Symmetry in Distributed Systems

Ralph E. Johnson
Ph.D. Thesis

87-855
August 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

SYMMETRY IN DISTRIBUTED SYSTEMS

Ralph Edward Johnson, PhD.

Cornell University 1987

A distributed computing system can be considered to be symmetric because of its topology or because of its behavior. Unfortunately, these different definitions can categorize the same system differently. The choice of definition becomes important when one is trying to prove that certain problems, such as the Dining Philosophers problem, cannot be solved on symmetric distributed computing systems. Since the behavioral definitions are based on the possible patterns of computation, it is much easier to use them for these proofs. However, behavioral definitions frequently are undecidable; topological definitions admit straightforward decision procedures.

This thesis presents a new definition for symmetry, called similarity, that, while based on behavior of processes, is decidable given the initial configuration of the system. The decision procedure for similarity depends partly on the model of computation being used, but a way to discover these decision procedures is given and is used to find decision procedures for a wide range of models of distributed computation. Distributed versions of these decision procedures form the basis of solutions to the problem of selecting a process as the leader. The thesis also shows how to use similarity

to compare the relative power of different models of computation, including models with shared variables with and without locking, models using synchronous and asynchronous message-passing, models making different assumptions about fairness, and models based on probabilistic techniques.

Biographical Sketch

Ralph Johnson was born October 7, 1955 in the Panama Canal Zone. He attended a number of primary and secondary schools, most notably The American School of Kinshasa, finally graduating from Galesburg High School. He received a BA from Knox College in 1977. In 1978 he married the former Faith Terpstra and entered graduate school in computer science at Cornell University. While at Cornell he received a MS (1981), started a business (Johnson Software, 1982), and became a father (Joy Ellen, 1983). He is currently on the faculty of the University of Illinois at Urbana-Champaign.

Acknowledgements

Since this will probably be the last degree that I receive, I would like to thank the many teachers who have helped me through twenty-three years of being a student. Some stand out particularly in my mind. My first teacher, Sue Johnson, not only taught me reading, writing and arithmetic, but taught me a love for learning that has lasted these many years. Ken Maurizi, my high-school biology teacher, was the first teacher to push me to my limits, but fortunately not the last. William Ripperger and Richard Reno introduced me to the joys of programming and the excitement of seeing my creations actually work. While the entire Cornell Computer Science faculty was instrumental in teaching me computer science, Fred Schneider, my advisor, taught me to *write* computer science, for which I am greatful. I hope his high standards will remain with me. To all my teachers I offer my sincere thanks.

A large number of people have helped with this thesis, either by reading and criticising it or by helping me to solve problems that I encountered along the way. I have lost count of the number of times that Fred Schneider has read the thesis; the problems in readability that remain are in spite of his efforts. Bowen Alpern gave a lot of valuable advice when I needed it. Thanks are also due to Ozalp Babaoglu, Rance Cleaveland, Cynthia Dwork, Michael Fischer, David Gries, Simon Kaplan, Mark Krentel, Leslie Lamport, Ryan Stansifer and David Wright, all of whom read precursors or earlier versions of the thesis.

I would like to thank the National Science Foundation for supporting me with a fellowship and an assistantship. I would also like to thank the Cornell Campus Store for supporting me for several years and giving me an opportunity to learn about "the real world".

The most important support I received was the moral support from my family and friends. Thanks for putting up with me, Faith, and helping me to keep on working hard.

Most important, I thank God for answering my prayers for a vocation, for giving me whatever talents I have, for giving me inspiring teachers and a loving family, and for giving me the strength to finish this thesis.

I was pushed back and about to fall,
but the Lord helped me.
The Lord is my strength and my song;
he has become my salvation.

from Psalm 118

Contents

List of Figures

Notational Conventions

1. All variable names are either Greek letters or in Roman italics.
2. Sets and graphs are upper case.
3. Invariant names of the form *I3.2* mean the second part of the invariant of Algorithm 3.
4. Processors are p, q, r.
5. Shared variables are u, v, w.
6. Nodes that can be either processors or variables are x, y, z.
7. i, j, k, l are integers.
8. n, m are names (members of $NAMES$).
9. Ψ is a supersimilarity labeling, Φ is a subsimilarity labeling, and Θ is a similarity labeling.
10. α, β, and γ are labels.
11. φ, ϱ, and ε are schedules (sequences of processor names).
12. $\varphi\varrho$ is the concatenation of schedules φ and ϱ.
13. Σ and Γ are systems of processors.
14. π, μ, and ρ are functions from the nodes of one system to those of another.
15. Programs use the guarded-command notation of Djikstra [D76].
16. Quantified formulas, whether in programs or proofs, are of the form

$$(\forall x, y\ \exists z : x \in X, y, z \in Z : \mathcal{P}(x, y, z))$$

which means that for all x in set X and y in set Z there is a z in Z such that $\mathcal{P}(x, y, z)$ is true.

17. The subset of X whose members satisfy a predicate $\mathcal{P}$ is specified by

$$\{x : x \in X : \mathcal{P}(x)\}.$$

18. The cardinality of a set X is $|X|$.

19. Tuples are delimited by $\langle$ and $\rangle$.

Chapter 1

Introduction

While computer systems incorporating thousands of processors have been discussed [GLR83], most systems that have actually been built contain fewer than a hundred processors. It is easy to individualize a hundred or so processors by giving each a special program or initial state. However, VLSI technology uses photographic techniques to mass-produce identical components. The slightest change from one component to the next, such as requiring each to have a unique identifier, greatly increases the production cost of a system. Thus, as the number of processors increases, it becomes more and more important that processors be identical. In the extreme case, it is not even feasible for each processor to have a unique identifier.

One of the main reasons for building multiprocessor systems is to obtain machines powerful enough to solve larger problems than present-day computers can solve. Experience shows that we can always make use of more powerful machines. Thus, it is reasonable to try to imagine what a computing system with a million processors would be like. A mega-multiprocessor would probably have only a few types of processors. It would be cheapest if there were only one type of processor, i.e. if all processors were identical. Also, a mega-multiprocessor would probably have a regular interconnection network, since such a network would be easiest to build and to use.

Some regular interconnections of identical processors result in systems in which each processor will always be in the same state as every other. For example, consider a ring of identical processors where each can send messages to its left neighbor and receive messages from its right neighbor. Identical processors are initially in the same state, so if all processors execute in lock-step then they will always be in the same state–processors in the same state execute the same instruction, send the same messages to the left, and receive the same messages from the right. If each of the processors in the system are always in the same state as the others, then together they can do no more work than a single processor, so all but one of the processors are unnecessary. Such a system is not very useful.

Systems in which each processor is in the same state as every other are commonly described as being *symmetric*. Unfortunately, this description is not precise because symmetry means different things to different people. Most would agree, however, that a fundamental requirement of any definition for symmetry is that two systems should be considered symmetric if they behave in the same way, and two components of a system should be considered symmetric if they cannot distinguish themselves from each other. This is consistent with the definition of symmetry used in graph theory, where two graphs or components of a graph are considered symmetric if there is an isomorphism mapping one to the other. The graph-theoretic definition of symmetry has been applied to systems of interconnected processors [RFH72][A80], as have other definitions [B81][S82][B84]. Unfortunately, the various definitions are not equivalent, most are model sensitive, and each is either undecidable or fails to capture the interchangeability of symmetric systems or indistinguishability of components.

1.1. Characterizing Symmetry

A definition of symmetry for distributed systems can be either syntactic or semantic. A *syntactic* definition depends only on the topology of the system and on text of the program executed by each processor. For example, the set of operations available in the programming language is a syntactic property. A *semantic* definition depends on the behavior of the components of the system. For example, the set of states that processors can be in when running a particular program is a semantic property. A semantic definition of symmetry is more likely to capture what we believe to be fundamental–indistinguishability of system components–because the most important feature of any computing system is the way it computes, i.e. its behavior. Naturally, the behavior of a system will depend, to some extent, on the interconnection of its components.

The most common definition of symmetry for distributed systems is the graph-theoretic definition given above, which is syntactic. We call this definition *G-symmetry* to distinguish it from others. In order for components of a system to be G-symmetric, the system must be represented by a graph, called the *system graph*, with each component of the system represented by a node and communication paths between components represented by edges. Two components of a system are G-symmetric when there is an isomorphism of the graph mapping the node representing one component to the node representing the other.

G-symmetry is not an adequate definition of symmetry for distributed systems because G-symmetric components do not necessarily exhibit the same behavior. They do in some models, such as the networks of finite automata of [RFH72] called *intelligent graphs.* However, Angluin [A80], using a model based on CCS [MM79], showed that any system represented by a complete graph–including G-symmetric ones–could ensure that each processor eventually entered a unique state. Thus, a G-symmetric graph ensures that symmetric components are always in the same state as each other in one model (intelligent graphs), but not in another (CCS). Clearly, G-symmetry does not adequately characterize symmetry for distributed systems.

A different syntactic definition of symmetry for distributed systems was given by Burns [B81]. He examined published distributed algorithms in order to determine what their authors (probably) meant by symmetry. He then formulated two definitions, both applicable only to systems in which each processor can access every shared variable. The definitions cover algorithms that had been proposed up to that time and are used to give lower bounds on the number and types of shared variables needed to solve various synchronization problems in "symmetric" systems. However, the definitions are not intuitive, and they are not general since they require each processor to be able to access every shared variable.

If symmetry is defined in terms of the semantic properties of the program being executed, then determining symmetry is undecidable, as is the case for the definition proposed by Bouge [B84]. Bouge has defined semantic symmetry for programs written in CSP [H78], a language for describing systems that communicate using synchronous message-passing. Each communication in a CSP program involves two processes, the sender and the receiver, and a value to be communicated. The behavior of a CSP program is defined by a set of communication sequences. Simplifying Bouge's definition a little, two processes are *B-symmetric* if there is an isomorphism ρ of the system graph mapping one processor to the other such that for any possible communication sequence s, $\rho(s)$ is also a possible communication sequence of that program. B-symmetry is a semantic property, because it is based on communication sequences. Since this definition is based on execution of a particular program, B-symmetry is undecidable, though Bouge has both found necessary syntactic conditions and found sufficient syntactic conditions for B-symmetry in certain classes of CSP programs. In addition, he has been able to use his definitions to solve a number of interesting problems, such as proving that there can be no B-symmetric solution to the Dining Philosophers problem [D71] in CSP.

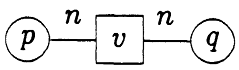


Figure 1.1: A Trivial System.

The question of symmetry in CSP is particularly interesting because a CSP process must explicitly name every other process with which it communicates. Thus, for communication to be possible at all, every CSP process must execute a different program. It runs counter to our intuition for processors with different programs to be symmetric. In addition, we believe that symmetry should depend only on the system components and their interconnection, not on the choice of a particular program.

In this thesis, we propose the *similarity relation* as a semantic definition of symmetry and show that it is well suited for understanding properties of interconnected processors. Two components of a system are *similar* if there is a schedule that, for any program, causes the two components to have the same state infinitely often.[1] Since the definition of similar is based on the behavior of system components, similarity is a form of semantic symmetry. However, we are able to show how to compute the similarity relation for practical systems given only their syntactic description, i.e. their interconnection network, their initial state, and the assumptions that programmers can make about allowable instructions and the scheduling policy.

1.2. Similarity is Subtle

The similarity relation of a system is related to the system's interconnection network, but not in a straightforward way. The relationship between a system's interconnection graph and its similarity relation depends on the model of computation: the processors' instruction set, the scheduling policy, the types of values that can be stored in shared variables, etc.

To illustrate the subtleties of computing the similarity relation for a system, let us examine several systems and, for each system, solve the *selection problem*. The selection problem (based on the coordinated choice problem of [R82]) is to decide whether or not there is a *selection algorithm* for a given

[1] Alternatively, components could have been defined to be similar if for every program there is a schedule that causes the two components to have the same state infinitely often. Our proofs that components are similar always provide a single schedule that works for all programs. Thus, the first definition is preferable. It also implies the second definition.

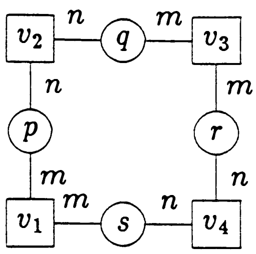


Figure 1.2: No Selection.

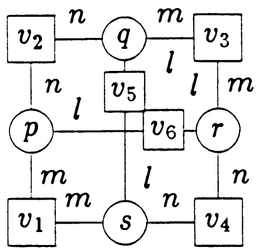


Figure 1.3: Selection.

system of processors, where a selection algorithm is one in which processors select exactly one *leader* by communicating with each other. The selection problem is typical of synchronization problems and is used in this thesis as a device for exploring similarity. The selection problem and the similarity relation complement each other because similarity provides the theoretical tool needed for solving the selection problem, while the selection problem is a concrete example that motivates and provides intuition for studying the similarity relation.

The set of instructions that components can perform is a syntactic property of a system. It influences similarity and therefore the feasibility of solving the selection problem. Consider the system in Figure 1.1. Processors p and q share a variable v that they both refer to by name n. If p and q can only read and write v and both have the same program and initial state, then no program can select either as a leader. This is because the schedule where p and q alternate instructions ensures that any state of one is the state of the other. On the other hand, if p and q can lock shared variables then both can attempt to lock the variable each calls n. The first processor to succeed (and only one can) is selected. Thus, there is a selection algorithm for the system of Figure 1.1 when processors can execute locking instructions and there is no selection algorithm when they cannot.

Locking instructions are not sufficient, however, to guarantee existence of a selection algorithm for a given system. For example, Figure 1.2 shows a system for which there is no selection algorithm, even when processors can lock neighboring variables. Although processor p can use locking instructions to distinguish itself from q and s, the schedule $p\,q\,r\,s\,p\,q\,r\,s\,\ldots$ causes any state of p to be a state of r, and *vice versa*. Figure 1.3, a slight modification to the system of Figure 1.2, depicts a system that does have a selection algorithm. There, exactly one processor can succeed in locking the variable it calls l, then the variable it calls m, and finally the variable it calls n. That processor is selected.

The systems of Figures 1.1 and 1.3 (for which selection algorithms exist) have one important property in common that the system in Figure 1.2 (for

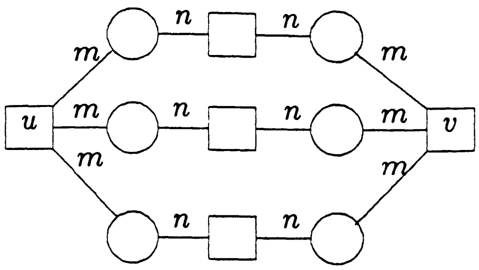


Figure 1.4: Counterexample to Conjecture

which no selection algorithm exists) lacks: every pair of processors share a variable. An obvious conjecture is:

> A G-symmetric system of processors that can lock shared variables can select a leader only when every pair of processors shares at least one variable.

The system in Figure 1.4 is a counterexample to this conjecture, since it is G-symmetric and has processors that share no variable, yet it has a selection algorithm. Processors can lock their n-neighbor and, if they succeed, lock their m-neighbor. Processors keep a count of the number of processors that attempt to lock u or v. A processor that locks u or v sets a flag stored there and then unlocks it. The second processor that succeeds in locking u or v is selected. The algorithm works because exactly three processors can succeed in locking their n-neighbor. When these processors then try to lock their m-neighbor, at least half of them will try to lock the same variable. This means that at least two processors will try to lock one of u or v, while no more than one will try to lock the other variable. Thus, exactly one processor can be selected.

The previous examples showed that some symmetric graphs describe systems that have selection algorithms, while others do not. We will see later that sometimes processors that are not even G-symmetric can be indistinguishable. In short, semantic symmetry is not trivially related to syntactic symmetry.

Calculating the similarity relation for a system must depend on the model of computation being used, since there are systems that have a selection algorithm only if locking instructions are allowed, but have none otherwise. This is unfortunate, because we would like model independent results. On the other hand, the notion of indistinguishability–similarity–is not model dependent. For each model that we examine, we will prove a theorem that tells when systems or components in that model are similar. This theorem becomes the basis for calculating the similarity relation for systems in that model, so it encapsulates the model-dependent aspects of similarity. Differences in these theorems (hence, what affects similarity) are a measure of the relative power of different models. For example, Figure 1.1 showed that processors that can execute locking instructions are more powerful than those that cannot, since a selection algorithm exists in that system only if locking instructions are available. In this thesis, we use similarity to explore the relative power of different instruction sets, different scheduling policies, and the use of randomization.

1.3. Organization of Thesis

The major contribution of this thesis is the similarity relation, a model-independent characterization of symmetry that can be used to solve various problems in distributed computing and to compare the power of various models. Most of this thesis concerns showing how to calculate the similarity relation, hence solve the selection problem, for a variety of different models of distributed computation. However, the specific solutions are not as important as the theory and techniques that are presented. The models were chosen to illustrate problems that arise when calculating the similarity relation. Other models should have algorithms for calculating the similarity relation like those algorithms presented in this thesis.

Chapter 2 describes the shared variable model used in most of the thesis. Chapter 3 formalizes the selection problem and similarity and explores their relationship. Chapter 4 examines the similarity relation in systems without locking. It provides an introduction to our techniques for calculating the similarity relation and for using it to solve the selection problem for systems without locking that satisfy certain fairness conditions. Chapter 5 shows how to generalize our technique to handle families of systems, instead of one system at a time. Chapter 6 completes the solution of the selection problem for systems without locking, while Chapter 7 solves the selection problem for systems with locking. Chapter 8 solves the selection problem for CSP and compares our work with that of Bouge, who also studied this problem. Chapter 9 examines the applicability of similarity to a number of problems in distributed systems. Similarity is used to show when the mutual exclusion problem has a solution, when processors can be given unique identifiers, and to prove that there is no distributed, symmetric, deterministic solution to the Dining Philosophers problem. There, we also argue that similarity is a good characterization of symmetry in distributed systems and discuss the meaning of "breaking symmetry". And, we show that probabilistic algorithms are ways of avoiding the assumptions that we have made about worst-case behavior and we use similarity to describe the power of probabilistic algorithms in distributed systems.

Chapter 2

Definitions

Throughout most of this thesis, a *system* $\Sigma = (N, init, I, SP)$ models a set of processors that communicate using shared variables.[1] N is a network connecting processors to shared variables; *init* is the initial state; I is an instruction set for processors; and SP is a set of possible schedules. As we will see, this four-tuple contains enough information about a system to compute its similarity relation. We now consider each element of the four-tuple in detail.

The *network* N is a connected bipartite graph where nodes correspond to processors P and shared variables V. An edge always connects a node in P to a node in V and is labeled by a function *naming* that maps the set of edges E to a set $NAMES$ of local names processors give to variables. A system $(M, init, I, SP)$ is a *subsystem* of $(N, init, I, SP)$ if M is a subgraph of N.

Given $p \in P$ and $v \in V$, p is an *n-neighbor of* v if there is an edge $e \in E$ from p to v such that $naming(e) = n$. A variable can be given different names by different processors; e.g. if processors are connected in a ring, one processor might refer to a variable as *left* and another might refer to that same variable as *right*. We require that each processor has exactly one n-neighbor for each element n in $NAMES$:

$$\begin{aligned}(\forall p, n : p \in P, n \in NAMES : \\ |\{v : v \in V : v \text{ is an } n\text{-neighbor of } p\}| = 1).\end{aligned} \tag{2.1}$$

This ensures that whenever a processor refers to a name n, it unambiguously refers to a single variable. Thus, there is a function nbr such that

$$nbr(n, p) \equiv \text{ the variable that is the unique } n\text{-neighbor of processor } p.$$

The *state* of a system is a function from nodes to states. Given a state *state* of a system and a node x, *state*(x) is the state of x in *state*. Processors

[1] Message-passing communication is considered in Chapter 8.

and shared variables each have their own states; the state of a processor might include private variables or registers. We make no assumptions about the number of possible states of a processor or variable.

A processor changes its state and the state of variables it can access by executing *instructions*. Each processor can execute a finite set of instructions satisfying the

Instruction Set Assumption: All processors in a system can perform the same set of instructions.

The instruction that a processor executes is determined by its program and its state, since the program counter is part of the state of the processor. All processors also satisfy the

Program Assumption: All processors in a system execute the same program.

Thus, processors in the same state will execute the same instruction. The Program Assumption is not really a limitation, since processors can have different initial states and so are capable of simulating processors with different programs.

Instruction sets are categorized by the kinds of instructions they contain. We look at two different instruction sets, S and L. S (simple instruction set) contains *local* instructions that only change the state of the processor executing them and the **read** and **write** instructions. Execution of

$$\textbf{read } i \textbf{ from } n$$

by processor p stores the value of $nbr(n,p)$ into p's local variable i, where $nbr(n,p)$ is the shared variable that p calls n. Execution of

$$\textbf{write } i \textbf{ to } n$$

by processor p stores the value of i into $nbr(n,p)$.

L (locking instruction set) consists of S extended by **lock** and **unlock** instructions, which implement binary semaphores associated with shared variables. These instructions use a *lock bit* associated with each shared variable. Execution of

$$\textbf{lock } n$$

by a processor p causes p to wait until the lock bit of $nbr(n,p)$ is cleared (zero) and then, in a single atomic action, sets it. Execution of

$$\textbf{unlock } n$$

by p resets the lock bit of $nbr(n, p)$. Note that any of the synchronization primitives that can be used to implement mutual exclusion (e.g. test-and-set, P and V) could have been used instead of **lock** and **unlock**.

An *execution* of a system can be viewed as a sequence of steps, where each *step* corresponds to a processor executing a single instruction. A *schedule* is a possibly infinite sequence of processor names, where a name signifies that the named processor executes a single step. Thus, a schedule defines a sequence of steps. The first step in the schedule is executed on the initial state of the system to produce the next state. Each succeeding step is executed on the state that results from the previous step. Any sequence of processor names is a schedule, so the catenation of two schedules is a schedule and any subsequence of a schedule is a schedule. We will write $p \in \varphi$ when p is one of the processors in schedule φ. Since the sequence of states that exist during execution of a program is a function of the schedule as well as of the program, the state of a system $\Sigma = (N, init, I, SP)$ after executing (finite) schedule ε is denoted by $init \circ \varepsilon$. The prefix of ε of length i is denoted by $\varepsilon|_i$. Thus, the state of Σ after executing the first i steps of ε is given by $init \circ \varepsilon|_i$.

In this thesis, we consider three types of schedules as SP: general (G), fair (F), and bounded fair (BF). A *general* schedule has no restrictions. A *fair* schedule contains each processor infinitely often. A *k-bounded fair* schedule contains each processor at least once in any substring of the schedule of length k. A system is *fair* if all schedules in SP are fair; it is *bounded fair* if there is a k such that all schedules in SP are k-bounded fair. A bounded fair scheduling policy corresponds to a synchronous system while a fair scheduling policy corresponds to an asynchronous system.

Chapter 3

The Selection Problem

With these preliminaries out of the way, the selection problem can be precisely formulated. Assume that each processor p has a local Boolean variable $selected_p$, which is initially false. A processor p is selected when it sets $selected_p$ to true. A *selection algorithm* for a system Σ is a program that always establishes

Uniqueness: Exactly one processor is selected.

and maintains

Stability: Once selected, a processor remains selected.

A selection algorithm is guaranteed to select exactly one processor, no matter which schedule is followed. However, different schedules may result in different processors being selected.

The *selection problem* for a system Σ is:

Selection Problem: Decide whether there is a selection algorithm for Σ and, if one exists, produce it.

Various researchers have shown how to select a leader when every processor has a unique identifier; this is a selection algorithm for a particular class of systems[G82][DKR82][P82][A85]. In contrast, we are interested in a decision procedure, not just a selection algorithm.

The selection problem is trivial for some models.

Theorem 1. *There is no selection algorithm for a system in S with general schedules.*

Proof: The proof is by contradiction. Suppose there is a selection algorithm for such a system. Thus, this algorithm eventually selects a processor, no matter which schedule is followed. Let schedule φ be a schedule that selects some processor, say p. Let ε be the prefix of φ up to the step where p is selected. Since general schedules are allowed, there must be a continuation of ε that will cause a selection even though p is not involved in any subsequent steps. Let q be the processor different from p that is selected in that schedule, and let ϱ be the schedule that can be appended to ε to select q. Thus, executing $\varepsilon\varrho$ selects q, and executing $\varphi = \varepsilon p$ selects p.

Since p executes only simple instructions, the instruction it executes when it is selected is either a local instruction or a **read** instruction, since a **write** instruction cannot change local variable $selected_p$. Execution of this local or **read** instruction changes only the state of p. Execution of ϱ cannot depend on p's state, because ϱ does not contain p. Therefore, $\varphi p \varrho$ would cause both p and q to be selected, which violates Uniqueness. So, the algorithm is not a selection algorithm–a contradiction. ∎

Theorem 2. *There is no selection algorithm for a system in L with general schedules unless there is a single variable that all processors can access.*

Proof: The proof is by contradiction. Suppose that there is a selection algorithm for Σ in L even though no variable in Σ is accessed by all processors. Let schedule φ be a schedule that selects some processor, say p. Let ε be the prefix of φ up to the step where p is selected and let v be the variable that p accesses in the step in which it selects itself. Since general schedules are allowed and there are processors that cannot access v, there must be a continuation of ε that will cause a selection even though no processor that accesses v is involved in any subsequent steps. Let q be the processor different from p that is selected in that schedule, and let ϱ be the schedule that is appended to ε to select q. Since the step in which p selects itself changes only p and v, and since no processor in ϱ accesses v, executing $\varepsilon\varrho p$ selects both p and q; a contradiction. ∎

Solution of the selection problem is much like the consensus problem described in [FLP83]. There, a set of processors are to decide on a particular value and the decision depends partly on each processor's initial state. The selection problem is equivalent to reaching a consensus on the processor to be selected, while consensus can be achieved by letting the selected processor make the decision. Thus, one might (incorrectly) think that Theorem 1 is also a proof of the main result of [FLP83], that there is no algorithm for achieving distributed consensus with one faulty processor that fails by halting. However, a selected processor can halt before it communicates its decision, so the consensus problem cannot be reduced to the selection problem. Moreover, the processor decided upon by the consensus can wait arbitrarily long, perhaps halting, so the selection problem cannot be reduced to the consensus problem. Thus, neither problem is reducible to the other for general schedules.

In light of Theorems 1 and 2, we henceforth ignore the case of general schedules and concentrate on fair and bounded fair schedules. For these classes, we can show that no processor can be selected in a system when there is a schedule that prevents some processor from entering a unique state. A particular schedule φ causes a set of processors or variables to *behave similarly* if, for any program, it causes the members of that set to have the same state at the same time infinitely often. Reformulating the definition of similarity in these terms:

> Nodes x and y (processors or variables) are *similar*, denoted $x \sim y$, if there is a schedule that causes them to behave similarly.

The following theorem shows how the notion of "behave similarly" can be used to solve the selection problem.

> **Theorem 3.** *If some schedule causes each processor in a system to behave similarly to some other processor, then there is no selection algorithm for that system.*

Proof: Suppose schedule φ causes each processor to behave similarly to some other and an algorithm selects some processor p when executed with schedule φ. By hypothesis, some processor q behaves similarly to p and so it will also be selected. If Stability is satisfied then Uniqueness is violated, so the algorithm cannot be a selection algorithm. ∎

For example, the system of Figure 1.1 with no locking instructions and a round-robin schedule causes p and q to behave similarly, so, according to Theorem 3, no program can select either.

It is tempting to try to formulate Theorem 3 in terms of "similar" instead of "behave similarly". This would be a mistake, because similarity is not necessarily transitive. Given $x \sim y$ and $y \sim z$, there might not be any schedule that causes x, y, and z (together) to behave similarly. Thus, a schedule that caused x and y to behave similarly might select z, and a schedule that caused y and z to behave similarly might select x. However, similarity is usually transitive, hence an equivalence relation, and can then be represented by a labeling of the network graph.

Let *PLABELS* be a set of labels of processors and *VLABELS* be a set of labels of variables; and assume these two sets are disjoint. A *supersimilarity labeling* Ψ is a labeling such that there is a schedule that causes nodes with the same label to behave similarly, thus nodes with the same label are similar.

A labeling that assigns a unique label to each node is a trivial supersimilarity labeling. In contrast, a *subsimilarity labeling* Φ is a labeling that gives nodes the same label if some schedule causes them to behave similarly, thus it gives similar nodes the same label. A labeling that assigns all nodes the same label is a trivial subsimilarity labeling. A *similarity labeling* Θ is one that is both a supersimilarity labeling and a subsimilarity labeling. Since similarity is not necessarily transitive, some systems have no similarity labeling. To summarize, when a similarity labeling Θ exists,

$$\begin{aligned}
&\Psi(x) = \Psi(y) \Rightarrow x \sim y, \\
&x \sim y \Rightarrow \Phi(x) = \Phi(y), \\
&x \sim y \iff \Theta(x) = \Theta(y).
\end{aligned}$$

Since there can be but one similarity relation for a system, a similarity labeling, if it exists, is unique up to isomorphism.

To help remember the difference between supersimilarity labelings and subsimilarity labelings, consider the partial ordering $<$ of labelings of a system graph where $\Psi < \Psi'$ if all nodes given the same label by Ψ' are also given the same label by Ψ. Thus, Ψ' has at least as large a range as Ψ. Under this partial ordering, the labelings of a system graph form a lattice. The top of the lattice is the trivial supersimilarity labeling that gives each node a unique label, while the bottom of the lattice is the trivial subsimilarity labeling that gives all nodes the same label. All supersimilarity labelings will be above the similarity labeling in this lattice, while all supersimilarity labelings will be below it.

Finding the similarity labeling is a major part of solving the selection problem, as shown by the following.

Theorem 4. *If there is a supersimilarity labeling for a system that gives every processor the same label as some other then there is no selection algorithm for that system.*

Proof: Follows from the definition of supersimilarity and Theorem 3. ∎

Lemma 5. *If Θ is a similarity labeling and Ψ is a supersimilarity labeling, then*

$$(\forall x \exists y : x, y \in P \cup V : \Psi(x) = \Psi(y) \Rightarrow \Theta(x) = \Theta(y)).$$

Proof: By definition of supersimilarity labeling, $\Psi(x) = \Psi(y) \Rightarrow x \sim y$. Since Θ is a similarity labeling, $x \sim y \Rightarrow \Theta(x) = \Theta(y)$. ∎

The similarity labeling for a system tells whether or not there is a a selection algorithm for the system and can be used to find a selection algorithm if it exists. Lemma 5 shows that the similarity labeling, if it exists, contains as much information about the similarity relation as any supersimilarity labeling. Thus, by Theorem 4, if the similarity labeling gives every processor the same label as some other then there is no selection algorithm. Otherwise, some processor is uniquely labeled by the similarity labeling. In this case, a selection algorithm can be constructed by having each processor learn its label under the similarity labeling. A processor selects itself if it is the one with a particular designated label that is known to be unique.

This suggests the following general approach for solving the selection problem for a particular model:

1. Describe supersimilarity labelings for systems.
2. Give an algorithm to find the similarity labeling, Θ.
3. Give a distributed program that allows each processor p to compute $\Theta(p)$ and select itself if $\Theta(p)$ is some unique value.

The first two steps are nearly the same for all models. The distributed program for the third step is usually derived from the algorithm of the second step, but differs from one model to another because means of interprocess communication varies across different models.

This three-step general approach will not solve the selection problem for all models, but only one extra idea is needed. The *generic selection problem* is to decide whether or not there is a single selection algorithm that works for any system in a *family* of systems. The selection problem for some models can be reduced to the generic selection problem for simpler models. For example, the selection problem for a locking system can be reduced to the generic selection problem for a family of simple systems having the same interconnection network as the original locking system. And, the generic selection problem for families of simple systems with the same interconnection network is the same as solving the selection problem for simple systems, which can be solved using the three-step general approach outlined above.

Chapter 4

Similarity in Simple Systems

The first model we study is systems of processors using instruction set S, described in Chapter 2. We start by giving some useful results about similarity for systems in S. These results are based on the intuitive idea that similar processors and variables should remain in the same state after executing the same instruction—otherwise, a program that recognized the different states could ensure that processors would never have the same state again, contradicting the assumption that they were similar. In particular, we consider execution of several types of instructions by similar processors and see what is necessary to ensure that all similar nodes remain in the same state after execution of each type of instruction.

If two processors or variables do not have the same initial state then they are not similar, because a program that prevents them from changing state will prevent them from having the same state infinitely often. Thus, if *init* is the initial state of a system containing nodes x and y, then

$$x \sim y \Rightarrow init(x) = init(y). \tag{4.1}$$

Similar processors will not have the same state after a **read** i **from** n instruction unless the variables they read have the same state. So, n-neighbors of similar processors must themselves be similar.

$$(\forall p, q : p, q \in P : p \sim q \Rightarrow (\forall n : n \in NAMES : nbr(n, p) \sim nbr(n, q))) \tag{4.2}$$

Similar variables will not have the same state after a set of similar processors execute a **write** i **to** n instruction unless those processors write the same value, and if one variable is written then any variable similar to it must also be written. Since similar processors have the same state, they will write the same value when executing the same instruction. Thus, if a round-robin schedule causes two variables to behave similarly then every n-neighbor of one must be similar to some n-neighbor of the other. This suggests that if two variables are similar then every n-neighbor of one must be similar to some n-neighbor of the other.

$$\begin{aligned}
&(\forall u, v : u, v \in V : \\
&\qquad u \sim v \Rightarrow \\
&\qquad (\forall n, p : n \in NAMES, p \in P : \\
&\qquad\qquad u = nbr(n,p) \Rightarrow (\exists q : q \in P : v = nbr(n,q) \wedge p \sim q)))
\end{aligned} \tag{4.3}$$

Since we have not shown that similar variables will always behave similarly under a round-robin schedule, (4.3) is not a theorem. In fact, we will see that (4.3) holds for bounded-fair systems and is useful in analyzing all systems.

If (4.1), (4.2), and (4.3) hold for any set of nodes that behave similarly then a supersimilarity labeling Ψ should satisfy the following:

$$\begin{aligned}
&(\forall x, y : x, y \in P \cup V : \Psi(x) = \Psi(y) \Rightarrow init(x) = init(y)) \\
&\wedge (\forall p, q : p, q \in P : \\
&\qquad \Psi(p) = \Psi(q) \Rightarrow (\forall n : n \in NAMES : \\
&\qquad\qquad\qquad \Psi(nbr(n,p)) = \Psi(nbr(n,q)))) \\
&\wedge (\forall u, v : u, v \in V : \\
&\qquad \Psi(u) = \Psi(v) \Rightarrow \\
&\qquad (\forall n, p : n \in NAMES, p \in P : \\
&\qquad\qquad u = nbr(n,p) \Rightarrow (\exists q : q \in P : \\
&\qquad\qquad\qquad v = nbr(n,q) \\
&\qquad\qquad\qquad \wedge\ \Psi(p) = \Psi(q)))).
\end{aligned} \tag{4.4}$$

When a labeling Ψ satisfies (4.2) and (4.3), (the second and third conjuncts of (4.4)) and $\Psi(x) = \Psi(y)$ we say that x and y *have the same environment under labeling* Ψ. An environment of a process or variable can be represented by a set of pairs consisting of a name n and the label under Ψ of one of its n-neighbors. Thus, if p is a process then the environment of p under labeling Ψ, $Env_\Psi(p)$, is given by

$$Env_\Psi(p) \equiv \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \Psi(nbr(n,p)) = \alpha\}. \tag{4.5}$$

If v is a variable then $Env_\Psi(v)$ is given by

$$\begin{aligned}
Env_\Psi(v) \equiv \{(n, \alpha) : n \in NAMES, \alpha \in PLABELS : \\
(\exists p : p \in P : v = nbr(n,p) \wedge \Psi(p) = \alpha\}.
\end{aligned} \tag{4.6}$$

The following lemma shows that (4.5) and (4.6) faithfully represent an environment under a labeling, as defined by (4.4).

Lemma 6. *Nodes x and y have the same environment under label Ψ if and only if $Env_\Psi(x) = Env_\Psi(y)$.*

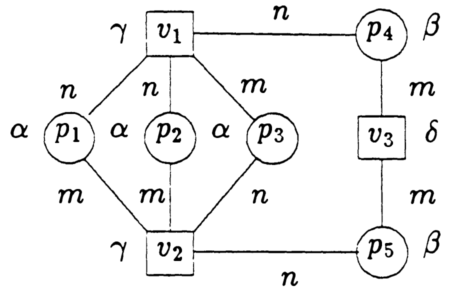


Figure 4.1: Environment Example

Proof: There are two cases, depending on whether x and y are processors or variables. The second conjunct of (4.4) holds for processors and is satisfied by definition (4.5). The third conjunct holds for variables and is satisfied by definition (4.6). ∎

Figure 4.1 illustrates environments and a labeling that satisfies characterization (4.4) of a supersimilarity labeling Ψ. Let Ψ be a labeling such that $\Psi(p_1) = \Psi(p_2) = \Psi(p_3) = \alpha$, $\Psi(p_4) = \Psi(p_5) = \beta$, $\Psi(v_1) = \Psi(v_2) = \gamma$, and $\Psi(v_3) = \delta$. Ψ satisfies the second conjunct of (4.4) because processors given the same label by Ψ have n-neighbors with the same label, for all names n. It satisfies the third conjunct of (4.4) because any n-neighbor of variable v is given the same label by Ψ as some n-neighbor of any other variable labeled $\Psi(v)$ by Ψ. The environment under Ψ of v_1 is the same as that of v_2, i.e. $\{(n, \beta), (n, \alpha), (m, \alpha)\}$. The environment under Ψ of p_1 is the same as that of p_2 and p_3, i.e. $\{(n, \gamma)(m, \gamma)\}$. Processors p_4 and p_5 also have the same environment $\{(m, \delta)(n, \gamma)\}$. Thus, all nodes with the same label under Ψ have the same environment under Ψ. If Ψ satisfies the first conjunct of (4.4) (e.g. if all nodes in the system have the same state), then Ψ is a supersimilarity labeling, as shown by the following theorem.

Theorem 7. *For any labeling Ψ on any fair system Σ in S with initial state init, if*

$$\begin{aligned}(\forall x, y : x, y \in P \cup V :& \\ &\Psi(x) = \Psi(y) \Rightarrow (Env_\Psi(x) = Env_\Psi(y) \wedge init(x) = init(y)))\end{aligned} \tag{4.7}$$

then Ψ is a supersimilarity labeling for Σ.

Proof: We construct a round-robin schedule φ that causes nodes with the same label under Ψ to behave similarly. Therefore, Ψ is a supersimilarity labeling.

The schedule φ consists of an infinite sequence of finite *small schedules*

$$\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i\ \ldots$$

Each small schedule ε_i consists of one step by every processor given a particular label by Ψ. Thus, if processor p is involved in a step in ε_i then there is a step in ε_i for every other processor also labeled $\Psi(p)$. In addition, φ is constructed so that small schedules for all labels occur before the small schedule for any label is repeated. Thus, φ is bounded-fair as well as fair.

Let T_i be the state after φ has been executed up through ε_i. Thus,

$$T_i = init \circ \varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_i.$$

We prove by induction that $\Psi(x) = \Psi(y) \Rightarrow T_i(x) = T_i(y)$ for all $i \geq 0$ and all nodes x and y. The base case, $i = 0$, follows from the conjunct $init(x) = init(y)$ in (4.7). The induction step is to show that if nodes given the same label by Ψ have the same state after ε_i, then they will have the same state after ε_{i+1}. Thus, we can assume

$$\Psi(x) = \Psi(y) \Rightarrow T_i(x) = T_i(y).$$

By construction,

$$(\forall p, q : p, q \in \varepsilon_{i+1} : \Psi(p) = \Psi(q)) \tag{4.8}$$

so, by the induction hypothesis, all processors in ε_{i+1} have the same state. Since all processors run the same program, all processors in ε_{i+1} will execute the same instruction. This can either be a local, a **read** or a **write** instruction. A local instruction can be considered to be the same as a **read** instruction that ignores the value it reads from the variable. Thus, we consider two cases.

read *Instruction.*

Since all processors in ε_{i+1} are in the same state, they all read shared variables with the same name, say n. By (2.1), each processor has exactly one n-neighbor, so by (4.7) all variables being read are labeled the same by Ψ, and, by the induction hypothesis, they must have the same value. Since all the processors in ε_{i+1} are performing the same instruction and all read the same values, they will all change to the same state. No other processors or variables change state, because no other processors appear in ε_{i+1}. Thus, $\Psi(x) = \Psi(y) \Rightarrow T_{i+1}(x) = T_{i+1}(y)$.

write *Instruction.*

Since all processors in ε_{i+1} are in the same state, they all write shared variables with the same name, say n, and all write the same value. By (4.7) and (4.8), all the variables being written are given the same label by Ψ, and all variables with that label have an n-neighbor involved in a step in ε_{i+1}. Thus, each variable with that label is being written, and all the variables being written have the same label, so exactly the variables with that label are given a new value. Processes with the same label as p will have the same state in T_{i+1} as p. No other nodes change state. Thus, $\Psi(x) = \Psi(y) \Rightarrow T_{i+1}(x) = T_{i+1}(y)$. ▮

4.1. Finding Similarity Labelings

Algorithm 1 (on page 22) computes a similarity labeling for bounded-fair systems in S. In the algorithm, a trivial subsimilarity labeling Φ_0 is first defined and then successively refined to $\Phi_1, \Phi_2, \ldots$ until it becomes a supersimilarity labeling. As long as a Φ_i does not satisfy (4.7), it is not a supersimilarity labeling. Moreover, Φ_i remains a subsimilarity labeling as long as similar nodes are not given different labels. When Φ_i is both a subsimilarity labeling and a supersimilarity labeling, it is a similarity labeling. The trivial subsimilarity labeling used is *init* (or, more precisely, isomorphic to *init*), which is a subsimilarity labeling because similar processors or variables must have the same initial state. A refinement step for constructing Φ_{i+1} from Φ_i consists of picking a set of nodes with the same label under Φ_i such that two members have different environments under Φ_i, then picking one node x from the set, and finally relabeling all nodes in the set that have environments under Φ_i different than that of x.[1]

Algorithm 1 will compute a similarity labeling for every system where similar nodes have the same environment under a subsimilarity labeling Φ_i and where Theorem 7 holds. It turns out that bounded-fair systems in S meet these requirements. Chapter 6 will explain how a fair, but not bounded-fair, system in S might have a subsimilarity labeling under which similar variables could have different environments. Chapter 7 will explain how processors in a system in L might be dissimilar even though they are given the same label by a labeling that meets the requirements of Theorem 7. However, the labeling produced by Algorithm 1, Θ, will be used to determine similarity for fair systems in S and for systems in L. Thus, Algorithm 1 will be useful even when its result is not a similarity labeling.

We will show in section 4.3 an algorithm for bounded-fair systems in S in which nodes with different environments under a subsimilarity labeling can differentiate themselves. This means that nodes with different environments under a subsimilarity labeling are dissimilar, so similar nodes have the same environment under a similarity labeling, as required by Algorithm 1. Until then, however, we will not have proven that Algorithm 1 computes the similarity labeling for bounded-fair systems. The following theorem gives the assumptions from which we can prove that Algorithm 1 computes a similarity

[1]The algorithm is essentially the same as that used for finite state machine state minimization [H71].

Algorithm 1. Compute similarity labeling Θ for a system in S.

Define $\{\alpha_1, \ldots, \alpha_j, \ldots\} \subseteq (PLABELS \cup VLABELS)$

S1.1: **for** $x \in P \cup V$:
$\quad \Phi_0(x) := init(x)$
rof;
$i := 0$;
$\{A1.1\text{: } (\Phi_i(x) = \Phi_i(y) \iff init(x) = init(y)) \wedge i = 0\}$
$\{I1.1 \wedge I1.2\}$

S1.2: **do** $(\exists x, y : x, y \in P \cup V : \Phi_i(x) = \Phi_i(y) \wedge Env_{\Phi_i}(x) \neq Env_{\Phi_i}(y)) \rightarrow$
$\quad$ **pick** $x \in P \cup V : (\exists y : y \in P \cup V :$
$\qquad \Phi_i(x) = \Phi_i(y) \wedge Env_{\Phi_i}(x) \neq Env_{\Phi_i}(y))$;
$\quad \Phi_{i+1} := \Phi_i$;
$\quad$ **for** $z \in P \cup V$ **such that** $\Phi_i(x) = \Phi_i(z) \wedge Env_{\Phi_i}(x) \neq Env_{\Phi_i}(z)$:
$\qquad \Phi_{i+1}(z) := \alpha_{i+1}$
$\quad$ **rof**;
$\quad i := i + 1$
$\quad \{I1.1 \wedge I1.2\}$
od;
$\Theta := \Phi_i$
$\{(\forall x, y : x, y \in P \cup V : \Theta(x) = \Theta(y) \Rightarrow Env_\Theta(x) = Env_\Theta(y))$
$\quad \wedge I1.1 \wedge I1.2\}$
$\{\Theta$ is a supersimilarity labeling $\wedge\ \Theta$ is a subsimilarity labeling$\}$

labeling. Note that Theorem 7 shows that the first assumption is satisfied for systems in S. Lemma 16 (and a small argument from the proof of Theorem 17) will show that the second assumption is satisfied by bounded-fair systems in S.

Theorem 8. *If, for system* Σ,

1. Ψ *is a supersimilarity labeling if*

$$(\forall x, y : x, y \in P \cup V : \\ \Psi(x) = \Psi(y) \Rightarrow (Env_\Psi(x) = Env_\Psi(y) \wedge init(x) = init(y))).$$

2. *Similar nodes have the same environment under any subsimilarity labeling.*

then Algorithm 1 computes a labeling Θ *such that for nodes* x *and* y, $x \sim y \iff \Theta(x) = \Theta(y)$.

Proof: *I1.1* and *I1.2* are loop invariants for loop *S1.2*.

I1.1: $(\forall x, y : x, y \in P \cup V : \Phi_i(x) = \Phi_i(y) \Rightarrow init(x) = init(y))$
I1.2: Φ_i is a subsimilarity labeling.

That *I1.1* is initially true follows trivially from *A1.1.* Since similar nodes have the same initial state, they will have the same label under Φ_0 by assertion *A1.1.* Thus, *I1.2* is initially true.

I1.1 is reestablished by executing the body of *S1.2* because for any node x, $\Phi_{i+1}(x)$ differs from $\Phi_i(x)$ only when $\Phi_{i+1}(x)$ is not in the range of Φ_i and the new label is assigned only to nodes y that had the same label as x. By *I1.1*, $init(x) = init(y)$.

To see that *I1.2* is reestablished, note that *I1.2* and assumption 2 of the theorem, that similar nodes have the same environment under any subsimilarity labeling, imply that similar nodes have the same environment under labeling Φ_i. If the **for** loop in *S1.2* relabels x then it will relabel all nodes with the same label and environment under Φ_i as x. Therefore, similar nodes will still have the same label and *I1.2* is a loop invariant.

The loop terminates because at most $|P \cup V|$ values are used as labels. Each iteration of the loop increases the number of labels by one, so the loop can be executed no more than $|P \cup V|$ iterations. When there are $|P \cup V|$ different labels every node has a unique label. Then, the guard of *S1.2* is false, and termination is assured.

Since *I1.2* is a loop invariant, Θ is a subsimilarity labeling. When *S1.2* terminates, its guard is false, so

$$(\forall x, y : x, y \in P \cup V : \Theta(x) = \Theta(y) \Rightarrow Env_\Theta(x) = Env_\Theta(y)).$$

Together with *I1.1* and the first precondition to the theorem, this implies that Θ is a supersimilarity labeling. Since Θ is a subsimilarity labeling and a supersimilarity labeling, it is a similarity labeling. ∎

Algorithm 1 has polynomial running time in the number of nodes in N, as shown by the following. Since there are a finite number of nodes in the network graph, an expression that is only quantified over nodes can be evaluated in time polynomial in the number of the nodes, where the degree of the polynomial is the number of quantifications in the expression. The loop of *S1.2* is executed no more times than the number of nodes. Thus, Algorithm 1 takes time polynomial in the number of nodes. In fact, there is a variant of this algorithm that takes time $O(n \; log \; n)$ in [H71].

4.2. Communication Using Shared Variables

In the previous section, we described supersimilarity labelings for systems in S and showed how to calculate similarity labelings (using Algorithm 1). The remaining step in solving the selection problem for bounded-fair systems in S is to develop a distributed algorithm that permits each processor to learn its similarity label. An algorithm to do this, given in Section 4.3, is based on replaying the execution of Algorithm 1, step by step. If step j (iteration j of loop *S1.2*) refines Φ_{j-1} by relabeling a set of processors then, in the distributed algorithm, each processor will examine its own environment and calculate its own label under Φ_j. If step j refines Φ_{j-1} by relabeling a set of variables then in the distributed algorithm, each processor will examine the environment of each neighboring variable and calculate the variable's label under Φ_j.

Replaying Algorithm 1 involves storing a list of labels in each variable. The j'th element on the list in a variable contains a set of labels–the label under Φ_j of each processor neighboring that variable. In S, processors can interfere with each others' attempts to update this list, since only **read** and **write** are atomic. This interference problem must be solved in order to implement the algorithm just outlined.

We first examine a restricted version of the interference problem:

> Initially, each processor p has a set of private values, $priv_p$. There is a single shared variable *shared_var*, which is eventually to contain $common = \bigcup_{p \in P} priv_p$.

In our solution to this problem, called the *communication substrate* (on page 25), each processor p keeps a local copy $local_p$ of an approximation to the list stored in *shared_var*. Each processor p continues to check *shared_var* to ensure that the value it wrote (i.e. $local_p$) is not overwritten by some other processor and adds new information from *shared_var* to $local_p$. If p discovers that $local_p$ contains elements that are not in *shared_var* then p writes $local_p$ to *shared_var*.

Communication Substrate for processor p in S:

$communicate(priv_p) \equiv$
  $local_p := priv_p$;
  write $local_p$ to $shared\_var$;
  do $\neg finished() \rightarrow$
    read $temp$ from $shared\_var$;
    $local_p := local_p \cup temp$;
    if $local_p \neq temp \rightarrow$ write $local_p$ to $shared\_var$
    $\square\ local_p = temp \rightarrow$ skip
    fi
  od

Our communication substrate does not terminate on its own, but requires some way (i.e. the Boolean function *finished*) to detect when all values have been communicated. If, for example, the number of values being communicated is known *a priori* then the loop can terminate when that many values are in *shared_var*. Or, in a bounded-fair system, if the slowest processor takes at most k steps to communicate a set of values and the fastest processor is l times faster than the slowest, then a process knows that all values have communicated by the time it finishes lk steps. Finally, note that even if there is no way to implement the loop guard *finished*, all processors using the substrate eventually learn the set of private values.

We now show that the communication substrate allows each processor eventually to learn the entire set of values being communicated by all participating processors. First recall that, by definition, *common* is the union of $priv_p$ for all processors p. Next, observe that for all processors p, $priv_p \subseteq local_p$. Thus, if $local_p = local_q$ for all processors p and q, then, for any p, $local_p = common$. We must therefore prove that eventually $local_p = local_q$ for all processors p and q.

The only way $local_p$ is changed in the communication substrate is by adding new values to it, so its size is non-decreasing. Unfortunately, the same cannot be said for the size of the set stored in *shared_var*, because a slow processor could write over values stored by a faster processor. However, the set of values in *shared_var* is always equal to $local_p$ for some processor p—the last one that wrote to it. Let a *most ignorant* processor p be one such that $local_p$ contains the set with fewest values. Note that there is always at least one most ignorant processor. (In fact, every processor can be a most ignorant processor.) Moreover, *shared_var* always contains at least as many values as a most ignorant processor, so as the sets of values in the local copies maintained by processors increase in size, the minimum possible size of the set in *shared_var* also increases. We can show that if there is a processor p

that does not know some value in $local_q$, for some processor q, then a most ignorant processor r will eventually learn a new value. Thus, the number of values in the local copies will increase until the entire set of values being communicated is known by each processor and the communication substrate has accomplished its goal.

The following two lemmas show that there is a bound on the number of write instructions that processors using the communication substrate can execute without some processor learning a value. Let a *write schedule* be a schedule with all steps deleted except some of those involving write instructions to *shared_var*. A *subschedule* of length n of a write schedule φ is any consecutive n elements of φ. Note that any subschedule of a write schedule is also a write schedule.

Lemma 9. *If every processor occurs more than once in a write schedule, then every most ignorant processor p in that schedule has increased the size of* $local_p$.

Proof: According to the program for the communication substrate, no processor q will write to *shared_var* more than once unless it has read a value from *shared_var* not contained in $local_q$. If every processor occurs more than once in a write schedule then each processor p read a set from *shared_var* different from $local_p$. Therefore, p has read the value of $local_q$, for some q, where $local_p \neq local_q$. If p is a most ignorant processor then, for any q, $local_q$ is at least as large as $local_p$. Therefore, p has learned a new value, which is added to $local_p$. ▮

Lemma 9 holds for any write schedule, even one that only contains a subset of the processors, since a most ignorant processor is relative to a set of processors. Consequently, Lemma 9 implies that if there is a subschedule such that every processor in the subschedule appears twice, then some processor p has learned something, i.e. increased the size of $local_p$. We can use this fact to bound the number of writes that can be performed before some processor learns a new value. In order for a schedule to prevent any processor from learning a new value, the hypothesis of Lemma 9 must be false for every subschedule.

Lemma 10. *If n processors are executing the communication substrate then during any write schedule of length* 2^n *every most ignorant processor p has increased the size of* $local_p$.

Proof: Let $\varphi(n)$ be the longest write schedule using n processors such that no processor learns a new value. By Lemma 9, for each subschedule of $\varphi(n)$, there is a processor that is involved in exactly one step of that subschedule.

There is a limit to the length a sequence can have if all its subschedules contain some processor exactly once. Let $f(n)$ be the length of $\varphi(n)$. For a base case, $f(1) = 1$. For the induction case, we know by hypothesis that some processor must appear in $\varphi(n)$ only once. Let p be that processor. The subschedules of $\varphi(n)$ to the right and left of the step involving p each involve $n-1$ processors and thus are each of length $f(n-1)$ or less, therefore $f(n) \leq 2f(n-1)+1$, so $f(n) \leq 2^n - 1$. A worst case equal to $2^n - 1$ can be found by recursively constructing each schedule of n processors by concatenating schedules of $n-1$ processors on either side of the n-th processor.[2] ∎

Theorem 11. *In a fair system in S, a set of processors sharing a common variable can use the communication substrate to communicate a finite set of values with each other.*

Proof: If a processor p knows a value not stored in *shared_var*, then that value will be written to *shared_var* by some processor. This is because when p reads *shared_var* either p will notice that the value is missing and write it to *shared_var*, or some other processor will replace the missing value before p discovers it is missing. By Lemma 10, every 2^n write instructions causes some processor to learn a value. If the number of values being communicated is finite then the number of writes until all processors have learned all values is bounded. If no processor is going to write to *shared_var* then *shared_var* contains the values of every processor. In that case, every processor will read *shared_var* and learn the entire set of values. ∎

Although we have bounded the number of **write** instruction executed by processors executing the communication substrate, we cannot bound the total number of instructions since one processor in a fair system can execute arbitrarily many more read instructions than another. In a bounded-fair system, however, the total number of instructions executed can be bounded.

Lemma 12. *In a bounded-fair system in S, a set of processors sharing a common variable can use the communication substrate to communicate a finite set of values to each other by the time any processor has executed $jlk2^k + il$ steps, where*

- *the last processor starts the communication substrate within i steps of the first,*

[2] The worst case occurs when each processor is twice as fast as the next fastest. If all the processors are the same speed then the bound will be much smaller, on the order $f(n) = O(n^2)$.

- *each processor takes j steps to execute the loop body of the communication substrate once,*
- *k processors share the variable,*
- *the fastest processor is l times faster than the slowest.*

Proof: By Theorem 11, the communication substrate can be used by processors to communicate a finite set of values to each other. If the first processor may start the communication substrate i steps before the last does and the fastest processor is l times faster than the slowest, then once each processor has taken il steps, all processors have started the communication substrate.

Suppose each of k processors has a value to communicate, that the fastest processor is l times faster than the slowest, and that a processor takes j steps to execute the loop body of the communication protocol once. Once all processors have started the communication substrate, if any processor executes jl steps then all have executed at least j steps, so all processors have executed the loop body at least once. If all processors have executed the loop body at least once but no processor wrote to *shared_var* then each $local_p$, hence $priv_p$, was in *shared_var*. Therefore, *shared_var* = *common* and all values have been communicated. Thus, anytime any processor executes jl steps either a write instruction occurs or all values have been communicated.

By Lemma 10, there are no more than 2^k write instructions before some processor learns a private value. If every processor knows at least n values then, after 2^k writes, every processor knows at least $n+1$ values, since any processor knowing only n values was a most ignorant processor and learned a value. Thus, after $k2^k$ write instructions, all processors have learned all k private values. Once all processors have started the communication substrate, all values are communicated before any processor has executed $(jl)(k2^k)$ steps. Therefore, by the time any processor executes $jlk2^k + il$ steps, all values have been communicated. ∎

Lemma 12 can be used to implement *finish* for bounded-fair systems. Once a processor has executed $jlk2^k$ steps inside the communication substrate, *finish* returns true. Note that if the communication substrate is then used a second time, i will be at least $jlk2^k$. Each time the communication substrate is then executed, i will be the total number of steps executed previously. In general, executing the communication substrate the n'th time will require at least $\sum_{i=1}^{n} jlk2^k l^{i-1}$ steps.

4.3. Selection for Bounded-Fair Systems in S

The communication substrate is used in Algorithm 2, a distributed algorithm that allows each processor in a bounded-fair system in S to find its label under similarity labeling Θ. Algorithm 2 (page 31) replays the execution of Algorithm 1 that produced Θ. It works as follows.

- Each processor knows its initial state and that of neighboring variables, so it knows its and their labels under Φ_0. It also knows the sequences $(\Phi_0, \ldots, \Phi_{n\_s})$ and $(\alpha_1, \ldots, \alpha_{n\_s})$ from Algorithm 1, where n_s is the number of steps it took Algorithm 1 to calculate Θ.

- Each shared variable v contains a list of sets, where the j'th set represents $Env_{\Phi_j}(v)$, the environment of v under Φ_j. The j'th set is created by processors when they replay step j: each processor p uses the communication substrate to store in the j'th set in v a record x containing two fields: $x.name$, the name that p gives v, and $x.suspect$, $\Phi_j(p)$.

- Each processor *self* running Algorithm 2 keeps copies of its label under each Φ_j in local variable $pec[j]$ (processor equivalence class) and the label of each of its m-neighbors in $vec[m][j]$ (variable equivalence class).

- Each processor uses bounded fairness to wait until all processors have finished replaying step j of Algorithm 1 before replaying step $j+1$.

  - Each processor *self* replays the relabeling of a set of variables by computing Φ_{j+1} for each neighboring variable. It reads each n-neighbor to obtain $Env_{\Phi_j}(nbr(n, self))$ and then uses a function *v-next* to calculate $\Phi_{j+1}(nbr(n, self))$, which is then stored in $vec[n, j+1]$.
  - Each processor *self* replays the relabeling of a set of processors by calculating $Env_{\Phi_j}(self)$ from the labels of its neighbors, using a function *p-next* to calculate $\Phi_{j+1}(self)$, and adding that label to the $j+1$'st set in each of its neighboring shared variables, as well as storing it in $pec[n, j+1]$.

The communication substrate is implemented by *write_to_variable* and *communicate*.[3] *write_to_variable* writes the record corresponding to the name-label pair $(n, pec[j])$ to each n-neighbor and to $local[n][j]$, thus starting the

[3] Splitting the communication substrate into two pieces will allow simpler presentations of subsequent algorithms.

communication substrate.

$$
\begin{array}{l}
\mathit{write\_to\_variable}(j, \mathit{local}, \mathit{pec}) \equiv \\
\quad \textbf{for } n \in \mathit{NAMES}: \\
\quad\quad \mathit{val.suspects} := \mathit{pec}[j];\ \mathit{val.name} := n; \\
\quad\quad \mathit{local}[n][j] := \mathit{local}[n][j] \cup \{\mathit{val}\}; \\
\quad\quad \textbf{write } \mathit{local}[n] \textbf{ to } n \\
\quad \textbf{rof}
\end{array}
$$

The rest of the communications substrate is implemented by *communicate*, which ensures that the records in $local[n]$ are communicated to other processors sharing $nbr(n, self)$, for each name n. Since *communicate* is a straightforward generalization of the communication substrate presented in Section 4.2, it is not described here. Because Algorithm 2 runs on a bounded-fair system, Lemma 12 provides an implementation of *finish*. We define *p-next* and *v-next* (used in Algorithm 2) after defining the loop invariant.

The loop invariant of *S2.2* of Algorithm 2 for processor *self* has three parts. The first two parts assert that each local copy of *self*'s labels and of its neighbors' labels have the correct label for all steps up to j.

I2.1: $(\forall i, n : 0 \leq i \leq j, n \in NAMES : vec[n][i] = \Phi_i(nbr(n, self)))$
I2.2: $(\forall i : 0 \leq i \leq j : pec[i] = \Phi_i(self))$

The third part of the loop invariant states that for all i, $i \leq j$, whenever *self* reads a variable v, the set of records stored in the i'th set on the list in v is $Env_{\Phi_i}(v)$, the environment of v under Φ_i. In other words, for each processor p, an n-neighbor of variable v, the i'th set in the local copy of the list stored in v contains the record corresponding to the pair $(n, \Phi_j(p))$.

I2.3: $(\forall i, n : 0 \leq i \leq j, n \in NAMES :$
$\quad \{(x.name, x.suspect) : x \in local[n][i]\}$
$\quad = Env_{\Phi_i}(nbr(n, self)))$

The following lemma gives conditions under which *I2* can be shown to be the loop invariant for *S2.2* of Algorithm 2. These conditions lead to the definitions of *p-next* and *v-next*.

Lemma 13. *If v-next satisfies*

$$
\begin{array}{l}
I2 \wedge \textit{variables labeled } vec[n][j] \textit{ are relabeled in step } j+1 \\
\Rightarrow \Phi_{j+1}(nbr(n, self)) = v\text{-}next(local, vec, j, n).
\end{array} \tag{4.9}
$$

and p-next satisfies

$$
\begin{array}{l}
I2 \wedge \textit{processors labeled } pec[j] \textit{ are relabeled in step } j+1 \\
\Rightarrow \Phi_{j+1}(self) = p\text{-}next(vec, pec, j)
\end{array} \tag{4.10}
$$

then I2 is a loop invariant for S2.2 of Algorithm 2.

Algorithm 2. Find $\Theta(self)$ for processor *self* in a bounded-fair system in S.

Program variables (for processor self):

```
pec[j] ∈ PLABELS /* Φ_j(self) */;
     pec[0] is initially init(self) = Φ_0(self);
vec[n][j] ∈ VLABELS /* Φ_j(nbr(n, self))*/;
local[n] /* local copy of value of variable nbr(n, self)*/;
local[n][j] /* set of records of step j, where each record x is of the form
               x.suspect ∈ PLABELS
               x.name ∈ NAMES */
```

Program for processor self

```
       j := 0;
S2.1:  for n ∈ NAMES :
          read vec[n][j] from n;
       rof;
       write_to_variable(j, local, pec);
       communicate(local)
S2.2:  do j ≤ number of steps in Algorithm 1 →
          if step j + 1 of Algorithm 1 relabels a set of variables →
             for n ∈ NAMES :
                if variables labeled vec[n][j] were split in step j + 1 →
                      vec[n][j + 1] := v-next(local, vec, j, n)
                □ otherwise → vec[n][j + 1] := vec[n][j]
                fi
             rof;
             pec[j + 1] := pec[j]
          □ step j + 1 of Algorithm 1 relabels a set of processors →
             if processors labeled pec[j] were relabeled in step j + 1
                            → pec[j + 1] := p-next(vec, pec, j)
             □ otherwise → pec[j + 1] := pec[j]
             fi;
             for n ∈ NAMES :
                   vec[n][j + 1] := vec[n][j]
             rof
          fi;
          write_to_variable(j + 1, local, pec);
          communicate(local);
          j := j + 1;
       od
```

Proof: *I2.1* and *I2.2* are true initially because the first step of Algorithm 1 (*S1.1*) is to make $\Phi_0(x) = \mathit{init}(x)$ for any node x. To show that *I2.3* is initially true, we argue as follows. By *I2.2*, *S2.1* causes each processor p to write $(n, \Phi_0(p))$ to its n-neighbor using the communication substrate, so *I2.3* will hold once each processor successfully communicates all values to every other neighboring processor. Since Algorithm 2 runs on a bounded-fair system, by Lemma 12 there is an implementation of *finish*, so *communicate* waits until all values have been communicated. Thus, *I2* is true at the start of loop *S2.2*.

After execution of the **if** statement in *S2.2*, we want

$$\begin{aligned}&(\forall n : n \in \mathit{NAMES} : \mathit{vec}[n][j+1] = \Phi_{j+1}(\mathit{nbr}(n, \mathit{self}))) \\ &\wedge \mathit{pec}[j+1] = \Phi_{j+1}(\mathit{self})\end{aligned} \tag{4.11}$$

for *I2.1* and *I2.2* to hold once $j := j + 1$ is executed. The first alternative of the **if** establishes (4.11) if *v-next* satisfies (4.9), since it sets $\mathit{pec}[j+1]$ equal to $\mathit{pec}[j]$ which, by *I2.2*, is $\Phi_j(\mathit{self})$ and, since step $j+1$ does not relabel processors, is also $\Phi_{j+1}(\mathit{self})$. Likewise, the second alternative of the **if** meets this condition if *p-next* satisfies (4.10). Therefore, *I2.1* and *I2.2* are maintained when j is incremented if *v-next* satisfies (4.9) and *p-next* satisfies (4.10).

After the **if** statement completes, *write_to_variable* writes $(n, \Phi_{j+1}(\mathit{self}))$ (encoded as $x.\mathit{name} = n$ and $x.\mathit{suspect} = \mathit{pec}[j+1]$, which, by *I2.1*, equals $\Phi_{j+1}(\mathit{self})$) to each n-neighbor. By Lemma 12, there is an implementation of *finish*. Therefore, the communications substrate can wait until each processor writes its record to each neighboring shared variable. In particular, at the end of *communicate*, for each name n, each neighbor of $\mathit{nbr}(n, \mathit{self})$ has written its record for step $j+1$ to $\mathit{nbr}(n, \mathit{self})$ and so *communicate* will have caused $\mathit{local}[n][j+1]$ to contain a record for step $j+1$ written by each neighbor of $\mathit{nbr}(n, \mathit{self})$. Thus, at the end of *communicate*,

$$\begin{aligned}&\{(x.\mathit{name}, x.\mathit{suspect}) : x \in \mathit{local}[n][j+1]\} \\ &= \{(m, \alpha) : m \in \mathit{NAMES}, \alpha \in \mathit{PLABELS} : \\ &\qquad (\exists p : p \in P : \mathit{nbr}(n, \mathit{self}) = \mathit{nbr}(m, \mathit{self}) \wedge \Phi_{j+1}(p) = \alpha).\end{aligned}$$

By definition (4.5) of $\mathit{Env}_{\Phi_{j+1}}$,

$$\{(x.\mathit{name}, x.\mathit{suspect}) : x \in \mathit{local}[n][j+1]\} = \mathit{Env}_{\Phi_{j+1}}(\mathit{nbr}(n, \mathit{self}))).$$

Therefore, *I2.3* is maintained when j is incremented. Thus, if we choose *v-next* and *p-next* so that (4.9) and (4.10) are satisfied, then *I2* is a loop invariant for Algorithm 2. ∎

Since all nodes given a particular label by similarity labeling Θ have the same environment under any subsimilarity labeling, it makes sense to extend the definition of an environment to the environment of a label (instead of simply a node). Thus,

$$Env_{\Phi}(\alpha) \equiv Env_{\Phi}(x) \text{ for any node } x \text{ such that } \Theta(x) = \alpha.$$

In the same way, $\Phi_j(\gamma)$ means the label that Φ_j gives a node labeled γ by Θ.

In order to maintain *I2*, we must choose *p-next* and *v-next* so that (4.9) and (4.10) are satisfied. If step $j+1$ of Algorithm 1 relabels the set of nodes labeled γ by Φ_j then it gives a new label α_{j+1} to all nodes labeled γ by Φ_j with a different environment than γ. Suppose Algorithm 2 is replaying a step $j+1$ that relabels the set of variables labeled $vec[n][j]$ by Φ_j. If $nbr(n, self)$ has the same environment as $vec[n][j]$ then it should keep $vec[n][j]$ as its label, otherwise it should be given α_{j+1} as a new label. By *I2.3*, the environment of $nbr(n, self)$ under Φ_j is represented by $local[n][j]$, so we get the following definition of *v-next*.

Lemma 14. *If v-next is defined as*

$$\begin{array}{l} v\text{-}next(local, vec, j, n) \equiv \\ \quad \textbf{if } \{(x.name, x.suspect) : x \in local[n][j]\} = Env_{\Phi_j}(vec[n][j]) \\ \qquad\qquad\qquad \rightarrow \textbf{return}(vec[n][j]) \\ \quad \square \textbf{ otherwise } \rightarrow \textbf{return}(\alpha_{j+1}) \\ \quad \textbf{fi} \end{array} \tag{4.12}$$

then

$$\begin{array}{l} \textit{I2} \wedge \textit{variables labeled } vec[n][j] \textit{ are relabeled in step } j+1 \\ \Rightarrow \Phi_{j+1}(nbr(n, self)) = v\text{-}next(local, vec, j, n). \end{array}$$

Proof: Suppose

$$\textit{I2} \wedge \text{variables labeled } vec[n][j] \text{ by } \Phi_j \text{ are relabeled in step } j+1.$$

Since step $j+1$ of Algorithm 1 (i.e. *S1.2* with $i = j$) relabels variables labeled $vec[n][j]$ by Φ_j, only those variables with an environment different from $Env_{\Phi_j}(vec[n][j])$ are relabeled. Thus, according to Algorithm 1,

$$\Phi_{j+1}(v) = \begin{cases} \Phi_j(v) & \text{if } Env_{\Phi_j}(v) = Env_{\Phi_j}(\Phi_j(v)) \\ \alpha_{j+1} & \text{if } Env_{\Phi_j}(v) \neq Env_{\Phi_j}(\Phi_j(v)). \end{cases}$$

By *I2.3*,

$$Env_{\Phi_j}(nbr(n, self)) = \{(x.name, x.suspect) : x \in local[n][j]\}.$$

Therefore, the function

$$
\begin{array}{l}
\textbf{if } \{(x.name, x.suspect) : x \in local[n][j]\} \\
\quad = Env_{\Phi_j}(\Phi_j(nbr(n, self))) \\
\qquad\qquad \to \textbf{return}(\Phi_j(nbr(n, self))) \\
\square \textbf{ otherwise} \to \textbf{return}(\alpha_{j+1}) \\
\textbf{fi}
\end{array}
\tag{4.13}
$$

returns $\Phi_{j+1}(nbr(n, self))$. By *I2.1*, $vec[n][j] = \Phi_j(nbr(n, self))$. Thus, (4.13) is equal to the definition of *v-next* in (4.12), so *v-next*($local, vec, j, n$) returns $\Phi_{j+1}(nbr(n, self))$. Therefore, (4.9) is satisfied. ∎

In the same way, if *p-next* is evaluated during the replay of step $j+1$, then step $j+1$ of Algorithm 1 relabels processors labeled $pec[j]$ by Φ_j that have a different environment than that of processors labeled $pec[j]$ by Θ. Thus, if *self* has the same environment as $pec[j]$, it should keep $pec[j]$ as its label, otherwise it should be given α_{j+1}, a new label. The environment of *self* under Φ_j can be computed from *vec*, since (by *I2.1*) $\Phi_j(nbr(n, self)) = vec[n][j]$, so we get the following definition of *p-next*.

Lemma 15. *If p-next is defined as:*

$$
\begin{array}{l}
p\text{-}next(vec, pec, j) \equiv \\
\quad \textbf{if } (\forall n : n \in NAMES : \Phi_j(nbr(n, pec[j])) = vec[m][j]) \to \textbf{return}(pec[j]) \\
\quad \square \textbf{ otherwise} \to \textbf{return}(\alpha_{j+1}) \\
\quad \textbf{fi}
\end{array}
$$

then

$$
\begin{array}{l}
I2 \wedge \textit{processors labeled } pec[j] \textit{ are relabeled in step } j+1 \\
\Rightarrow \Phi_{j+1}(self) = p\text{-}next(vec, pec, j).
\end{array}
$$

Proof: Suppose

$$I2 \wedge \text{processors labeled } pec[j] \text{ are relabeled in step } j+1.$$

Since step $j+1$ of Algorithm 1 (i.e. *S1.2* with $i = j$) relabels processors labeled $pec[j]$ by Φ_j, only those processors with an environment different from $Env_{\Phi_j}(pec[j])$ are relabeled. Thus, according to Algorithm 1,

$$\Phi_{j+1}(self) = \begin{cases} \Phi_j(self) & \text{if } Env_{\Phi_j}(\Phi_j(self)) = Env_{\Phi_j}(self) \\ \alpha_{j+1} & \text{if } Env_{\Phi_j}(\Phi_j(self)) \neq Env_{\Phi_j}(self). \end{cases}$$

Note that definition (4.5) can be rewritten as

$$Env_\Psi(p) = \{(n, \Psi(nbr(n, p))) : n \in NAMES\},$$

so

$$Env_{\Phi_j}(\Phi_j(self)) = \{(n, \Phi_j(nbr(n, \Phi_j(self)))) : n \in NAMES\}$$

and

$$Env_{\Phi_j}(self) = \{(n, \Phi_j(nbr(n, self))) : n \in NAMES\}.$$

Since $Env_{\Phi_j}(self)$ and $Env_{\Phi_j}(\Phi(self))$ each have one element $(n, f(n))$ for each name n,

$$\begin{aligned} & Env_{\Phi_j}(\Phi_j(self)) = Env_{\Phi_j}(self) \\ & \iff (\forall n : n \in NAMES : \Phi_j(nbr(n, \Phi_j(self))) = \Phi_j(nbr(n, self))). \end{aligned}$$

Therefore, the function

$$\begin{aligned} & \textbf{if } (\forall n : n \in NAMES : \Phi_j(nbr(n, \Phi_j(self))) = \Phi_j(nbr(n, self))) \rightarrow \\ & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{return}(\Phi_j(self)) \\ & \Box\, \textbf{otherwise} \rightarrow \textbf{return}(\alpha_{j+1}) \\ & \textbf{fi} \end{aligned}$$

returns $\Phi_{j+1}(self)$. By *I2.1*, $(\forall n : n \in NAMES : vec[n][j] = \Phi_j(nbr(n, self)))$. By *I2.2*, $pec[j] = \Phi_j(self)$. Therefore, $p\text{-}next(vec, pec, j)$ returns $\Phi_{j+1}(self)$, so (4.10) is satisfied. ∎

To show interference-freedom for Algorithm 2, we must show that no processor can invalidate the proof of correctness of the program at another. The proof of correctness of Algorithm 2 for one processor, including the invariant, *I2*, does not depend on private variables of other processors or on shared variables, so Algorithm 2 is trivially interference-free. It might seem odd that a distributed program using shared variables should have such a trivial proof of freedom from interference, but the communication substrate completely encapsulates synchronization among processes and so allows Algorithm 2 to be trivially interference-free.

Lemma 16. *Algorithm 2 allows each processor p in a bounded-fair system Σ to learn $\Theta(p)$, where Θ is the labeling calculated by Algorithm 1.*

Proof: By Lemmas 13, 14, and 15, *I2* is a loop invariant of Algorithm 2. Thus, if Algorithm 2 terminates then each processor p in Σ learns $\Theta(p)$. *S2.2* iterates once for each step in Algorithm 1, so it is bounded by the size of Σ. The only part of *S2.2* whose running time is not obviously bounded is *communicate*. Since Σ is bounded-fair, Lemma 12 implies that *communicate* terminates, thus Algorithm 2 terminates and each processor p in Σ learns $\Theta(p)$. ∎

Note that we have not yet proved that Algorithm 2 lets each processor calculate its label under the similarity labeling for the system because we have not yet proven that Θ is a similarity labeling. The next theorem completes the proof.

Theorem 17. *Algorithm 1 calculates a similarity labeling Θ for any bounded-fair system Σ in S,*

Proof: Since Algorithm 1 gives nodes with different environments under any Φ_i different labels, and since (by Lemma 16) Algorithm 2 lets each node replay Algorithm 1, nodes with different environments are dissimilar. Thus, similar nodes will have the same environments. So, by Theorems 7 and 8, Algorithm 1 calculates a similarity labeling Θ for Σ. ∎

Algorithm 2 can be used as a basis for a selection algorithm when there is a processor with no other processor similar to it.

Theorem 18. *Selection in Non-locking Systems.*

There is a selection algorithm (which turns out to be Algorithm 2) for a bounded-fair system in S if and only if there is a processor p such that no other processor is similar to p.

Proof:
if:

If there is a processor p such that no processor is similar to p then, by Theorem 17, each processor can use Algorithm 2 to determine its label and the processor labeled $\Theta(p)$ by Θ can be selected. Exactly one processor, p, has this label so exactly one processor will be selected. Thus, there is a selection algorithm.

only if:

Suppose that every processor has some other that is similar to it. Thus, a similarity labeling would give no processor a unique label. By Theorem 17, Θ is a similarity labeling, thus it is a supersimilarity labeling. So, by Theorem 4, there is no selection algorithm. ∎

The selection algorithm of Theorem 18 always selects the same processor. However, the algorithm can be converted into one that selects any of a set of processors, depending on the schedule. Instead of selecting a processor after running Algorithm 2, the distinguished processor becomes an *elector*. The elector then uses some measure based on the schedule, such as the number of steps it took to run Algorithm 2, to pick a label to be selected and sends that label to all processors. When a processor receives the label that it computed when it executed Algorithm 2, it selects itself. Of course, there must be only one processor with that label.

This new version of the algorithm is not very satisfying, because it still relies on a single processor. Another possible solution might be for each processor to use a value resulting from some execution as its "initial state" for the purposes of Algorithm 2. Different schedules can produce different values, so there is a set of resulting systems, one for each value. It might be possible for a single program to be a selection algorithm for all the resulting systems. The next chapter discusses how to find a single program that can be a selection algorithm for any of a set of systems.

Chapter 5

Families of Systems

The selection algorithm in Chapter 4 is designed for a particular system. It will be useful to have selection algorithms that work for a set of systems. A *family* of systems $\mathcal{F}$ is a set of systems, each with the same instruction set and the same scheduling policy, and with interconnection networks that use the same set of variable names. Only the network topology and the initial states of the systems in a family differ. Since all systems in the family have the same instruction set and the same set of variable names, a program for one system in the family will be a program for any other system in the family. A program is a selection algorithm for a family of systems if it is a selection algorithm for each system in the family. The *generic selection problem* for a family $\mathcal{F}$ is to decide whether there is a selection algorithm for $\mathcal{F}$.

The generic selection problem for families of bounded fair systems in S can be solved using the similarity relation. Unfortunately, the similarity relation has been defined only for nodes in a single system. In order to relate nodes in different systems, we define *disjoint-union* systems as follows: given two systems $\Sigma = (N_\Sigma, init_\Sigma, I, SP)$ and $\Pi = (N_\Pi, init_\Pi, I, SP)$ where interconnection networks N_Σ and N_Π use the same set of variable names, the disjoint-union system $\Sigma \cup \Pi = (N_\Sigma \cup N_\Pi, init_{\Sigma\cup\Pi}, I, SP)$. For each node z in $\Sigma \cup \Pi$,

$$init_{\Sigma\cup\Pi}(z) = \begin{cases} init_\Sigma(z) \text{ if } z \in N_\Sigma \\ init_\Pi(z) \text{ if } z \in N_\Pi. \end{cases}$$

Since Σ and Π are disjoint, a processor has the same behavior in the disjoint-union system as in the original system, so we use disjoint-union systems to extend similarity to relate nodes from different systems. Two nodes from different systems are similar if and only if they are similar in their disjoint-union system. A similarity labeling Θ for a disjoint-union system can be computed using Algorithm 1, since we made no assumptions in Algorithm 1 about connectivity of the communication network. Algorithm 2 lets each processor in a bounded-fair disjoint-union system in S learn its similarity label, because assumptions about connectivity were not made there, either.

We now show that the generic selection problem for families of bounded-fair systems in S is just like the selection problem for bounded-fair systems in S.

Theorem 19. *Assume that each process in a system in a family $\mathcal{F}$ can learn its label under a similarity labeling Θ. Then, there is a generic selection algorithm for $\mathcal{F}$ if and only if there is a set of processor similarity labels ELITE such that each member of $\mathcal{F}$ has exactly one processor with a label under Θ in ELITE.*

Proof:
if:

Assume that each member of $\mathcal{F}$ has exactly one processor with a label under Θ in *ELITE*. Since each processor p can learn $\Theta(p)$, the generic selection algorithm is one in which each processor p learns $\Theta(p)$ and then selects itself if $\Theta(p) \in ELITE$.
only if:

Assume that there is a generic selection algorithm for $\mathcal{F}$. Since similarity labeling Θ is a supersimilarity labeling, there is some schedule φ that makes any set of processors given the same label by Θ behave similarly in the disjoint-union system for $\mathcal{F}$. Let *ELITE* be the set of labels under Θ of the processors in systems in $\mathcal{F}$ selected by the generic selection algorithm when φ is executed. A generic selection algorithm, when run on the disjoint-union system for $\mathcal{F}$, must select exactly one processor from each system in $\mathcal{F}$ for any schedule, including φ. If $\Theta(p) = \Theta(p)$ then any algorithm that selects p when executing schedule φ will also select q. Thus, each member of $\mathcal{F}$ has exactly one processor with a label under Θ in *ELITE*. ∎

Since Algorithm 2 lets processors in families of bounded-fair systems in S learn their label under Θ, there is a generic selection algorithm for a family $\mathcal{F}$ of bounded-fair systems in S, if and only if there is a set of processor similarity labels *ELITE* such that each member of $\mathcal{F}$ has exactly one processor with a label under Θ in *ELITE*.

We can now solve the problem posed at the end of Chapter 4—that of finding a selection algorithm that does not depend on any particular processor, but whose choice of processor depends on the schedule. Suppose that a bounded-fair system Σ in S has a selection algorithm. Then, there is a selection algorithm whose choice of processor depends on the schedule, as follows. Each variable is treated as if it had an extra one-bit field, r. Since we have assumed that variables can have an arbitrary number of states, this has no effect on Σ unless r was being used to represent the initial state. The number of different initial states of variables is no more than the number of variables, so some bit can always be chosen that is not needed to represent the initial state. Each processor inverts the r field of each n-neighbor by reading the variable, complementing the r field, and then writing the resulting value back to the variable. Since many processors will be doing this at once, processors can interfere with each other. Thus, it is not possible to determine the final value of each variable, except that a variable's final value will be the same as its original value but for its r field, which will be either 0 or 1. When all processors are finished inverting the r fields of their neighbors, the system will be in one of a set of states $FINAL$. The original system has been converted to a system in a family

$$\mathcal{F} = \{(N, \mathit{final}, S, BF) : \mathit{final} \in FINAL\}.$$

Since variables can have the same final state only if they had the same initial state, any supersimilarity labeling for a system in $\mathcal{F}$ will be a supersimilarity labeling for $(N, \mathit{init}, S, BF)$. Since the similarity labeling for the initial system gives some processor a unique label, there will be a set of processor similarity labels $ELITE$ such that each member of $FINAL$ has exactly one processor with a label under Θ in $ELITE$. Thus, there is a generic selection algorithm for $FINAL$. If $ELITE$ is chosen so that some labels are given to different processors (which is not always possible) then the processor that is selected depends on the schedule.

Families are commonly derived from a single system. In Chapter 6, we will see how a fair system in S is like a family of bounded-fair systems in S. In Chapter 7 we will see how a system in L is like a family of systems in S.

Chapter 6

Selection for Simple Fair Systems

A fair system Σ in S is like a family of bounded-fair systems in S. This is because during the time some processor p has not yet executed–due to the use of a fair but not bounded-fair schedule–Σ is equivalent to a subsystem of Σ that does not include p. Since any subset of the processors can be delayed, a fair system Σ is like the family made up of the subsystems of Σ. The behavior of a processor in such a system will depend on the set of processors that are delayed, though it will be the same as the behavior of a processor in one of the systems in the family.

Selection in fair systems is more difficult than in bounded-fair systems. Even if a processor q in Σ is given a unique label α by the similarity labeling of Σ, there might be a subsystem of Σ in which two or more processors are given α by that subsystem's similarity labeling. If there were such a subsystem, then any algorithm that selects q could select all the nodes that are similar to q in that subsystem. Then, if some of the processors in Σ were delayed due to the weak guarantee implied by fairness, two or more processors in Σ would be selected.

We can formalize the problem of processes assuming different identities with the *mimics* relation. Given a system $\Sigma = (N, init, S, F)$, let π be an isomorphism from elements of N to elements of a disjoint network N' that preserves *init* and *naming*. For any $M \subseteq N$, let $\Pi(M) = (\pi(M), init, S, F)$. Since π preserves *init* and *naming*, *init* and *naming* are both defined for $\Pi(M)$, so $\Pi(M)$ is a system. Since N and $\pi(N)$ are disjoint, we can form the disjoint-union of Σ and $\Pi(M)$, (which we could not do for Σ and $(M, init, S, F)$ because M is a subsystem of N). Let x and y both be processors or both be variables in $(N, init, S, F)$. We say that

x mimics y if there is a subset M of N (containing both x and y) such that in the disjoint-union system $\Sigma \cup \Pi(M)$, x, $\pi(x)$ and $\pi(y)$ can behave similarly.

If $M = N$ then x and $\pi(x)$ are isomorphic, so $\pi(x) \sim \pi(y) \Rightarrow x \sim \pi(y)$. Therefore, if $x \sim y$ then x mimics y. The mimics relation can also be defined for labels of nodes. If Θ is the labeling produced by Algorithm 1 for a system containing components x and y then $\Theta(x)$ mimics $\Theta(y)$ if and only if x mimics y.

The following lemma shows that if x mimics y then any algorithm that ensures that x enters a particular state can also cause y to be in that state at the same time. There might be some algorithm for which there is some schedule that causes x to be in a state that y can never enter, but no algorithm can cause x to enter that state for every schedule.

Lemma 20. *For any system $\Sigma = (N, init, S, F)$, any program, and any state c, if x mimics y then*

$$(\forall \varphi\ \exists i : \varphi \text{ a schedule}, i \text{ an integer} : init \circ \varphi|_i(x) = c) \Rightarrow$$
$$(\exists \varphi, i : \varphi \text{ a schedule}, i \text{ an integer} : init \circ \varphi|_i(x) = init \circ \varphi|_i(y) = c).$$

Proof: Since x mimics y, there is a network $M \subseteq N$ such that x, $\pi(x)$, and $\pi(y)$ can all behave similarly in the disjoint union system $\Sigma \cup \Pi(M)$. By definition of behave similarly, there is a schedule φ for the union system $\Sigma \cup \Pi(M)$ that, for every program, causes x, $\pi(y)$, and $\pi(x)$ to all have the same state infinitely often. For any program $\mathcal{P}$, let $\mathcal{P}'$ be a version of $\mathcal{P}$ modified so that processors never exit state c once they enter it. Since φ will cause x, $\pi(x)$ and $\pi(y)$ to behave similarly for any program, including $\mathcal{P}'$, if x enters state c while executing $\mathcal{P}'$ then $\pi(x)$ and $\pi(y)$ will enter that state, too.

For any schedule φ, let φ' be derived from φ by omitting all steps in which a processor in state c executes. Since processors running $\mathcal{P}'$ in state c do not change their state, the sequence of system states during execution of φ with $\mathcal{P}'$ is equivalent to the sequence of system states during execution of φ' with $\mathcal{P}$ except that some system states might be omitted if they are equal to their predecessor in the sequence. So, for any program $\mathcal{P}$, there is a schedule φ' that causes x, $\pi(x)$, and $\pi(y)$ to simultaneously have state c. Thus, there is some i such that

$$init \circ \varphi'|_i(x) = init \circ \varphi'|_i(\pi(x)) = init \circ \varphi'|_i(\pi(y)) = c.$$

By construction, the elements of N are not connected to elements of $\pi(M)$, so the states of processors in N cannot be affected by steps involving processors in $\pi(M)$. A schedule ϱ such that $init \circ \varrho|_j(x) = init \circ \varrho|_j(y) = c$ for some j is found by deleting those steps in φ involving processors p not in $\pi(M)$ and then replacing each remaining process p with $\pi^{-1}(p)$. The prefix of ϱ up to step j is turned into a fair schedule by appending a fair schedule to it. Thus, if there is an algorithm that guarantees that x eventually is in a particular state, then there is a fair schedule (with a prefix that may only use a subset of the processors) that causes x and y to be in the same state simultaneously. ∎

Note that if there is a schedule φ that does not force x to enter a particular state, then the above lemma does not guarentee that y will enter that state, so if x is caused to enter a state by only some schedules, instead of by all schedules, then y might not necessarily enter that state.

Although the similarity relation is symmetric, the mimics relation is not. If x mimics y then it is possible that y can tell it is different from x even though x can never tell it is different from y. In contrast, similar nodes can never distinguish themselves from each other.

Even though mimic and similarity are different, mimic can be used to tell when no selection algorithm exists. The following theorem does this; it is analogous to Theorem 3.

Theorem 21. *No selection algorithm exists for a fair system in S if*

$$(\forall p \exists q : p, q \in P : (p \neq q) \wedge (p \text{ mimics } q)).$$

Proof: A selection algorithm causes some processor to enter a state that no other processor is in or will ever be in. Lemma 20 shows that if every schedule causes a processor p to be in a selected state then there is a schedule that causes both p and some processor q (where p mimics q) to be in that state. Therefore, Uniqueness is violated and there is no selection algorithm. ∎

Lemma 20 (hence, Theorem 21) does not hold for bounded-fair systems because ϱ in the proof of Lemma 20 is not bounded fair. A processor in a bounded-fair system can know how long to wait until all neighbors of a variable have had time to write to it and therefore it can determine the absence of a neighbor. Thus, in a bounded-fair system, nodes x and y can be differentiated even if x mimics y. Processors in a bounded-fair system mimic each other only if they are similar.

Theorem 21 implies that there are fair systems for which no program can cause each processor to know its label under similarity labeling Θ. This is because if p mimics q, then p can never learn that its label under Θ differs from $\Theta(q)$, or else it would be possible to solve the selection problem, contradicting

Theorem 21. Since no algorithm can always let each processor learn its label under Θ, Algorithm 2 does not work for fair systems. However, in the next section we show an algorithm for fair systems that, like Algorithm 2, can be used as a selection algorithm.

6.1. The Algorithm

Algorithm 2 used bounded-fairness to implement *finish*. Although there is no implementation of *finish* for a fair system, we can devise a program that replays Algorithm 1. In the new algorithm, as in Algorithm 2, a processor replaying step j stores its own label under Φ_j in each neighboring variable, then reads the collection of labels that have been stored in each variable and uses them to compute that variable's environment. Without bounded-fairness, a processor cannot be sure when all processors neighboring the variable have stored their own label under Φ_i in the variable. Thus, a processor cannot be sure when each variable contains its own environment under Φ_i for any $i < j$. If a processor reads a variable too early then the processor will read only a subset of the labels that eventually will be stored in that variable.

If a processor can detect that the set read from a variable is not a valid environment in the given system then the processor can wait until more processors contribute labels to the set. Unfortunately, due to Lemma 20, if variable v mimics u then any set certain to be stored in v might also be stored in u, so u can contain a set of labels representing the environment of v. Thus, a variable can sometimes contain a set that, though a valid environment, is not the correct environment for that variable. Moreover, if processor p mimics q then it is possible that each n-neighbor of q has a set stored in it that is the same as the environment of the n-neighbor of p. In this case, some processors have not yet executed, so q should wait. However, since p and q have the same program, this means that p has to wait if each of its n-neighbors have their own environments stored in them. Clearly, p will wait forever before it sees any changes in these environments.

We are on the horns of a dilemma: if p mimics q then either q might decide incorrectly that it is p, or p might wait forever before deciding on its label. We choose the first alternative and let processors make mistakes temporarily. Thus, a processor will not necessarily compute its label under Φ_j, but will only compute an *estimate*. We will see that p's estimate of $\Phi_j(p)$ becomes more accurate as j increases, and we later prove that when j is large enough, the estimate mimics $\Phi_j(p)$. When a processor detects that its estimate is mistaken, it *backtracks* to the earliest step that it can tell is erroneous (in the case of q, step j) and *retries* that step. Retrying a step could invalidate the estimates of other processors, causing them to also backtrack. Thus, we need to define:

1. How backtracking to earlier steps is implemented.

2. How processors detect that an estimate at an earlier step was wrong.

To implement backtracking, values stored in shared variables include *version numbers.* When a processor retries an earlier step, it recalculates the values it stored in shared variables, increments the version numbers attached to these values, and stores the values with a new version number back in the shared variables. Version numbers make it possible for processors to distinguish between outdated and current values of shared variables, since only the value with the largest version number is current. Other than the addition of version numbers, data in shared variables is stored as it was in Algorithm 2. Hence, each variable contains a list of sets of records, where the i'th set represents the environment of the variable under Φ_i. In that set, each record x written by processor p to variable v contains two fields: $x.name$, the name that p gives v, and $x.suspect$, processor p's most recent estimate of $\Phi_i(p)$, where i is the number of the set that p is replaying when it writes x. When a processor retries step i, it shortens the list of sets in a variable to length i, since the data for the later steps are invalidated when the earlier step is retried. Version numbers are independent of step numbers, since the value of a variable consists of a list of sets, one for each step that its neighbors have replayed, but contains only one version number.

A processor p detects that its estimate of its label under some Φ_i is mistaken by reading neighboring shared variables. If any of the values upon which p made its decision about the value of $\Phi_i(p)$ have been changed, then p has to backtrack and retry the decision.

The only values that can be changed by other processors are shared variables. There are two ways that a set of records in a shared variable can change: it can get larger when a new record is added or it can be changed arbitrarily in size or disappear when some neighboring processor retries a step. A processor retrying a step increases the version numbers of surrounding variables, so replacing a value increases the associated version number. Thus, p has to backtrack after estimating Φ_i only when

1. the version number of a variable increases (because some other processor was replacing an erroneous record) or

2. the size of the i'th element of a shared variable increases (because a slow processor added a (probably) correct record).

Algorithm 3. Find $\Theta(self)$ for processor *self* in a fair system Σ in S.

Program variables (for processor self):

```
pec[j] ∈ PLABELS /* Φ_j(self) */;
     pec[0] is initially init(self) = Φ_0(self);
vec[m][j] ∈ VLABELS /* Φ_j(nbr(m, self))*/;
local[m] /* local copy of value of variable nbr(m, self)*/;
local[m].version /* version number of value from variable nbr(m, self)*/;
local[m][j] /* set of records from step j, where each record x is of the form
            x.suspect ∈ PLABELS
            x.name ∈ NAMES */;
```

Program for processor self

```
       j := 0;
S3.1:  for m ∈ NAMES :
          read vec[m][j] from m;
       rof;
       write_to_variable(j, local, pec);
       prepare_for_step(j, local);
S3.2:  do j ≤ number of steps in Algorithm 1 →
          if step j + 1 of Algorithm 1 relabels a set of variables →
             for m ∈ NAMES :
                if variables labeled vec[m][j] were split in step j + 1
                               → vec[m][j + 1] := v-next(local, vec, j, m)
                □ otherwise  → vec[m][j + 1] := vec[m][j]
                fi
             rof;
             pec[j + 1] := pec[j]
          □ step j + 1 of Algorithm 1 relabels a set of processors →
             if processors labeled pec[j] were relabeled in step j + 1
                            → pec[j + 1] := p-next(vec, pec, j)
             □ otherwise  → pec[j + 1] := pec[j]
             fi;
             for m ∈ NAMES :
                  vec[m][j + 1] := vec[m][j]
             rof
          fi;
          write_to_variable(j + 1, local, pec);
          prepare_for_step(j + 1, local);
          j := j + 1
       od
```

When a processor backtracks because a version number has increased, we say that a backtrack has *propagated.* When a processor backtracks because the size of the i'th element of a shared variable increases, we say that a backtrack has been *initiated.* The initiation of a backtrack occurs when a processor observes the effects of a slower processor, so there are never more backtracks to a particular step initiated at a variable than there are neighbors. However, each initiation of a backtrack can result in many propagations.

We can now describe Algorithm 3, a program that lets each processor p in a fair system compute a label that mimics $\Theta(p)$, and eventually compute $\Theta(p)$. As in Algorithm 2, for any $i \leq j$, local variable *pec*$[i]$ contains the processor's estimate of its label under Φ_i, *vec*$[m][i]$ contains its estimate of its m-neighbor's label under Φ_i, and *local*$[m][i]$, the most recent version of the value of the processor's m-neighbor, contains the processor's estimate of the environment of its m-neighbor under Φ_i. The main body of Algorithm 3 (page 46) is almost the same as that of Algorithm 2 (page 31). Here, the functions of the communication substrate are performed by *prepare_for_step* and *write_to_variable.* In addition, *prepare_for_step* deals with the complications caused by fairness without bounded-fairness, such as retrying steps, and reestablishes *I3*, the loop invariant of *S3.2*, before permitting a loop iteration to terminate.

The outermost loop of *prepare_for_step* (page 48) repeats until there is a processor label γ such that *self* has the same environment under Φ_j as does γ and each n-neighbor has the exact same environment under Φ_j as does $nbr(n, \gamma)$. When this is true, *self* can replay step j of Algorithm 1. While this is false, *self* executes the body of *prepare_for_step*, *i.e.* reads all its neighboring variables, checks to see if it needs to backtrack, and runs the communication substrate. It does this in two parts.

The first part of the body of *prepare_for_step* (*L1*) reads each neighboring variable and compares the value read with its local copy. The variable k is set to the largest number such that the value read for all steps before k is the same as that read previously. Thus, *self* will not have to backtrack to any step before step k.

The second part of the body of *prepare_for_step* acts on the results of the comparison of the current value of the variable and its previous value. The four guards of the **if** statement correspond to four possible results of the comparison. If the version number of a variable just read is too small then the value of the variable is obsolete, so *G1* replaces the variable with a current value. This might happen if some processor was delayed and wrote over a value with a larger version number. If the version number of a variable is too large then the value used by *self* is obsolete, so *G2* backtracks and propagates the retry of step $j+1$, the first obsolete step. If the version number is correct, *temp* is the local value read from a shared variable.

```
prepare_for_step(j, local, pec) ≡
  do ¬(∃γ, ∀n,i : γ ∈ PLABELS, n ∈ NAMES, i ≤ j :
                    pec[i] = Φ_i(γ)
                    ∧ vec[n][i] = Φ_i(nbr(n,γ))
                    ∧ {(x.name, x.suspect) : x ∈ local[n][i]} = Env_{Φ_i}(nbr(n,γ)))
    →
    for n ∈ NAMES:
L1:   read temp from nbr(n, self);
      if (∀i : i ≤ j : local[n][i] = temp[i]) → k := j
      □ otherwise → k := min({i : i ≤ j : local[n][i] ≠ temp[i]})
      fi
      {(∀i : i < k : local[n][i] = temp[i])}
G1:   if temp.version < local[n].version → write local[n] to nbr(n, self)
G2:   □ temp.version > local[n].version →
          local[n] := temp;
          j := k;
          backtrack(j + 1)      /* backtrack to step j + 1 */
G3:   □ temp.version = local[n].version
         ∧ k < j ∧ local[n][k] ≠ local[n][k] ∪ temp[k] →
          local[n][k] := local[n][k] ∪ temp[k];
          j := k;
          backtrack(j + 1)      /* backtrack to step j + 1 */
G4:   □ temp.version = local[n].version
         ∧ (k = j ∨ local[n][k] = local[n][k] ∪ temp[k]) →
          local[n][k] := local[n][k] ∪ temp[k];
          if temp[k] ≠ local[n][k] → write local[n] to nbr(n, self)
          □ otherwise → skip
          fi
      fi
    rof
  od;

  backtrack(j) ≡
    for n ∈ NAMES :
      if length(local[n]) ≤ j → skip
      □ length(local[n]) > j → shorten local[n] to j steps;
                               local[n].version := local[n].version + 1
      fi
      write_to_variable(j, local, vec)
    rof
```

but the set representing an environment that *self* has already used (in step k) contains a value that *local* does not contain, then a slow processor has belatedly added its record to the environment since *self* last read it, so *G3* backtracks to initiate a retry of the step in which that environment was used. *G4* implements the communication substrate, i.e. it ensures that the set of values stored in a shared variable contains every value known and busy-waits.

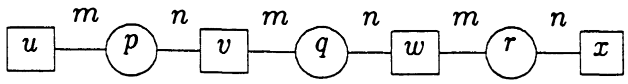
Although there are a number of possible error conditions for which *prepare_for_step* does not explicitly check, it still detects and corrects these conditions. For example, suppose a very slow processor writes on $nbr(n, self)$, causing j at *self* to be larger than $length(temp)$. If the version number written by the slow processor is less than $local[n].version$ then *G1* restores the value of $nbr(n, self)$; if one of the sets in the list is too small or if the list is too short then *G4* restores the value of $nbr(n, self)$. As another example, suppose that some processor neighboring $nbr(n, self)$ propagates a backtrack at the same time that *self* does, and both processors choose the same new version number. When *self* reads its n-neighbor, some element of *temp* will probably differ from the corresponding element in *local*, so *G3* will cause *self* to propagate the backtrack. If no element of *temp* differs from the corresponding element in *local* then the two processes agree on their n-neighbors environment and are in the same state as they would have been if one of them had propagated the backtrack to the other.

6.2. Loop Invariant

The label that a processor calculates at step j for itself or a neighboring variable reflects information that the processor has learned about the region of distance j around that node. In Algorithm 2, $pec[j]$ is $\Phi_j(self)$, but in Algorithm 3, $pec[j]$ is only *self*'s estimate for $\Phi_j(self)$. In Algorithm 3, $pec[j]$ mimics $\Phi_j(self)$ in the region of distance j around *self*, where a *region* is defined as follows. A *mirage by* μ of a system $(N, init, S, F)$ is a system $(T, init, S, F)$, where T and N are disjoint and μ is a homomorphism[1] from T to N that preserves *init* and *naming*. Note that $\mu(T)$ might be only a subset of N. If μ and *init* are implicit then we say T is a mirage of N. Figure 6.1 shows a system Σ and a mirage by μ of Σ, where $\mu(a_i) = a$ for every node a_i in the mirage.

We use mirages to describe the view of each processor running Algorithm 3 in a fair system. The label that processor *self* in a system Σ calculates for $\Phi_j(self)$ is always the label of some processor p in a mirage T of Σ by μ such

[1] A homomorphism from one graph to another is a function on its nodes that preserves edges and labelings.



System Σ

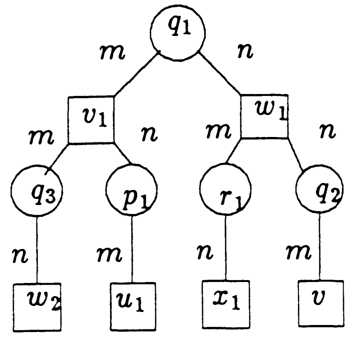


Figure 6.1: A Mirage

that *self* $= \mu(p)$ and T is a tree. Such tree mirages are defined recursively as follows:

- A *tree mirage for a variable v by μ* of depth 1 is a mirage of Σ consisting of a variable r such that $v = \mu(r)$.

- A *tree mirage for a processor p by μ* of depth i is a mirage of Σ consisting of a root r such that $p = \mu(r)$ and, for each neighbor of p, a subtree with an edge to r. The subtree corresponding to $nbr(n,p)$ is a mirage for $nbr(n,p)$ of depth $i-1$. The edge from r to the subtree corresponding to $nbr(n,p)$ is labeled n.

- A *tree mirage for a variable v by μ* of depth i, $i > 1$, is a mirage of Σ consisting of a root r such that $v = \mu(r)$ and a set of subtrees with edges to r. For each descendant s of r, $\mu(s)$ is a neighbor of v and s is the root of a tree mirage for $\mu(s)$ of depth $i-1$ with the subtree corresponding to v ($\mu(r)$) deleted. This subtree for v is deleted to ensure that each process has a unique n-neighbor, and therefore the tree is a system. The edge from r to s is labeled $naming(e)$, where e is the edge from v to $\mu(s)$.

Figure 6.1 shows a system Σ and a tree mirage for processor q in Σ of depth 4. Tree mirages for variables are of odd depth while tree mirages for processors are of even depth.

A processor in Σ replaying step i of Algorithm 1 computes labels for a tree mirage of depth $2i+2$, not labels for Σ. Thus, when processor *self* finishes replaying step i and writes $\Phi_i(self)$ to shared variables, that label is based on the mirage of Σ of depth $2i+2$ for *self*. A processor reads variables that contain labels under Φ_i for a mirage of depth $2i+2$ before replaying step

A Version of Algorithm 3 Annotated with Auxiliary Variables

Program variables (for processor self):

$pec[j] \in PLABELS$ /* $\Phi_j(self)$ */;
    $pec[0]$ is initially $init(self) = \Phi_0(self)$;
$vec[m][j] \in VLABELS$ /* $\Phi_j(nbr(m, self))$*/;
$local[m]$ /* local copy of value of variable $nbr(m, self)$*/;
$local[m].version$ /* version number of value from variable $nbr(m, self)$*/;
$local[m][j]$ /* set of records of step j, where each record x is of the form
            $x.suspect \in PLABELS$
            $x.name \in NAMES$
            $x.tree$ is a mirage of depth $2j + 2$ */
$tree[j]$ /* auxiliary variable containing mirage of *self* */
    $tree[0]$ is initially the mirage of *self* of depth 2;

Program for processor self

```
        j := 0;
S3.1:   for m ∈ NAMES :
           read vec[m][j] from m
        rof;
        /* When x is written to the 0'th element of each nbr(n, self),
           write_to_variable sets x.tree = tree[0] */
        write_to_variable(j, local, pec);
        prepare_for_step(j, local);
S3.2:   do j ≤ number of steps in Algorithm 1 →

            /* Replay next step of Algorithm 1, performed by */
              first if statement in S3.2 on page 46. */

            tree[j + 1] := new_tree(pec, vec, local, j + 1);
            /* When x is written to the i'th element of each nbr(n, self),
               write_to_variable sets x.tree = tree[i] */
            write_to_variable(j + 1, local, pec);
            prepare_for_step(j + 1, local);
            j := j + 1
        od
```

$i+1$, so we can use the mirage of depth $2i+2$ associated with those labels to define the mirages of depth $2i+4$ representing the information the processor obtains for the next step of Algorithm 1.

We can construct the mirage that a processor views by adding to Algorithm 3 an auxiliary variable *tree* that is an array of tree mirages for *self*; *tree*$[i]$ is the mirage representing *self*'s view of Σ when *self* replays step i of Algorithm 1. Each value x written into a shared variable v by any processor p during step j will contain an auxiliary field $x.tree$ which is equal to *tree*$[j]$ of p. This auxiliary field is in addition to the fields $x.name$ (the name that p gives v) and $x.suspect$ (the value of *pec*$[j]$ of p) defined in Algorithm 2. Thus, when *self* reads a neighboring variable it also learns the tree mirages of the processors that neighbor that variable and can use these mirages to construct its own tree mirage. The auxiliary variable and fields are only needed for the proof of correctness of Algorithm 3 and are not actually used to compute the label of a processor. They are included in a revised version of Algorithm 3 on page 51 and in a revised version of *write_to_variable* on page 71. We do not describe exactly how tree mirages are represented; any of the standard representations of trees or graphs will suffice.

The function *new_tree* in the version of Algorithm 3 on page 51 constructs a processor's tree mirage for step i, *tree*$[i]$, from the tree mirages for step $i-1$ that are stored in neighboring variables. Note that the tree mirage is a system such that execution of Algorithm 3 on the tree mirage after i steps could cause the root of the tree to have the same state that execution of Algorithm 3 on Σ could cause *self* to have. The children of *tree*$[i]$ are roots of mirages for neighboring variables, while the grandchildren are roots of mirages of processors that neighbor the neighboring variables. A more precise definition of *new_tree* (modulo the fact that we do not precisely define trees) is:

> $new\_tree(pec, vec, local, i) \equiv$ a tree with:
>   a root r such that $init(r) = init(self)$,
>   for each name n,
> r has an n-neighbor v, where $init(v) = init(nbr(n, self))$,
>   for each x in $local[n][i-1]$,
> v has the root of $x.tree$ as an $x.name$-neighbor.

Due to *write_to_variable*, (as required in the comments in *S3.2* on page 51) whenever *self* stores a record x as part of the j'th element of a shared variable, $x.tree = tree[j]$. Because *tree*$[i]$ contains a root processor and variables as well as the various $x.trees$, *tree*$[i]$ is of depth 2 more than *tree*$[i-1]$. Since *tree*$[0]$ is of depth 2, *tree*$[i]$ is of depth $2i+2$.

Since *tree*$[i]$ is a system, Algorithm 1 can compute a labeling Θ for *tree*$[j]$ and thus a sequence of Φ_i's. Unfortunately, this sequence of Φ_i's will not necessarily be related to the sequence produced for Σ. However, we can use Algorithm 1 to compute a sequence of Φ_i's for both $tree[j]$ and Σ such that $\Phi_j(self) = \Phi_j(root(tree[j]))$, as we show next. This will allow us to assert that

$$pec[j] = \Phi_j(r) \text{ for } r \text{ the root of } tree[j]$$

is part of the loop invariant for Algorithm 3. We will later show that when j gets large enough, $\Phi_j(self) = \Phi_j(root(tree[j]))$.

Let Algorithm 1 compute the similarity labeling of the disjoint-union system $\Sigma \cup tree[j]$ and let each x that *S1.2* of Algorithm 1 picks to partition sets of nodes be chosen from Σ (*i.e.* x is not in $tree[j]$). Since Algorithm 1 makes Φ_0 isomorphic to *init*, and *init* is defined on nodes in the mirage, for each node $z \in tree[j]$, $\Phi_0(z) = \Phi_0(\mu(z))$. Because Algorithm 1 gives nodes different labels only if they have different environments, if x has the same environment under Φ_i as a node y in Σ then $\Phi_{i+1}(x) = \Phi_{i+1}(y)$. Thus, the sequence of Φ_i's produced by Algorithm 1 on the disjoint system $\Sigma \cup tree[j]$ gives the same values to nodes in Σ as the sequence produced when Algorithm 1 computes the similarity labeling of Σ.

I3, the loop invariant for Algorithm 3, is much like *I2*, the loop invariant for Algorithm 2. For processor *self*, *I3* is

I3: $tree[j]$ is a tree mirage for *self* of depth $2j + 2$.
  $\wedge$ let r be the root of $tree[j]$,
    $(\forall i : 0 \le i \le j :$
      I3.1: $(\forall n : n \in NAMES : vec[n][i] = \Phi_i(nbr(n, r)))$
      I3.2: $pec[i] = \Phi_i(r)$
      I3.3: $(\forall n : n \in NAMES :$
        $\{(x.name, x.suspect) : x \in local[n][i]\}$
        $= Env_{\Phi_i}(nbr(n, r)))$
      I3.4: $(\forall n, x : n \in NAMES, x \in local[n][i] :$
        $\Phi_i(root(x.tree)) = x.suspect$
        $\wedge$ $x.tree$ is a mirage of depth $2i + 2$ of a
          $x.name$-neighbor of $nbr(n, self)))$.

I3 states that the behavior of running Algorithm 3 on Σ is like the behavior of running Algorithm 2 on $tree[j]$. Intuitively, $tree[j]$ consists of those processors in communication with each other. These processes use the knowledge of each others' existence to simulate bounded-fairness.

We will now show that *I3* always holds when *prepare_for_step* terminates. It turns out that the negation of the loop guard of *prepare_for_step* (page 48), which obviously holds when *prepare_for_step* terminates, is all that we need to know about *prepare_for_step* to show that *I3* holds.

Lemma 22. *If I3.4 holds before prepare_for_step then I3 holds after prepare_for_step.*

Proof: Let r be the root of $tree[j]$. By construction of $tree$, (i.e. by definition of new_tree on page 52) each m-neighbor of an n-neighbor of r is the root of $x.tree$ for some x in $local[n][j-1]$ such that $x.name = m$. We construct homomorphism μ from $tree[j]$ to Σ by defining $\mu(r) = self$, $\mu(nbr(n,r)) = nbr(n, self)$, and $\mu(z) = \pi(z)$ for each z in some $x.tree$, where π is the homomorphism mapping $x.tree$ to Σ. Since each x is in $local[n][j-1]$, by *I3.4*, each $x.tree$ is a mirage of depth $2(j-1)+2 = 2j$ for a $x.name$-neighbor of $nbr(n, self)$. Since $tree[j]$ has a root with children that are variables with childred than are roots of the various $x.tree$'s, $tree[j]$ is of depth $2j+2$. Thus, $tree[j]$ is a mirage for $self$ of depth $2j+2$, so the first part of *I3* holds.

Next, we show that *I3.3* holds. Let r be the root of $tree[j]$. By definition (4.6) of Env_{Φ_i}, for any i

$$\begin{aligned} Env_{\Phi_i}(nbr(n,r)) = \{(m,\alpha) : m \in NAMES, \alpha \in PLABELS :& \\ (\exists p : p \text{ a processor in } tree[j] :& \\ nbr(n,r) = nbr(m,p) \wedge \Phi_i(p) = \alpha)\}.& \end{aligned}$$

By construction of $tree[j]$, $nbr(n,r)$ has an edge to some processor p if and only if there is an $x \in local[n][j]$ such that p is the root of $x.tree$. Moreover, the processor that wrote x is an $x.name$-neighbor of $nbr(n, self)$, so

$$\begin{aligned} Env_{\Phi_j}(nbr(n,r)) = \{(m,\alpha) : m \in NAMES, \alpha \in PLABELS :& \\ (\exists x : x \in local[n][j] :& \\ x.name = m \wedge \Phi_j(root(x.tree)) = \alpha)\}.& \end{aligned}$$

Substituting for $\Phi_j(root(x.tree))$ based on *I3.4*,

$$\begin{aligned} Env_{\Phi_j}(nbr(n,r)) = \{(m,\alpha) : m \in NAMES, \alpha \in PLABELS :& \\ (\exists x : x \in local[n][j] :& \\ x.name = m \wedge x.suspect = \alpha)\}.& \end{aligned}$$

This can be rewritten as

$$Env_{\Phi_j}(nbr(n,r)) = \{(x.name, x.suspect) : x \in local[n][j]\}. \tag{6.1}$$

When *prepare_for_step* terminates, the negation of its loop guard holds.

$$\begin{aligned} (\exists \gamma,\ \forall n, i : \gamma \in PLABELS, n \in NAMES, i \leq j :& \\ pec[i] = \Phi_i(\gamma)& \\ \wedge\ vec[n][i] = \Phi_i(nbr(n,\gamma))& \\ \wedge\ \{(x.name, x.suspect) : x \in local[n][i]\}& \\ = Env_{\Phi_i}(nbr(n,\gamma)))& \end{aligned} \tag{6.2}$$

Let γ be an element of *PLABELS* that satisfies (6.2). It follows that

$$\{(x.name, x.suspect) : x \in local[n][j]\} = Env_{\Phi_j}(nbr(n, \gamma)). \tag{6.3}$$

Combining (6.1) and (6.3), we conclude

$$Env_{\Phi_j}(nbr(n, r)) = Env_{\Phi_j}(nbr(n, \gamma)). \tag{6.4}$$

Once Algorithm 1 calculates a labeling that gives two nodes different environments, these nodes will never have the same environment under any of the labelings calculated by later steps. Thus,

$$\begin{aligned}(\forall i : 0 < i \leq j : Env_{\Phi_i}(nbr(n, r)) &= Env_{\Phi_i}(nbr(n, \gamma)) \\ \Rightarrow Env_{\Phi_{i-1}}(nbr(n, r)) &= Env_{\Phi_{i-1}}(nbr(n, \gamma))).\end{aligned}$$

Therefore, by (6.4),

$$(\forall i : 0 \leq i \leq j : Env_{\Phi_i}(nbr(n, r)) = Env_{\Phi_i}(nbr(n, \gamma))). \tag{6.5}$$

So, by (6.2) and transitivity, *I3.3* holds.

Next, we show that *I3.1* holds. Since Algorithm 1 gives nodes different labels only if they have different environments, (6.5) implies that $\Phi_i(nbr(n, \gamma)) = \Phi_i(nbr(n, r))$. By (6.2), $vec[n][i] = \Phi_i(nbr(n, \gamma))$. So, by transitivity, *I3.1* holds.

Finally, we show that *I3.2* holds. By definition (4.5) of Env_{Φ_i},

$$Env_{\Phi_i}(r) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \Phi_i(nbr(n, r)) = \alpha\}.$$

Thus, by *I3.1*,

$$Env_{\Phi_i}(r) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : vec[n][i] = \alpha\}.$$

By (6.2),

$$Env_{\Phi_i}(r) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \Phi_i(nbr(n, \gamma)) = \alpha\}.$$

By definition (4.5) of Env_{Φ_i}, $Env_{\Phi_i}(r) = Env_{\Phi_i}(\gamma)$. Since Algorithm 1 only gives nodes different labels if they have different environments, $\Phi_i(\gamma) = \Phi_i(r)$. By (6.2), $pec[i] = \Phi_i(\gamma)$, so *I3.2* holds. ∎

Since $tree[j+1]$ is a mirage, it is a system. Further, $self = r \Rightarrow (I3 \Rightarrow I2)$. By replacing Σ with $tree[j+1]$ and *self* with r, we can prove:

Lemma 23.

I3 $\wedge$ *variables labeled* $vec[n][j]$ *are relabeled in step* $j+1$
$\Rightarrow \Phi_{j+1}(nbr(n, r)) = v\text{-}next(local, vec, j, n)$.

Proof: A trivial corollary of Lemma 14. ∎

Lemma 24.

> *I3* $\wedge$ *processors labeled pec*$[j]$ *are relabeled in step* $j+1$
> $\Rightarrow \Phi_{j+1}(r) = p\text{-}next(vec, pec, j)$.

Proof: A trivial corollary of Lemma 15. ∎

Lemma 25. *I3 is a loop invariant for S3.2 of Algorithm 3.*

Proof:

I3 holds initially, as follows. Initially $j = 0$. Algorithm 1 causes $\Phi_0(x) = init(x)$ for each node x in Σ and Algorithm 3 initializes $pec[0]$ to $init(self) = \Phi_0(self)$, so $x.suspect = \Phi_0(p)$ for each process p writing a value x to element 0 of a variable. Since $tree[0]$ is assumed initially to be the mirage of depth 2 for *self*, for each x in $local[n][0]$, $x.tree$ is a mirage of depth 2 of the processor that wrote x. If p writes record x then $\Phi_0(root(x.tree)) = init(root(x.tree)) = init(p) = \Phi_0(p)$, so $x.suspect = \Phi_0(root(x.tree))$. Also, $x.name$ is the name p gives to $nbr(n, self)$. Thus, *I3.4* holds. By Lemma 22, after *prepare_for_step* terminates, *I3* holds.

S3.2 reestablishes *I3*, as follows. If processors labeled $pec[j]$ were relabeled in step $j+1$ then by Lemma 24, $p\text{-}next(local, pec, j) = \Phi_{j+1}(r)$, where r is the root of $tree[j]$. If processors labeled $pec[j]$ were not relabeled in step $j+1$ then $\Phi_{j+1}(r) = \Phi_j(r)$. Thus, each iteration of the loop in *S3.2* makes $pec[j+1] = \Phi_{j+1}(r)$. For each value x that *self* writes to the $j+1$ element of its n-neighbor in *write_to_variable*, $x.suspect = \Phi_{j+1}(r)$, $x.tree = tree[j+1]$, and $x.name = n$. Therefore, each value x written to the j'th element of a variable by processor p is such that $x.suspect$ is the label given to the root of $x.tree$ by Φ_j, p is an $x.name$-neighbor of the variable, and $x.tree$ is a mirage for p of depth $2j+4$. Thus, *I3.4* holds. By Lemma 22, after *prepare_for_step* terminates, *I3* holds. ∎

6.3. Termination and Selection

The following lemma implies that each processor executing Algorithm 3 will eventually learn its label.

Lemma 26. *Eventually for all processors self and integers* i, $i \le j$, *where* j *is the number of the step of Algorithm 1 being replayed by Algorithm 3,* $pec[i] = \Phi_i(self) \wedge (\forall n : n \in NAMES : vec[n][i] = \Phi_i(nbr(n, self)))$.

Proof: By induction on j.
Base case:

The base case is $j = 0$. Due to initialization of $pec[0]$, $pec[0] = \Phi_0(self)$. Due to *S3.1*, for each name n, $vec[n][0] = init(nbr(n, self))$. By definition of Φ_0 in Algorithm 1, for each name n, $init(nbr(n, self)) = \Phi_0(nbr(n, self))$. By transitivity, the lemma holds for $j = 0$.
Induction step:

The induction step is to show that if the lemma is true for all $i \leq j$ then it is true for all $i \leq j + 1$. Thus, we can assume that

$$(\forall i : i \leq j : (\forall n : n \in NAMES : vec[n][i] = \Phi_i(nbr(n, self)))).$$

Then, for $i \leq j$,

$$\begin{aligned} &\{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \alpha = \Phi_i(nbr(n, self))\} \\ = &\{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \alpha = vec[n][i]\}. \end{aligned} \tag{6.6}$$

By definition (4.5) of Env_{Φ_j},

$$Env_{\Phi_j}(self) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \alpha = \Phi_j(nbr(n, self))\}.$$

By (6.6),

$$Env_{\Phi_j}(self) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \alpha = vec[n][j]\}. \tag{6.7}$$

By definition (4.5) of Env_{Φ_j},

$$Env_{\Phi_j}(r) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : \Phi_j(nbr(n, r)) = \alpha\}.$$

If $r = root(tree[j])$, then, by *I3.1*,

$$Env_{\Phi_j}(r) = \{(n, \alpha) : n \in NAMES, \alpha \in VLABELS : vec[n][j] = \alpha\}.$$

By (6.7), $Env_{\Phi_j}(self) = Env_{\Phi_j}(r)$.

Algorithm 1 gives nodes different labels only if they have different environments. Therefore, since $Env_{\Phi_j}(self) = Env_{\Phi_j}(r)$, $\Phi_{j+1}(self) = \Phi_{j+1}(r)$. By Lemma 24, *p-next* will compute $\Phi_{j+1}(r)$, so $pec[j + 1] = \Phi_{j+1}(self)$, proving the first part of the induction hypothesis.

In *prepare_for_step*, each processor p uses the communication substrate to write records containing $\Phi_j(p)$ to all neighboring variables. Suppose that all processors neighboring $nbr(n, self)$ eventually communicate their label under Φ_{j+1} to that variable. When that happens, each variable v will have as its value $Env_{\Phi_j}(v)$, since, by definition (4.6),

$$\begin{aligned} Env_{\Phi_j}(v) = \{(n, \alpha) : n \in NAMES, \alpha \in PLABELS : \\ (\exists p : p \in P : v = nbr(n, p) \wedge \Psi(p) = \alpha\}. \end{aligned}$$

Suppose that step $j+1$ relabels variables labeled $vec[n][j]$. In that case, $vec[n][j+1] = v\text{-}next(local, vec, j, n)$, so, by Lemma 23, $vec[n][j+1] = \Phi_{j+1}(nbr(n, self))$. Otherwise, since $\Phi_j(nbr(n, self)) = vec[n][j]$ by the induction hypothesis, $\Phi_j(nbr(n, self)) = \Phi_{j+1}(nbr(n, self))$, so $vec[n][j+1] = \Phi_{j+1}(nbr(n, self))$. Thus, if Algorithm 3 causes each processor *self* to communicate $\Phi_{j+1}(self)$ to each variable then the lemma holds for $j+1$.

Since Algorithm 3 has the communication substrate embedded in it, if Algorithm 3 writes only a finite set of values to each variable then, by Theorem 11, all processors neighboring a variable communicate their labels to that variable. To establish that Algorithm 3 writes only a finite set of values to each variable, notice that a processor initiates a backtrack only when it learns that a variable has more neighbors than the processor knew about previously, so backtracking can be initiated at a variable no more than once for each of its neighbors. Therefore, the total number of initiations of backtracks can be no more than $|E|$. In *G3* of *prepare_for_step*, a backtrack to a step k at a variable causes neighboring processors of that variable to backtrack to step $k+1$, *i.e.* the step number of a backtrack increases as the backtrack propagates. A backtrack quits propagating once its step number is as big as that of any step being replayed, so the total number of backtracks is finite. Therefore, each processor writes only a finite set of values to a shared variable, so Algorithm 3 can use *communicate* to allow each processor *self* to communicate $\Phi_j(self)$ with each neighboring variable. Thus, each processor will eventually be able to calculate the label of neighboring variables and of itself under Φ_{j+1}. ∎

Unfortunately, Lemma 26 does not imply that each processor can use Algorithm 3 as the basis for a selection algorithm. While each processor will eventually find its label, it might first calculate an incorrect label and incorrectly select itself. However, we will show below that there is an l such that $pec[l]$ mimics $\Phi_l(self)$, and that l can be calculated from the number of iterations of Algorithm 1. Thus, if processor p mimics no other processor then $pec[l] = \Theta(p)$ only when $self = p$, so Algorithm 3 can be used as the basis for a selection algorithm by executing it for l steps. Conversely, by Theorem 21, there is no selection algorithm unless one processor mimics no other.

Although all values of $tree[j]$ are tree mirages for *self*, not all tree mirages for *self* can actually be values of $tree[j]$. Ideally, $tree[j]$ would be a tree mirage whose root was given a label by Φ_j that mimicked $\Phi_j(self)$, since this would imply that $pec[j]$ mimics $\Phi_j(self)$. Unfortunately, this ideal is not always the case. However, we can show that when j gets big enough, Φ_j will give a label to the root of $tree[j]$ that is "very close" to the label that mimics $\Phi_j(self)$. We will then make a slight refinement of Algorithm 3 that enables processes to distinguish between labels that are "very close". Thus, those labels that can *not* be so distinguished are actually similar, *i.e.* the labeling Θ produced by Algorithm 1 is not necessarily a similarity labeling for fair but not bounded-fair systems in S.

If Γ is a mirage by μ for Σ, where x is a node in Σ, then an *image* of x is any node $\hat{x}$ in Γ such that $\mu(\hat{x}) = x$.

The following three lemmas constrain the values that a labeling Φ_i provided by Algorithm 1 can give to a processor's images in a mirage. In the proofs of these lemmas, we must compare the mirages of various processors executing Algorithm 3. We will always refer to the mirage for *self* as *tree*$[j]$. To avoid subscripts, the mirage for some other processor p will be referred to only as "the mirage for p".

The next lemma shows that if two processors have images distance $2k$ apart in a mirage then one processor can be replaying a step at most k ahead of the other.

Lemma 27. *If self replays step j and if processor p has an image that is distance $2k$ from the root of tree$[j]$ then processor p has replayed step $j-k$ and p's own mirage of depth $2j-2k+2$ includes the mirage for self of depth $2j-4k+2$, i.e. tree$[j-2k]$.*

Proof: Let $\hat{p}$ be the image of p that is distance $2k$ from the root of *tree*$[j]$. By *I3*, *tree*$[j]$ is a tree mirage of depth $2j+2$, so $\hat{p}$ is the root of a tree of depth $2j-2k+2$. Thus, by *I3*, the tree rooted at $\hat{p}$ was written by p after p replayed step $j-k$.

We show that p includes *tree*$[j-2k]$ in its own mirage by induction on k.

Base case:

The base case is $k=1$. When an image of p is distance 2 from the root of *tree*$[j]$ then *local*$[n][j-1]$ contains a value written by p, where $v = nbr(n, self)$ is the variable shared by *self* and p. In *communicate*, p copies its entire local copy of v to v. It not only copies the record that it is creating for step $j-1$, but it copies all known records for steps earlier than $j-1$, as well.

Suppose that *self* has replayed step j but p has not read any value x from v during step $j-2$ such that $x.suspect = pec[j-2]$. However, such a value was written by *self*, *i.e.* if *self* is an n-neighbor of v then by *I3.3* there will be a record x in *local*$[n][j-2]$ such that $x.suspect = pec[j-2]$. Thus, when *self* reads v after p writes to v, *local*$[n][j-2] \neq temp[j-2]$ so *G4* will cause *self* to retry step $j-1$. Thus, *self* will not replay step j—a contradiction—so p must have read some value x from v such that $x.suspect = pec[j-2]$.

We now want to argue that $x.tree = tree[j-2]$. Strictly speaking, there might be n-neighbors of $nbr(n, self)$ other than *self* whose label under Φ_{j-2} equals $\Phi_{j-2}(self)$; they, not *self*, might have created x, so $x.tree$ would not equal $tree[j-2]$. The actual processor that created x is chosen nondeterministically by the execution of the communication substrate. However, since $x.tree$ is only an auxiliary variable and is not used in an actual implementation, the communication substrate will consider two records to be equal even if their *tree* fields are different, so if several processors write records whose *suspect* and *name* fields are the same then only one of those records will be communicated. The system is in the exact same state no matter which processor created x, so we arbitrarily assume that *self* created it. Since *self* created x, $x.tree = tree[j-2]$.

Induction step:

Assume that for all $i < k$, if any processor p has an image that is distance $2i$ from the root of $tree[j]$ then p's own mirage of depth $2j - 2i + 2$ includes the mirage for *self* of depth $2j - 4i + 2$, i.e. $tree[j - 2i]$. We will show that this holds for $i = k$.

Pick a processor q that is distance $2l$ from *self* on the path of length $2k$ between *self* and p, so q is distance $2k - 2l$ from p. By hypothesis, all the processors on the path between *self* and p have images in $tree[j]$. In particular, since q is distance $2l$ from *self*, q has an image $\hat{q}$ distance $2l$ from the root of $tree[j]$. $\hat{q}$ is the root of a mirage of depth $2(j-l)+2$ for q, where $k < l$, and that mirage contains $\hat{p}$, an image of p, distance $2(k-l)$ from the root. In turn, $\hat{p}$ is the root of a mirage of depth $2j - 2k - 2$. By the induction hypothesis, since p is distance $2k - 2l$ from q, the mirage for p of depth $2j - 2k + 2$ includes the mirage of depth $2j - 2k - (2k - 2l) + 2 = 2j - 4k + 2l + 2$ for q. Also by the induction hypothesis, since q is distance $2l$ from *self*, the mirage for q of depth $2j - 4k + 2l + 2$ includes the mirage for p of depth $(2j - 4k + 2l + 2) - 2l = 2j - 4k + 2$, i.e. $tree[j - 2k]$. Since $tree[j-2k]$ is in q's mirage, which is in p's mirage, $tree[j - 2k]$ is in p's own mirage of depth $2j - 2k + 2$. ∎

Suppose that a processor x has two images in $tree[j]$ that are given different labels by some Φ_i, *i.e.*, there are processors p and q in $tree[j]$, a mirage of depth $2j + 2$ for *self* by μ, such that $\mu(p) = \mu(q) = x$ and $\Phi_i(p) \neq \Phi_i(q)$. Since, by *I3.2*, the label under Φ_k of a root of a mirage of depth $2k$ for a processor x is always the same as the label under Φ_k that x estimated to be its own, this means that at one time x estimated its label under Φ_i to be $\Phi_i(p)$ but later it estimated its label to be $\Phi_i(q)$. The only way that x could have changed its estimate for its label under Φ_i is if x had backtracked to step i or earlier. Furthermore, this backtrack has propagated to *self* along the path containing p, causing $\Phi_i(p)$ to be in $tree[j]$. This backtrack will eventually propagate to *self* along the path containing q, at which time *self* will backtrack. Until then, *self* has an inconsistent view of Σ.

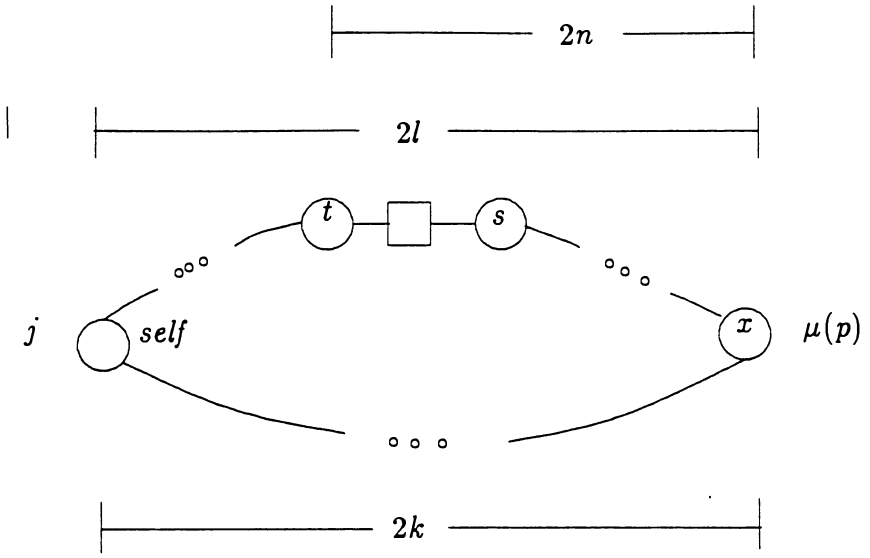


Figure 6.2: Proof of Lemma 28

It turns out that inconsistencies in a processor's mirage can only occur far from the root. As the following lemma shows, since processors distance $2k$ from each other are replaying steps within k of each other, images of a processor that are sufficiently close to the root of the mirage will be given the same label by Φ_i.

Lemma 28. *For all processors p and q in tree$[j]$ during execution of Algorithm 3 by self, if*

1. $\mu(p) = \mu(q)$,
2. *p is distance $2k$ from the root,*
3. *q is distance $2l$ from the root,*

then for all $i \leq j - k - l$, $\Phi_i(p) = \Phi_i(q)$.

Proof: Suppose that $\Phi_i(p) \neq \Phi_i(q)$. By the argument just before this lemma, $\mu(p)$ has backtracked to step i and *self* has received two tree mirages that give $\mu(p)$ two different labels, *i.e.* $\Phi_i(p)$ and $\Phi_i(q)$. Therefore, there must be at least two paths from $\mu(p)$ to *self*; one of length $2k$ and the other of length $2l$. Figure 6.2 shows this system. Here, $\mu(p)$ is x and $\Phi_i(p)$, the label more recently learned by *self*, was propagated to *self* along the lower path. Since $\Phi_i(q)$ is the old estimate of $\Phi_i(x)$, the new value has not yet propagated to *self* along the upper path. Therefore, some processor on that path must have the new value in a variable on one side of it but the old value on the other.

Let t be that processor; t is not executing or else it would have propagated the new value to all its neighboring variables.

Let the length of the path from t to x be $2n$. If s is a neighbor of t on the path from x then s is distance $2n-2$ from x. By Lemma 27 (substituting $i+2n-2$ for j, $n-1$ for k, x for *self*, and s for p), if x replays step $i+2n-2$ then s has replayed step $i+n-1$ and, more importantly, has learned x's new mirage for step i. Since information about x's new mirage is propagated along with a backtrack, when s learns x's new mirage for step i, it will backtrack to step $i+n-1$, unless it needs to replay step i for the first time. Also by Lemma 27, s will not replay step $i+n$ until t backtracks to step $i+n-1$. Consequently, s can be replaying at most step $i+n-1$, x can be replaying at most step $i+2n-2$, and *self* can be replaying at most step $i+2n+k-2$. Since t is not executing, it cannot have learned about the new estimate for $\Phi_i(x)$ from *self*, either. Since *self* replayed step $i+k$ when it learned $\Phi_i(q)$, this means that t has not learned *self*'s mirage for step $i+k$. Since the distance from *self* to t is $2l-2n$, by Lemma 27, *self* can be replaying at most step $i+k+2(l-n)$. The value of n maximizing the minimum of $i+k+2l-2n$ and $i+k+2n-2$ is $n=(l+1)/2$. Therefore, if $(\mu(p)=\mu(q)=x)\wedge\Phi_i(p)\neq\Phi_i(q)$ then *self* is replaying at most step $i+k+l-1$. Thus, if *self* is replaying a step j after $i+k+l-1$, then $\mu(p)=\mu(q)\Rightarrow\Phi_i(p)=\Phi_i(q)$. If $j>i+k+l-1$ then $\Phi_i(p)=\Phi_i(q)$. However, (since i, j, k, and l are all integers) $j>i+k+l-1$ is the same as $i\leq j-k-l$, so for all $i\leq j-k-l$, $\Phi_i(p)=\Phi_i(q)$. ∎

We now show that if a tree mirage of a system Σ becomes deep enough, the root will have the same label under Θ as the image of *self* in a *split mirage* of Σ. A split mirage of a system Σ is a mirage (not necessarily a tree mirage) such that the processors in Σ are in one-to-one correspondence with the processors in the mirage. However, there may be more variables in the mirage than in the system, so a processor is not necessarily similar to its image in a split mirage. Hence, a processor and its image in a split mirage might have different labels under similarity labeling Θ. As we prove in the next lemma, Algorithm 3 ensures that each processor learns the label of its image in some split mirage.

The next lemma requires that each processor replay three times as many steps in Algorithm 3 as there were in Algorithm 1. These extra steps can be obtained by extending the sequence of Φ_i's obtained from Algorithm 1 with $2l$ Θ's. Thus, if n_s is the number of steps in Algorithm 1,

$$(\forall i : n\_s \leq i \leq 3n\_s : \Phi_i = \Theta).$$

These refinements require no change to the text of Algorithm 3.

Lemma 29. *Let n_s be the number of steps that Algorithm 1 takes to calculate Θ from system Σ. If self ever replays step $3n\_s$ during execution of Algorithm 3 then there is a processor p in Γ, a split mirage by ρ of Σ, such that $\rho(p) = self$ and $pec[3n\_s] = \Theta(p)$, where Θ is the labeling that Algorithm 1 would calculate for $\Gamma \cup \Sigma$.*

Proof: Suppose that *self* has computed $tree[3n\_s]$, *i.e.* it has replayed $3n\_s$ steps. By Lemma 28 (letting $j = 3n\_s$ and $k = l = n\_s$), for any processors p and q in $tree[3n\_s]$ at a distance $2n\_s$ or less from the root, $\mu(p) = \mu(q) \Rightarrow \Phi_i(p) = \Phi_i(q)$, for $i \leq n\_s$. Since $\Phi_{n\_s} = \Theta$, $\mu(p) = \mu(q) \Rightarrow \Theta(p) = \Theta(q)$. Since Θ is a supersimilarity labeling, $\Theta(p) = \Theta(q) \Rightarrow Env_\Theta(p) = Env_\Theta(q)$. Thus, $\mu(p) = \mu(q) \Rightarrow Env_\Theta(p) = Env_\Theta(q)$, so all images of a processor at a distance $2n\_s$ or less from the root have the same label and environment under Θ.

In contrast, two images of the same variable that are at a distance $2n\_s$ from the root do not necessarily have the same label. However, all images of a variable that neighbor an image of a particular processor have the same label under Φ_{n_s}, *i.e.*

$$(\forall p, q : p, q \text{ processors at a distance } 2n\_s \text{ or less from the root of } tree[3n\_s] : \\ (\forall n : n \in NAMES : \mu(p) = \mu(q) \Rightarrow \Phi_l(nbr(n,p)) = \Phi_l(nbr(n,q))))$$

because p and q have the same environment under $\Phi_{n\_s} = \Theta$, so their n-neighbors have the same label.

The images of a particular variable can be divided into equivalence classes according to their label (hence, environment) under Θ. Each pair (v, α) defines such an equivalence class, where v is a variable in Σ and α is a label given to images of that variable in $tree[3n\_s]$. These equivalence classes of variables will be used to define a split mirage of Σ.

Let system Γ consist of a processor for each processor in Σ and a variable for each equivalence class of images of a variable in $tree[3n\_s]$. There is an edge from a processor representing p (p in Σ) to a variable representing (v, α) if there is an edge from p to v, and the two edges have the same label. We show that Γ is a split mirage of Σ by defining a homomorphism ρ from Γ to Σ. Define ρ to map each processor in Γ to its corresponding processor in Σ. Recall that each equivalence class of images of a variable in $tree[3n\_s]$, hence each variable in Γ, corresponds to a pair (v, α), where v is a variable in Σ and α is a label. Define ρ to map each variable in Γ corresponding to equivalence class (v, α) to v in Σ. There is an edge from p to v in Γ only if there is an edge from $\rho(p)$ to $\rho(v)$ with the same label, so ρ is a homomorphism from Γ to Σ. Thus, Γ is a mirage by ρ of Σ. Since ρ is one-to-one on processors, Γ is a split mirage of Σ.

We will show that there is a processor p in Γ such that $\rho(p) = self$ and $\Theta(root(tree[3n\_s])) = \Theta(p)$ by finding a mapping σ from $tree[3n\_s]$ to Γ such that σ preserves Θ and Env_Θ and then letting p be $\sigma(root(tree[3n\_s]))$. For a processor p at a distance $2n\_s$ or less from the root of $tree[3n\_s]$, define $\sigma(p)$ to be the unique processor q in Γ such that $\mu(p) = \rho(q)$, so $\rho(\sigma(p)) = \mu(p)$. Define $nbr(n, \sigma(p)) = \sigma(nbr(n,p))$ for all processors p at a distance $2n\_s$ or less from the root of $tree[3n\_s]$. Note that σ is only defined on nodes at a distance $2n\_s$ or less from the root of $tree[3n\_s]$.

The definition $\Phi_{n\_s}(\sigma(x)) = \Phi_{n\_s}(x)$ for each node x distance $2n\_s$ or less from the root of $tree[3n\_s]$ assigns only one label under Φ_{n_s} to each node in Γ, because

1. For all processors p and q at a distance $2n\_s$ or less from the root of $tree[3n\_s]$, $\sigma(p) = \sigma(q) \Rightarrow \mu(p) = \mu(q)$, so by Lemma 28, $\sigma(p) = \sigma(q) \Rightarrow \Phi_{n\_s}(p) = \Phi_{n\_s}(q)$.

2. Variables in Γ correspond to equivalence classes of variables in $tree[3n\_s]$ under Φ_{n_s}, so all variables in $tree[3n\_s]$ that are mapped to the same variable in Γ have the same label under Φ_{n_s}.

Since $\Phi_{n\_s} = \Theta$, σ preserves Θ for all nodes at a distance $2n\_s$ or less from the root of $tree[3n\_s]$.

Next, we show that σ preserves $Env_{\Phi_{n\_s}}$. By definition (4.5) of $Env_{\Phi_{n\_s}}$,

$$Env_{\Phi_{n\_s}}(\sigma(q)) =$$
$$\{(n,\alpha) : n \in NAMES, \alpha \in VLABELS : \Phi_{n\_s}(nbr(n,\sigma(q))) = \alpha\}.$$

By definition,

$$nbr(n,\sigma(q)) = \sigma(nbr(n,q)) \text{ and } \Phi_{n\_s}(\sigma(nbr(n,q)) = \Phi_{n\_s}(nbr(n,q)).$$

So, by (4.5), $Env_{\Phi_{n\_s}}(\sigma(q)) = Env_{\Phi_{n\_s}}(q)$ for any processor p at a distance $2n\_s$ or less from the root of $tree[3n\_s]$. In the same way, for any variable v at a distance $2n\_s$ or less from the root of $tree[3n\_s]$, q is an n-neighbor of v if and only if $\sigma(q)$ is an n-neighbor of $\sigma(v)$, and $\Phi_{n\_s}(\sigma(q)) = \Phi_{n\_s}(q)$, so by (4.6), $Env_{\Phi_{n\_s}}(\sigma(v)) = Env_{\Phi_{n\_s}}(v)$.

A processor executing Algorithm 3 will suspect a label under Φ_{n_s} only if it has the same environment as other processors with that label, so $\Phi_{n\_s}(x) = \Phi_{n\_s}(y) \Rightarrow Env_{\Phi_{n\_s}}(x) = Env_{\Phi_{n\_s}}(y)$ for any x and y at a distance $2n\_s$ or less from the root of $tree[3n\_s]$ and, by transitivity, for any x and y in Γ. Thus, Θ (*i.e.* Φ_{n_s}) is the labeling that Algorithm 1 would calculate for Γ, just as it is for Σ.

By definition, $\Phi_{n\_s}(r) = \Theta(\sigma(r))$ and $\sigma(r) = p$, where r is the root of $tree[3n\_s]$ and p is a processor in Γ such that $\rho(p) = self$. Thus, $\Phi_{n\_s}(r) = \Theta(p)$. Since $\Phi_{n\_s} = \Phi_{3n\_s}$, $\Phi_{3n\_s}(r) = \Theta(p)$. By *I3.1*, $pec[3n\_s] = \Phi_{3n\_s}(r)$, so, by transitivity, $pec[3n\_s] = \Theta(p)$. ∎

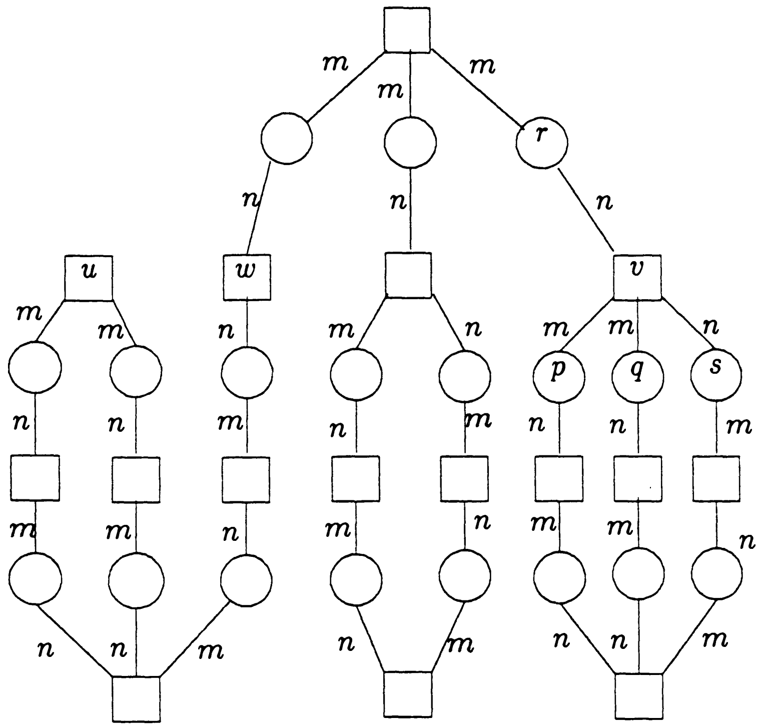


Figure 6.3: Problem Variables

A system behaves like a split mirage only when processors sharing a variable are not able to communicate with each other. In this case, each value written by one processor is always overwritten by other processors without being read. This problem is illustrated by the system in Figure 6.3. Here, four processors, p, q, r, and s, all neighbor a variable v. Suppose that whenever p or q reads from v, the other has just written to v. Further, suppose that whenever r or s reads from v, the other has just written. Neither pair of processors will ever read a value written by the other pair. Thus, all four processors will have incorrect estimates of the environment of v. In particular, the incorrect values that p and q read from v are the same as the correct value of u, so p and q might conclude that their m-neighbor is u. In the same way, r and s might conclude that their n-neighbor is w.

Since neither p nor r can read the values that the other is writing, it is as if they do not neighbor the same variable. In fact, they are behaving as if they were in the split mirage of Σ, shown in Figure 6.4, where v is replaced by v_1 and v_2, p and q have edges to v_1, and r and s have edges to v_2. If the neighbors of v read values from v that differed from all environments under Φ_j then they would quit writing on v and the problem would be solved. However, v_1 is similar to u and v_2 is similar to w so p, q, r, and s cannot deduce that

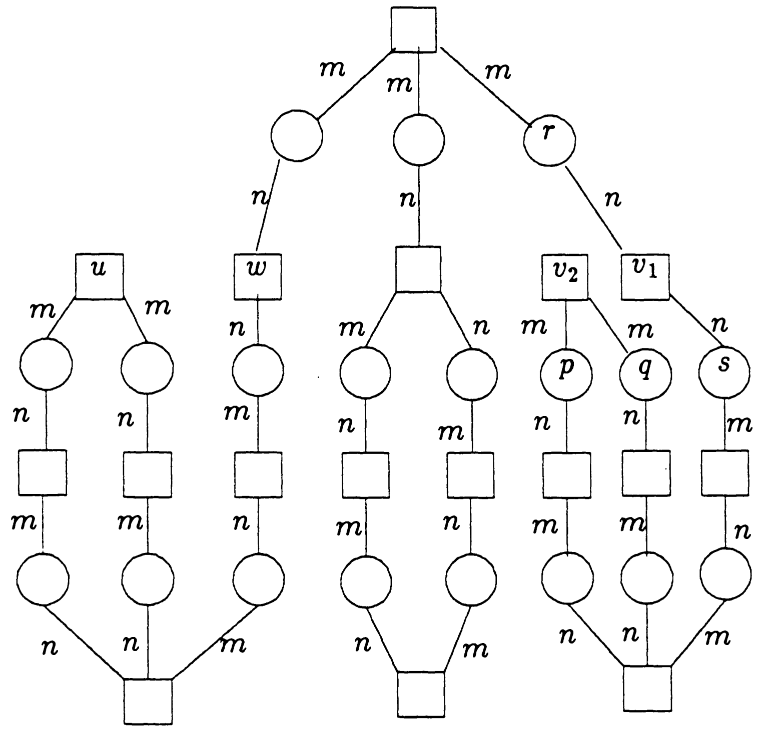


Figure 6.4: A Split Mirage Γ

there is a problem using Algorithm 3 as it is now defined. In fact, each of p, q, r, and s gets a consistent view, even though it is an incorrect one.

It turns out that v and w can be differentiated if Algorithm 3 is changed so that a process that suspects that its m-neighbor is u will replay the next five steps without writing on its m-neighbor. The number five was chosen because it is large enough to ensure that if p suspects that its m-neighbor is u then p will wait until s is forced to write to v. Thus, this algorithm will ensure that p will read from v the value that was written by s and so p will learn that its m-neighbor is v, not u. After replaying these five steps, if a processor still estimates that its m-neighbor is u rather than v then its estimate is correct. This algorithm requires that there be some path between v_1 and v_2 that contains no variable similar to u, since such a variable would not be written and so its presence on a path would prevent processors on either side of it from using the path to synchronize steps. However, if all paths between v_1 and v_2 contain a variable similar to u then there is no way to differentiate between v and u, as the following lemma shows. The variables u, v, w, v_1, and v_2 in this lemma correspond roughly to the variables with the same name in the systems of Figures 6.3 and 6.4, but any system Σ of Lemma 30 will be much larger than those systems, since every path between v_1 and v_2 must contain both a variable similar to u and one similar to v.

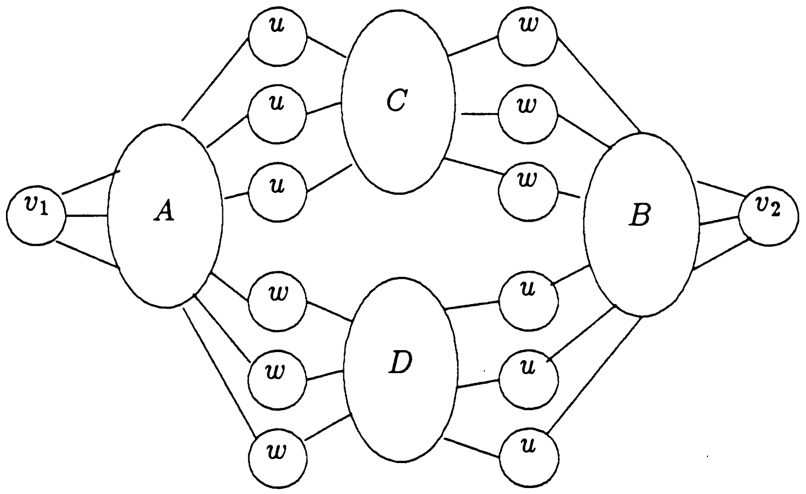


Figure 6.5: A Partitioning of Γ

Lemma 30. *If*

1. *Γ is a split mirage by μ of a system Σ,*
2. *Θ is the labeling that Algorithm 1 calculates for $\Gamma \cup \Sigma$,*
3. *for some variables $v, u, w \in \Sigma$ and $v_1, v_2 \in \Gamma$,*
   (a) $\mu(v_1) = \mu(v_2) = v$,
   (b) $\Theta(v_1) = \Theta(u)$,
   (c) $\Theta(v_2) = \Theta(w)$,
   (d) every path from v_1 to v_2 goes through a variable similar to w and a variable similar to u

then each n-neighbor of u is similar to some n-neighbor of v.

Proof: Since $\Theta(v_1) = \Theta(v_2)$, the proof of Theorem 7 gives a round-robin schedule φ for Γ that causes each n-neighbor of v_1 to behave similarly to some n-neighbor of v_2. We will show that a certain fair (but not bounded-fair) schedule ϱ simulates this round-robin schedule on Γ and so causes any set of processes in Σ to behave similarly if their images in Γ are given the same label by Θ. In Γ, each n-neighbor of u is given the same label by Θ as some n-neighbor of v_1, so each n-neighbor of u is similar to some n-neighbor of v. The rest of the proof constructs ϱ from φ.

Let Γ be a split mirage by μ of Σ with variables v_1 and v_2 such that $\mu(v_1) = \mu(v_2) = v$, $\Theta(v_1) = \Theta(u)$, $\Theta(v_2) = \Theta(w)$, and every path from v_1 to v_2 contains a variable similar to w and a variable similar to u. Processors in Γ can be partitioned into four sets, as shown in Figure 6.5.

1. A is the set of processors with a path to v_1 containing no variable similar to u or w.

2. B is the set of processors with a path to v_2 containing no variable similar to u or w.

3. C is the set of processors not in A with a path to v_1 containing no variable similar to w.

4. D is all other processors in Γ.

Since each path from v_1 to v_2 passes through a variable similar to w and a variable similar to u, A and B are disjoint and at least one of C and D are nonempty. $A \cup D$ and $B \cup C$ each define a region of Γ bounded by variables similar to u, while $B \cup D$ and $A \cup C$ each define a region bounded by variables similar to w. Processors in split mirage Γ are one-to-one with processors in Σ and therefore A, B, C, and D also correspond to sets of processors in Σ.

The difficult part of constructing a schedule ϱ for Σ that simulates execution of φ on Γ is preventing interference between processes in Σ whose images in Γ neighbor v_1 and processes in Σ whose images neighbor v_2. This interference occurs because $\mu(v_1) = \mu(v_2)$, *i.e.* the two variables v_1 and v_2 must be simulated by a single variable in Σ. Interference is prevented by separating the steps of processors of Σ in A from those in B so that a processor in A writes to v, then processors in A execute, then a processor in B writes to v, then processors in B execute, etc. A schedule ϱ that simulates φ must be constructed so that whenever processors in A are executing, v has the value of v_1; whenever processors in A are not executing, there is a processor in A whose next step is to write the value of v_1 to v.

This is done as follows. Without loss of generality, assume that we can partition φ into a sequence of finite schedules $\varepsilon_1\varepsilon_2\ldots$ such that:

- Steps in ε_i for i odd do not write to variables similar to u.

- Steps in ε_i for i even do not write to variables similar to w.

- Each ε_i starts with a processor writing to v_1 or v_2. This might require a rearrangement of the first few processors in ε_i.

We will define (by induction) a sequence of schedules $\varrho_0, \varrho_1, \ldots$ such that execution of any ϱ_i on Σ will simulate execution of that part of φ up through ε_{2i} on Γ *i.e.*

$$(\forall p : p \in \Gamma : \mathit{init} \circ \varepsilon_0 \ldots \varepsilon_{2i}(p) = \mathit{init} \circ \mu(\varrho_i)(\mu(p)))$$

and $init \circ \varepsilon_0 \ldots \varepsilon_{2i}(v_1) = init \circ \mu(\varrho_i)(v)$. Each ϱ_i will cause processors in Γ with the same label under Θ to have the same state i times, so if ϱ is the infinite sequence of which each ϱ_i is a prefix then ϱ will cause all processors with the same label under Θ to behave similarly. Thus, execution of ϱ on Σ will cause each n-neighbor of u to behave similarly to some n-neighbor of v.

Two schedules are *equivalent* if execution of either schedule results in the same system state. Note that two schedules are equivalent when one is derived from the other by exchanging adjacent steps such that if one step changes the value of a particular shared variable then the other step either does not access that variable or writes the same value to it. We can use this rule to convert φ to ϱ.

Base case:

The base case is $i = 0$. Let ϱ_0 be the empty sequence. Since $init(p) = init(\mu(p))$ and $init(v_1) = init(\mu(v))$,

$$(\forall p : p \in \Gamma : init(p) = init \circ \mu(\varrho_0)(\mu(p)))$$

and $init(v_1) = init \circ \mu(\varrho_0)(v)$.

Induction step:

Each ε_i is an interleaving of steps involving processors in A, B, C, and D. However, since an ε_i for i odd does not cause variables similar to u to change and since processors in $B \cup C$ are separated from processors in $A \cup D$ by variables similar to u, steps in ε_{2i+1} involving processors in $B \cup C$ can have no effect on steps involving processors in $A \cup D$. Thus, ε_{2i+1} is equivalent to $\varepsilon_{2i+1,AD}\varepsilon_{2i+1,BC}$, where $\varepsilon_{2i+1,BC}$ is the steps of ε_{2i+1} involving processors in $B \cup C$. In the same way, ε_{2i+2} is equivalent to $\varepsilon_{2i+2,BD}\varepsilon_{2i+2,AC}$.

Assume that ϱ_i satisfies the induction hypothesis. Recall that ε_{2i+1} does not write on variables in Γ similar to u, such as v_1, while ε_{2i+2} does not write on variables in Γ similar to w. Since $\varepsilon_1 \ldots \varepsilon_{2i}$ is equivalent to ϱ_i,

$$(\forall p : p \in \Sigma : init \circ \varepsilon_1 \ldots \varepsilon_{2i+2}(p) = init \circ \varrho_i \varepsilon_{2i+1,AD}\varepsilon_{2i+1,BC}\varepsilon_{2i+2,BD}\varepsilon_{2i+2,AC}(p)).$$

The rest of the induction hypothesis that must be proved is

$$(\forall p : p \in \Gamma : init \circ \varepsilon_0 \ldots \varepsilon_{2i+2}(p) = init \circ \mu(\varrho_i)(\mu(p))).$$

To prove this we must show that every time a processor in Σ reads v it will read the same value as it would if it were reading v_1 or v_2 in Γ. Since the last step in ϱ_i writes on v_1, each processor in $\varepsilon_{2i+1,AD}$ that reads v will read the same value as if it were reading v_1. Since the first set of steps in $\varepsilon_{2i+1,BC}$ are writing to variables similar to w, which includes v_2, each processor in $\varepsilon_{2i+1,BC}$ and $\varepsilon_{2i+2,BD}$ that reads v will read the same value as if it were reading v_2. The first set of steps in $\varepsilon_{2i+2,AC}$ write to variables similar to u (or v_1), so ϱ_{i+1} concludes with v having the same value as if it were v_1. ∎

The discussion before Lemma 30 suggests a strategy for distinguishing a variable from its images in a split mirage. The adversary schedule ϱ of the proof of Lemma 30 achieves its effect by causing each set of processes to overwrite the result of the other set of processes. If we could prevent one of these sets of processes from writing to the variable then we could force one of the sets of processes to learn the result of the processes in the other set. It turns out that this is possible when the hypothesis of Lemma 30 does not hold.

Let v_1 and v_2 be images of variable v in a split mirage of Σ. Suppose that a path of length k between v_1 and v_2 contains no variables similar to v_1. By Lemma 27, neighbors of v_1 have to replay step $i-k$ before neighbors of v_2 can replay step i, and *vice versa*. If, between these steps, the neighbors of v_1 and v_2 do not write v_1 then all the neighbors of v will read the same value for v_1 and v_2. Some processor would learn that it had calculated the wrong environment for the variable and backtrack to an earlier step. Processors sharing a variable must write on that variable to ensure that they are replaying adjacent steps, so this strategy will work only if there is no variable similar to v_1 on the path from v_1 to v_2. In the same way, v_1 and v_2 can be distinguished from v if there is no variable similar to v_2 on the path from v_1 to v_2. If neighbors of v_1 and v_2 are not similar to neighbors of v then, by Lemma 30, there is a path from v_1 to v_2 either containing no variable similar to v_1 or no variable similar to v_2. Thus, either v is similar to one of its images or it can be distinguished from any of them.

The above strategy is implemented by refining *write_to_variable.* Let $\beta_1, \beta_2, \ldots, \beta_k$ be the set of labels that Θ incorrectly gives to variables in some split mirage of Σ, where the order of the labels is arbitrary, but fixed. Let $length(\beta_i)$ be the largest distance between two images of some variable in a split mirage such that one of the images is labeled β_i by Θ. Let

$$\begin{aligned}
start(\beta_1) &= n_s \\
start(\beta_i) &= start(\beta_{i-1}) + 2 \cdot length(\beta_{i-1}) + 1 \\
stop(\beta_{i-1}) &= start(\beta_i) \\
stop(\beta_k) &= start(\beta_k) + 2 \cdot length(\beta_k) + 1.
\end{aligned}$$

Thus, for any pair of images v_1 and v_2 of the same variable, $stop(\Theta(v_1)) - start(\Theta(v_1))$ is larger than twice the distance from v_1 to v_2, so each neighbor *self* of v_1 can be sure that each neighbor of v_2 will have written to v_2 between the time that *self* has replayed steps $start(\beta_i)$ to $step(\beta_i)$. The new definition of *write_to_variable* is:

$$
\begin{array}{l}
\textit{write\_to\_variable}(j, \textit{local}, \textit{pec}) \equiv \\
\quad \textbf{for } n \in \textit{NAMES}: \\
\quad\quad \textit{val.suspects} := \textit{pec}[j];\ \textit{val.name} := n;\ \textit{val.tree} := \textit{tree}[j] \\
\quad\quad \textit{local}[n][j] := \textit{local}[n][j] \cup \{\textit{val}\}; \\
\quad\quad \textbf{if } \neg(\textit{start}(\textit{vec}[n][j]) < j \le \textit{stop}(\textit{vec}[n][j])) \\
\quad\quad\quad\quad \rightarrow \textbf{write } \textit{local}[n] \textbf{ to } \textit{nbr}(n, \textit{self}) \\
\quad\quad \square\ (\textit{start}(\textit{vec}[n][j]) < j \le \textit{stop}(\textit{vec}[n][j]) \rightarrow \textbf{skip} \\
\quad\quad \textbf{fi} \\
\quad \textbf{rof}
\end{array}
$$

Also, *prepare_for_step* needs to be changed so that a processor will not wait for a variable (in a problem set) that is not being written to. If v is not being written on during step j then $\Phi_j(v) = \Phi_{j-1}(v)$, for any j between $start(\Phi_j(v))$ and $stop(\Phi_j(v))$. Therefore, when v is not being written to, the value of v for step j can be assumed to be the same as for step $j - 1$ unless its version number has changed. The appropriate changes to *prepare_for_step* are left to the reader.

Lemma 29 indicated that Algorithm 3 needs to replay at least $3n\_s$ steps, where n_s is the number of steps in Algorithm 1. The maximum value of *stop* might be bigger than $3n\_s$, in which case Algorithm 3 will need to replay even more steps. Let *stop_max* be the maximum value of *stop*. To ensure that all backtracks will propagate before any processor decides on its label, Algorithm 3 must take at least $stop\_max + n\_s$ steps. Thus, Algorithm 3 should replay k steps, where k is the maximum of $3n\_s$ and $stop\_max + n\_s$.

Lemma 31. *During execution of Algorithm 3, if self is similar to no other processor and pec*$[j]$ *does not mimic* $\Phi_j(self)$ *then*

$$j < max(3n\_s, stop\_max + n\_s).$$

Proof: Assume that *self* is similar to no other processor. By Lemma 29, the root of $tree[3n\_s]$ is given the same label by Θ as the processor p in a split mirage by μ of Σ such that $\mu(p) = self$. If $pec[j]$ mimics $\Phi_j(self)$ then there are variables v, u, and w in Σ such that v has two images v_1 and v_2 in the split mirage such that $\Theta(v_1) = \Theta(u)$ and $\Theta(v_2) = \Theta(w)$. When *self* is between steps $start(vec[n][j])$ and $stop(vec[n][j])$, it will not write to its n-neighbor but will continue to read it. By Lemma 30 and the assumption that *self* is similar to no other processors, there is a path between the variables either containing no variables similar to u or containing no variables similar to w. Thus, either during steps $start(\Theta(u))$ to $stop(\Theta(u))$ or during steps $start(\Theta(w))$ to $stop(\Theta(w))$, all neighbors of $nbr(n, self)$ will be simultaneously replaying some step in which they are not writing on $nbr(n, self)$. All these neighbors will read the same value or read some value not written by a processor that believes it is neighboring $nbr(n, self)$. This value will cause some processor to retry a step before n_s. Thus, unless *self* has calculated a label that mimics $\Theta(self)$, there will be some processor before step n_s. This processor is within a distance $2n\_s$ of *self*, so $j < stop\_max + n\_s$. ∎

The major difference between Algorithms 2 and 3 is that, whereas Algorithm 2 finds the similarity label of each node, Algorithm 3 only finds a label that mimics the node's similarity label. Although Algorithm 3 will eventually find the correct label, there is no way for a processor to know whether its correct label has been found unless the label that has been found mimics no other label.

Theorem 32. *There is a selection algorithm for a fair system in S if*

$$(\exists p\, \forall q : p, q \in P : (p = q) \vee (p \text{ does not mimic } q)).$$

Proof: The selection algorithm uses Algorithm 3 to let each processor learn its label; a processor selects itself when $pec[k] = \Phi(p)$, where k is the constant of Lemma 31. By Lemma 26, p will eventually compute its label and select itself. By Lemma 31, for no processor except p can $pec[k]$ be $\Theta(p)$. Thus, p will eventually select itself, and no processor except p will ever select itself, so there is a selection algorithm. ∎

Theorem 32 complements Theorem 21, so we have necessary and sufficient conditions for selection in fair systems in S.

While we have not explicitly described how to compute the mimics relation, a straightforward algorithm follows from the definition of mimics, Lemma 30, and Algorithm 1. The definition for the mimics relation depends on the similarity relation and, by Lemma 30, computing the similarity relation requires knowing the mimics relation. Fortunately, these recursive requirements can be met, since calculating the mimics relation for processors in Σ requires the similarity relation for each subset of Σ while calculating the similarity relation for processors in Σ requires the mimics relation for each split mirage of Σ. Each subset of Σ has fewer edges than Σ but the same number of variables (if we delete only processors), while each split mirage of Σ has more variables but no more edges. The number of edges eventually goes to zero, at which point calculating the similarity relation is trivial. Thus, the similarity and mimics relation can be calculated for fair systems in S, so using Theorems 21 and 32, the selection problem can be solved for any fair system in S.

Chapter 7

Locking

In this chapter, we solve the selection problem for systems of processors using the locking instruction set, L. Recall, L consists of S plus the **lock** and **unlock** instructions defined in Chapter 2.

The similarity relation for systems in L is different from that of systems in S. Processors in L that share a variable and give it the same name n are not similar because they can distinguish between themselves by locking their n-neighbor. Thus, in L

$$p \sim q \Rightarrow (\forall n : n \in NAMES : nbr(n, p) \neq nbr(n, q)). \tag{7.1}$$

Either of p or q could be the 'winner' of a competition to lock a variable they share, so the result of each processor attempting to lock each of its neighboring variables is a family of systems, containing one system for each possible outcome of success and failure at locking. Each member of the family has the property that no two processors with the same state give the same variable the same name.

The following theorem is key to this phenomenon.

Theorem 33. *For any labeling Ψ on any fair system Σ in L with initial state init, if*

$$\begin{aligned}(\forall p, q : \ & p, q \in P : \\ & \Psi(p) = \Psi(q) \Rightarrow (Env_{\Psi}(p) = Env_{\Psi}(q) \wedge init(p) = init(q)) \\ & \qquad\qquad\qquad \wedge (\forall n : n \in NAMES : nbr(n, p) \neq nbr(n, q)))\end{aligned}$$

then Ψ is a supersimilarity labeling.

Proof: The proof is exactly the same as that for Theorem 7 except that the case of locking instructions must be considered. There are three cases.
read *instructions.*

Same as proof of Theorem 7.

write *instructions.*

Same as proof of Theorem 7.

Locking instructions.

Let the processors involved in steps in small schedule ε_i, as defined in the proof of Theorem 7, be performing a locking instruction on a shared variable named n. Let T_i be the state after the system has executed ε_1 through ε_i. The induction hypothesis can be written as

$$\Psi(x) = \Psi(y) \Rightarrow (T_i(x) = T_i(y) \wedge T_i(nbr(n,x)) = T_i(nbr(n,y))).$$

Since all processors involved in a step in ε_i have the same label under Ψ, $\Psi(p) = \Psi(q)$ for any $p, q \in P$ involved in a step in ε_i. By hypothesis, for any $p, q \in P$ involved in a step in ε_i,

$$Env_\Psi(p) = Env_\Psi(q) \wedge (\forall n : n \in NAMES : nbr(n,p) \neq nbr(n,q)).$$

All processors involved in steps in ε_i are locking their n-neighbors, so they are all locking different variables. Since $Env_\Psi(p) = Env_\Psi(q)$, by (4.2), n-neighbors of similar processors are similar. Thus, all the variables called n by processors in ε_i are similar and, due to (4.1), have the same value. By the induction hypothesis and because a locking instruction changes the states of a processor and variable as a function of their original states,

$$\Psi(x) = \Psi(y) \Rightarrow (T_{i+1}(x) = T_{i+1}(y) \wedge T_{i+1}(nbr(n,x)) = T_{i+1}(nbr(n,y))).$$

∎

Systems in L do not necessarily have similarity labelings. Consider the system in Figure 7.1. If this system were in S, processors p, q, r, and s (among others) would all be similar. However, if the system were in L, p and q could not be similar, nor could r and s, since each of those pairs of processors shares a variable to which they give the same name. Theorem 33 can be used to show that there are supersimilarity labelings that give p and r the same label (e.g. the one shown in Figure 7.1) and ones that give p and s the same label (e.g. swapping the labels on p and q in Figure 7.1.) Therefore, $p \sim r$ and $p \sim s$ but $r \not\sim s$. Since similarity is not an equivalence relation in this system, there can be no similarity labeling. Although most systems in L have similarity labelings, the fact that some systems do not means that some parts of our theory must be extended to handle this. As we will see, this is not hard.

The following lemma states that if a supersimilarity labeling gives at least one processor or variable a unique label then it gives every processor and variable a unique label. It is used to describe when there is no selection algorithm.

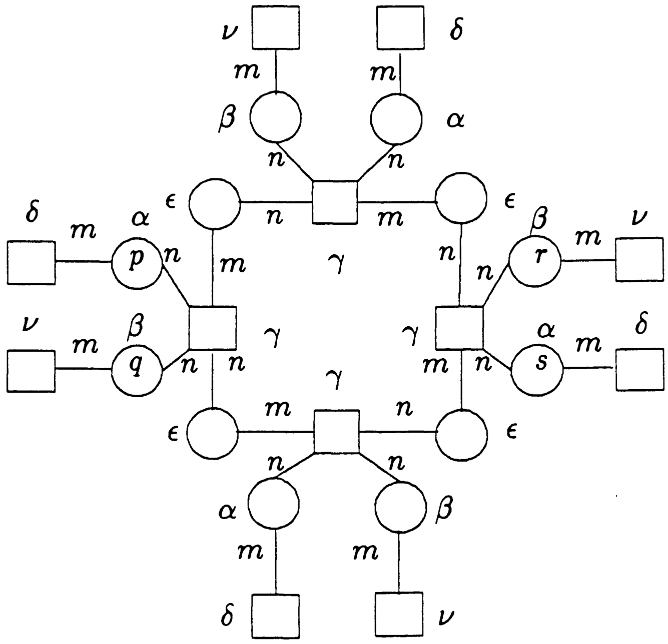


Figure 7.1: No Similarity Labeling

Lemma 34. *For any supersimilarity labeling Ψ of a connected system $\Sigma = (N, init, L, F)$, if any node is uniquely labeled by Ψ, then all nodes are.*

Proof: We show that every neighbor of a node uniquely labeled by Ψ is uniquely labeled. Since N is connected, if any node is uniquely labeled by Ψ then all are.

There are two cases, since a node is either a processor or a variable. Assume that p is a uniquely labeled processor with n-neighbor v. All variables labeled $\Psi(v)$ by Ψ must be similar, by definition of supersimilarity labeling. By (4.3) any variable similar to v must have an n-neighbor similar to p, so any variable labeled $\Psi(v)$ by Ψ must have an n-neighbor similar to p. But, no other processor is similar to p and p has exactly one n-neighbor, so v must be uniquely labeled, as well. Therefore, all neighbors of a uniquely labeled processor are uniquely labeled.

Assume that v is a uniquely labeled variable with n-neighbor p. Since v is uniquely labeled and each node similar to p must have an n-neighbor similar to v, each processor similar to p must be an n-neighbor of v. However, by (7.1), no two n-neighbors of v have the same label under Ψ, so all neighbors of v are uniquely labeled. Thus, all neighbors of a uniquely labeled variable are uniquely labeled. Thus, if any node is uniquely labeled, all nodes are uniquely labeled. ∎

Theorem 33 describes properties of supersimilarity labelings for systems in L. Consequently, if Ψ is a supersimilarity labeling in S and no two processors given the same label by Ψ give the same variable the same name, then Ψ is a supersimilarity labeling in L. If $(\forall p\, \exists q : p, q \in P : p \neq q \wedge \Psi(p) = \Psi(q))$ then, by Theorem 4, there is no selection algorithm. Lemma 34 implies that either any processor can be selected or else none can. This is summarized by the following theorem.

Theorem 35. *For any connected fair system in L with labeling Ψ, if*

$$\begin{aligned}(\forall p, q : p, q \in P :\ & (\Psi(p) = \Psi(q) \wedge p \neq q) \\ & \Rightarrow (Env_{\Psi}(p) = Env_{\Psi}(q) \wedge init(p) = init(q)) \\ & \quad \wedge (\forall n : n \in NAMES : nbr(n, v) \neq nbr(n, q))) \\ \wedge\ (\exists p, q : p, q \in P : \Psi(p) = \Psi(q)) &\end{aligned}$$

then there is no selection algorithm.

Proof: Given above. ∎

7.1. Finding Labels in L

To see how each processor in a system in L can compute its label, note the following. Assume that

$$\text{if processor } p \text{ mimics processor } q \text{ in } \Sigma, \text{ a system in } L, \text{ then } p \text{ is similar to } q. \tag{7.2}$$

Recall that Algorithm 3 lets each processor find a label that mimics its own label under similarity labeling Θ. Any program for a system in S is also a program for a system in L, so Algorithm 3 can also run on systems in L. If one processor mimics another only when they are similar, then Algorithm 3 lets each processor find its label under Θ. Thus, if (7.2) is true then Algorithm 3 lets each processor in a system in L find its label under similarity labeling Θ, provided that such a labeling exists.

We will need to use (4.1), i.e.

$$x \sim y \Rightarrow init(x) = init(y)$$

(4.2), i.e.

$$(\forall p, q : p, q \in P : p \sim q \Rightarrow (\forall n : n \in NAMES : nbr(n, p) \sim nbr(n, q))),$$

and (4.3) to prove the following lemmas. However, (4.3) was only a "suggestion" to motivate the definition of environment, but was not proven because it did not hold for all systems. The next lemma proves it for systems in L.

Lemma 36.

$$\begin{aligned}
&(\forall u, v : u, v \in V : \\
&\qquad u \sim v \Rightarrow \\
&\qquad (\forall n, p : n \in NAMES, p \in P : \\
&\qquad\qquad v = nbr(n,p) \Rightarrow (\exists q : q \in P : u = nbr(n,q) \wedge p \sim q))).
\end{aligned}$$

Proof: By contradiction. If v and u are similar but there is an n-neighbor of v, p, such that there is no n-neighbor of u that is similar to p then there must be some program that will ensure that eventually p will have a different state from any n-neighbor of u. This program can incorporate a version of Algorithm 3 that uses locking to ensure that updates to shared variables are atomic. By fairness, all processors will eventually write to all variables so each variable will eventually contain one name/state pair for each neighboring processor. If variable v has an n-neighbor labeled α by *init* but variable u does not then v will always contain the pair $\langle n, \alpha \rangle$ but u will never, hence the states of v and u will differ. Thus, u and v are not similar, a contradiction. Therefore, if v and u are similar then for each n-neighbor of v there is an n-neighbor of u that is similar. ∎

Assumption (7.2) will be shown to be valid using *fully branched mirages.* The fully branched mirage of depth k of x, $FB(x,k)$, is the largest possible tree mirage of x of depth k such that no two descendents of a variable giving it the same name are similar. Note that if p is a processor and r is the root of $FB(p,k)$ then

$$nbr(n,r) \text{ is the root of } FB(nbr(n,x), k-1). \tag{7.3}$$

Lemma 37. *In a system in L, for any $k > 0$ and all nodes x and y,*

$$x \sim y \Rightarrow (FB(x,k) \textit{ is isomorphic to } FB(y,k)).$$

Proof: The proof proceeds by induction on k.
Base case:

The base case is when $k = 0$. If x and y are similar then, by (4.1), $init(x) = init(y)$. $FB(x,0)$ is a single node r such that $\mu(r) = x$, and similarly for y. The trivial mapping of $FB(x,0)$ to $FB(y,0)$ preserves *init* and is a one-to-one mapping, so it is an isomorphism.
Induction step:

Suppose that $x \sim y$. By (4.1), $init(x) = init(y)$. There are two cases, since x and y are either both processors or both variables.

If x and y are both processors then let r be the root of $FB(x,k)$ and s be the root of $FB(y,k)$. By definition of fully branched mirages, $init(r) =$

init(x). By (7.3), $nbr(n,r)$ is the root of $FB(nbr(n,x),k-1)$ and $nbr(n,s)$ is the root of $FB(nbr(n,y),k-1)$. Since $x \sim y$, by (4.2),

$$(\forall n : n \in NAMES : nbr(n,x) \sim nbr(n,y)).$$

By induction, $FB(nbr(n,x),k-1)$ and $FB(nbr(n,y),k-1)$ are isomorphic, for all n. The function mapping r to s and, for each n, the mirage rooted at $nbr(n,r)$ to $nbr(n,s)$ preserves *init* (since $init(r) = init(x) = init(y) = init(s)$), *naming* and edges. It is one-to-one since r and s each have exactly one n-neighbor, so it is an isomorphism.

If x and y are variables then, since $x \sim y$, by Lemma 36

$$\begin{aligned}&(\forall n, p : n \in NAMES, p \in P :\\ &\qquad x = nbr(n,p) \Rightarrow (\exists q : q \in P : y = nbr(n,q) \wedge p \sim q))).\end{aligned}$$

By (7.1), no two processors giving x (or y) the same name are similar. Therefore, for each processor p that is an n-neighbor of x there is exactly one n-neighbor q of y such that $p \sim q$. We denote this unique neighbor q by $f(p)$. Since $p \sim f(p)$, by the induction hypothesis, $FB(p,k-1)$ is isomorphic to $FB(f(p),k-1)$. The function mapping the root of $FB(x,k)$ to the root of $FB(y,k)$ and, for each neighbor p of x, $FB(p,k-1)$ to $FB(f(p),k-1)$ is an isomorphism, since it preserves *init*, *naming*, edges, and is one-to-one. ∎

We would like to show that each processor can always tell if its mirage is different from its fully branched mirage, i.e. if a neighboring processor has not yet executed. However, if a variable has two neighbors that are similar processors that give it the same name then it may not be possible to ensure that both processors have executed. The next lemma takes advantage of the assumption that no two processors giving a variable the same name have the same initial state to show that if some processor has not yet executed then every processor will compute a mirage that is smaller than its fully branched mirage.

Lemma 38. *Consider systems $(N, init_1, L, F)$ and $(\pi(M), init_2, L, F)$, where $M \subseteq N$. If M is missing a processor distance k from some processor p in N then $FB(\pi(p),k)$ is smaller than $FB(p,k)$.*

Proof: By induction on k.
Base case:

The base case is $k = 2$. By (7.1), no two m-neighbors of a variable are similar. Therefore, for any processor p, $FB(p,2)$ has as many leaves as

$$\sum_{n \in NAMES} \text{number of neighbors of } nbr(n,p).$$

If M is missing a processor distance 2 from some processor p in N then one of the processors neighboring a neighbor of $\pi(p)$ in system $(\pi(M), init, L, F)$ is missing. Thus, $FB(\pi(p), 2)$ has fewer leaves than $FB(p, 2)$, i.e. $FB(\pi(p), 2)$ is smaller than $FB(p, 2)$.
Induction step:

The induction step is to assume that $FB(\pi(q), k-2)$ is smaller than $FB(q, k-2)$ for $k > 2$ and prove that $FB(\pi(p), k)$ is smaller than $FB(p, k)$. By definition, the root r of $FB(\pi(p), k)$ has an edge to a variable for each n-neighbor of $\pi(p)$. Since each of a variable's m-neighbors is dissimilar to any other, if $\pi(q)$ is an m-neighbor of $nbr(n, \pi(p))$ then there is an edge from $nbr(n, r)$ labeled m to $FB(\pi(q), k-2)$. One of these m-neighbors $\pi(q)$ is distance $k-2$ from one of the processors missing from M. By induction, $FB(\pi(q), k-2)$ is smaller than $FB(q, k-2)$. Thus, $FB(\pi(p), k)$ is smaller than $FB(p, k)$. ∎

Lemma 39. *If processor p mimics processor q in a connected system Σ in L then p is similar to q.*

Proof: Since p mimics q, there is a subset M of N such that in $(N \cup \pi(M), init, L, F)$, $p \sim \pi(p)$ and $p \sim \pi(q)$. Since no two processors giving a variable the same name have the same initial state, by Lemma 37, $FB(\pi(p), k)$ is isomorphic to $FB(p, k)$ for k the diameter of Σ. M and N must be equal, since if M is missing a processor in N then, by Lemma 38, $FB(\pi(p), k)$ and $FB(p, k)$ will not be isomorphic—a contradiction. Since M and N are equal, by the definition of mimics, $p \sim q$. ∎

Homogeneous families consist of systems with the same network topology and differing initial states. As we will show in the next section, selection in systems in L is equivalent to generic selection in homogeneous families of systems in L in which no processors with the same initial state give the same variable the same name. The following lemmas will prove helpful in showing that Algorithm 3 can be used to let each processor in a homogeneous family find its label under the similarity labeling of that family.

Lemma 40. *Given a system Σ in L, if there is a path from node x to node y in Σ, then there is a path from every node similar to x to some node similar to y.*

Proof: By induction on the length of the path.
Base case:

Let y be distance 1 from x, i.e. y is a neighbor of x. If x and y are processors then by (4.2),

$$(\forall p, q : p, q \in P : p \sim q \Rightarrow (\forall n : n \in NAMES : nbr(n, p) \sim nbr(n, q))).$$

Thus, if x is a processor with n-neighbor y and $t \sim x$ then t has an n-neighbor similar to y. If x and y are variables then by Lemma 36,

$$\begin{aligned}
&(\forall u, v : u, v \in V : \\
&\qquad u \sim v \Rightarrow \\
&\qquad (\forall n, p : n \in NAMES, p \in P : \\
&\qquad\qquad v = nbr(n,p) \Rightarrow (\exists q : q \in P : u = nbr(n,q) \wedge p \sim q)))
\end{aligned}$$

so if x is a variable with n-neighbor y and $t \sim x$ then t has an n-neighbor similar to y.

Induction step:

Assume the lemma is true for paths of length less than k, and suppose there is a path of length k from node x to node y. Let the first node in this path be z, an n-neighbor of x. Then, there is a path of length $k-1$ from z to y. Suppose t is similar to x. There are two cases based on whether x is a processor or a variable. If x is a processor then, by (4.2), t has an n-neighbor s similar to z. If x is a variable with n-neighbor z then, by Lemma 36, t has an n-neighbor s similar to z. Thus, t has an n-neighbor s similar to z. By hypothesis, s has a path of length $k-1$ to a node similar to y. Thus, there is a path of length k from t to a node similar to y. ∎

Observe that Lemma 40 involved no assumptions about the connectedness of Σ. Therefore, it holds for disjoint-union systems. Given a family of systems with similarity labeling Θ that labels two connected nodes in one system α and β, Lemma 40 implies that any system in the family with a node labeled α by Θ will have a node labeled β by Θ.

Lemma 41. *If processor p mimics processor q in a homogeneous family in L then $p \sim q$.*

Proof: If p and q are in the same system then, by Lemma 39, p is similar to q. Therefore, assume p is in $\Sigma = (N, init_1, L, F)$ and q is in $(N, init_2, L, F)$. Since p mimics q, there is a subset M of N such that $p \sim \pi(q)$ in $\Sigma \cup (\pi(M), init_2, L, F)$. By Lemma 40, each processor in Σ must be similar to some processor in $(\pi(M), init_2, L, F)$. If M is missing a processor in N then, by Lemma 38, for each z in N, $FB(\pi(z), k)$ is smaller than $FB(z, k)$, so any processor in N with largest fully branched mirage will have a different mirage than any processor in $\pi(M)$, and by Lemma 37 be dissimilar to any processor in $\pi(M)$—a contradiction. Hence, M is not missing a processor in N, so M and N are equal. By the definition of mimics, $p \sim q$. ∎

7.2. Selection for Systems in L

Since systems in L do not necessarily have similarity labelings, we need a different selection algorithm than one that first lets each processor find its label under the similarity labeling and then causes the processor with the predetermined label to select itself. Instead, we first use locking to change the state of the system so that processors giving the same variable the same name have different states.

The following algorithm is used to relabel processors that give a variable the same name. Processes are relabeled in the order in which they lock the variable. If there are i processors that give v the same name then the algorithm labels those processors 1 to i. In the algorithm, each variable contains an integer that indicates the number of processors that have successfully locked it. All variables are initially zero. A processor p locks v, reads *count* from v, labels itself with *count*, increments v, and unlocks v. If processor p is the j'th processor to lock its n-neighbor then the new state of p can be represented by $\langle init(p), (n, j)\rangle$. This is summarized by the following subroutine:

$relabel(self) \equiv$
  $init'(self) := init(self)$;
  for each $n \in NAMES$:
    lock n;
    read *count* **from** n;
    let $init'(self)$ be $\langle init'(self), (n, count)\rangle$;
    write $count + 1$ **to** n;
    unlock n
  rof;

Lemma 42. *Subroutine relabel changes init to init' such that*

$$init'(x) = init'(y) \Rightarrow init(x) = init(y)$$

and

$$(\forall p, q, n : p, q \in P, n \in NAMES : \\ init'(p) = init'(q) \Rightarrow nbr(n, p) \neq nbr(n, q)).$$

Proof: Since $init'(x)$ is a list whose first element is $init(x)$,

$$init'(x) = init'(y) \Rightarrow init(x) = init(y),$$

because if two lists are equal then their first elements must be equal. If processors p and q are n-neighbors of different variables, then the rest of the lemma follows. Otherwise, p and q are both n-neighbors of variable v, and they both read v and incremented its value of *count*. Each must have read a different value of *count*, because locking is used in *relabel* to ensure at most one processor reads the value of *count* before incrementing it. Since the state of a processor includes the value of *count* that it read, p and q have different states. Thus, if p and q have the same states after *relabel* has executed then they are not both n-neighbors of v. ∎

Since the order that processors lock variables is determined by the schedule, the state of a system Σ after executing *relabel* will be one of a set of possible states. Thus, execution of *relabel* results in a member of a family of homogeneous systems. It turns out that this family has a similarity labeling.

Lemma 43. *A system $(N, init, L, F)$ has a similarity labeling if*

$$(\forall p, q, n : p, q \in P, n \in NAMES : (init(p) = init(q) \Rightarrow nbr(n,p) \neq nbr(n,q))).$$

Proof: Every program in S is a program in L so if some property holds for every program in L then it holds for every program in S. Thus, if x and y in N are similar in $(N, init, L, F)$ then they are similar in $(N, init, S, F)$. This implies that a subsimilarity labeling for $(N, init, S, F)$ is a subsimilarity labeling for $(N, init, L, F)$. By Theorem 33, if

$$(\forall p, q, n : p, q \in P, n \in NAMES : (init(p) = init(q) \Rightarrow nbr(n,p) \neq nbr(n,q))).$$

then a supersimilarity labeling for $(N, init, S, F)$ is a supersimilarity labeling for $(N, init, L, F)$. Thus, any similarity labeling for $(N, init, S, F)$ is a similarity labeling for $(N, init, L, F)$. By Theorem 8, Algorithm 1 will find the similarity labeling for $(N, init, S, F)$, so it will also find it for $(N, init, L, F)$. ∎

Thus, by Lemma 41, we can use Algorithm 3 to let each processor find its label under the similarity labeling of the family. By Theorem 19, if we can find a set of processor labels, *ELITE*, such that each member of the family produced by *relabel* has exactly one member with a label in *ELITE*, then Σ has a selection algorithm. This happens when every supersimilarity labeling of Σ gives some processor a unique label.

Lemma 44. *Let A be a set of supersimilarity labelings of members of a homogeneous family of systems $\mathcal{H}$ in L such that no two n-neighbors of the same variable have the same initial state. There is a set ELITE of processor labels such that each supersimilarity labeling in A gives exactly one processor a label in ELITE.*

Proof: *ELITE* is found by adding labels from members of A that have no labels in *ELITE*, as follows.

$ELITE := \phi$;
$A :=$ the set of all versions of Θ that could have been produced by *relabel*;
do $(\exists\Psi : \Psi \in A : (\forall p : p \in P : \Psi(p) \notin ELITE)) \rightarrow$
    pick $\Psi \in A$ such that $(\forall p : p \in P : \Psi(p) \notin ELITE)$;
    pick a processor p such that $(\forall q : q \in P : p \neq q \Rightarrow \Psi(p) \neq \Psi(q))$;
    $ELITE := ELITE \cup \{\Psi(p)\}$;
od
$\{(\forall\Psi : \Psi \in A : (\exists self : self \in P : \Psi(self) \in ELITE))\}$

The loop invariant is

$$(\forall\Psi : \Psi \in A : \text{ for no more than one processor } p \text{ is } \Psi(p) \in ELITE).$$

ELITE is initially empty, so the invariant holds initially. By hypothesis, each Ψ in A has a processor that is uniquely labeled. By Lemma 34, all processors are uniquely labeled by each Ψ in A. If $(\forall q : q \in P : \Psi(q) \notin ELITE)$, then, by Lemma 40, there is no path from any processor labeled $\Psi(p)$ to any processor with a label in *ELITE*. Thus, for any labeling Ψ in A such that $(\forall q : q \in P : \Psi(q) \in ELITE)$, addition of $\Psi(p)$ to *ELITE* for any $p \in P$ will maintain the invariant.

The loop terminates because A has a finite number of members. Each iteration of the loop causes at least one more member of A to have a processor whose label is in *ELITE*. When all members have one, the loop terminates. When it terminates, each member of A has a processor with a label in *ELITE*, but, by the invariant, no member has more than one processor with a label in *ELITE*. Thus, each member of A has exactly one processor with a label in *ELITE*. ∎

Theorem 45. *If a fair system Σ in L has no supersimilarity labeling Ψ such that*

$$(\forall p\, \exists q : p, q \in P : p \neq q \wedge \Psi(p) = \Psi(q))$$

then Σ has a selection algorithm.

Proof: By Lemma 42, *relabel*(*self*) produces a member of a family of systems that, by Lemma 43, has a similarity labeling. Therefore, *relabel* induces a set of similarity labelings, just as it induces a family of systems. By Lemma 44, there is a set of processor labels *ELITE* such that each member of the family has exactly one processor with a label in *ELITE*. By Theorem 19, there is a generic selection algorithm for this family. By Lemma 41, Algorithm 3 can be used as a basis of such a selection algorithm. ∎

We have so far avoided instructions that can access several shared variables simultaneously. However, it is easy to see how to describe systems in which processors can lock a list of variables. In a system in which processors can lock two variables at once, similar processors can not neighbor the same variable. Otherwise, if p were the n-neighbor of v and q were the m-neighbor of v then p and q could be distinguished by the program locking n and m in a single instruction. Other than this difference, all results are the same as the results for L.

Chapter 8

Selection in CSP

CSP[H78] is a language for distributed programming in which processes communicate with each other by message-passing. The selection problem for CSP has been independently studied by Bouge [B84], whose work has many parallels to ours. Bouge distinguished between syntactic symmetry, such as G-symmetry, and semantic symmetry, such as similarity, though his definition for semantic symmetry differs from similarity. In addition, Bouge observed the relationship between the selection problem and the problem of showing that there is no distributed, symmetric, deterministic solution to the Dining Philosophers problem, which we discuss in Section 9.3.

This chapter demonstrates how to compute similarity for message-passing systems. It also shows how to compute similarity for systems in which some of our assumptions are relaxed, since CSP violates the Program Assumption and the behavior of a CSP system is usually described as a sequence of communications instead of as a sequence of states. Since Bouge has a different semantic definition of symmetry, this chapter also shows how to use similarity to calculate relations other than similarity that are semantic definitions of symmetry. Finally, this chapter solves a problem posed by Bouge that hitherto had only a partial solution–determining when a symmetric system of processes in CSP can select a leader.

CSP combines synchronous communication primitives with a modified version of the guarded command notation that we have been using.[1] There are two communication primitives; execution of the *output statement*

$$p!y$$

[1]To minimize confusion, we will continue to use **if fi** and **do od** instead of the brackets of CSP. Those familiar with CSP may assume that it is the confusion of the author that is being minimized.

by process q causes the value of y to be sent to process p, and execution of the *input statement*

$$p?x$$

by q causes q to receive a value from process p and store it in local variable x. Since communication is synchronous, execution of $p!y$ by q is simultaneous with the execution of $q?x$ by p and results in x in p being set to the value y.

Communication primitives are not only statements, but can also appear in guards of guarded commands. When a process q executes communication primitive $p!x$ or $p?y$ in a guard, if p has terminated then the primitive evaluates to false, if p is ready to communicate with q then the primitive evaluates to true and the communication occurs, otherwise q waits. Bouge examines two variants of CSP. *Strict* CSP, the original version by Hoare[H78], allows only input statements in guards, while *extended* CSP allows both input and output statements in guards.

The interconnection between processes is not explicit but is derived from the CSP program. Thus, it is not possible to separate a program and the system of processes that executes it. However, symmetry can be defined without reference to the interconnection graph by defining it relative to schedules, as follows.

A schedule of a CSP program is a sequence of pairs of processes, where the first process sends the message and the second receives it. Local computation that involves no message exchange is assumed to occur simultaneously with the preceding message exchange, so schedules reflect the sequence of states of the program. A function ρ from processes to processes can be extended to be a function on schedules by defining $\rho(\langle r, s\rangle\varphi) = \langle\rho(r)\rho(s)\rangle\rho(\varphi)$ for any schedule φ, and processes r and s.

Let $SCHED(\Sigma)$ be the set of legal schedules for program Σ. Processors p and q in program Σ are *B-symmetric* (i.e. symmetric according to the definition of Bouge) if there is an automorphism ρ on processes of Σ such that $p = \rho(q)$ and $(\forall\varphi : \varphi \in SCHED(\Sigma) : \rho(\varphi) \in SCHED(\Sigma))$. This definition is partly syntactic, because the set of automorphisms of Σ can be determined from the syntax of the program. It is a semantic definition because a schedule corresponds to the input/output behavior of the processes in the system. B-symmetry expresses the idea that any symmetric process is as likely to enter a particular state as any other. Thus, a symmetric selection algorithm is one in which each process has an equal chance of being selected, assuming that all schedules are equally likely.

Systems of processes in CSP violate the Program Assumption. A program for a CSP process p explicitly names the processes with which p communicates, so usually all processes will have different programs. An unlimited ability to assign different programs to different processes permits any process to be trivially selected. Also, it is undecidable whether or not processes are B-symmetric in a given program. In order to avoid these problems, Bouge assumes that all processes use the same *program template*. Given a pair of functions *in* and *out* from processes and names (i.e. elements of set *NAMES*) to processes, a program template is a program for process p that always sends messages to process $out(p, n)$ for some name n and receives messages from process $in(p, n)$ for some n.

The introduction of program templates allows the program that a system of processes is executing to be separated from the interconnection graph of the processes. The functions *in* and *out* correspond to the network graph of our earlier model while program templates correspond to the programs. CSP defines an instruction set and scheduling policy. The only part of our earlier models without a corresponding feature in CSP is the initial state, which we define next. Thus, even though CSP did not originally seem to fit the assumptions we make about the model of computation, it was not hard to change it to fit.

Let $OUTPUT(p)$ be the set of processes to which p is trying to send a message and let $INPUT(p)$ be the set of processes to which p is trying to receive a message. Processes p and q have *symmetric opportunities* if

$$(\forall n : n \in NAMES : in(p,n) \in INPUT(p) \iff in(q,n) \in INPUT(q)) \wedge$$
$$(\forall n : n \in NAMES : out(p,n) \in OUTPUT(p) \iff out(q,n) \in OUTPUT(q)).$$

Lemma 46. *Processes with symmetric opportunities for any program template are in the same state.*

Proof: Suppose that processes p and q are in different states. The program that causes processes in the state of p to send to a single process and processes in the state of q to receive from a single process will cause $INPUT(p)$ to be empty and so be different from $INPUT(q)$, which has one member. Thus, if processes are in different states then there are programs that prevent them from having symmetric opportunities. ∎

Bouge extends B-symmetry to systems using program templates. In addition to the previous requirements, an automorphism ρ must preserve *in* and *out*, i.e. if p and q are B-symmetric then for every n, $in(p,n) = \rho(in(q,n))$ and $out(p,n) = \rho(out(q,n))$. Bouge shows that use of program templates ensures that automorphic processes are B-symmetric, though the converse is not necessarily true. We can show that B-symmetric processes in strict CSP are similar by showing that they have symmetric opportunities infinitely often. The following theorem is the same as Theorem 4 of [B84].

Theorem 47. *B-symmetric processes in strict CSP are similar.*

Proof: We show a schedule that causes B-symmetric processes to have symmetric opportunities infinitely often for any program unless all processes are deadlocked. This schedule, like the schedule of Theorem 7, is an infinite sequence of small schedules. Each small schedule ε_i consists of a sequence of pairs such that the sending processes are all B-symmetric and the receiving processes are all B-symmetric. Part of the proof is to show the existence of such a schedule. The proof is by induction on the number of small schedules. The induction hypothesis is that there is a schedule φ consisting of k small schedules $\varepsilon_1 \dots \varepsilon_k$ such that all B-symmetric processes have symmetric opportunities after execution of φ.

Base case:

The null schedule is the base case for $k = 0$. We show that B-symmetric processes initially have symmetric opportunities. Suppose that there is a schedule that starts with process p sending a message to process q. If process r is B-symmetric to p then there is an automorphism ρ such that $r = \rho(p)$, so there is a schedule that starts with process r sending a message to $\rho(q)$. Thus,

$$(\forall q : q \in P : q \in OUTPUT(p) \iff \rho(q) \in OUTPUT(r))$$

Suppose that $q = out(p, n)$. Since B-symmetry preserves *out*,

$$\rho(q) = out(\rho(p), n) = out(r, n).$$

Thus,

$$(\forall n : n \in NAMES : out(p, n) \in OUTPUT(p) \iff out(r, n) \in OUTPUT(r)).$$

By the same argument,

$$(\forall n : n \in NAMES : in(p, n) \in INPUT(p) \iff in(r, n) \in INPUT(r)).$$

Thus, B-symmetric processes initially have symmetric opportunities.

Induction step:

Assume that the induction hypothesis is true for φ, a schedule consisting of k small schedules. If the system is deadlocked then B-symmetric processes remain in the same state. Otherwise, some process p is able to send a message to some process q. Suppose that $p = in(q, n)$ and $q = out(p, m)$. Let the small schedule ε_{k+1} consist of a sequence of pairs $\langle r, out(r, m)\rangle$ for each process r that is B-symmetric to p. By the induction hypothesis, all B-symmetric processes have symmetric opportunities, so all processes B-symmetric to p are ready to send a message to their n-neighbor and all processes B-symmetric to q are ready to receive a message from their m-neighbor. No process in

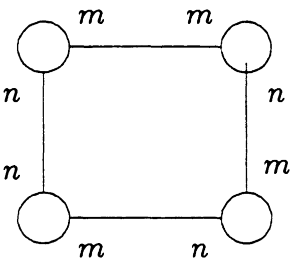


Figure 8.1: No Selection

strict CSP can be both ready to send and ready to receive, so each process involved in ε_{k+1} either sends one message or receives one message, but not both. Thus, $\varphi\varepsilon_{k+1}$ does not deadlock and the states of the processes involved in steps in ε_{k+1} are independent of the order in which they are scheduled within ε_{k+1}.

Since ρ is a homomorphism, $\rho(\varphi)\rho(\varepsilon_{k+1})\langle\rho(p),\rho(q)\rangle$ is a legal schedule if $\varphi\varepsilon_{k+1}\langle p,q\rangle$ is a schedule. By the induction hypothesis and Lemma 46, each process is in the same state after $\rho(\varphi)\rho(\varepsilon_{k+1})$ as after $\varphi\varepsilon_{k+1}$, so if $\varphi\varepsilon_{k+1}(p,q)$ is a schedule, $\varphi\varepsilon_{k+1}\langle\rho(p),\rho(q)\rangle$ is a schedule. Therefore, if p can send a message to q then $\rho(p)$ can send a message to $\rho(q)$, so B-symmetric processes have symmetric opportunities after execution of ε_{k+1}. Thus, the induction hypothesis is true for $\varphi\varepsilon_{k+1}$, which has $k+1$ small schedules. ∎

Dissimilar processes in a system in strict CSP cannot always tell themselves apart just as dissimilar processes in fair systems in S cannot always tell themselves apart. This is because there are systems in strict CSP for which any program will deadlock. For example, consider a system in which for every initial state and every name n there are processes p and q with that initial state such that $out(p,n) = q$ and $out(q,n) = p$. Figure 8.1 shows such a system. Some process r must send a message to its n-neighbor, for some n, before any program can receive a message. A strict CSP process that attempts to execute an output instruction must finish it before executing any other instruction. Therefore, processes p and q with initial state $init(r)$ such that $out(p,n) = q$ and $out(q,n) = p$ will try to send to each other and deadlock, so every program causes some pair of processes to deadlock. We can design such a system so that for any two processes in the same initial state there is some n such that their n-neighbors are in different initial states. Thus, even though this system is very asymmetric, no algorithm can cause every process to learn its similarity label.

Extended CSP differs in power from strict CSP in two ways. First, deadlock can always be avoided. Second, neighboring processes are always dissimilar. Therefore, in contrast with the result of Theorem 47, B-symmetric processes in extended CSP are not necessarily similar. A system in extended CSP, as in L, is equivalent to a family of systems. We converted systems in L to families of systems in S, but we have no such simpler model into which we can convert systems in extended CSP. Therefore, we convert a system in extended CSP to a family of systems in extended CSP in which no two neighboring processes have the same state.

We will discover how to compute similarity labelings by considering execution of several types of instructions by similar processes, as we did in Chapters 4 and 7. As in the shared variable models, similar processes must have the same initial state, otherwise a program in which no process ever changes state will prevent supposedly similar processes from having the same state infinitely often. Thus,

$$p \sim q \Rightarrow init(p) = init(q). \tag{8.1}$$

When similar processes receive a value from an n-neighbor, the values received must be the same, so

$$p \sim q \Rightarrow in(p,n) \sim in(q,n). \tag{8.2}$$

Each process can label values sent with the name that the process gives the receiving process, so

$$p \sim q \Rightarrow ((p = out(r,n) \wedge q = out(s,n)) \Rightarrow r \sim s). \tag{8.3}$$

As in the shared variable models, a supersimilarity labeling must satisfy the above properties, i.e. processes with the same label must have the same environment under the labeling. An environment is a set of tuples, each denoting the label of a neighbor and the name that each process gives the other.

$$\begin{aligned} &CSPenv_\Psi(p) \equiv \\ &\quad \{\langle m,n,\alpha\rangle : m,n \in NAMES, \alpha \in LABELS : \\ &\qquad\qquad (\exists q : q \in P : \Psi(q) = \alpha \wedge q = in(p,m) \wedge p = out(q,n))\} \end{aligned}$$

Since (8.1), (8.2) and (8.3) must hold for any set of nodes that behave similarly, a similarity labeling Ψ must ensure that $\Psi(p) = \Psi(q) \Rightarrow CSPenv_\Psi(p) = CSPenv_\Psi(q)$.

Just as in L, a labeling that gives the same environment to every pair of processes with the same label is not necessarily a supersimilarity labeling. A supersimilarity labeling for a system in extended CSP, like that for a system in L, must ensure that neighbors are dissimilar.

Theorem 48. *For any system Σ in extended CSP, Ψ is a supersimilarity labeling for Σ if*

$$\begin{aligned}(\forall p, q : p, q \in P : \Psi(p) = \Psi(q) \Rightarrow (&p \text{ is B-symmetric to } q\\ &\wedge CSPenv_{\Psi}(p) = CSPenv_{\Psi}(q)\\ &\wedge (\forall n : n \in NAMES : in(p, n) \neq q)))\end{aligned}$$

Proof: The round-robin schedule of Theorem 47 causes processes with the same label to have symmetric opportunities infinitely often for any program, as we now show. By Lemma 46, it follows that processes are in the same state infinitely often, so Ψ is a supersimilarity labeling.

Just as in Theorem 47, the induction hypothesis is that there is a schedule φ consisting of k small schedules $\varepsilon_1 \ldots \varepsilon_k$, except that here processes in each ε_i are not only B-symmetric but have the same label under Ψ. In addition, for each automorphism ρ, $\Psi(p) = \Psi(q)$ implies that p and q have the same state after execution of φ or of $\rho(\varphi)$. We show that all processes have symmetric opportunities after execution of φ. The proof is by induction on k.

Base case:

The base case is $k = 0$. Processes with the same label are B-symmetric, by hypothesis. The base case of the proof of Theorem 47 shows that B-symmetric processes initially have symmetric opportunities.

Induction step:

Assume that the induction hypothesis is true for φ, a schedule consisting of k small schedules. We will show that it is true for a schedule consisting of $k+1$ small schedules. If the system is deadlocked then processes with the same label remain in the same state. Otherwise, some process p is able to send a message to some process q. Suppose that $p = in(q, n)$ and $q = out(p, m)$. Let the small schedule ε_{k+1} consist of a sequence of pairs of processes $\langle r, out(r, m)\rangle$ for each process r such that $\Psi(r) = \Psi(p)$. By the induction hypothesis, for each such r, $CSPenv_{\Psi}(p) = CSPenv_{\Psi}(r)$ so $\Psi(out(p, m)) = \Psi(out(r, m))$. Since $\Psi(r) \neq \Psi(out(r, m))$, the processes in ε_{k+1} sending messages are disjoint from the processes receiving messages, so $\varphi\varepsilon_{k+1}$ does not deadlock. Moreover, no process appears in ε_{k+1} more than once, all sending processes in ε_{k+1} execute the same instruction and send the same value to their n-neighbor, and all receiving processes in ε_{k+1} execute the same instructions and receive the same value from their m-neighbor. Therefore, the states of the processes involved in steps in ε_{k+1} are independent of the order in which they are scheduled within ε_{k+1}.

Since $\Psi(r) = \Psi(p)$, r and p are B-symmetric, so there is an automorphism ρ of the system such that $\rho(p) = r$ and if $\varphi\varepsilon_{k+1}\langle p, q\rangle$ is a schedule then $\rho(\varphi)\rho(\varepsilon_{k+1})\langle\rho(p), \rho(q)\rangle$ is a schedule. By the induction hypothesis, p and r have the same states after execution of φ as they would have after execution of $\rho(\varphi)$. Since the states of the processes involved in steps in ε_{k+1} are independent of the order in which they are scheduled within ε_{k+1}, p and r have the same states after execution of $\varphi\varepsilon_{k+1}$ as they would have after execution of $\varphi\rho(\varepsilon_{k+1})$. By transitivity, $\rho(\varphi)\rho(\varepsilon_{k+1})\langle\rho(p),\rho(q)\rangle$ has the same effect as $\varphi\varepsilon_{k+1}\langle\rho(p),\rho(q)\rangle$. Thus, if $\varphi\varepsilon_{k+1}\langle p,q\rangle$ is a schedule then $\varphi\varepsilon_{k+1}\langle\rho(p),\rho(q)\rangle$ is a schedule, i.e. if p can send a message to q then $\rho(p)$ can send a message to $\rho(q)$, so processes with the same label under Ψ have symmetric opportunities after execution of ε_{k+1}. Therefore, the induction hypothesis is true for $\varphi\varepsilon_{k+1}$, which has $k+1$ small schedules. ∎

Algorithm 1 can be used to find a labeling such that processes have the same label if and only if they have the same environment and initial state. This labeling is a supersimilarity labeling if no two neighboring processes have the same label. First, we will show how processes in extended CSP can replay an execution of Algorithm 1. Then, we will show how to ensure that no two neighboring processes have the same label under the labeling produced by Algorithm 1. Thus, we will have shown how each process can find its similarity label.

Processes in extended CSP replay an execution of Algorithm 1 as follows. For each process, let *pec*$[j]$ be the process's label under Φ_j for each $j \leq i$, where i is the step most recently replayed. Each process replays step $i+1$ by sending *pec*$[i]$ to each neighbor and receiving labels from each neighbor in return, using these labels as its environment under Ψ_i. If Algorithm 1 does not relabel processes labeled *pec*$[i]$ in step $i+1$ then *pec*$[i]$ = *pec*$[i+1]$. If Algorithm 1 does relabel processes labeled *pec*$[i]$ in step $i+1$ then the process uses its environment under Φ_i to pick *pec*$[i+1]$. Algorithm 4 (page 93) shows the program template that lets each process learn its label under the labeling produced by Algorithm 1.

Lemma 49. *Algorithm 4 causes each process in a system in extended CSP to learn its label under the labeling produced by Algorithm 1.*

Proof: The loop invariant of Algorithm 4 is that *pec*$[i] = \Phi_i(\mathit{self})$ for each $i \leq j$. Thus, each process *self* sends to each neighbor a pair consisting of $\Phi_k(\mathit{self})$ and the name by which *self* calls the neighbor. Thus, after *self* receives from its n-neighbor, *neighbor*$[n]$ contains a pair $\langle\Phi_j(in(\mathit{self},n)),m\rangle$, where m is the name that its n-neighbor calls it. In short, *neighbor* contains $CSPenv_{\Phi_j}(\mathit{self})$. Function *next_label* is not explicitly defined, but computes the next label that Algorithm 1 would assign to *self*. Since *pec*$[j+1]$ will become $\Phi_{j+1}(\mathit{self})$, the loop invariant will be maintained when j is incremented.

No process finishes step j until all elements of *need* are false and of *given* are true. Thus, no process finishes step j until it has sent its results from step

Algorithm 4. Template for processor *self*.

```
j := 0;
pec[0] := init(self);
do Algorithm 1 does not relabel processes labeled pec[j] in step j + 1
    ∧ j < number of steps in Algorithm 1 →
            pec[j + 1] := pec[j];
            j := j + 1;
□   Algorithm 1 does relabel processes labeled pec[j] in step j + 1 →
            for n ∈ NAMES : need(n) := true; given(n) := false rof;
            do for each n
            □ in(self, n)?neighbor[n] ∧ need(n) → need(n) := false
            □ out(self, n)!(pec[j], n) ∧ ¬given(n) → given(n) := true
            od;
            pec[j + 1] := next_label(pec[j], neighbor, j);
            j := j + 1
od
```

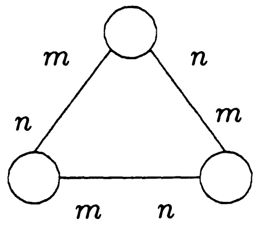


Figure 8.2: A System

```
if  in(self, n)?x → y := 0;
                    out(self, m)!0
    out(self, m)!1 → y := 1;
                    out(self, n)?x
fi
```

Figure 8.3: A Program

$j - 1$ to all its neighbors and has received results from step $j - 1$ from all its neighbors. Consequently, the slowest process is no more than one step behind any of its neighbors and all processes will finish Algorithm 4. Therefore, all processes will eventually know their label under the labeling produced by Algorithm 1. ∎

The program in Figure 8.3 running on the system in Figure 8.2 shows how a system can ensure that neighboring processes have different environments. Here, each process has access to two channels, named n and m. After executing the program in Figure 8.3, each process is in a unique state, since the first process to receive a message has $x = 1$ and $y = 0$, the first process to send a message has $y = 1$, the third process has $x = 0$, and x of the first process to send equals y of the third process. Thus, none of the processes are similar to each other. Note that the system in Figure 8.2 is G-symmetric.

The program of Figure 8.2 can be generalized to be an algorithm that relabels a system in CSP to produce a member of a family of systems. Each member of this family has the property that no supersimilarity labeling of the system gives the same label to neighboring processes. Each process numbers the messages it sends, so that the first message is numbered 1, the second 2, and so on. Each process saves a list of received messages in the order that they were received. This list becomes a part of the new state.

The relabeling algorithm consists of a single loop with a pair of guards for each edge. One guard receives a message along the edge while the other sends a message. A process receives only one message along each edge.

$relabel(self) \equiv$
  for $n \in NAMES : read(n)$:=**false** **rof**;
  $my\_count := 0$;
  do for each n;
  $\square$ $in(self, n)?count \wedge \neg read(n) \rightarrow list := list \circ \langle n, count\rangle$;
  $read(n) :=$ **true**
  $\square$ $out(self, n)!my\_count \wedge my\_count < |NAMES|$
  $\rightarrow my\_count := my\_count + 1$
  od

The following lemmas imply that no two neighboring processes are given the same label by any supersimilarity labeling Ψ that uses the result of executing relabel as the initial state.

Lemma 50. *After relabel has executed there is no cycle of processes with the same state such that each is an i-neighbor of the next.*

Proof: By contradiction. Assume that there is a cycle of processes with the same state such that each is an i-neighbor of the next. Since the processes are all in the same state, there is an integer j such that each process has received a message with $count = j$ from its i-neighbor. Therefore, each sent a message with $count = j$. In order for a process to both send and receive a message with $count = j$, it had to send before receiving. However, not all the processes in the cycle could have sent before receiving, because some process had to send last. All other processes have sent, and CSP uses synchronous message-passing, so the process that sent last received a message with $count = j$ before it sent. Thus, it sent a message with $count > j$, a contradiction. ∎

Lemma 51. *If init is the state of system Σ after relabel has executed, and Ψ is a labeling such that*

$$(\forall p, q : p, q \in P : \Psi(p) = \Psi(q) \Rightarrow (p \text{ is B-symmetric to } q \\ \wedge\ CSPenv_{\Psi}(p) = CSPenv_{\Psi}(q))$$

then Ψ is a supersimilarity labeling for Σ.

Proof: We will show, by contradiction, that no two neighboring processes are given the same label by Ψ. Thus, by Theorem 48, Ψ is a supersimilarity labeling for Σ.

Suppose $\Psi(p) = \Psi(q)$ and q is the i-neighbor of p. Then, by the definition of *CSPenv*, q has the same label under Ψ as its i-neighbor. By induction, there is an infinite chain of processes, each with the same label under Ψ as its i-neighbor. There are only a finite number of processes, so this infinite chain must be a cycle. However, by Lemma 50, one of these processes must have a different state after execution of *relabel* then the others. By Lemma 46, these processes are not B-symmetric, so Ψ gives them different labels; a contradiction. ∎

The previous lemma implies that if Ψ is a supersimilarity labeling of the system after *relabel* has executed then Ψ is a supersimilarity labeling of the original system. Thus, *relabel* converts a system into a member of a family of homogeneous systems with the property that if Ψ gives processes with the same label the same environment under Ψ then Ψ is a supersimilarity labeling. Moreover, each member of the family has a similarity labeling that can be computed using an algorithm like Algorithm 1. This algorithm can be replayed to let each process determine its label under the similarity labeling of the member of the family resulting from execution of *relabel*.

Theorem 52. *There is a selection algorithm for a system Σ if and only if there is no supersimilarity labeling Ψ such that for all processes p there is a process q such that $\Psi(p) = \Psi(q)$.*

Proof: If there is such a supersimilarity labeling then each process has another process similar to it, so, by Theorem 4, there is no selection algorithm. Otherwise, by Lemma 51, execution of *relabel* will cause the system state to be such that Algorithm 1 would find a supersimilarity labeling for the system. Thus, by Lemma 49, Algorithm 4 causes every process to learn its label under the supersimilarity labeling created by Algorithm 1. By Lemma 44, there is a set of process labels *ELITE* such that each member of the family has exactly one process with a label in *ELITE* and so, by Theorem 19, there is a selection algorithm.

Although Lemma 44 was proven for L, it turns out that it holds for CSP as well, since its dependency on L is limited to its use of Lemma 34 and 40. Lemma 34 states that if one processor is uniquely labeled by a supersimilarity labeling than all are, while Lemma 40 states that if one processor labeled α by a supersimilarity labeling Ψ has a path to a node labeled β then all do. Both lemmas hold for CSP, so the algorithm of Lemma 44 can be used to find *ELITE*. ∎

The original problem proposed by Bouge was to decide when systems of B-symmetric processes can select a leader. This occurs when B-symmetric processes are dissimilar. Given any system of B-symmetric processes in strict CSP, we can decide if processes are dissimilar by enumerating all labelings and making sure that each supersimilarity labeling gives some process a unique label. Although examining each labeling takes exponential time in the size of the system, the selection problem for some systems can be solved much faster. For example, a ring of j B-symmetric processes has a selection algorithm only if j is prime, otherwise there is a supersimilarity labeling with i labels, where i is a divisor of j, such that j/i processes have each label.

One measure of the usefulness of similarity is the fact that Bouge was not able to completely solve the selection problem for CSP, though he was able to solve it for a number of special cases. However, similarity permits solutions that are simple and relatively easy to understand. Thus, similarity provides a powerful tool for solving arbitration and synchronization problems in distributed systems, regardless of the model being used.

Chapter 9

Application to Related Problems

Similarity is related to a number of concurrent programming problems in addition to the selection problem. In this chapter we show how the similarity relation sheds light on three of these problems.

9.1. Mutual Exclusion

The mutual exclusion problem is a central problem in concurrent programming and one of the earliest to be explored[D65]. Given a set of processors and a set of program-counter states called the *critical region*, the mutual exclusion problem is to find a program such that

- no more than one processor at a time is ever in the critical region,
- if a set of processors is attempting to enter the critical region then one of them will eventually enter it,
- processors outside the critical region can halt without affecting other processors and
- there is no static ordering of processors.

If two processors in a system are similar, then no program exists that can guarantee that their states differ. If one processor is in the critical region then the other can be, too. Thus, no processor can be guaranteed entry to the critical region unless it is similar to no other processor. Assuming that all processors are to have an opportunity to enter the critical region, no two processors can be similar. Moreover, since processors outside the critical region can halt without affecting other processors, some processor may never execute. Thus, no two processors in any subsystem may be similar.

In Dijkstra's original formulation, the lack of static ordering of processors was called symmetry. We earlier noted that the usual definition of symmetry in a distributed system is G-symmetry. The following theorem shows that G-symmetric nodes in a system in S are similar.

Theorem 53. *G-symmetric nodes in a system in S are similar.*

Proof: Recall that two nodes are G-symmetric in a system network graph if there is an automorphism of the graph mapping one node to the other, where the automorphism preserves labelings (in our case, *init* and *naming*). As we now show, the equivalence classes of nodes under G-symmetry form a supersimilarity labeling. Let σ be an automorphism of a system Σ and Ψ be some labeling on Σ such that $\sigma(x) = y \Leftrightarrow \Psi(x) = \Psi(y)$. Since σ preserves *init*,

$$\Psi(x) = \Psi(y) \Rightarrow init(x) = init(y).$$

Since σ preserves *naming*, for any processors p and q

$$\sigma(p) = q \Rightarrow \sigma(nbr(n,p)) = nbr(n,q).$$

Since $\sigma(x) = y \Leftrightarrow \Psi(x) = \Psi(y)$,

$$\Psi(p) = \Psi(q) \Rightarrow \Psi(nbr(n,p)) = \Psi(nbr(n,q)).$$

Thus,

$$\begin{aligned}&(\forall x, y : x, y \in P :\\ &\qquad \Psi(x) = \Psi(y) \Rightarrow (Env_\Psi(x) = Env_\Psi(y) \wedge init(x) = init(y))).\end{aligned}$$

Since σ is an automorphism,

$$(\forall u \in V : (\forall p : p \in P : \sigma(u) = nbr(n,p) \Rightarrow v = nbr(n, \sigma(p)))$$

so

$$\begin{aligned}&\sigma(u) = v\\ &\Rightarrow (\forall p : p \in P : u = nbr(n,p) \Rightarrow (\exists q : q \in P : v = nbr(n,q) \wedge \sigma(p) = q))\end{aligned}$$

and

$$\begin{aligned}&\Psi(u) = \Psi(v)\\ &\Rightarrow (\forall p : p \in P : u = nbr(n,p) \Rightarrow (\exists q : q \in P : v = nbr(n,q) \wedge \Psi(p) = \Psi(q))).\end{aligned}$$

Thus,

$$\begin{aligned}&(\forall x, y : x, y \in P :\\ &\qquad \Psi(x) = \Psi(y) \Rightarrow (Env_\Psi(x) = Env_\Psi(y) \wedge init(x) = init(y))).\end{aligned}$$

Since Σ is in S, by Theorem 7, Ψ is a supersimilarity labeling of Σ. Therefore, G-symmetric nodes are similar in systems in S. ∎

Since a solution to the mutual exclusion problem has no similar processors, there is no G-symmetric solution to the mutual exclusion problem. However, Dijkstra's definition of symmetry has no relation to the topology of the solution system, only to the program. Dijkstra's solution to the mutual exclusion problem permitted each processor to have a unique identifier, but the choice of processor to enter the critical region depended only on the execution sequence and did not give priority to any processor. Thus, Dijkstra's definition of symmetry is different from G-symmetry.

9.2. Generating Unique Names

The second problem that similarity can be used to solve is that of finding a unique name for each processor in a system. Suppose each processor p has an uninitialized integer-valued local variable id_p. The problem is to ensure that for all processors p and q, $id_p \neq id_q$. If two processors are similar then they can have the same state infinitely often, so there is no algorithm that can give them unique names. If no two processors are similar then the similarity labeling gives each processor a unique label, so by using labels as names, each processor can find a unique name by using an algorithm to find its label under the similarity labeling. Thus, for any system we can either find an algorithm to uniquely name processors or prove that no such algorithm exists.

It does not suffice simply to elect a leader and have it number the rest of the processors. It is possible (as shown in Figure 6.3) for a system to have a unique processor and yet have a pair of similar processors. In such a system, there is no way for the leader to give unique numbers to the similar processors.

9.3. Dining Philosophers

Similarity can be used to prove the claim [LR80]:

DP: There is no symmetric, distributed, deterministic solution to the Dining Philosophers problem in L.

In the Dining Philosophers problem [D71], five philosophers alternately think and eat.

- Philosophers eat at a table with a plate of spaghetti for each philosopher.
- There is a fork between every adjacent pair of plates. A philosopher needs the fork on both sides of its plate to eat.

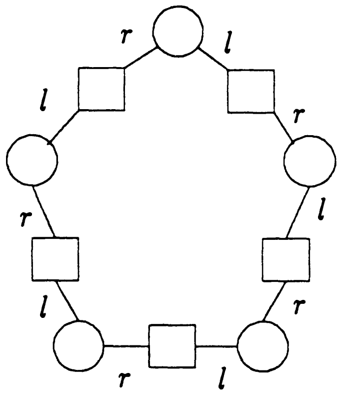


Figure 9.1: Five Dining Philosophers.

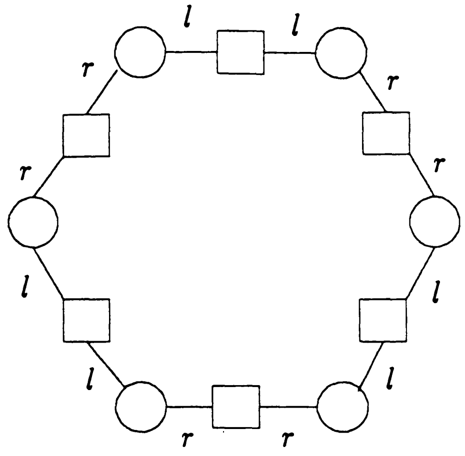


Figure 9.2: Six Dining Philosophers.

- Only one philosopher can use a particular fork at any given time.

Thus, two adjacent philosophers cannot eat at the same time.

The system in Figure 9.1 might be proposed as a solution to the Dining Philosophers problem, and it is distributed and G-symmetric.[1] The round-robin schedule $p_1\, p_2\, p_3\, p_4\, p_5\, \ldots$ causes all processors to behave similarly, hence all are similar. Any time one philosopher is eating, all are, which violates the problem constraints. Thus, there is no program that can solve the Dining Philosophers problem for this system [L80]. This argument, however, is not a proof of DP, since some other interconnection of processors might be a solution.

It is disturbing that a small change to the Dining Philosophers problem—changing the number of philosophers—allows a distributed, symmetric, deterministic solution:

DP′: There is a symmetric, distributed, deterministic solution to the six-philosopher Dining Philosophers problem in L.

Figure 9.2 shows a system that is distributed and G-symmetric for which there is a solution to the six-philosopher problem. This is because processors in L that give the same variable the same name are dissimilar.

Similarity can be used to prove both DP and DP′, as we now show. A distributed, G-symmetric system in L with a prime number of G-symmetric

[1] G-symmetry is the definition of symmetry used in DP [L85].

processors must behave just like the same system in S. Any distributed, G-symmetric solution to the Dining Philosophers problem must be such a system.

Theorem 54. *Let Σ be a distributed, G-symmetric, deterministic system in L containing an equivalence class C of j G-symmetric processors, where j is prime. Then, all j processors in C are similar.*

Proof: The automorphisms of the system graph of Σ that permute the processors in C form a group. Processors in C are G-symmetric to each other, so for any pair of processors in C there is an automorphism in the group that maps one to the other. Since j is prime there is a single member of this group that can generate a group that can map any element of C to any other element of C. This automorphism σ is of order j, i.e. σ^j is the identity mapping. The nodes of Σ are partitioned into equivalence classes by σ, and these equivalence classes define a supersimilarity labeling Ψ by the argument of Theorem 53. Since σ is of order j, the number of nodes in each equivalence class must divide j. However, j is prime, so the number of nodes in each equivalence class must be either 1 or j. If variable v has k n-neighbors labeled α then any variable labeled $\Psi(v)$ by Ψ is G-symmetric to v and also has k n-neighbors labeled α. Thus, the number of processors labeled α is equal to k times the size of the equivalence class of v. Since the equivalence classes are of size one or of size j, $k = j$ or $k = 1$. The system is distributed, so $k \neq j$. Thus, $k = 1$, i.e. no two processors with the same label under Ψ give the same variable the same name. By Theorem 33 a supersimilarity labeling for a system in S is a supersimilarity labeling for that system in L if no two processors with the same label give the same variable the same name. Thus, all processors in C are similar. ∎

Since five is a prime number, it follows that in any distributed, G-symmetric, deterministic solution to the Dining Philosophers problem in L, the five processors representing philosophers must be similar.

However, adjacent philosophers must be dissimilar in any solution to the Dining Philosophers problem. If philosophers are similar then there is a schedule that causes all to be eating whenever one is eating, so any solution to the Dining Philosophers problem must make adjacent philosophers be dissimilar. However, five G-symmetric processors are similar by Theorem 54, so there is no solution and DP is true.

DP$'$ relies on the fact that the number of philosophers is composite, so Theorem 54 does not apply. Alternate philosophers put their backs to the table, as in Figure 9.2, so that each philosopher's right fork is its neighbor's right fork. Philosophers now form two equivalence classes: those with their backs to the table and those facing the table. Variables also fall into two equivalence classes: those that are right forks to both their neighbors and those that are left forks to both their neighbors. All philosophers are G-symmetric, since automorphisms corresponding to a reflection through a line through a variable and to rotation about the axis map each processor to any other. However, adjacent philosophers are dissimilar, since similar processors in L cannot give the same variable the same name.

A simple algorithm for six philosophers in the system of Figure 9.2 is for each philosophers to wait until its right fork is available, pick it up, wait until its left fork is available, pick it up, eat, then put all forks down. Waiting for a fork and picking it up is accomplished by repeatedly attempting to lock a variable until success is achieved. Putting down a fork is accomplished by unlocking the variable. This algorithm does not deadlock because the variables are in two equivalence classes. A philosopher with the left fork is eating, so it will eventually put the fork down. Thus, a philosopher with a right fork that is waiting for a left fork will eventually acquire the fork and eat (modulo the lack of fairness), eventually putting the right fork down. Therefore, a philosopher waiting for a right fork will eventually acquire it and, by previous argument, eventually eat. The only way a philosopher will not be able to acquire a variable is if the other philosopher eats infinitely often.

Earlier we argued that adjacent philosophers must be dissimilar in order for the Dining Philosophers problem to have a solution. It turns out that the converse is also true.

DP$''$: A system can solve the Dining Philosophers problem if and only if philosophers that share a fork are dissimilar.

To see this, assume that every pair of philosophers that share a fork are dissimilar. Each processor can find its label under the similarity labeling and assign a priority to itself based on the label. Thus, no two philosophers that share a fork will have the same priority. This is exactly the condition required by Chandy and Misra to solve the Drinking Philosophers problem, hence the Dining Philosophers problem [CM84]. By DP$''$, this condition is not only sufficient for solving the Dining Philosophers problem, but necessary.

9.4. Similarity as a Definition of Symmetry

We believe that similarity is the essence of symmetry in a distributed system. One argument supporting this is its usefulness in solving the fundamental problems in distributed systems, as demonstrated above. Another argument is that similarity is a generalization of a proof technique that has been frequently used to show that processors were indistinguishable–using round-robin schedules to show that processors can have the same state infinitely often. The first such proof showed that symmetric intelligent graphs were indistinguishable [RFH72]. Other proofs showed that certain configurations of processors in CCS [MM79] (a message-passing model much like extended CSP) were not capable of "finding the center", i.e. selecting a leader [A80]. A proof outline for DP (above) also used round-robin schedules [LR80]. These proofs were similar in purpose and technique to the proofs of Theorems 7 and 33, and we first began to study similarity because of them. Thus, similarity is an idea which, though never before formalized, has often been useful in proofs about distributed systems.

One reason that G-symmetry, instead of similarity, has often been used as the definition of symmetry for distributed systems is that in many models G-symmetric components are always similar. For example, Theorem 53 showed that G-symmetric nodes in a system in S are similar. Also, Theorem 47 showed that B-symmetric processors in strict CSP are similar, and B-symmetry implies G-symmetry.

As we have seen, there are many models in which G-symmetric nodes are not necessarily similar. We say that a system Σ can *break symmetry* if nodes in Σ that are G-symmetric are not similar. Theorem 47 and Theorem 53 can be restated as saying that systems in strict CSP and S cannot break symmetry. Systems in L can break symmetry, as shown by the systems of Figures 1.1 and 1.3. These systems are G-symmetric, yet none of their components are similar. Systems in extended CSP can also break symmetry, as shown by the system of Figure 8.2. L differs from S in that processors in L that give the same variable the same name are not similar, while extended CSP differs from strict CSP in that neighboring processors in extended CSP are not similar. These differences reflect the properties of models that break symmetry.

Another common characteristic of the symmetry-breaking models that we have studied is that the selection problem for a system Σ in such a model must be reduced to the generic selection problem for a family of systems derived from Σ. There are usually a number of different ways that a graph can be labeled so that adjacent nodes have different labels. Requiring adjacent nodes to be dissimilar seems to be the heart of symmetry breaking models, so these models will naturally produce families of systems after the symmetry breaking instructions.

There are a number of possible objections to using similarity as the definition of similarity in distributed systems. One is that similarity is too limited a concept of having the same behavior. For example, we might be concerned with sequences of operations instead of with states. However, we saw in Section 8 that similarity can be used to characterize other semantic definitions of symmetry. We believe that similarity is not a limited concept but that it can be used to explain any other semantic definition of symmetry.

Another objection might be that we allow any processors that are not identical to be completely different. It might be argued that some systems consist of processors that are almost, but not quite, the same, and that we cannot adequately represent such systems. For example, both producers and consumers in the producers and consumers problem take buffers, process them, and return them. The only difference is that producers process a buffer by filling it, while consumers process a buffer by emptying it. A similar type of problem was solved in Section 8, where every process in a CSP program runs a different part of the program, by factoring program differences into the network topology. Alternatively, we might have attempted to change the instructions allowed or performed some other change making the system amenable to our techniques. Thus, it is frequently possible to change a model that does not meet our assumptions into one that does. However, similarity is limited by the fact that it is program independent, so it cannot necessarily be used to solve program-dependent problems.

The basic electronic components with which we build computers cannot break symmetry [CM73]. In fact, there is no deterministic implementation of locking for asynchronous processors, since it can take arbitrarily long for an asynchronous processor to decide whether or not it has acquired a lock. Synchronous systems can solve this problem by defining an ordering on processors neighboring each variable or by using a schedule as an arbiter, but both these solutions introduce asymmetry. Since the electronics with which we implement distributed systems cannot break symmetry, any implementation of a system that can break symmetry has encapsulated the necessary asymmetry.

Any model that breaks symmetry cannot be implemented by a model that does not break symmetry without introducing asymmetry (such as ordering processors) or using probabilistic techniques. Thus, while there is a symmetric implementation of strict CSP [S84], an implementation of extended CSP requires asymmetry [B80] or random numbers [FR80].

Instructions that can break symmetry have the advantage that programs using them can be thought of as being symmetric, and it is generally easier to reason about symmetric systems. They have the disadvantage that they may trick the programmer into erroneously thinking that a program using them can be implemented symmetrically. The solution to the Drinking and Dining Philosophers problem [CM84] demonstrates a method in which systems are designed by explicitly encapsulating the necessary asymmetry. There, processors all execute the same program and do not have unique identifiers. The system can be thought of as symmetric, but the initial state is carefully designed to ensure fair use of resources by being asymmetric. This initial state is equivalent to an acyclic directed graph covering the system, giving an ordering for any two neighboring processors. Therefore, no two neighboring processors are similar. This method has the advantage of making the program easier to reason about without hiding the necessary asymmetry.

We have argued that all systems of processors require asymmetry in order to guarantee that processors are distinguished. While there are a number of models that seem to allow symmetric processors to be distinguished, these are just methods of hiding asymmetry. It is often useful to hide details that make it difficult to reason about programs, so these models are useful. Nevertheless, it is a mistake to expect that assumptions about symmetry made using these models can be propagated to the level of electrical circuits.

9.5. The Power of Probability

Probabilistic techniques have been used to break symmetry in distributed systems[IR81][LR80]. One example is a probabilistic solution to the Dining Philosopher's problem that is symmetric and distributed [LR80]. Each of the five philosophers is represented by a processor and each fork by a variable. Each fork is accessed by two philosophers, one of which calls it a right fork and one of which calls it a left fork. Philosophers randomly choose left or right, and attempt to pick up that fork (i.e. lock that variable). If they succeed then they pick up the other fork. If they fail to pick up either fork then they drop all forks and start again. If the probability of picking up a fork, including picking up the second fork after having picked up the first, is greater than some positive constant, then with probability 1.0 all philosophers will eventually pick up both forks and eat.

The above example illustrates the features of a probabilistic distributed program.

1. Each processor is able to independently choose random numbers.
2. "Safety", i.e. no matter which numbers are chosen, the program never produces the wrong behavior.
3. "Liveness", i.e. with probability 1.0, the program eventually produces the right behavior.

In the Dining Philosopher's problem used as an example, the wrong behavior is to have two adjacent philosophers eat at the same time; the right behavior is that each philosopher eventually eats. It is usually not possible to bound the time that a program will take to generate the right behavior.

This section contains two main results. The first is that similarity shows when a probabilistic algorithm can solve a problem that no deterministic algorithm can solve, since every system has a probabilistic selection algorithm. The second result is that allowing processors to choose random numbers is no more powerful than assuming randomly selected schedules. In one sense, probabilistic techniques are a way of avoiding the worst-case assumptions we have made about schedules in the definition of similarity.

The following algorithm is a probabilistic selection algorithm for a bounded fair system in S. Each processor randomly chooses a number. Processors exchange these numbers with each other, using shared variables, and so try to learn the set of numbers that have been chosen. Since the system is bounded fair, eventually the processors know that they have learned all the numbers that have been chosen. If there are more processors than numbers then the processors know that two or more processors chose the same number, so they start the algorithm over. If there are as many numbers as processors, then the processor that chose the largest number is selected.

If the system is fair, but not bounded fair, then it is impossible to tell whether two similar processors sharing the same variable have guessed the same number or whether one of them has not yet executed. This problem is solved by having processors wait a random amount of time, then starting over. The number that each processor communicates is a sequence of all its earlier choices. Sequences are compared lexicographically to determine the largest, so numbers from different rounds can be compared in such a way that once a processor has the largest number, its number will always be largest. Unfortunately, the sequences can get arbitrarily large, so this solution requires shared variables of unbounded size. Nevertheless, it is a probabilistic solution to the selection problem for any fair system in S and thus for any fair system in L.

There is a simpler probabilistic selection algorithm for fair systems in L that uses shared variables of bounded size. In it, each processor guesses a number between 1 and some constant d that becomes its initial state, converting the system into a member of a homogeneous family. Each label given by the similarity labeling of this family falls into one of three sets:

- *FAIL* is the set of all labels belonging to systems with two or more similar processors.

- *ELITE* is exactly one label from each system with every processor uniquely labeled.

- *PEON* is the set of all other labels.

Processors use Algorithm 3 to find their similarity label. A processor with a similarity label in *FAIL* starts the algorithm over. A processor with a similarity label in *ELITE* selects itself. Otherwise, some other processor is selected. By Lemma 40, if one processor has a similarity label in *FAIL*, then all do, so either all processors start the algorithm over or none do. Also by Lemma 40, there is an set *ELITE* such that each member of the family with every processor uniquely labeled has exactly one process with a label in *ELITE*. Thus, processors in a system in L can probabilistically select a leader using shared variables of bounded size.

Since most methods of generating random numbers require asymmetry, probabilistic techniques can be considered a way of packaging asymmetry. However, a better way of thinking about them is that they are a way of changing some of our assumptions about schedules. Instead of assuming that the worst case schedule will occur, we assume a random distribution of schedules. In the rest of this section, we will show that assuming a random distribution of schedules is equivalent to using probabilistic techniques. This is nearly the same as Pnueli's description of probabilistic algorithms as those with "extremely fair" scheduling policies [P83].

First, we will review the probabilistic solution to the Dining Philosopher's problem given above. A deterministic version of this algorithm is one in which a philosopher tries to pick up his right fork and then his left fork. If the left fork is not available then the philosopher puts down the right fork and starts over. The reason why this deterministic algorithm does not work is that if all philosophers are in lock step then no philosopher ever gets both forks. Thus, in this example, probabilistic techniques are used to outwit the adversary schedule.

The above deterministic algorithm solves the Dining Philosopher's problem with probability 1.0 under the assumption that all fair schedules (or even all k-bounded fair, for k larger than the number of processors) have equal probability of occurring. Deadlock is not possible, since philosophers never wait for events. If no philosopher is eating then any philosopher can acquire its right fork. There is a non-zero probability that the philosopher will acquire its left fork, as well, since frequently a philosopher is holding its right fork and its left fork is free; occasionally that philosopher will acquire both forks since all fair schedules have equal probability of occurring. Thus, philosophers occasionally get both forks, so, with probability 1.0 there will be no starvation.

This algorithm can be generalized to obtain random sequences of bits in locking systems where every schedule has an equal probability of occurring. Each shared variable v has a bit r that is either 0 or 1. A processor can make a random choice by locking v, reversing r, unlocking v, then using the value of r as a random number. The probability of r being 1 depends on the number of processors sharing the variable, but will approach 0.5 as the number of processors gets large. If every schedule has an equal probability of occurring then each processor has the same chance of reading a 1 as any other processor.

A random sequence of bits can also be generated in a nonlocking system. Processors reverse r, as above, but do not lock v. Naturally, processors can interfere with each others' attempts to write on v, but this is irrelevant to the goal of generating random sequences of bits unless the schedules are close to being round-robin, in which case all processors could generate the same sequences of bits.

If some schedule is more likely to occur than others then there will be a bias in favor of some bit sequences. The techniques of Von Neumann[V51] can be used to convert a biased string of independent bits into an unbiased string of bits. This is done by picking pairs of random bits, throwing out $\langle 0,0\rangle$ and $\langle 1,1\rangle$ and letting $\langle 0,1\rangle$ represent 0 and $\langle 1,0\rangle$ represent 1. Since the probability of occurrence of these pairs is the same, we have a random number generator that produces 0 and 1 with equal probability. This can be used to generate random numbers of any given size by using it to generate each bit of the needed number. A biased string of bits that is not independently generated can still be used. A technique of Vazirani [V85] can be used to change the algorithm to behave the same as if the bits were independently generated. Thus, use of random number generators (and probabilistic techniques) has been reduced to the assumption that all schedules are equally likely.

Probabilistic algorithms can select a processor when a deterministic algorithm cannot only when some processors are similar. Thus, similarity is a measure of the power of probabilistic techniques. Probabilistic techniques are a way to avoid the adversary schedules that cause processors to behave similarly and are equivalent to assuming that the adversary schedules are, with probability 1.0, impossible.

Chapter 10

Conclusion

Within its intended domain of applicability, similarity is a powerful tool. It has proven useful in solving four important distributed synchronization problems: the selection problem, deciding whether or not a particular system can solve the mutual exclusion problem or generate unique ids for processors, and showing that there is no distributed, symmetric, deterministic solution to the Dining Philosophers problem. Since having the same behavior is intuitively an attribute of symmetric components in distributed systems, similarity is a characterization of symmetry.

One reason why similarity is useful is that it is easily computed for a wide range of models. This thesis showed how to compute the similarity relation for systems of processors communicating with shared variables and systems of processors communicating with message-passing, for systems in which processor could break symmetry and systems in which they could not, and for systems where relative processor speeds were bounded and systems where they were not. For each model we characterized supersimilarity labelings. If the model could not break symmetry then we used Algorithm 1 to find a similarity labeling Θ and then found a distributed algorithm that let each processor learn its label under Θ. If the model could break symmetry then we reduced the problem to one for a family of systems that could not break symmetry. This technique, developed in the main body of the thesis for systems in S and L, was successfully applied to systems in CSP, a very different model than S or L.

An important use of similarity is to compare different models of computation. For example, we can see that systems in L differ from systems in S only in that no two similar processors give the same variable the same name. In Section 9.4, we used similarity to characterize systems in which processors can break symmetry. The fact that the definition of breaking symmetry agrees so well with the way the term is commonly used is further justification of our claim that similarity captures the important characteristics of symmetry. We used similarity in Section 9.5 to argue that probabilistic algorithms are essentially the same as assuming a random distribution of schedules.

At present, similarity is more important to the theoretician than to the designer of distributed systems–almost all the problems we solved in this thesis were problems posed by theoreticians rather than system builders. Designers do not worry about building a system in which processors are similar because they usually design systems that are asymmetric. However, as larger and larger collections of processors are built, it will become more important for them to be symmetric, and the notion of similarity will be useful to ensure that the processors, though perhaps syntactically symmetric, are not similar. Thus, similarity may become generally useful to both theoreticians and practitioners.

Bibliography

[A85] Afek, Y. and E. Gafni. Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks. *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Systems* (1985), 186-195.

[A80] Angluin, D. Local and Global Properties in Networks of Processors. *Proceedings of 12th Conference on Principles of Progamming Languages* (1980), 82-93.

[B80] Bernstein, A. Output Guards and Nondeterminism in "Communicating Sequential Processes". *ACM Transactions on Programming Languages and Systems 2,* 2 (April 1980), 234-238.

[B84] Borge, L. Symmetric Election in CSP. Tech. Report 84-31, Laboratory of Information Theory and Programming, University of Paris (June 1984).

[B81] Burns, J.E. Symmetry in Systems of Asynchronous Processes. *Proceedings of 22nd Symposium on Foundations of Computer Science* (1981), 169-174.

[CM84] Chandy, K.M. and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems 6,* 4 (Oct. 1984), 632-646.

[CM73] Chaney, T.J., and C.E. Molnar. Anomalous Behaviour of Synchronizer and Arbiter Circuits. *IEEE Transactions of Computers, C-22*, 4 (April 1973), 421-422.

[D65] Dijkstra, E.W. Solution of a Problem in Concurrent Programming Control. *C. ACM 8*, 9 (Sept. 1965), 569.

[D71] Dijkstra, E.W. Hierarchical Ordering of Sequential Processes. *Acta Informatica 1* (1971), 115-138.

[D76] Dijkstra, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, (1976).

[DKR82] Dolev, D., M. Klawe and M. Rodeh, An O(n log n) Unidirectional Distributed Algorithm for Extrema-finding in a Circle. *Journal of Algorithms 3*, 3 (Sept. 1982) 245-260.

[FLP83] Fischer, M., N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Proceedings Second ACM Symposium on the Principles of Database Systems* (1983), 1-7.

[FR80] Francez, N. and M. Rodeh. A Distributed Abstract Data Type Implemented by a Probabilistic Communication Scheme. *21st Symposium on Foundations of Computer Science* (Oct. 1980), 373-379.

[G82] Garcia-Molina, H. Elections in a Distributed Computing System. *IEEE Transactions on Computers C-31,* 1 (Jan. 1982) 48-59.

[GLR83] Gottlieb, A., B.D. Lubachorsky, L. Rudolph. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems 5,* 2 (April 1983), 164-189.

[H78] Hoare, C.A.R. Communicating Sequential Processes. *C. ACM 21,* 8 (Aug. 1978), 666-677.

[H71] Hopcroft, J.E. An *n log n* algorithm for minimizing states in a finite automata. *Theory of Machines and Computations* Z. Kohavi and A. Paz, ed., Academic Press, New York, (1971) 189-196.

[IR81] Itah, A. and M. Rodeh. The Lord of the Ring, or Probabilistic Methods for Breaking Symmetry in Distributive Networks. RJ 3110, IBM (April 1981).

[L85] Lehmann, D.J. Private communication with F. B. Schneider, New Orleans, Jan. 1985.

[LR80] Lehmann, D.J., and M.O. Rabin. On the Advantages of Free Choice. *Proceedings of 12th Conference on Principles of Progamming Languages* (1980), 134-138.

[L80] Lynch, N. Fast Allocation of Nearby Resources in a Distributed System. *Proceedings of the Twelfth Annual ACM Symposium of Theory of Computing* (1980), 70-81.

[MM79] Milne, G.A. and R. Milner. Concurrent Processes and their Syntax. *J. ACM 26*, (1979), 302-321.

[P82] Peterson, G.L. An O(n log n) Unidirectional Algorithm for the Circular Extrema Finding Problem, *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan. 1982), 758-762.

[P83] Pnueli, A. On the Extremely Fair Treatment of Probabilistic Algorithms. *Proceedings of the Fifteenth Annual ACM Symposium of Theory of Computing,* (April 1983), 278-290.

[R82] Rabin, M.O. The Choice Coordination Problem. *Acta Informatica 17*, (1982), 121-134.

[RFH72] Rosenstiehl, P., J.R. Fiksel, and A. Holliger. Intelligent Graphs. *Graph Theory and Computing*, R. Read, ed., Academic Press, New York (1972), 219-265.

[S84] Sistla, A.P. Distributed Algorithms for Ensuring Fair Interprocess Communications. *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* (August 1984), 266-277.

[S82] Stark, E.W. Semaphore Primitives and Starvation-Free Mutual Exclusion. *J. ACM 29,* 4 (Oct. 1982), 1049-1072.

[V85] Vazirani, U.V. Towards a Strong Communication Complexity Theory, or Generating Quasi-Random Sequences from Two Communicating Slightly-Random Sources. *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing,* (May 1985), 366-378.

[V51] von Neuman, J. Various Techniques Used in Connection with Random Digits. Notes by G.E. Forsythe, National Bureau of Standards, Applied Math Series, Vol. 12, 36-38 (1951). Reprinted in von Neuman's Collected Works, Vol 5, Pergoman Press (1963), 768-770.