algorithm of [CM]), and a modular proof of correctness. The two modules were an arbitrary dining philosophers algorithm and explicit code to manipulate it. By substituting a time-efficient dining philosophers module, a time-efficient drinking philosophers algorithm can be obtained.

In Chapter 5, the minimum spanning tree algorithm of [GHS] was proved correct, using a lattice-structured proof technique. The algorithm is an important one, and had not previously had a rigorous proof. The proof technique is a general one that enables modularity to be used in proving algorithms that do not break apart easily.

2. Future Work

2.1. Partial Synchrony

In the area of partial synchrony, much work remains to be done in defining other partially synchronous models and showing relationships between the models. Real computer systems should provide inspiration for these models; one example would be to suppose a probability distribution on the message delays.

2.1.1. Transaction Commit

One could consider strengthening the time lower bound for the transaction commit problem. We proved that for every constant B, there is some adversary and some set of inputs such that the expected number of processor steps is larger than B. It may be possible to prove that there is some adversary and some set of inputs such that in most or all of the executions (with that adversary and inputs), processors take more than B steps.

Given any asynchronous agreement protocol that takes constant expected time, we can create a constant expected time transaction commit protocol by resolving the differences between the input-output relations of transaction commit and agreement, as was done in the protocol of Chapter 2. Chor, Merritt and Shmoys [CMS] have an asynchronous, constant expected time agreement protocol that tolerates just under $n/6$ of the n processors crashing. It would be interesting to devise an asynchronous constant expected time agreement protocol that tolerates just under $n/2$ of the processors crashing, which would enable us to obtain a transaction commit protocol with the same fault tolerance as the one in Chapter 2.

More generally, our approach to modeling partial synchrony — letting the behavior of the system during a run affect the problem statement — could be applied to other problems.

2.1.2. Simulating Synchronous Processors

The definition of simulation between systems could be extended to cover randomized algorithms, and complexity measures. One could try to show that the asynchronous model simulates the partially synchronous model with respect to these extensions.

The original simulation, or new ones, could be applied to other problems, for instance, the lower bounds in Chapter 2, which were shown for synchronous processors. (Of course, using the simulation here is only worthwhile if there is a simpler proof of the result in the asynchronous model.)

2.2. Modular Decomposition

In the realm of modular decomposition, there are doubtless other forms of modularity to be studied. The two studied in this thesis materialized by analyzing specific algorithms; I would expect the analysis of more algorithms to suggest other types.

2.2.1. Drinking Philosophers

The drinking philosophers algorithm of Chapter 4 only satisfies the weakest of the concurrency definitions presented in that chapter. It would be interesting to see how much concurrency can be achieved using a dining philosophers subroutine in a modular way. Along the same lines, one could design an algorithm (not necessarily using a dining philosophers algorithm) to achieve the maximal concurrency.

The time measure applied to the drinking philosophers algorithm is a function of the size of the entire resource graph; one should be able to find a better time measure for the dynamic problem that more accurately reflects the size of the "involved" graph.

More generally, there are many distributed algorithms, purporting to use other algorithms as subroutines, that could profit from a careful, modular description (e.g., parallel algorithms for computational geometry).

2.2.2. Minimum Spanning Tree

The work in Chapter 5 could be extended to discover what parts of the lattice and existing proof for the [GHS] minimum spanning tree algorithm is applicable to proofs of related algorithms. Some candidate algorithms include those in [A2] [CT]

[G], which describe modifications of the algorithm of [GHS] designed to decrease the running time. In particular, the algorithm in [G] might fit well into the current lattice, because the main difference is how the levels of fragments are computed, a facet of the algorithm which is encapsulated into a single high-level description in the lattice.

We believe the lattice-structured proof technique could profitably be applied to other algorithms that do not easily break apart into modules that can be studied independently. In fact, work in [T] is applying this technique to prove the correctness of a distributed deadlock recovery algorithm.

Our three techniques for verifying the liveness condition were all motivated by particular situations in the lattice used to verify [GHS]. Presumably, other techniques will be useful in other situations.

One could also investigate to what extent mechanical help (i.e., theorem-proving programs) can help reduce the tedium of these sorts of proofs. Once the algorithms in the lattice, the mappings between the algorithms, and the predicates are been chosen (using one's intuition about the algorithm), it remains to check that the predicates are invariants and that the mappings have the desired properties, which could probably be shunted off to a theorem-prover.

References

[A1] B. Awerbuch, "Complexity of Network Synchronization," *JACM* vol. 32, no. 4, pp. 804–823, 1985.

[A2] B. Awerbuch, "Optimal Distributed Algorithms for Minimum Weight Spaning Tree, Counting, Leader Election and Related Problems," *Proc.* 19th *Ann. ACM Symp. on Theory of Computing,* pp. 230–240, 1987.

[AG] B. Awerbuch and R. Gallager, "Distributed BFS Algorithms," *Proc.* 19th *Ann. ACM Symp. on Theory of Computing,* pp. 230–240, 1987.

[Be] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols," in *Proc.* 2nd *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 27–30,1983.

[Br] G. Bracha, "An $O(\log n)$ Expected Rounds Randomized Byzantine Generals Algorithm," *Proc.* 17th *Ann. ACM Symp. on Theory of Computing,* pp. 316–326, 1985.

[CC] B. Chor and B. Coan, "A Simple and Efficient Randomized Byzantine Agreement Algorithm," *IEEE Trans. on Software Engineering,* vol. SE-11, no. 6, pp. 531–539, 1985.

[CL] B. Coan and J. Lundelius, "Transaction Commit in a Realistic Fault Model," *Proc.* 5th *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 40–51, 1986.

[CM] K. M. Chandy and J. Misra, "The Drinking Philosophers Problem," *ACM Trans. on Programming Languages and Systems,* vol. 6, no. 4, pp. 632–646, 1984.

[CMS] B. Chor, M. Merritt, and D. Shmoys, "Simple Constant-Time Consensus Protocols in Realistic Failure Models," *Proc.* 4th *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 152–162, 1985.

[CT] F. Chin and H. F. Ting, "An Almost Linear Time and $O(n \log n + e)$ Messages Distributed Algorithm for Minimum-Weight Spanning Trees," *Proc.* 26th *Ann. IEEE Symp. on Foundations of Computer Science,* pp. 257–266, 1985.

[D1] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Comm. ACM,* vol. 8, p. 569, 1965.

[D2] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica,* vol. 1, pp. 115–138, 1971.

[DDS] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *J. ACM,* vol. 34, to appear.

[DLS] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Proc.* 3rd *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 103–118, 1984.

[DS] C. Dwork and D. Skeen, "The Inherent Cost of Nonblocking Commitment," *Proc.* 2nd *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 1–11, 1983.

[F] N. Francez, ***Fairness***, Springer-Verlag, New York, 1986, Chapter 2.

[FLP] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM,* vol. 32, no. 2, pp. 374–382, 1985.

[FLS] A. Fekete, N. Lynch, L. Shrira, "A Modular Proof of Correctness for a Network Synchronizer," *Proc.* 2nd *International Workshop on Distributed Algorithms*, 1987.

[Ga] E. Gafni, "Improvements in the Time Complexity of Two Message-Optimal Election Algorithms," *Proc.* 4th *Ann. ACM Symp. on Principles of Distributed Computing,* pp. 175–185, 1985.

[GHS] R. Gallager, P. Humblet and P. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 1, pp. 66–77, 1983.

[Gr] J. Gray, "Notes on Data Base Operating Systems," Research Report RJ2188(300001)2/23/78, IBM Research Laboratory, San Jose, California, 1977.

[HM] J. Halpern and Y. Moses, "Knowledge and Common Knowledge in a Distributed Environment," *Proc. 3rd Ann. ACM Symp. on Principles of Distributed Computing*, pp. 50–61, 1984 (revised as of January 1986 as IBM-RJ-4421).

[HO] B. Hailpern and S. Owicki, "Verifying Network Protocols Using Temporal Logic," *Proc. Trends and Applications 1980: Computer Networks,* IEEE Computer Society, pp. 18–28, 1980.

[K] R. Kurshan, "Reducibility in Analysis of Coordination," *Proc. IIASA Workshop on Discrete Event Systems,* 1987.

[La1] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, pp. 558–565, 1978.

[La2] L. Lamport, "Specifying Concurrent Program Modules," *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 2, pp. 190–222, 1983.

[La3] L. Lamport, "What Good is Temporal Logic?" *Proc. 9th IFIP World Computer Conference*, Sept. 1983, pp. 657–668.

[Ly] N. Lynch, "Upper Bounds for Static Resource Allocation in a Distributed System," *JCSS,* vol. 23, no. 2, pp. 254–278, 1981.

[LF] N. A. Lynch and M. J. Fischer, "On Describing the Behavior and Implementation of Distributed Systems," *Theoretical Computer Science,* vol. 13, pp. 17–43, 1981.

[LM] N. Lynch and M. Merritt, "Introduction to the Theory of Nested Transactions," to appear in *Theoretical Computer Science.* (Also available as technical report MIT/LCS/TR-367, Laboratory for Computer Science, Massachusetts Institute of Technology, 1986.)

[LPS] D. Lehmann, A. Pnueli, and J. Stavi, "Impartiality, Justice and Fairness: The Ethics of Concurrent Termination," *Proc. 8th International Colloquium on Automata, Languages and Programming*, July 1981, pp. 264–277.

[LS] S. Lam and U. Shankar, "Protocol Verification via Projections," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, July 1984, pp. 325–342.

[LSP] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. on Programming Languages and Systems,* vol. 4, 1982.

[LT] N. A. Lynch and M. R. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing,* pp. 137–151, 1987. (Also available as technical report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.)

[MC] J. Misra and K. M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. on Software Engineering*, vol. SE-7, no. 4, July 1981.

[MCS] J. Misra, K. M. Chandy, and T. Smith, "Proving Safety and Liveness of Communicating Processes with Examples," *Proc. 1st Ann. ACM Symp. on Principles of Distributed Computing,* pp. 201–208, 1982.

[NDGO] V. Nguyen, A. Demers, D. Gries, and S. Owicki, "A Model and Temporal Proof System for Networks of Processes," *Distributed Computing*, vol. 1, pp. 7–25, 1986.

[NT] G. Neiger and S. Toueg, "Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems," *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing,* pp. 281–293, 1987. (Also available as technical report TR86-790, Department of Computer Science, Cornell University, 1986.)

[PF] G. L. Peterson and M. J. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. 9th Ann. ACM Symp. on Theory of Comp.,* pp. 91–97, 1977.

[PSL] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *J. ACM*, vol. 27, 1980.

[R] M. Rabin, "Randomized Byzantine Generals," *Proc. 24th Ann. IEEE Symp. on Foundations of Computer Science,* pp. 403–409, 1983.

[RL] M. Rabin and D. Lehmann, "On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem," *Proc. 8th ACM Symp. on Principles of Programming Languages,* pp. 133–138, 1981.

[Sk] D. Skeen, "Crash Recovery in a Distributed Database System," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1982. (Also available as technical report UCB/BRL M82/45.)

[St] E. Stark, "Foundations of a Theory of Specification for Distributed Systems," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, 1984. (Also available as technical report MIT/LCS/TR-342.)

[SR] F. Stomp and W. de Roever, "A Correctness Proof of a Distributed Minimum-Weight Spanning Tree Algorithm," manu- script, April 1987.

[T] G. Troxel, "A Hierarchical Proof of an Algorithm for Deadlock Recovery in a System Using Remote Procedure Calls," S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, in progress.

[W1] J. Welch, "Simulating Synchronous Processors," *Information and Computation,* vol. 74, no. 2, pp. 159–171, 1987.

[W2] J. Welch, "Synthesis of Efficient Mutual Exclusion Algorithms," manuscript, 1987.

Appendix

In this Appendix, we review the aspects of the model from [LT] that are relevant to this paper.

An *input-output automaton* A is defined by the following four components. (1) There is a (possibly infinite) set of *states* with a subset of *start states*. (2) There is a set of *actions*, associated with the state transitions. The actions are divided into three classes, *input*, *output*, and *internal*. Input actions are presumed to originate in the automaton's environment; consequently the automaton must be able to react to them no matter what state it is in. Output and internal actions (or, *locally-controlled* actions) are under the local control of the automaton; internal actions model events not observable by the environment. The input and output actions are the *external* actions of A, denoted $ext(A)$. (3) The transition relation is a set of (state, action, state) triples, such that for any state s' and input action π, there is a transition (s', π, s) for some state s. (4) There is an equivalence relation $part(A)$ partitioning the output and internal actions of A. The partition is meant to reflect separate pieces of the system being modeled by the automaton. Action π is *enabled* in state s' if there is a transition (s', π, s) for some state s.

An *execution* e of A is a finite or infinite sequence $s_0\pi_1s_1\ldots$ of alternating states and actions such that s_0 is a start state, (s_{i-1}, π_i, s_i) is a transition of A for all i, and if e is finite then e ends with a state. The *schedule* of an execution e is the subsequence of actions appearing in e.

We often want to specify a desired behavior using a set of schedules. Thus we define an *external schedule module* S to consist of input and output actions, and a set of schedules $scheds(S)$. Each schedule of S is a finite or infinite sequence of the actions of S. Internal actions are excluded in order to focus on the behavior visible to the outside world. External schedule module S' is a *sub-schedule module* of external schedule module S if S and S' have the same actions and $scheds(S') \subseteq scheds(S)$.

Automata can be composed to form another automaton, presumably modeling a system made of smaller components. Automata communicate by synchronizing on shared actions; the only allowed situations are for the output from one automaton to be the input to others, and for several automata to share an input. Thus, automata to be composed must have no output actions in common, and the internal actions of each must be disjoint from all the actions of the others. A state of the composite automaton is a tuple of states, one for each component. A start state of the composition has a start state in each component of the state. Any output action of a component becomes an output action of the composition, and similarly for an internal action. An input action of the composition is an action that is input for every component for which it is an action. In a transition of the composition on action π, each component of the state changes as it would in the component automaton if π occurred; if π is not an action of some component automaton, then the corresponding state component does not change. The partition of the composition is the union of the partitions of the component automata.

Given an automaton A and a subset Π of its actions, we define the automaton $Hide_{\Pi}(A)$ to be the automaton A' differing from A only in that each action in Π becomes an internal action. This operation is useful for hiding actions that model interprocess communication in a composite automaton, so that they are no longer visible to the environment of the composition.

An execution of a system is fair if each component is given a chance to make progress infinitely often. Of course, a process might not be able to take a step every time it is given a chance. Formally stated, execution e of automaton A is *fair* if for each class C of $part(A)$, the following two conditions hold. (1) If e is finite, then no action of C is enabled in the final state of e. (2) If e is infinite, then either actions from C appear infinitely often in e, or states in which no action of C is enabled appear infinitely often in e. Note that any finite execution of A is a prefix of some fair execution of A.

The following result from [LT] is very useful: If e is a fair execution of a composition of automata, and A is one of the components, then $e|A$ is a fair execution of

A. (If $e = s_0\pi_1 s_1 \ldots$, we define $e|A$ to be the sequence obtained from e by deleting $\pi_i s_i$ if π_i is not an action of A, and replacing the remaining s_i with A's component.)

The ***fair external behavior*** of automaton A, denoted $Fbeh(A)$, is the external schedule module with the input and output actions of A, and with set of schedules $\{\alpha|ext(A) : \alpha \text{ is the schedule of a fair execution of } A\}$.[1] A ***problem*** is (specified by) an external schedule module. Automaton A ***solves*** the problem P if $Fbeh(A)$ is a sub-schedule module of P, i.e., the behavior of A visible to the outside world is consistent with the behavior required in the problem specification. Automaton A ***satisfies*** automaton B if $Fbeh(A)$ is a sub-schedule module of $Fbeh(B)$.

[1] If α is a sequence from a set S and T is a subset of S, then $\alpha|T$ is defined to be the subsequence of α consisting of elements in T.