Symmetry and Similarity in Distributed Systems*

Ralph E. Johnson†
Fred B. Schneider‡

Computer Science Department
Cornell University

Abstract: Similarity is introduced as a model-independent characterization of symmetry. It can be used to decide when a concurrent system has a solution to the selection problem. It can also be used to compare different models of parallel computation, including differences in scheduling policy and instruction set, and the consequences of using randomization.

1. Introduction

Symmetry means different things to different people. Fundamentally, however, two systems should be considered *symmetric* if they behave identically, and two components of a system should be considered *symmetric* if they are indistinguishable. This is consistent with the definition of symmetry used in graph theory, where two graphs are considered symmetric if there is an isomorphism mapping one to the other and two components of a graph are symmetric if there is an automorphism of the graph mapping one component to the other.[1] The graph-theoretic definition of symmetry has been applied to concurrent systems [RFH72][A80], as have other definitions [B81][S82][B84]. Unfortunately, the various definitions are not equivalent, most are model-sensitive, and none completely captures the fundamental notion of symmetry given above as it applies to concurrent systems.

This paper introduces a characterization of symmetry—the *similarity relation*—that is well suited for understanding properties of concurrent systems. We show how to compute the similarity relation for a variety of systems. We use similarity to derive necessary and sufficient conditions for solving a generalization of the coordinated choice problem [R80], called the *selection problem*, where processors select exactly one *leader* by communicating with each other. Solutions to many other synchronization problems and to certain types of distributed programming problems can be found using similarity in the same way. The paper also explains how similarity gives insight into different models of interprocess communication, different scheduling disciplines, and randomization.

*Work supported by NSF grant DCR-8320274.

†Also supported in part by a NSF Fellowship.

‡Also supported in part by a Faculty Development Award from IBM.

Authors' address: Computer Science Department, Cornell University, Ithaca, NY 14853.

[1] Recall that an isomorphism ρ on graphs is a 1-to-1 function from one graph to another that has an inverse and that preserves edges and the labels on nodes and edges. An automorphism of a graph is an isomorphism from the graph to itself.

2. Concurrent Systems

A *system* $\Sigma = (N, state_0, I, SP)$ is a set of processors that communicate using shared variables. (The results we develop, however, are easily applied to message-passing communications.) N is a network connecting processors to shared variables; $state_0$ is an initial state; I is an instruction set for processors; and SP is a set of possible schedules.

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* which maps the set of edges E to a set $NAMES$, the local names that processors give to variables. Given $p \in P$ and $v \in V$, p is an ***n*-neighbor *of v*** if $\overline{pv} \in E$ and $naming(\overline{pv}) = n$. A variable might 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 $NAMES$. This ensures that whenever a processor refers to a name, it unambiguously refers to some variable. Thus, there is a function n-nbr on processors such that

$$n\text{-nbr}(p) \equiv \text{ the unique } n\text{-neighbor of } p.$$

The *state* of a processor or variable x is denoted by $state(x)$. We make no assumptions about the number of possible states, although in practice it is bounded.

We assume all processors in a system can execute the same set of instructions I. Three instruction sets are considered: S, L, and Q. We say that a system is in S (L or Q) if its processors can execute the S (L or Q) instruction set. S (simple instruction set) consists of **read** and **write** instructions on shared variables and arbitrary instructions on local variables. Execution of

read i **from** n

by processor p stores the value of the variable that p calls n into p's local variable i. Execution of

write i **to** n

stores the value of i in the variable that p calls n.

L (locking instruction set) consists of S plus **lock** and **unlock**, which use a *lock bit* associated with each shared variable. Execution of

lock$(n, success)$

by p sets the lock bit of the variable that p calls n and, if the lock bit was already set, stores **false** in *success*, otherwise it stores **true**. Execution of

unlock(n)

resets the lock bit of n.

Instruction set Q (quasi-locking instruction set) is more powerful than S but less powerful than L. Q is useful to study, even though it is not very realistic, because L and S can be treated as variants of Q. The distinguishing feature of Q is an unusual kind of shared variable that is accessed by instructions **peek** and **post**. A variable v consists of a multi-set of subvalues—a subvalue $v\langle p\rangle$ for each processor p that can access v. Initially, a variable has no subvalues. The first time p executes **post** on a variable it creates its subvalue in that variable. Execution of

post i **to** n

by p, where v is the n-neighbor of p, stores i in $v\langle p\rangle$. Execution of

peek i **from** n

by p stores in i the unordered multiset of the n-neighbor of p. A processor cannot directly determine the number of neighbors of a variable, although the number of subvalues returned from a **peek** is a lower bound for that number.

In characterizing SP, the set of schedules, we assume that all processors in a system execute the same program. This means that processors in the same state execute the same instruction. (The program counter is part of the processor's state.) An *execution* of the system can be viewed as a sequence of steps, where a *step* is an atomic action corresponding to executing a single instruction. A *schedule* is a possibly infinite sequence of processor names, where a name signifies that that processor executes a single step. We consider three types of schedules: general (G), fair (F), and bounded-fair (BF). A *general* schedule has no restrictions; a *fair* schedule contains each processor infinitely often; and 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 its schedules are fair. It is *bounded-fair* if there is a k such that all its schedules are k-bounded fair.

3. The Selection Problem

With these preliminaries out of the way, the selection problem can be stated precisely. 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, but different schedules may result in different selections.

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.

The selection problem is trivial for some classes of systems.

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 which selects a processor after a finite execution, 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 selected in that schedule, and let ϱ be the schedule that is appended to ϵ to select q. Thus, execution of $\epsilon\varrho$ selects q and execution of $\varphi = \epsilon p$ selects p. Since p executes only instructions in S, the instruction it executes when it is selected assigns **true** to local variable $selected_p$ and so must be either an instruction that involves no shared variables or a **read**. Thus, only the state of p is changed. Therefore, $\varphi p \varrho$ would cause both p and q to be selected because ϱ cannot depend on p's state. This schedule violates Uniqueness, so the algorithm is not a selection algorithm; a contradiction. ∎

The existence of a solution to the selection problem when general schedules are permitted is a special case of the consensus problem when processors can fail by halting, as described in [FLP83]. This is because a halting failure can be viewed as an infinite schedule where a faulty processor appears only a finite number of times. The consensus being reached concerns determining the selected processor. Thus, Theorem 1 is also a proof that there is no algorithm for achieving distributed consensus with one faulty processor, the main result of [FLP83].

In light of Theorem 1, we will ignore the case of general schedules and concentrate on fair and bounded-fair sched-

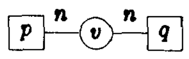

Figure 1: A Trivial System.

ules. We can show that there can be no selection for a system when there is a schedule that prevents any processor from entering a unique state. A schedule φ causes processors and variables to *behave similarly* if it causes them to have the same state at the same time infinitely often, for any program. A set of processors (variables) are *similar* if there is a schedule that causes them to behave similarly. Given this, we can now prove

Theorem 2. *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 this violates Uniqueness, so the algorithm cannot be a selection algorithm. ∎

For example, the system of Figure 1 (assuming instruction set S or Q) with a round-robin schedule causes p and q to behave similarly, so no program can select either.

The difficulty in solving the selection problem is deciding when processors will behave similarly. Certain labelings of the nodes of a system graph are useful for this purpose. A *supersimilarity labeling* is a labeling such that 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* is a labeling such that similar nodes have the same label. A labeling that assigns all nodes the same label is a trivial subsimilarity labeling.

Theorem 3. *If there is a supersimilarity labeling for a system Σ that gives every processor the same label as some other then there is no selection algorithm for Σ.*

Proof: Follows from the definition of supersimilarity and Theorem 2. ∎

A *similarity labeling* is one that is both a supersimilarity labeling and a subsimilarity labeling. It is unique up to isomorphism.

A solution to the selection problem for a collection of systems satisfying some given properties has two parts, both based on a similarity labeling. The first is an algorithm to compute the similarity labeling for any system Σ in that collection. By Theorem 3, if the similarity labeling gives every processor the same label as some other then there is no selection algorithm. Supersimilarity labelings and subsimilarity labelings can be used to find the similarity labeling by constructing successive approximations to it. The second part of a solution to the selection problem is a proof that if some processor is uniquely labeled by the similarity labeling of a system Σ in that collection then some program $SELECT(\Sigma)$ is a selection algorithm for Σ. $SELECT(\Sigma)$ is based on a distributed program in which each processor is able to learn its label under the similarity labeling and select itself if it is the one with some designated label that is known to be unique.

4. Selection for Systems in Q

The following theorem lets us detect supersimilarity labelings for systems in Q and is used to calculate similarity labelings. It is based on the notion of an *environment*. Given a labeling Ψ of nodes in a system, if x and y satisfy

(1) $state_0(x) = state_0(y)$,
(2) $x, y \in P \Rightarrow$
$(\forall n : n \in NAMES : \Psi(n\text{-nbr}(x)) = \Psi(n\text{-nbr}(y)))$,
(3) $x, y \in V \Rightarrow$
$(\forall n : n \in NAMES :$
$|\{r : r \in P : n\text{-nbr}(r) = x \wedge \Psi(r) = \alpha\}| =$
$|\{t : t \in P : n\text{-nbr}(t) = y \wedge \Psi(t) = \alpha\}|)$.

then we say they have the same environment under Ψ.

Theorem 4. *Ψ is a supersimilarity labeling for a system Σ in Q if for any nodes x and y, $\Psi(x) = \Psi(y)$ implies that x and y have the same environment under Ψ.*

Proof: There is a round-robin schedule that gives all nodes with the same label the same state infinitely often. For details, see [J85]. ∎

Computing a similarity labeling for systems in Q is possible using the set partition algorithm of [H71]. A trivial subsimilarity labeling Φ is defined and then refined until it is a similarity labeling. A refinement step of Φ consists of picking a set of nodes with the same label under Φ such that two members have different environments under Φ, then picking one node x from the set, and finally relabeling all nodes in the set that have environments different from x under Φ. The labeling remains a subsimilarity labeling as long as similar nodes are not given different labels. Recall, when Φ is both a subsimilarity labeling and a supersimilarity labeling, it is a similarity labeling.

Algorithm 1. *Compute Similarity Labeling Θ.*

```
$\Phi$ := trivial subsimilarity labeling;
do nodes $x$ and $y$ have the same label but
  different environments under $\Phi$ $\rightarrow$
    pick new label $\alpha$;
    for each $y$ with the same label as $x$ but
        a different environment under $\Phi$ :
      $\Phi(y) := \alpha$;
    rof
od
$\Theta := \Phi$
```

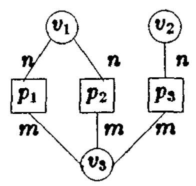

Figure 2: Complicated Alibis.

Theorem 5. *There is an $O(|P \cup V|\ log|P \cup V|)$ algorithm to compute a similarity labeling Θ for any system Σ in Q.*

Proof: Algorithm 1 is not $O(|P \cup V|\ log|P \cup V|)$, but [H71] shows how to make it so. [J85] contains a proof of correctness for Algorithm 1. ∎

Algorithm 1, which calculates similarity labeling Θ for a system Σ, can be modified to yield a distributed algorithm in which each processor computes its own label under Θ. This algorithm is specific for the system Σ, but can be generated automatically from the bipartite graph specification of Σ. In the algorithm (given below), processors keep lists of labels they suspect that they and their neighboring variables might have. The neighboring shared variables are then used to communicate a processor's list of suspected labels to other processors. A processor p determines suspected labels for a variable v by: reading v, learning the suspected labels for each processor that can access v, and using that information (and Θ) to rule out some labels it suspects for v. Then, p refines its guess of its own suspected labels by using Θ and the suspected labels of its neighboring variables.

Knowledge of the network topology and initial state of Σ is used to determine which labels are possible for a given processor or variable. A node x has an *alibi* for a label α if x can determine that it cannot be labeled α by Θ. The set of labels suspected by a node x consists of all labels for which there is no alibi. Obviously, not being in this set of suspects means there is an alibi for a label.

The system in Figure 2 illustrates the type of reasoning needed to find alibis. There, processor p_1 is similar to p_2 and p_1 is not similar to p_3, so there are two equivalence classes of processors under the similarity labeling Θ. Initially, all processors suspect both equivalence classes. Eventually, after writing these sets to their n-neighbors, p_1 and p_2 are able to tell that their n-neighbor v_1 has two neighbors. This can be used as an alibi for $\Theta(p_3)$. However, p_3 has no alibi for $\Theta(p_1) = \Theta(p_2)$, since its state would be the same as that of p_2 if p_1 had not yet written to v_1. If p_1 and p_2 write their set of suspects (now containing only $\Theta(p_1)$) to v_3, then p_3 can find an alibi for $\Theta(p_1)$ from the information in v_3. Since v_3 has three neighbors, two of which suspect only $\Theta(p_1)$, p_3 can tell that it must be the third processor. Thus, p_3 can learn its label.

Algorithm 2 (see below) is the distributed algorithm that lets each processor learn its own label. It uses ***p-alibi***, which computes a set of labels for processors for which there are alibis, and ***v-alibi***, which computes a set of labels for variables for which there are alibis. *PLABELS* and *VLABELS* are the sets of labels Θ gives to processors and variables, respectively. Function ***neighborhood_size***(n, α, β) returns the number of n-neighbors labeled α of a variable labeled β and depends on the topology of the system.

Two types of alibis are found by ***p-alibi***. First, a processor has an alibi for α if its n-neighbor has an alibi for n-nbr(α). For example, in Figure 2, p_1 has an alibi for $\Theta(p_3)$ because v_1 has an alibi for v_2, since it is written by two processors. Second, a processor has an alibi for label α if it can tell that all the processors labeled α know their label, and it does not yet know its own label. For example, in Figure 2, p_3 uses the second type of alibi to learn that it is not p_1.

There is just one type of alibi for variable labels. A variable has an alibi for label β if too many of its neighbors suspect only labels in some set ***Lab*** of processor labels. Since a processor suspects its own label, this means the variable has more neighbors with labels in ***Lab*** than β has. For example, in Figure 2, v_1 has an alibi for $\Theta(v_2)$ because v_1 has more than one neighbor. In this case, ***Lab*** is *PLABELS*.[2]

A node can never find an alibi for its own label, so Algorithm 2 never terminates with a wrong answer. However, it is not obvious that it always terminates. It does, when run in a connected system with fair schedules, because it simulates the partitioning performed by Algorithm 1. Each partitioning is modeled by processors and variables learning which half they are in. This is accomplished by computing alibis for labels of processors and variables in the other half.

Theorem 6. *If a system Σ in Q is connected and fair or is unconnected and bounded-fair then there is a program for the processors of Σ that lets each learn its label under similarity labeling Θ.*

Proof: Algorithm 2 is the program. A proof of its correctness is in [J85]. ∎

Note that while systems are generally assumed to be connected, we will find it useful later to assume that the system is not connected but that each processor knows the number of neighbors of each neighboring shared variable. This assumption is equivalent to assuming bounded fairness.

We can now define *SELECT*(Σ), the program to select a leader in Σ, to be a program that uses Algorithm 2 to find the label of each processor and then selects the processor with a distinguished, unique label. Algorithm 1 can find similarity labelings, hence it can be used to tell when *SELECT*(Σ) will work and to find the label of the processor to be selected. Thus, we can decide if a system Σ in Q has a selection algorithm and use Algorithm 2 if it does.

[2]The specification of *p-alibi* in Algorithm 2 looks exponential because it iterates over a powerset. However, only a linear number of these sets are needed, so *p-alibi* is polynomial in the sizes of *PLABELS*, *VLABELS*, and *NAMES*. [J85]

Note that any selection algorithm for a fair system is a selection algorithm for a bounded-fair system, so Theorem 6 holds for connected systems with bounded-fair schedules. Thus, there is no difference between connected bounded-fair systems in Q and connected fair systems in Q as far as the selection problem is concerned.

Algorithm 2. *Find Similarity Label of Processor k.*

```
Program variables (for processor $k$):
$PEC \subseteq PLABELS$ /* suspects for $\Theta(k)$ */
$VEC[n] \subseteq VLABELS$ /* suspects for $\Theta(n\text{-nbr}(k))$ */
$local[n]$ /* local copy of value of variable $n$-nbr($k$) */
Given $x$, a value posted to a shared variable
  $x.suspects \subseteq PLABELS$,
  $x.name \in NAMES$.

Program for processor $k$:
$PEC := \{\alpha : \alpha \in PLABELS : state_0(\alpha) = state_0(k)\}$
for $n \in NAMES$:
  $VEC[n] := \{l : \alpha \in VLABELS :$
                    $state_0(\alpha) = state_0(n\text{-nbr}(k))\}$;
rof;
do $|PEC| > 1 \rightarrow$
  for $n \in NAMES$:
    peek $local[n]$ from $n$;
    $VEC[n] := VEC[n] - v\text{-}alibi(local[n])$
  rof;
  $PEC := PEC - p\text{-}alibi(VEC, local, PEC)$;
  for $n \in NAMES$:
    $val.suspects := PEC$; $val.name := n$;
    post $val$ to $n$
  rof
od

$p\text{-}alibi(VEC, local, PEC) \equiv$
  $\{\alpha : \alpha \in PLABELS :$
    $(\exists n : n \in NAMES :$
      $n\text{-nbr}(\alpha) \notin VEC[n] \vee$
      $((|PEC| > 1) \wedge$
        $|\{x : x \in local[n] :$
              $x.suspects = \{\alpha\} \wedge x.name = n\}| =$
        $neighborhood\_size(n, n\text{-nbr}(\alpha), \alpha)))\}$.

$v\text{-}alibi(v) \equiv$
  $\{\beta : \beta \in VLABELS :$
    $(\exists n, Lab : n \in NAMES, Lab \in \mathcal{P}(PLABELS) :$
      $|\{x : x \in VAL(v) :$
            $x.name = n \wedge$
            $x.suspects \subseteq Lab\}| >$
      $\sum_{\alpha \in Lab} neighborhood\_size(n, \alpha, \beta))\}$.
```

5. Selection for Systems in L

A *family* of systems is a set of systems, each with the same instruction set, the same types of schedules, and the same set of variable names *NAMES*. Only the network topology and the initial states of processors and variables in the systems of a family may differ. Since all systems in a family have the same instruction set and the same set of variable names, a program for the processors in one system of the family is also a program for the processors of any other system of that family. A program is a selection algorithm for a family of systems if it is a selection algorithm for each system in the family. *Homogeneous* families consist of systems with the same network topology and thus differ only in the initial states of processors and variables.

We are able to solve the selection problem for homogeneous families of systems in Q, and this turns out to be useful in solving the selection problem for systems in L. Solving the selection problem for a family requires a similarity labeling for the family. This is simply the similarity labeling for the system made up of the union of all the systems in the family. Algorithm 1 can be used to calculate the similarity labeling of this system.

Algorithm 2 cannot be used without modification for a basis of *SELECT* for a homogeneous family of systems. The proof of termination of Algorithm 2 (Theorem 6) requires either that the system be connected or that every processor know the number of neighbors of each of its neighboring variables, but the union system is not connected. We can circumvent this problem by using Algorithm 2 twice: first, to find the number of neighbors of each variable, and then to find the label of each processor. The first time Algorithm 2 is used, the initial state of each node is ignored. Since systems in a homogeneous family differ only in their initial state, any distributed algorithm that ignores the initial state will have the same effect on each system in the family. Thus, a version of Algorithm 2 that ignores the initial state of the system will find the same labeling for every system in the family. This labeling might not be a similarity labeling, but it will have the property that variables with different numbers of neighbors will have different labels, i.e. each processor knows the number of neighbors of each of its neighboring variables. Then, the similarity label of each processor can be determined using Algorithm 2 again, this time with the label from the first labeling as the initial state of each variable.

Algorithm 3. *Find label for processor k in a homogeneous family in Q.*

```
Use Algorithm 2 to find the number
    of neighbors of each variable;
Change the initial state of each variable
    to reflect the number of its neighbors;
Use Algorithm 2 to find $\Theta(k)$, using
    the new initial state.
```

There is a selection algorithm for a homogeneous family of systems in Q when there is a set *ELITE* of processor similarity labels such that each system in the family contains exactly one processor with a similarity label in *ELITE*. Given *ELITE*, Algorithm 3 can form the basis for a selection algorithm to select the unique processor with a label in *ELITE*. If no such set of labels exists then there is a system in the family such that every processor has a similar processor in the system. Thus, by Theorem 2, there can be no selection algorithm.

Theorem 7. *A family of systems in Q has a selection algorithm if and only if there is a set ELITE of processor labels such that each system in the family contains exactly one processor with a similarity label in ELITE.*

Proof: An outline is given above. Details are in [J85]. ∎

We can now turn to the selection problem for systems in L, which is much like that for homogeneous families in Q. Processors in L that share a variable and give it the same name n cannot be similar because they can distinguish between themselves by locking their n-neighbor—only one will succeed. If processors do this, the result is a system in which processors that give the same variables the same names have different states. A similarity labeling for such a system in Q is also a similarity labeling for a system in L with the same network topology, initial state, and types of schedules.

Theorem 8. *If Ψ is a supersimilarity labeling for $(N, state, Q, F)$ and if*

$$(\Psi(p) = \Psi(q) \Rightarrow (\forall v : v \in V : naming(\overline{pv}) \neq naming(\overline{qv}))).$$

then Ψ is a supersimilarity labeling for a system $(N, state, L, F)$.

Proof: Just as in Theorem 4, there is a round-robin schedule that gives each processor with the same label the same state infinitely often for any program. Details are in [J85]. ∎

When locking is used to relabel processors in L that give a variable the same name, the resulting state of the system is one of a set of possible states R, depending on the schedule. Any two n-neighbors of a variable necessarily lock it in a different order, so each system state in R has the property that no two processors giving a variable the same name have the same state. Since processors with different states are dissimilar, no two processors giving a variable the same name have the same state. Thus, by Theorem 8, if *state* is in R then any supersimilarity labeling of $(N, state, Q, F)$ is a supersimilarity labeling of $(N, state, L, F)$. Two nodes that are similar in $(N, state, L, F)$ are clearly similar in $(N, state, Q, F)$, so systems $(N, state, L, F)$ and $(N, state, Q, F)$ have the same similarity labeling.

A processor computes its initial state in R by locking each neighboring variable, recording the number of processors that have previously locked that variable, and using that count as part of its new state. Thus, the state of the processor is a function of the numbers it read from neighboring variables when it locked them. This procedure is summarized by the following subroutine:

```
relabel(k) ≡
  for each $n \in NAMES$:
    success := false;
    do ¬success → lock(n-nbr(k), success)  od;
    read $count_n$ from n;
    write $count_n + 1$ to n;
    unlock(n-nbr(k))
  rof
```

The set $\{(N, state, L, F) | state \in R\}$ is a family of homogeneous systems. By the previous argument, the similarity labeling of any system in this family can be computed using Algorithm 1 because the system satisfies Theorem 8. Thus, the labeling is a supersimilarity labeling of Σ. If each system in the family has a processor that is dissimilar to any other in the system, then there is a selection algorithm for the family.

Algorithm 4. *Selection algorithm for processor k in a system in L.*

```
relabel(k);
Use Algorithm 3 as selection algorithm for family
    of systems produced by relabel.
```

Theorem 9. *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 Algorithm 4 is a selection algorithm.

Proof: Let H be a homogeneous family of systems, each of which could be produced by an execution of *relabel* on Σ. Following Theorem 7, we construct the set *ELITE* as follows.

```
ELITE := ∅;
VERSIONS := the set of similarity labelings
              of members of H
do $(\exists \Psi : \Psi \in VERSIONS:$
        $(\forall p : p \in P : \Psi(p) \notin ELITE)) \rightarrow$
            pick $\Psi \in VERSIONS$ 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
$\{(\forall \Psi : \Psi \in VERSIONS:$
        $(\exists k : k \in P : \Psi(k) \in ELITE))\}$
```

The invariant for this loop is that each system in the family has no more than one processor with a label in *ELITE*. It is true initially. The invariant would be maintained when $\Psi(p)$ is added to *ELITE* provided

1. p is uniquely labeled by Ψ and

2. no member of *VERSIONS* gives some processor the label $\Psi(p)$ and also gives some processor a label in *ELITE*.

By hypothesis of the theorem, each supersimilarity labeling Ψ of Σ has a node that is uniquely labeled and the algorithm picks p to be such a node. Thus, (1) is satisfied. Since *VERSIONS* is a set of similarity labelings for systems in H, it is also a set of supersimilarity labelings for Σ. Therefore, each Ψ in *VERSIONS* gives some node a unique label. If a supersimilarity labeling labels one processor α and another β then every supersimilarity labeling that labels a processor α must label another β. Ψ gives no processor a label in *ELITE*, so no supersimilarity labeling with a label in *ELITE* labels a processor $\Psi(p)$. Thus, (2) is satisfied. Therefore, when the loop terminates, each system has exactly one processor with a label in *ELITE*.

The loop terminates because each iteration reduces the number of systems with no processor with a label in *ELITE*. Eventually all systems have a processor with a label in *ELITE*, so the loop will terminate.

We have constructed a set of process labels *ELITE* such that the similarity labeling of each member of the family H gives exactly one process a label in *ELITE*. By Theorem 7, Algorithm 3 is a selection algorithm for the family of systems produced by ***relabel***. ∎

6. Selection for Systems with Other Instruction Sets

Extended Locking

We have not discussed instruction sets containing instructions that involve access to several shared variables simultaneously. It is easy to find similarity labelings for systems in which processors can indivisibly lock a list of variables. In a system in which a processor can lock two variables at once, similar processors cannot be neighbors of 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. Except for this difference, all results for the extended locking instructions are the same as the results for systems in L.

Systems in S

The solution to the selection problem in systems in S, unlike Q or L, depends on whether schedules are fair or bounded-fair. Computing the similarity labeling for bounded-fair schedules and instruction set S is almost the same as for Q. The difference is that similar variables in a system in Q must have the same number of n-neighbors with a particular label, but if v is similar to u in a system in S then the only requirement is that every n-neighbor of v be similar to some n-neighbor of u, and *vice versa*. The distributed algorithm for finding similarity labels is nearly the same as the one given above for Q, and it too can be used as the basis for a selection algorithm.

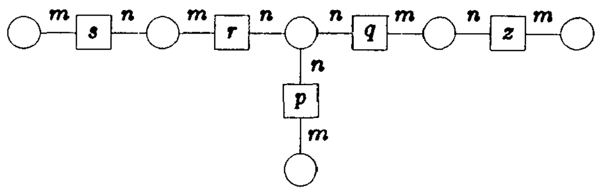

Figure 3: A System in S.

Fair Systems in S

Selection in fair systems in S that are not also bounded-fair is quite different. Dissimilar processors are not necessarily able to distinguish between themselves, and there is no distributed algorithm to find the similarity labels of each processor. The problem is illustrated in Figure 3. There, if z has not executed, then processors p and q behave as if they were similar, and p cannot tell whether z has executed. Thus, even though a similarity labeling Θ can be computed as before, there is no distributed algorithm (analogous to Algorithm 2) that lets processors learn their labels. Given a system Σ, we say that x *mimics* y if there is a subsystem of Σ such that x is similar to the image of y in the subsystem. If x mimics y then as long as the processors in Σ that are not in the subsystem do not execute, y can be in any state that x could have been in. Thus, x can never learn its similarity label without the chance of y incorrectly deciding that it was labeled the same as x. There is a selection algorithm for a fair system Σ in L if and only if there is a process that mimics no other process.

Message Passing

Similarity is a useful concept in message-passing systems. Similarity labelings and distributed algorithms for finding labels can be easily computed for any fair system that uses asynchronous message-passing. There, the environment of a processor depends only on the processors that can send messages to it. This is responsible for a difference between similarity labelings for bidirectional and unidirectional message-passing models. A unidirectional, fair, not strongly-connected system in which no processor knows the number of processors that can send messages to it suffers from the same problems as fair systems in S; all other asynchronous message-passing systems give results like those of Q.

Synchronous message-passing systems are more complicated. Two versions that we have studied are CSP [H78] and CSP extended with output guards. A supersimilarity labeling for an asynchronous, bidirectional message-passing system is a supersimilarity labeling for that same system using the operations of extended CSP if no two neighboring processors have the same label. Thus, systems in extended CSP are to asynchronous bidirectional message-passing systems as systems in L are to systems in Q. This analogy extends to the way selection algorithms can be found. Any supersimilarity labeling of an asynchronous bidirectional message-passing system is a supersimilarity labeling of a CSP system with no output guards. Unfortunately, we have

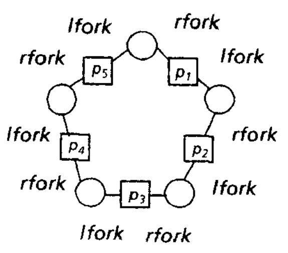

Figure 4: Five Dining Philosophers

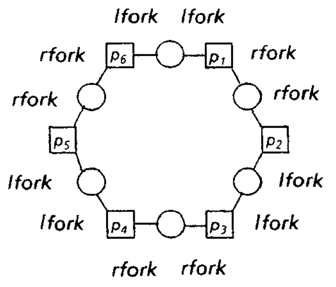

Figure 5: Six Dining Philosophers.

not yet found a general algorithm free from deadlock that will let processors in a system using CSP with no output guards learn their label.

7. Dining Philosophers

Our interest in symmetry originated with the claim [LR80]:

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

In the Dining Philosophers problem [D71], five philosophers alternately eat and think at a table with five plates of spaghetti, one for each philosopher, and five forks, one between every adjacent pair of plates. A philosopher needs two forks to eat and always uses the forks on either side of its plate. Thus, two adjacent philosophers cannot eat at the same time. The interconnection of processes and variables in Figure 4 might be a solution to the Dining Philosophers problem. It is *distributed* because no variable is accessed by all processors. It is symmetric because, for any pair of processors there is an automorphism mapping one to the other.[3] Unfortunately, for any possible program, the round-robin schedule $p_1 p_2 p_3 p_4 p_5 p_1 \ldots$ causes all philosophers to have the same state at the same time infinitely often, so all philosophers are similar. Moreover, any time one philosopher is eating, all are, which violates the problem constraints. Thus, there is no program that can solve the Dining Philosopher problem for this system. This argument, however, is not a proof of DP, since some other interconnection might be a solution. DP does happen to be true, though.

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.

Figure 5 shows an interconnection that is distributed and symmetric (in the sense of [LR80]) for which there is a solution to the six-philosopher problem. Not all philosophers are similar here.

[3]The graph theoretic definition of "symmetry" is the one used in DP [L85].

Since the graph-theoretic definition of symmetry is used in DP and DP′, two nodes in a system are *symmetric* if there is an automorphism of the system graph mapping one node to the other. The set of automorphisms of a graph is a group, so symmetry of nodes is an equivalence relation, and nodes are partitioned into equivalence classes under symmetry.

Theorem 10. *Symmetric nodes (in the graph-theoretic sense) in a system in Q are similar.*

Proof: The equivalence classes of nodes under symmetry form a supersimilarity labeling. This is because an automorphism that maps node x to node y is a one-to-one mapping from the edges of x to the edges of y that preserves the labels on the edges and the nodes. If nodes are labeled with their equivalence class under symmetry then variables with the same label have the same number of n-neighbors with any given label, for all names n, and processors with the same label have n-neighbors with the same label. Thus, by Theorem 4, this labeling is a supersimilarity labeling in Q. Therefore, symmetric nodes are similar in Q. ∎

We can now show that a distributed, symmetric system in L with a prime number of symmetric processors must behave just like the same system in Q. Any distributed, symmetric solution to the Dining Philosophers problem must be such a system.

Theorem 11. *Let Σ be a distributed, symmetric, deterministic system in L containing an equivalence class C of j 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 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 10. 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 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 8 a supersimilarity labeling for a system in Q 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, 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. This does not follow directly from the definition of similar, since the state that philosophers are in together infinitely often could be the state of not eating, which would satisfy the constraints of the problem. For similarity labelings for L, the round-robin schedule causes processors with the same labelings to have the same state after every round of the schedule. If p and q are similar then every state that p is in can simultaneously be the state of q. Thus, 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 philosophers be dissimilar. However, five symmetric processors are similar by Theorem 11, so DP is true.

DP′ relies on Theorem 11 and the fact that the number of philosophers is composite. Then, we can let alternate philosophers put their backs to the table, as in Figure 5, 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 and those that are left forks. All philosophers are symmetric, since the automorphism corresponding to a reflection through a line through a variable keeps all variables in the same equivalence class, but changes the equivalence class of all philosophers. Since there are two equivalence classes of philosophers, it is possible for each philosopher to be dissimilar to its neighbors.

8. Discussion

Related Work

Special cases of some of our results have appeared in the literature. [G80][A80][IR81] are subsumed by *SELECT*, although these other results, being less general, are more efficient. [A80][RFH72] contain proofs that some problems are not solvable on "symmetric" graphs, similar in purpose and technique to the proofs of Theorems 4 and 8. Bouge [B84] has examined CSP and extended CSP and found some necessary conditions and some sufficient conditions for the selection problem. He also noticed the relationship between the selection problem and other problems, including the Dining Philosophers problem. Bouge distinguishes between syntactic symmetry, such as symmetry of the system graph, and semantic symmetry, such as similarity, but chose a definition for semantic symmetry more complicated than similarity.

Encapsulating Asymmetry

The simple characterization of semantic symmetry in terms of similarity has many uses. One use is to tell when operations are inherently asymmetric. We say that a system Σ can *break symmetry* if nodes in Σ that are symmetric using the graph-theoretic definition are not similar. Note that Theorem 10 can be restated as saying that systems in Q cannot break symmetry. If a set of components can be used to build a system that breaks symmetry then some of those components can break symmetry. If a system that can break symmetry is implemented using components that cannot break symmetry then the implementation must explicitly introduce asymmetry. The implementation is encapsulating the asymmetry so that programs running on the symmetry-breaking system do not have to be aware of the underlying asymmetry. Since the basic electronic components with which we build computers cannot break symmetry [CM73], any implementation of a system that can break symmetry has encapsulated the necessary asymmetry. It follows that locking operations encapsulate asymmetry. Extended CSP also encapsulates asymmetry [J85][B84][LR80].

Operations that encapsulate asymmetry have the advantage that programs using them can be thought of as being symmetric, and it is generally easier to reason about symmetric systems. A method is proposed in [CM84] for designing systems by explicitly encapsulating the necessary asymmetry. There, processors all execute the same program and have no explicit labels. The system can be thought of as symmetric, but the initial state is carefully designed to ensure fair use of resources and is 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, and problems like the Dining Philosopher's problem can be solved.

Randomized algorithms have been used to break symmetry in distributed systems [IR81][LR80][FR80]. These algorithms can solve synchronization problems that deterministic algorithms cannot. By characterizing symmetry in terms of similarity, we are able to identify systems that could not deterministically solve problems but can using randomized algorithm. Thus, we can describe the added power of randomization.

9. Conclusion

Similarity labelings provide a simple syntactic characterization of the power of different models. In this paper, we have shown that any similarity labeling for a system Σ in L is a similarity labeling for Σ in Q, but that the reverse is not true. Thus, L is strictly more powerful than Q. In the same way we can show that Q is more powerful than bounded-fair S which is more powerful than fair S. Operations that encapsulate asymmetry are more powerful than those that do not, and randomized programs are more powerful than deterministic ones. Even more important, the rules used to define similarity labelings give a qualitative comparison of the different models. For example, systems in L differ from systems in Q only because processors that give the same name to the same variables can tell themselves apart, and systems in Q differ from bounded-fair systems in S only because processors can eventually learn the number of neighbors of each variable. Thus, similarity provides a way to compare and contrast different models of concurrent programming.

Acknowledgments

O. Babaoglu, D. Gries, and L. Lamport made helpful comments on earlier drafts of this paper.

References

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

[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.

[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.

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

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

[FLP83] Fischer, M., N. Lynch, and M. Patterson. Impossibility of Distributed Concensus with One Faulty Process. *Proceedings of Second ACM SIGACT-SIGMOD Symposium on 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.

[G80] Garcia-Molina, H. Elections in a Distributed Computing System. TR-280, Princeton University (Dec. 1980).

[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).

[J85] Johnson, R.E. *Symmetry in Distributed Systems.* Ph.D. Dissertation, in preparation.

[L85] Lehmann, D.J. Private communication with F. 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.

[R80] Rabin, M.O. The Choice Coordination Problem. Memorandum No. UCB/ERL M80/38, Univ. of California Berkeley (Aug 1980).

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

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