

Introduction to Distributed Systems

Dr. Paul Sivilotti
Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH, 43210-1277

Spring 2007

Contents

1	Booleans, Predicates, and Quantification	1
1.1	References	1
1.2	Preliminaries	1
1.2.1	Functions	1
1.2.2	Booleans and Predicates	2
1.2.3	Lifting	3
1.2.4	Everywhere Brackets	4
1.3	The Predicate Calculus	6
1.3.1	Equivalence	6
1.3.2	Disjunction	7
1.3.3	Proof Format	8
1.3.4	Conjunction and Implication	9
1.3.5	Negation and false	10
1.3.6	Discrepance	11
1.4	Quantification	13
1.4.1	Syntax	13
1.4.2	Informal Interpretation	14
1.4.3	Definition	15
2	The Computational Model	17
2.1	References	17
2.2	Programs	17
2.3	Actions	18
2.3.1	Simple Assignments	18
2.3.2	Multiple Assignments	18
2.3.3	Guarded Actions	19
2.3.4	Sequential Composition	19
2.4	Operational Intuition	19
2.4.1	Program Execution	19
2.4.2	Termination	20
2.4.3	An Example: Find Max	21
2.5	Visualizing a Program	22
2.5.1	Directed Graphs	22
2.5.2	An Exercise	22

2.6	Fairness	24
2.6.1	Motivation	24
2.6.2	Weak Fairness	24
2.6.3	Strong Fairness	24
3	Reasoning About Programs	27
3.1	References	27
3.2	Introduction	27
3.2.1	Motivational Example	27
3.2.2	Specification	28
3.3	Reasoning About a Single Action	28
3.3.1	Hoare Triples	28
3.3.2	The Assignment Axiom	29
3.3.3	Guarded Actions	30
3.4	Safety	31
3.4.1	Next	32
3.4.2	Stable	33
3.4.3	Invariant	34
3.4.4	Unless	35
3.5	Progress	36
3.5.1	Transient	36
3.5.2	Ensures	38
3.5.3	Leads-to	40
3.5.4	Induction	42
4	Small Example Programs	45
4.1	Proof Structure	45
4.2	FindMax	45
4.3	Sorting	47
4.4	Earliest Meeting Time	49
4.5	Greatest Common Divisor	50
5	Time, Clocks, and Synchronization	53
5.1	References	53
5.2	Introduction	53
5.3	Logical Time	54
5.3.1	Happens Before	54
5.3.2	Timelines	55
5.3.3	Logical Clock	56
5.3.4	Algorithm	57
5.3.5	Total Ordering	58
5.4	Vector Clocks	59
5.4.1	Motivation	59
5.4.2	Central Idea	60
5.4.3	Algorithm	62
5.5	Synchronization of Physical Clocks	64

5.5.1	Messages with Unbounded Delay	65
5.5.2	Messages with Bounded Delay	68
6	Diffusing Computations (Gossip)	71
6.1	References	71
6.2	Introduction	71
6.3	Operational View	72
6.4	Specification	74
6.5	Algorithm	74
6.6	Proof of Correctness	74
6.6.1	Safety	75
6.6.2	Progress	76
7	Mutual Exclusion	77
7.1	Reference	77
7.2	Introduction	77
7.3	Specification	78
7.3.1	A Trivial Solution	79
7.4	Distributed Atomic Variables	79
7.4.1	Introduction	79
7.4.2	Algorithm	81
7.5	Nontoken-Based Solutions	82
7.5.1	Lamport's Algorithm	82
7.5.2	Optimization #1	83
7.5.3	Optimization #2: Ricart-Agrawala	83
7.6	Token-Based Solutions	84
7.6.1	Introduction	84
7.6.2	Simple Token Ring	84
7.6.3	Token Ring with Requests	85
7.6.4	Token Tree (Raymond)	87
7.6.5	Token Graph	88
7.6.6	Summary of Key Ideas for Token-based Solutions	88
8	Dining Philosophers	91
8.1	References	91
8.2	Introduction	91
8.3	Specification	92
8.4	Naive Solutions	92
8.5	Hygienic Solution	93
8.6	Refinement of Specification	94
8.6.1	Safety (Forks)	94
8.6.2	Priority (Clean vs. Dirty)	94
8.6.3	Neighbor Hunger (Request Tokens)	96
8.7	Algorithm	96
8.7.1	Message-Passing Pseudocode	97
8.8	Proof of Correctness	99

8.9	Summary of Key Points	100
9	Snapshots	101
9.1	References	101
9.2	Problem Description	101
9.3	The Naive Approach	102
9.4	Consistent Cuts	103
9.5	Solution #1: Logical Time	104
9.6	Utility of Snapshots	105
9.7	Solution #2: Marker Algorithm	106
9.7.1	Algorithm	106
9.7.2	Proof of Correctness	107
10	Termination Detection	109
10.1	Problem Description	109
10.2	Algorithm	110
10.3	Specification	112
10.4	Proof of Correctness	112
11	Garbage Collection	115
11.1	Reference	115
11.2	Problem Description	115
11.3	Application to Memory Storage	116
11.4	Relationship to Termination Detection	117
11.5	Formal Definitions	117
11.6	Principle of Superposition	119
11.7	Propagator - First Attempt	119
11.8	Propagator - Second Attempt	119
11.9	Specification of Propagator	120
11.10	Proof of Correctness	121
12	Byzantine Agreement	123
12.1	References	123
12.2	Background: Two-Generals Problem	123
12.3	Faults in Distributed Systems	124
12.4	Binary Consensus	124
12.5	Asynchronous Consensus with Process Failures	125
12.6	Synchronous Agreement with Crash Faults	126
12.7	Synchronous Agreement with Byzantine Faults	127
12.7.1	With Authenticated Signatures	127
12.7.2	Without Authenticated Signatures	129
13	Discrete-Event Simulation	131
13.1	References	131
13.2	Background: Sequential DES	131
13.3	A Sequential Algorithm	133

13.4 Time-Driven Simulation	134
13.5 Conservative Approach	134
13.5.1 A Naive Algorithm	134
13.5.2 A Correct Algorithm	135
13.6 Extensions	136
13.7 Optimistic Approach	136
13.8 Extension: Lazy Cancellation	138
13.9 Summary of Key Ideas	138
A Order of Operations	139

List of Figures

1.1	Venn Diagram of the <i>is_tall</i> Predicate	3
1.2	The Conjunction of Two Predicates	4
1.3	Graphical Representation of <i>is_tall</i> \equiv <i>is_heavy</i>	5
2.1	Programs as Directed Graphs	22
2.2	A Program with Five States and Three Actions	23
2.3	Directed Graph Corresponding to Program Example 1	23
3.1	Guaranteeing that Q holds after $x := E$	29
3.2	Two Preconditions that Guarantee the Postcondition <i>even.x</i>	30
3.3	Graphical Representation of P next Q	33
3.4	Graphical Representation of P unless Q	35
3.5	Graphical Representation of transient . P	37
3.6	Graphical Representation of P ensures Q	38
3.7	Graphical Representation of $P \leadsto Q$	40
5.1	Timeline for a computation with four processes.	56
5.2	Equivalent timeline for a computation with four processes.	56
5.3	Assigning timestamps using logical time.	58
5.4	A partial timeline with two events: A and B	59
5.5	Timestamping an event with a vector clock.	60
5.6	Timestamping a receive event with a vector clock.	61
5.7	Assigning timestamps using vector clocks.	63
5.8	Using vector clocks to determine whether $A \longrightarrow B$	64
5.9	Single time message with unbounded delay.	65
5.10	Request and reply protocol for obtaining current time.	65
5.11	Multiple requests and replies for obtaining current time.	66
5.12	Multiple simultaneous requests for obtaining current time.	67
5.13	Tallying counts for candidate current times.	67
5.14	Possible result of tallying several intervals.	68
5.15	A physical clock with drift.	69
6.1	Barrier synchronization	72
6.2	Example of a given topology for diffusing computation	72

6.3	Possible race condition in gossip algorithm	73
7.1	Mutual exclusion layer arbitrates user process conflicts	77
7.2	State transition diagram for user processes	78
7.3	Protocol between user process and mutual exclusion layer	79
7.4	Mutual exclusion layer as a distributed system	80
7.5	Updates to a distributed copies of a shared variable	80
7.6	Request queues for sorting update requests	81
7.7	First optimization: reducing the number of acknowledgements	83
7.8	Ricart-Agrawala optimization: deferring acknowledgements	84
7.9	Simple token ring	85
7.10	Token ring with requests	86
7.11	Multiple pending requests	88
8.1	State transitions for a philosopher	91
8.2	Partial order of philosophers	93
8.3	Possible position and state of a shared fork, given $u \leq v$	95
9.1	Simplified timeline indicating recording of local state	102
9.2	Wavey cut through a timeline	103
9.3	Transfer of funds in a distributed bank	103
9.4	Consistent cut	104
9.5	Inconsistent cut	104
9.6	A possible consistent cut	105
9.7	Computation during a snapshot	106
9.8	Reachability of S_{snap}	106
9.9	Result of swapping adjacent actions	107
10.1	Topology with directed channels	109
10.2	State transitions allowed for a process	110
10.3	Process for detecting termination	111
10.4	Local topology of a single process	112
11.1	A directed graph with a designated root	115
11.2	Heap of dynamically allocated storage	116
11.3	Edges between Food, Garbage, and Manure	118
11.4	A pair of vertices for which $\neg ok[x, y]$	120
12.1	Byzantine Agreement with Authentication (Tolerates 1 Fault)	128
12.2	Consensus Tree of Message Chains	128
13.1	Three basic kinds of nodes	131
13.2	A simple simulation system with three servers	132
13.3	Possible deadlock with naive algorithm	135
13.4	Roll-back caused by the arrival of a straggler	137
13.5	Lazy cancellation of optimistically generated events	138

List of Tables

1.1	Common Symbols for Quantification	14
-----	---	----

Chapter 1

Booleans, Predicates, and Quantification

1.1 References

The following references are both very good:

1. “Predicate Calculus and Program Semantics”, by E.W. Dijkstra and C.S. Scholten, Springer-Verlag 1990 [DS90]. This chapter of the lecture notes is based on this reference (primarily chapter 5). The earlier chapters of the reference give a nice introduction to some fundamentals such as everywhere brackets, boolean structures, and the format for calculational proofs. The later chapters of the reference deal with program semantics and predicate transformers (*i.e.*, weakest precondition semantics). It is a classic reference for weakest precondition.
2. “A Logical Approach to Discrete Math”, by D. Gries and F. Schneider, Springer-Verlag, 1993 [GS93]. Chapters 3 and 8 are most useful. This book is a more introductory text but has a very nice exposition of booleans, predicates, predicate calculus, and quantification.

1.2 Preliminaries

1.2.1 Functions

A function is a mapping, or relationship between elements from two sets. A function maps values in one set (the *domain*) to values in the other (the *range*). By definition, a function maps each element in its domain to at most one element in its range. We use the notation $f : \mathcal{A} \rightarrow \mathcal{B}$ to indicate a function f whose domain is the set \mathcal{A} and whose range is the set \mathcal{B} . For example, the familiar

square-root function would be written:

$$\text{sqrt} : \mathbb{R}^+ \rightarrow \mathbb{R}^+$$

This describes a function named *sqrt* that maps elements from the positive real numbers to the positive real numbers.

The most basic operator for a function is *function application*. This operation is denoted by a dot (*i.e.*, “.”). For example, we would write:

$$\text{sqrt}.16$$

Contrast this with the more common notation for function application, which appears something like *sqrt*(16). Although the later notation has the advantage of similarity with several common imperative programming languages (*e.g.*, C, Pascal, and Java), we choose the former notation because it lends itself nicely to Currying.

This does mean, however, that we must exercise care when dealing with functions with more than one argument. For example, consider the function *max2* that takes the maximum of two integers. The signature of this function is:

$$\text{max2} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

Application of such a function to its arguments will be written as *max2*.(*a*, *b*) (and not *max2*.*a*, *b* which would force us to distinguish two very similar characters—“.” and “,”—to resolve the meaning of such expressions).

The dot operator associates to the left, so *f*.*x*.*y* should be interpreted as (*f*.*x*).*y*. For such an expression to make sense, the result of applying *f* to *x* must be a function! This function is then applied to *y*. Any functions with multiple arguments can be redefined in this manner. For example, consider the *max2* function above. We can define a new function, *max*, where we would write *max*.*a*.*b*. What is the type of *max*? Make sure you can graph *max*.4 (which is a function in one argument).

1.2.2 Booleans and Predicates

Boolean refers to a set with precisely two elements: **true** and **false**. Many different symbols have been used historically (and are still popular today in various disciplines) to represent these two values, including T/F, \top / \perp , and 1/0.

There are many basic operations defined on booleans, all of which you are certainly familiar with. They include:

- conjunction (\wedge , pronounced “and”)
- disjunction (\vee , pronounced “or”)
- negation (\neg , pronounced “not”)
- equivalence (\equiv , pronounced “equivalents”)

- implication (\Rightarrow , pronounced “implies”)
- etc...

These operators can be viewed as functions from booleans to boolean. For example, \wedge .**true**.**false** is **false**. For convenience, we use infix notation. The previous expressions would therefore be written:

$$\mathbf{true} \wedge \mathbf{false}$$

Predicate. A predicate is function whose range is boolean. Thus, we can write:

$$P : S \rightarrow \text{boolean}$$

where S is an arbitrary domain.

For example, consider a function *is_tall*. The domain of this function is the set of students at Ohio State. For students taller than 6', this function evaluates to **true**, for all others it evaluates to **false**. This function is therefore a predicate, and the expression *is_tall.Sindhu* is therefore a boolean.

One tool for visualizing predicates is a Venn diagram. This is a crutch and can really hold you back if it is the only way you can reason about logical expressions. Nevertheless, crutches can be helpful. See Figure 1.1 for the graphical representation of the *is_tall* predicate.

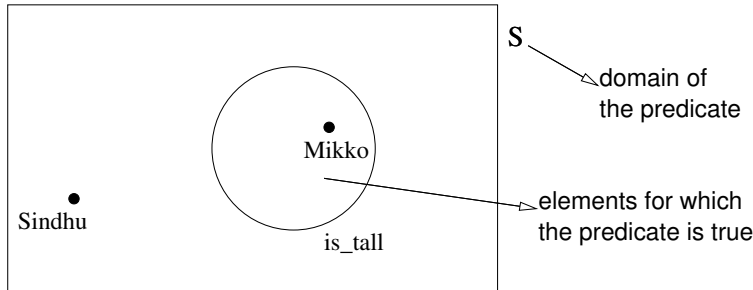


Figure 1.1: Venn Diagram of the *is_tall* Predicate

In computer science, the domain of a predicate is quite often the state space of a program. For example, in the context of a program with two variables, x and y , both integers, we might write expressions such as *even.x*, *prime.y*, and $x + y = 3$. These are all boolean expressions that are either true or false for each point in the state space.

1.2.3 Lifting

The distinction between boolean (a set with two elements: **true** and **false**) and predicate (a function with a particular range, *i.e.*, boolean) seems basic

and obvious. Nevertheless, this distinction quickly becomes blurred when we consider operations. The familiar operations on booleans (*e.g.*, \wedge , \vee) can also be applied to predicates! They are technically different operations, since they have different signatures, but confusion can arise because the same symbol is usually used.

$$\begin{aligned}\wedge & : \text{boolean} \times \text{boolean} \rightarrow \text{boolean} \\ \wedge & : \text{predicate} \times \text{predicate} \rightarrow \text{_____}\end{aligned}$$

Typically, the operator being used is clear from context (*i.e.*, from the types of the arguments). We could add subscripts to distinguish these operators, but this would needlessly clutter our expressions and mask many of the commonalities between these different versions of fundamentally similar functions (with fundamentally similar properties).

For example, consider the predicates *is_tall* and *is_heavy*. The expression *is_tall* \wedge *is_heavy* denotes a predicate! It is therefore a function that can be applied to particular elements of its domain, for example:

$$(\textit{is_tall} \wedge \textit{is_heavy}).\textit{Jeff}$$

This results in a boolean. See Figure 1.2 for a graphical representation.

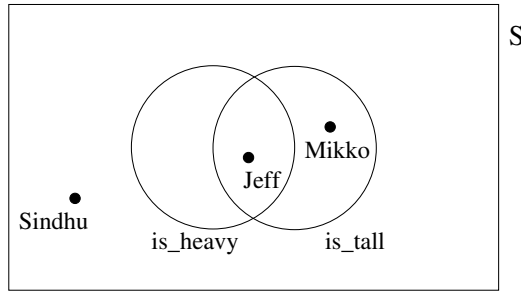


Figure 1.2: The Conjunction of Two Predicates

In this way, a simple operator on boolean (\wedge) has been elevated to operate on functions that map to boolean. This overloading of a single symbol is called *lifting*. Of course, the process of lifting can continue, elevating the operator to apply to functions that map to functions that map to boolean. And so on.

Lifting can be applied to the constants **true** and **false** as well. These symbols have been introduced as the elements of the set of boolean. But they can also be lifted to be predicates (the former mapping to **true** everywhere, and the latter to **false**). This leads us into the topic of what is meant by “everywhere”...

1.2.4 Everywhere Brackets

Consider the following expression:

$$\textit{is_tall} \equiv \textit{is_heavy}$$

Notice that this expression is a _____. Thus, we can evaluate it at various points in its domain, for example:

$(is_tall \equiv is_heavy).Joe$ is _____
 $(is_tall \equiv is_heavy).Mikko$ is _____
 $(is_tall \equiv is_heavy).Sindhu$ is _____

In this way, you can complete Figure 1.3 by shading the areas in which the element is mapped to **true**.

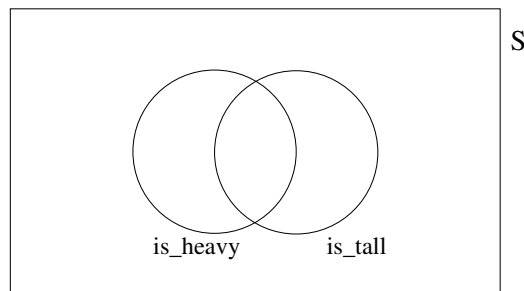


Figure 1.3: Graphical Representation of $is_tall \equiv is_heavy$

But what if we wanted to talk about the equivalence of the two predicates themselves? In other words, what if we want to state the claim “Being tall is the same as being heavy”? This claim is either true or false. It is not true sometimes and false other times. So what is going on here?

The problem is that the claim really involves an implicit quantification. Claiming that “Being tall is the same as being heavy” can more precisely be restated as “For every person, being tall is the same as being heavy”. Thus, we could use explicit quantification to state our claim. This issue turns out to be so pervasive, however, that it is worth introducing a short-hand for this quantification. We write:

$$[is_tall \equiv is_heavy]$$

The domain over which the quantification occurs is understood from context. When it matters, it is almost always the state space of some program under consideration. The square brackets are known as “everywhere brackets”.

Notice that the expression is now a _____. Indeed we can enclose any predicate with everywhere brackets, resulting in a _____.

$[is_heavy]$ is _____
 $[is_tall \vee is_heavy]$ is _____
 $[is_tall \vee \neg is_tall]$ is _____

1.3 The Predicate Calculus

1.3.1 Equivalence

We begin the introduction of the predicate calculus with arguably its most fundamental operator: equivalence. This operator is written \equiv and pronounced “equivalents”. Intuitively, $X \equiv Y$ means that both are true or both are false.

Why not use the more common operator $=$ (pronounced “equals”)? The main reason for choosing a different symbol is that the equals operator has a special meaning when there are multiple occurrences on the same line. For example, one might see:

$$\text{sqrt}.16 = 4 = 2^2$$

This is called “chaining” and it is really a short-hand for writing two conjuncts:

$$(\text{sqrt}.16 = 4) \wedge (4 = 2^2)$$

Equivalents can also appear multiple times on the same line, but—in general—we do not intend this to represent chaining. An expression with multiple occurrences of equivalents is evaluated (or simplified) directly. For example:

$$\begin{array}{c} \underbrace{\text{false} \equiv \text{true}} \equiv \text{false} \\ \underbrace{\hspace{1.5cm}} \equiv \text{false} \\ \hline \end{array}$$

But, rewriting this expression as if chaining were the intended interpretation leads to a different result:

$$\begin{array}{c} \underbrace{(\text{false} \equiv \text{true})} \wedge \underbrace{(\text{true} \equiv \text{false})} \\ \underbrace{\hspace{1.5cm}} \\ \hline \end{array}$$

Formally, equivalence is defined by its axioms.

Axiom 1. *Associativity of \equiv .*

$$[(X \equiv Y) \equiv Z] \equiv (X \equiv (Y \equiv Z))$$

This means that in an expression with multiple \equiv ’s, it doesn’t matter how it is parenthesized. Therefore, the parentheses will be omitted, and we are free to evaluate the equivalences in any order we wish. For example, we might write $[X \equiv Y \equiv Z]$, and we would then be free to interpret this as either $[(X \equiv Y) \equiv Z]$ or as $[X \equiv (Y \equiv Z)]$.

Axiom 2. *Commutativity of \equiv .*

$$[X \equiv Y \equiv Y \equiv X]$$

Here we make use of the associativity of \equiv to write the axiom without parentheses. The resulting axiom is quite rich. It captures the property of commutativity, with:

$$[(X \equiv Y) \equiv (Y \equiv X)]$$

It also, however, can be parenthesized as follows:

$$[X \equiv (Y \equiv Y) \equiv X]$$

This exposes an interesting fact: the expression $(Y \equiv Y)$ is both the left and right identity of \equiv . An identity element is worth naming, and we do so next.

Axiom 3. *Definition of **true**.*

$$[Y \equiv Y \equiv \mathbf{true}]$$

Again, notice the effect of different placements of parentheses. By interpreting this axiom as $[Y \equiv (Y \equiv \mathbf{true})]$ we see that **true** is indeed the (right) identity of \equiv . In other words, there is no difference between writing $[Y]$ and $[Y \equiv \mathbf{true}]$.

1.3.2 Disjunction

While \equiv is interesting, just one operator is a little limiting. So, we introduce a new operator, called disjunction and written \vee (pronounced “or”). To reduce the number of parentheses, we define an order of operations: \vee is defined to bind more tightly than \equiv .

We define this new operator by the following four axioms.

Axiom 4. *Associativity of \vee .*

$$[X \vee (Y \vee Z) \equiv (X \vee Y) \vee Z]$$

Axiom 5. *Commutativity of \vee .*

$$[X \vee Y \equiv Y \vee X]$$

Axiom 6. *Idempotence of \vee .*

$$[X \vee X \equiv X]$$

The last axiom describes how our two operators interact with each other.

Axiom 7. *Distribution of \vee over \equiv .*

$$[X \vee (Y \equiv Z) \equiv (X \vee Y) \equiv (X \vee Z)]$$

The definition of these two operators (*i.e.*, the axioms given above) form the foundation of our presentation of the predicate calculus. There are many other choices for where to begin this presentation. A different set of axioms could have been chosen and then the associativity of \vee , for example, derived as a theorem.

1.3.3 Proof Format

The axioms above give the basic properties of our operators. They can be used to derive further properties of these operators as theorems. A theorem must be proven from a set of axioms and theorems that have already been proven. To carry out these proofs, we will use a precise notation. This notation will help us structure our proofs so they are easy to read and verify. Perhaps surprisingly, this format also frequently makes proofs easier to write since it will often suggest the next step to be carried out.

Proofs are carried out in a calculational style. Each step in the proof involves the application of one (or more) axioms or theorems. Each step is annotated with a hint for the reader justifying that step. Each step should be small enough that a reader can be easily convinced the step is correct. Beware of steps that involve the application of multiple axioms or theorems! These steps are often where trouble arises. As you write proofs, you will develop a feel for the appropriate granularity of these steps: Too small and the proof is pedantic and hard to follow, but too large and the proof is obscure and hard to follow. Some axioms are used so ubiquitously they are frequently omitted (*e.g.*, commutativity of \equiv and of \vee). As a rule of thumb, error on the side of thoroughness.

As an general example of our calculational proof notation, consider a proof of $[A \equiv B]$. One such proof might take the form:

$$\begin{array}{l} A \\ \equiv \quad \{ \text{reason why } [A \equiv C] \} \\ C \\ \equiv \quad \{ \text{reason why } [C \equiv B] \} \\ B \\ \square \end{array}$$

A (less nice) alternative for such a proof would be:

$$\begin{array}{l} A \equiv B \\ \equiv \quad \{ \text{reason why } [A \equiv B \equiv D] \} \\ D \\ \equiv \quad \{ \text{reason why } [D \equiv \mathbf{true}] \} \\ \mathbf{true} \\ \square \end{array}$$

Writing proofs in this way will give us a common structure in which to read each other's proofs. It will also encourage a nice discipline of justifying steps in a precise manner. Notice, also, that expressions written in this format have a very different meaning than their straight-line counterparts. That is, the meaning of

$$\begin{array}{l} A \\ \equiv \\ C \\ \equiv \\ B \end{array}$$

is *not* the same as $[A \equiv C \equiv B]$. The former represents implicit chaining and should be interpreted as $[A \equiv C] \wedge [C \equiv B]$

As an example of a calculational proof, consider the following theorem.

Theorem 1. *true is the zero of \vee .*

$$[X \vee \mathbf{true} \equiv \mathbf{true}]$$

Proof.

$$\begin{aligned} & X \vee \mathbf{true} \\ \equiv & \{ \text{definition of } \mathbf{true} \} \\ & X \vee (Y \equiv Y) \\ \equiv & \{ \text{distribution of } \vee \text{ over } \equiv \} \\ & (X \vee Y) \equiv (X \vee Y) \\ \equiv & \{ \text{definition of } \mathbf{true}, \text{ with } Y \text{ as } X \vee Y \} \\ & \mathbf{true} \end{aligned}$$

□

1.3.4 Conjunction and Implication

Consider an expression written using only \equiv and \vee and with two or fewer variables. There are only a small number of structurally different expressions we can write that cannot be simplified using the properties of these operators. In fact, there are really only two that seem complex enough to warrant further investigation. They are: $X \vee Y \equiv X \equiv Y$ and $X \vee Y \equiv Y$.

Each of these expressions can be seen as an operation on two variables. So, we introduce two new binary operators as shorthands for these longer expressions.

Axiom 8. *Golden rule (also definition of \wedge).*

$$[X \vee Y \equiv X \equiv Y \equiv (X \wedge Y)]$$

Axiom 9. *Definition of \Rightarrow .*

$$[X \vee Y \equiv Y \equiv (X \Rightarrow Y)]$$

Again, to reduce the number of parentheses, we extend the order of operations to include these new operators. The symmetry in the Golden Rule suggests that \wedge have the same binding as \vee . \Rightarrow is defined to have a lower binding than these two, but higher than \equiv . With this order of operations, the two previous axioms are more succinctly written:

Axiom 10 (rewritten). *Golden rule (also definition of \wedge).*

$$[X \vee Y \equiv X \equiv Y \equiv X \wedge Y]$$

Axiom 11 (rewritten). *Definition of \Rightarrow .*

$$[X \vee Y \equiv Y \equiv X \Rightarrow Y]$$

From these axioms, we can now prove many interesting properties of \wedge and \Rightarrow .

Theorem 2. *true is the identity of \wedge .*

$$[X \wedge \mathbf{true} \equiv X]$$

$$\begin{aligned} \text{Proof. } & X \wedge \mathbf{true} \\ \equiv & \{ \\ & \\ \equiv & \{ \\ & \\ \equiv & \{ \\ & X \end{aligned}$$

□

Theorem 3. *Associativity of \wedge .*

$$[X \wedge (Y \wedge Z) \equiv (X \wedge Y) \wedge Z]$$

$$\begin{aligned} \text{Proof. } & X \wedge (Y \wedge Z) \\ \equiv & \{ \text{Golden Rule} \} \\ \equiv & \{ \text{Golden Rule twice} \} \\ \equiv & \{ \vee \text{ over } \equiv \text{twice} \} \\ & X \vee Z \equiv Y \vee Z \equiv X \vee Y \vee Z \equiv X \equiv Y \equiv X \vee Y \equiv Z \\ \equiv & \{ \vee \text{ over } \equiv \text{twice} \} \\ \equiv & \{ \text{Golden Rule twice} \} \\ \equiv & \{ \text{Golden Rule} \} \\ & (X \wedge Y) \wedge Z \end{aligned}$$

□

1.3.5 Negation and false

We have been examining binary operators. If we consider unary operators, there are only four possible choices. Three of these do not seem very interesting (the identity operator and two constant operators). The fourth is called negation. We introduce the symbol \neg and, since it is unary, give it a higher binding power than any of the previously introduced operators. It is defined by the following two axioms.

Axiom 12. *Law of Excluded Middle.*

$$[\neg X \vee X]$$

Axiom 13. *Distribution of \neg over \equiv .*

$$[\neg(X \equiv Y) \equiv X \equiv \neg Y]$$

Again, many interesting properties can be derived for this new operator.

Theorem 4. *\neg is an involution.*

$$[\neg\neg X \equiv X]$$

$$\begin{aligned} \text{Proof. } \neg\neg X &\equiv X \\ &\equiv \{ \quad \quad \quad \} \\ &\equiv \{ \quad \quad \quad \} \\ &\equiv \{ \quad \quad \quad \} \\ &\equiv \{ \quad \quad \quad \} \\ &\quad \text{true} \end{aligned}$$

□

Theorem 5. $[X \Rightarrow Y \equiv \neg X \vee Y]$

Proof. (For proof, see homework.)

□

We introduced a special symbol for the identity of \equiv (*i.e.*, **true**). It turns out to be useful to have a special symbol for the negation of this identity. It is written **false**, and defined by a single axiom.

Axiom 14. *Definition of false.*

$$[\neg\text{true} \equiv \text{false}]$$

Not surprisingly, **false** enjoys many interesting properties with respect to \vee , \wedge , \Rightarrow , \equiv , ...

1.3.6 Discrepance

The symbols introduced so far have been very familiar. It can also be a useful exercise for honing your skills with this calculational approach to consider less common operators. In particular, we now examine the discrepancy operator, written \neq (and pronounce “differs from”). We pursue this investigation primarily as an exercise in using and working within the calculation style.

Discrepance is defined by a single axiom.

Axiom 15. *Definition of \neq .*

$$[\neg(X \equiv Y) \equiv (X \neq Y)]$$

The following two properties are not very surprising.

Theorem 6. *Commutativity of \neq .*

$$[(X \neq Y) \equiv (Y \neq X)]$$

Theorem 7. *Associativity of \neq .*

$$[(X \neq (Y \neq Z)) \equiv ((X \neq Y) \neq Z)]$$

This operator “feels” (and looks!) a lot like \equiv , so a natural question would be how do these two interact? In particular, how does equivalence distribute over discrepancy (*i.e.*, $[X \equiv (Y \neq Z) \equiv ?]$)? It turns out, the following (perhaps somewhat surprising) property can be proven.

Theorem 8. *Mutual associativity of \equiv and \neq .*

$$[(X \equiv (Y \neq Z)) \equiv ((X \equiv Y) \neq Z)]$$

This means that in a chain of \equiv ’s and \neq ’s, we can leave off the parentheses! For example, we might write:

$$[A \equiv B \equiv C \neq D \equiv E \neq F \neq G]$$

The intuition behind the “meaning” of such an expression can be tricky, but the calculational style lets us manipulate it, massage it, and (possibly) simplify it. One useful property for working with discrepancy is:

Theorem 9. *Parity of \neq .*

$$[(X \neq Y \neq Z) \equiv \underline{\hspace{4cm}}]$$

Another name that is frequently used for discrepancy is “exclusive or”, sometimes written \oplus . This is an example of bad notation! This bad notation makes some basic transformations appear strange. For example, consider the distribution of \oplus over \equiv . Confronted with the expression $X \oplus (Y \equiv Z)$ it is tempting to write $\underline{\hspace{4cm}}$. It takes considerable force of will to write the correct result of the distribution, $(X \oplus Y) \equiv Z$. Good notation should suggest the rule.

Of course, even with good notation our intuition can lead us astray. For example, consider the distribution of \vee over \neq . A reasonable, intuitive, guess for such a rule might be:

$$[X \vee (Y \neq Z) \equiv \underline{\hspace{4cm}}]$$

But this is not correct. When we go to verify our intuition through calculation, we discover:

$$\begin{aligned} X \vee (Y \neq Z) \\ \equiv \{ \hspace{10em} \} \\ \equiv \{ \hspace{10em} \} \end{aligned}$$

$$\begin{array}{lcl}
\equiv & \{ & \} \\
\equiv & \{ & \} \\
\equiv & \{ & \} \\
\equiv & \{ & \}
\end{array}$$

1.4 Quantification

1.4.1 Syntax

You are certainly familiar with summation and product notation. For example, you can immediately calculate:

$$\begin{array}{lcl}
\sum_{i=2}^5 i & = & \text{_____} \\
\prod_{i=1}^4 i & = & \text{_____}
\end{array}$$

But you may not have thought very carefully about the definition of these symbols. The trouble with intuitive understandings is that they can be ambiguous. For example, what would you say is the value of $\prod_{i=4}^1 i$?

The summation and product expressions above are actually both examples of quantification! You’ve certainly encountered quantification in the form of “for all” and “there exists”, but in this section we will present a general definition (and syntax) of quantifications that subsumes these particular forms. The general notation will allow us to introduce and work with new forms of quantification that will prove to be convenient short-hands.

A quantification has the form:

$$(\mathbf{Q} i : r.i : t.i)$$

where \mathbf{Q} is the “operator”, i is the “bound variable”, $r.i$ is the “range”, and $t.i$ is the “term”. In order to be a valid quantification expression, these elements of the expression must satisfy certain constraints.

- The operator must be a binary, symmetric, associative operator with an identity element.
- The range must be a predicate on the bound variable i .

- The term must be an expression (that may contain i) and the type of this expression must be the same as the type of the operands of the operator.

Thus, for a bound variable of type T , the range is a predicate on T , the term is an expression of some type D , and the quantifier an operation of type $D \times D \rightarrow D$.

1.4.2 Informal Interpretation

To understand the meaning of a quantified expression, an informal understanding of its expansion is a useful place to start. The predicate $r.i$ defines a set of values for the bound variable i (*i.e.*, the set of values for which the predicate is true). Now, if that set of values for which $r.i$ holds is $\{i_0, i_1, i_2, \dots, i_N\}$, the quantified expression is a short-hand for the expanded expansion:

$$u \mathbf{Q} t.i_0 \mathbf{Q} t.i_1 \mathbf{Q} t.i_2 \mathbf{Q} \dots \mathbf{Q} t.i_N$$

where u is the identity element of the operator \mathbf{Q} .

For example, consider the expansion of:

1.

$$\begin{aligned} & (* i : 1 \leq i \leq 4 : i) \\ & = \\ & = \end{aligned}$$

2.

$$\begin{aligned} & (\wedge n : n \geq 0 : even.n) \\ & = \\ & = \end{aligned}$$

For common quantifications, special symbols have been used historically as the quantification symbol. See Table 1.1.

Operator	Quantification Symbol
\wedge	\forall
\vee	\exists
$+$	\sum
$*$	\prod

Table 1.1: Common Symbols for Quantification

Our definition of quantification is more general than these four symbols, however. Any appropriate operator (*i.e.*, satisfying the requirements) can be used.

For example, you should now understand the following quantified expressions and be able to write out their expansions:

1. $(\cup i : i \in \mathbb{Z} \wedge \text{even}.i : \{i\})$
2. $(\mathbf{Min} n : n \in \mathbb{N} \wedge \text{even}.n : (n - 3)^2)$

As a short-hand, the “type” of the bound variable is often understood from context and/or from convention. For example, i is often used as a bound variable ranging over the integers, while n is used as a bound variable ranging over the naturals. Thus, the second quantification above could be more succinctly written:

$$(\mathbf{Min} n : \text{even}.n : (n - 3)^2)$$

As a further short-hand, when the range is the predicate **true**, it can be omitted all together. For example, we write:

$$(\forall i :: i^2 > 0)$$

1.4.3 Definition

This section began by posing the question concerning the meaning of $\prod_{i=4}^1 i$. Armed now with our informal understanding of the expansion of quantification, we are prepared to write out the solution.

$$\begin{aligned} & \prod_{i=4}^1 i \\ = & (*i : 4 \leq i \leq 1 : i) \\ = & \end{aligned}$$

Like the operators in the predicate calculus, quantification is properly defined by its axioms. The most basic axiom for quantification addresses the question posed at the beginning of this section.

Axiom 16. *Empty range.*

$$(\mathbf{Q} i : \mathbf{false} : t.i) = u$$

(Recall that u is the identity element of the **Q** operator.)

As another example of the application of this axiom, consider:

$$(\forall x : x^2 < x : x = 23) \equiv \underline{\hspace{2cm}}$$

So, it is important that the operator of a quantification have an identity element. For example, consider the quantification expression using minimum. What is the identity element of the **Min** operator? Knowing this identity element is what allows us to expand the following quantification:

$$(\mathbf{Min} x : \mathbf{false} : t.x) = ???$$

In other words, we need to find a u such that $u \mathbf{Min} x = x$, for any x . This suggests the correct value for u is _____. Therefore, the previous expression is properly simplified as:

$$(\mathbf{Min} x : \mathbf{false} : t.x) = \underline{\hspace{2cm}}$$

Axiom 17. *One-point rule.*

$$(\mathbf{Q} i : i = E : t.i) = t.E$$

Several more axioms are required for a complete definition of quantification. These axioms can be used to prove many useful theorems for manipulating quantification in exactly the same manner as the previous section dealt with the basic predicate calculus. The remaining axioms for quantification, however, are not presented here. We will be content with an informal understanding of quantification (and an understanding of how a more rigorous definition would be presented and used).

Chapter 2

The Computational Model

In this chapter, we introduce a notation for writing distributed programs. We also present the abstract computational model in which these programs execute. This notation is not directly supported by any real compiler and is therefore not appropriate for implementing distributed systems. It is convenient, however, for discussing distributed algorithms and reasoning about their correctness.

2.1 References

Our programming notation and model of computation follows very closely the presentation in “Parallel Program Design: A Foundation” by K.M. Chandy and J. Misra (Addison-Wesley, 1988) [CM88]. This is the canonical reference for UNITY logic.

2.2 Programs

Programs consist of two things:

1. A set of typed variables.
2. A finite set of assignments (also called “actions”).

The first thing to notice is that the collection of assignments is a *set*. Assignments are therefore not ordered!

The variables are declared at the beginning of the text of the program, in a section beginning with the keyword **var**. The assignments are listed in the next section (called the “assign” section), indicated by the keyword **assign**. A “fatbar” (\parallel) is used to separate assignments.

As a simple example of this notation, consider the following program:

Program	<i>Trivial</i>
var	$x, y : \text{int}$

```

assign
   $x := 2$ 
   $\parallel y := f.7$ 

```

The type of the variables will be omitted when understood from context.

By convention, all programs implicitly contain the simplest assignment of all: **skip**. This assignment leaves the state unchanged (*i.e.*, it does not modify any variables). Even if **skip** is not explicitly listed in the assign section, it is still part of the program's set of assignments. As a consequence, no program has an empty set of assignments.

Optionally, programs may also contain some initial conditions. These are listed in the “initially section”, denoted by the keyword **initially**. This section (if present) is included after the variable declarations and before the assign section. Note that the initially section is a predicate (on the program variables).

2.3 Actions

2.3.1 Simple Assignments

Each action must terminate. Since an action is just an assignment, this is not difficult to guarantee. An assignment can, however, be nondeterministic. For example, the following action assigns a random integer in the interval $[1..10]$ to x :

$$x := \text{rand}(1, 10)$$

Actions must, however, be total. That is, they must be defined for every possible program state. So, in a program with two integer variables, x and y , the following assignment is not permitted:

$$x := x/y$$

This action is not defined when y is 0.

2.3.2 Multiple Assignments

There can also be multiple assignments in a single action. Two notations exist for writing multiple assignments. The first is more compact:

$$x, y := 2, f.3$$

The meaning of such a multiple assignment is that, in one action, all expressions on the right-hand side are evaluated and then the assignments to the variables listed on the left-hand side are made. So, the values of two variables are swapped by the action:

$$x, y := y, x$$

The second notation is a little less compact, but lends itself to quantification. A parallel operator (\parallel) is used to separate the multiple assignments. The swapping example above would be written:

$$x := y \parallel y := x$$

2.3.3 Guarded Actions

The execution of an assignment can be conditional on a predicate (called the “guard”) being true. For example:

$$x > 0 \longrightarrow x, y := 2, f.3$$

The guard is a predicate on the state space of the program. If the guard is true in some state, the action is said to be “enabled” in that state.

2.3.4 Sequential Composition

Recall that actions are not ordered. Ordinary sequential execution of a collection of actions is therefore not part of the basic notation. It is not difficult, however, to simulate sequential execution. In order to do this, an explicit program counter is introduced and guarded statements are used to ensure that only the correct (*i.e.*, “next”) action is enabled at any time.

For example, the following program is a sequential implementation of the traditional swap algorithm for two variables.

Program	<i>SequentialSwap</i>
var	$x, y, temp : \text{int},$
	$pc : \text{nat}$
initially	$pc = 1$
assign	
	$pc = 1 \longrightarrow temp, pc := x, 2$
	$\parallel pc = 2 \longrightarrow x, pc := y, 3$
	$\parallel pc = 3 \longrightarrow y, pc := temp, 4$

2.4 Operational Intuition

2.4.1 Program Execution

Although we will reason about our programs and algorithms assertionally, it can be helpful to have an informal, operational intuition of the model of execution of these programs.

A program can begin in any state satisfying the initially predicate. Execution then proceeds by an action in the set of program actions being nondeterministically selected and executed. Once this action has completed, an action is again non-deterministically selected and executed. This process is repeated infinitely.

You can think of each action being written on a separate piece of paper. All the pieces of paper are placed in a hat. Program execution consists of reaching into the hat, selecting an piece of paper, and executing the specified action. The piece of paper is then placed *back* into the hat, and the selection process repeated.

There are very few restrictions on which action is chosen. An action could be chosen several times in a row, for example. There are, however, some restrictions. These are termed “fairness” properties and will be discussed at the end of this chapter.

What if the selected action is guarded and the guard is false? In this case, the action is simply a skip (*i.e.*, it does not change the program state).

Another natural question to ask is when does a program terminate? When do we consider the computation to be “over”? We address this issue next.

2.4.2 Termination

The nondeterministic selection of actions continues without end. There is, therefore, no “last” action picked and executed. Consider, however, the following program:

```

Program      Boring
initially     $x = 2 \wedge y = f.7$ 
assign
     $x := 2$ 
     $\parallel y := f.7$ 

```

Clearly this program is not very exciting. Whichever action is selected, the state of the program does not change. The selection of actions continues forever, but the state of the program remains fixed. The computation can certainly be viewed as “done”.

So what we care about, then, for termination is the arrival of the program execution to a fixed point (abbreviated *FP*). A fixed point is a state in which the execution of any action leaves the state unchanged.

By looking at the assign sections of programs, we can calculate their fixed points. For example, consider a program with an assign section:

```

assign
     $x := y$ 
     $\parallel y := f.7$ 

```

The *FP* is: _____. Notice that *FP* is a predicate that characterizes the states in which the program is at a fixed point. The calculation of this *FP* is indicative of a common paradigm: take each assignment and change the $:=$ operator to equality, then take the conjunction of the resulting set of predicates.

When some actions are guarded, we have to be a little more careful. Consider a program with an assign section:

assign
 $x = y \longrightarrow x := 2$
 $\parallel y := f.7$

In this case, $FP = \underline{\hspace{2cm}}$.

In general, to calculate FP , require each action to have no effect. So, for a guarded action of the form $g \longrightarrow x := E$, we require: $\underline{\hspace{2cm}}$.

2.4.3 An Example: Find Max

Given A , an array $0..N - 1$ (where $N \geq 1$) of integers, we are required to calculate the maximum integer in the array. In a sequential programming language, we would write a loop that iterates through the array. In our notation here, however, we do not have sequential composition (let alone iteration).

As an exercise, it is worthwhile to try and write the solution yourself. Once you have come up with something, take a look at the following program.

Program *FindMax*
var $A : \text{array } 0..N - 1 \text{ of int,}$
 $result : \text{int}$
initially $result = A[0]$
assign
 $(\parallel x : 0 \leq x \leq N - 1 : result := \max(result, A[x]))$

The first thing to notice about this example program is the assignment section. The one line in this section is actually a quantification! The operator is \parallel , the free variable is x , the range is $0 \leq x \leq N - 1$, and finally the term is $result := \max(result, A[x])$. As an exercise, can you write out the expansion of this quantification? You should see the result is a set of assignment actions. What do you suppose is the identity element for the \parallel operator?

Next, consider the fixed point of this example. In order for the first action in the set (*i.e.*, in the quantification) to leave the state unchanged, it must be the case that:

$$result = \max(result, A[0])$$

Equivalently, this can be written:

$$result \geq A[0]$$

Therefore, in order for no action to change the state of the program, it must be the case that:

$$(\forall x : 0 \leq x \leq N - 1 : result \geq A[x])$$

The “path” taken to this fixed point, however, is nondeterministic.

2.5 Visualizing a Program

2.5.1 Directed Graphs

Programs have been defined as textual elements, with a specific notation for variable declarations, assignment actions, etc. It can be helpful to also consider a more visual representation of a program. Again, this visual representation should be viewed as a crutch. The idea is to help build some informal intuition about programs and program executions.

To begin, each state in the program state space is represented by a vertex. Each action in the program is represented by directed arrows. A program is therefore represented by a directed graph, for example see Figure 2.1.

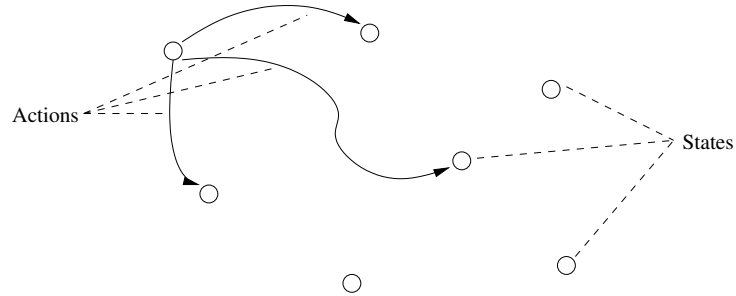


Figure 2.1: Programs as Directed Graphs

Recall that there were several requirements placed on programs in the previous section. These requirements each have implications on the nature of the directed graphs that we end up drawing to represent programs. In particular:

- All programs include **skip**. This means that all directed graphs include _____. These edges corresponding to the implicit **skip** action are usually omitted from the graph (and understood to be implicitly present).
- All actions are total (*i.e.*, defined for every state). This means that each vertex has _____.

You can think of each edge as being labeled (or colored) by its corresponding action. Complete Figure 2.2 with an example of a possible program, with five states in its state space and three actions (not including the implicit **skip**).

2.5.2 An Exercise

Consider the following program:

Program	<i>Example1</i>
var	<i>b</i> : boolean,

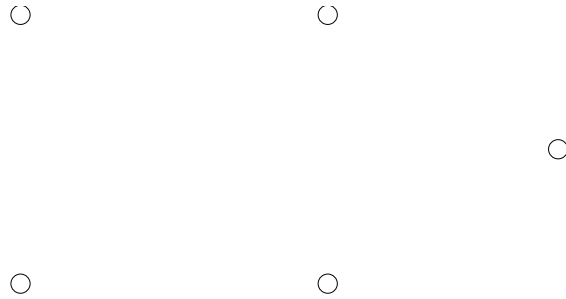


Figure 2.2: A Program with Five States and Three Actions

```

           $n : \{0, 1, 2, 3\}$ 
initially     $n = 1$ 
assign
     $b \longrightarrow n := n +_4 1$ 
     $\parallel n = 0 \longrightarrow b := \mathbf{false}$ 

```

The addition in the first assignment is performed modulo 4. Draw the corresponding directed graph in the space provided by Figure 2.3. Draw the vertices and the directed edges. Identify which edges correspond to which action (by labels or by color). Also, circle the set of states that satisfy the initially predicate.

Figure 2.3: Directed Graph Corresponding to Program Example 1

2.6 Fairness

2.6.1 Motivation

Because every program includes implicitly a **skip** action, every program (apart from the trivial one, with an empty assign section) consists of multiple actions. How do we choose which action is executed next? Recall our informal model of computation had us reaching into a hat each time and picking an action. What if we are extremely unlucky and happen to pick the **skip** action each and every time? None of the other actions are ever chosen and the program does nothing.

To prevent this kind of “unlucky” selection, we impose a *fairness* requirement on the selection of actions. There are two kinds of fairness: weak and strong.

2.6.2 Weak Fairness

Under weak fairness, every action is guaranteed to be selected infinitely often. This means that between any two selections of some particular action (A), there are a finite (but unbounded) number of selections of other (*i.e.*, not A) actions.

For example, in the Find Max example, weak fairness requires each action to be selected (eventually). There is no guarantee (or finite bound) on how quickly (*i.e.*, how many selections) will be required before all actions have been selected at least once.

In the context of the directed graph representation of programs, what does weak fairness require? A computation is an infinite path in the directed graph and weak fairness says that in this path, each edge label _____.

As another example, consider program Example 1. This program begins execution in a state satisfying the predicate $n = 1$. If the initial state is the state $\langle \mathbf{false}, 1 \rangle$ (*i.e.*, the state in which $b = \mathbf{false}$ and $n = 1$), then every possible computation leaves the state unchanged. If, on the other hand, the initial state is the state $\langle \mathbf{true}, 1 \rangle$, then what can we say about the behavior of this program? More specifically, is a fixed point reached and if so, what is this fixed point?

Under an assumption of weak fairness, there is no guarantee that the Example 1 program reaches a fixed point when it begins in the state $\langle \mathbf{true}, 1 \rangle$. This is because it can cycle between the four states: $\langle \mathbf{true}, 1 \rangle$, $\langle \mathbf{true}, 2 \rangle$, $\langle \mathbf{true}, 3 \rangle$, and $\langle \mathbf{true}, 0 \rangle$. The action that assigns **false** to b must be selected an infinite number of times, but it may be selected every time $n = 3$, when its guard is false and the action therefore has no effect. In such a computation, every action is selected infinitely often and it therefore satisfies the weak fairness requirement.

2.6.3 Strong Fairness

The previous example suggests that a stronger fairness requirement can be made. Weak fairness was introduced to prevent an extraordinarily unlucky (one might

even say malicious) selection in which one action (or more) is perpetually ignored. In the previous example, the selection was also extraordinarily unlucky (one might even say malicious). One action was only chosen at particular times, when it happened to be unenabled.

Strong fairness requires that each action be selected infinitely often (like weak fairness), and furthermore, that if an action is enabled infinitely often, it is selected infinitely often. Under a model of computation with strong fairness, the program Example 1 would not be allowed to cycle through the state $\langle \mathbf{true}, 0 \rangle$ infinitely often without selecting the action that assigns **false** to b . Under this model, therefore, the program would be guaranteed to reach a fixed point.

To further contrast “strong” and “weak” fairness, you should notice that if we reason about a program using weak fairness, and prove some property, P , then P is indeed a property of the program regardless of whether the actual execution is strongly or weakly fair. Conversely, however, if we prove a property of a program assuming strong fairness, then this property may not hold under a weakly fair model of execution.

In our model, we will assume weak fairness.

Chapter 3

Reasoning About Programs

In this chapter, we address the question: what does a given program do? Often it is tempting to use operational arguments, that is, arguments that try to follow all possible executions. Such arguments are cumbersome, however, and frequently rely heavily on case analysis and long chains of causality. Instead, we would prefer to answer the question “what does this program do” in a manner that convinces ourselves (and others) that our answer is correct.

3.1 References

- The best reference for this material is again the classic “Parallel Program Design: A Foundation”, by K. M. Chandy and J. Misra [CM88]. Our presentation here, however, is quite different. For example, our basic temporal operators, **transient** and **next**, do not appear in the Chandy & Misra book. Nevertheless, the fundamental ideas (such as the Assignment Axiom) are the same.
- For another presentation of **transient** and **next**, you can look at Jay Misra’s book “A Discipline of Multiprogramming” [Mis01]. (The **next** operator that we use is introduced as **co** in this reference.)

3.2 Introduction

3.2.1 Motivational Example

Recall the example given in the previous chapter for finding the maximum element in an array.

Program	<i>FindMax</i>
var	$A : \text{array } 0..N - 1 \text{ of int,}$
	$r : \text{int}$
initially	$r = A[0]$

assign
 ($\parallel x : 0 \leq x \leq N - 1 : r := \max(r, A[x])$)

Intuitively, this program seems “right”. In other words, it seems to calculate the maximum of an array of integers. We can convince ourselves, with operational and intuitive arguments, that it will eventually terminate. But what is the fixed point of this program? Recall our recipe for calculating the fixed point, where each assignment is an equality. Therefore:

$$[FP \equiv (\forall x : 0 \leq x \leq N - 1 : r = \max(result, A[x]))]$$

In the previous chapter, we simplified this expression and saw that:

$$[FP \equiv r \geq (\mathbf{Max} x : 0 \leq x \leq N - 1 : A[x])]$$

But we believe that the program is “correct” and that r is eventually *equal* to the maximum, not greater than it.

In this chapter we will formalize our intuition of what it means for a program to be correct. We will see the tools for proving that our programs are indeed correct without relying on operational arguments.

3.2.2 Specification

Reasoning about a program involves convincing ourselves, and others, that the program meets its specification. A specification is some kind of higher-level description of program behavior. The question is how to write such higher-level descriptions.

We will use *program properties* to specify our programs. Recall that non-determinism is fundamental (even inescapable) in our model of computation. There are therefore many possible executions (also called “computations” or “traces”). A program property is a predicate on an execution. We say that a program property R holds for a program P exactly when R holds for every possible execution P .

There are two fundamental categories of program properties that we will use to describe program behavior: safety properties, and progress properties. Before considering these properties on computations (consisting of an infinite sequence of actions), we first examine how to reason about the behavior of a single action, executed once.

3.3 Reasoning About a Single Action

3.3.1 Hoare Triples

A Hoare triple is an expression of the form

$$\{P\} \ S \ \{Q\}$$

where S is a program action and P and Q are predicates. The predicate P is often called the “precondition” and Q the “postcondition”. Informally, this triple says that if S begins execution in a state satisfying predicate P , it is guaranteed to terminate in a state satisfying Q .

As an aside, a distinction is usually made at this point between *total* and *partial* correctness. The definition we have given above suggests total correctness. For partial correctness, the statement S must either terminate in a state satisfying Q , or not terminate at all. For our purposes, no distinction is required between total and partial correctness. This is because our statements are _____.

3.3.2 The Assignment Axiom

Our programming notation has a single kind of action, an assignment. To reason about a single action, then, we must understand how to reason about assignment. In particular, we must answer the question: “When does the triple $\{P\} \ x := E \ \{Q\}$ hold?”

Put another way, this question asks: “In what state must we begin execution of $x := E$ in order for Q to be true afterwards?” This question is represented graphically in Figure 3.1.

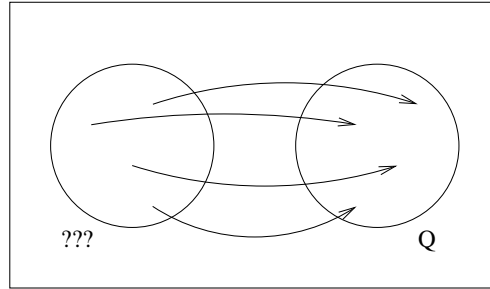


Figure 3.1: Guaranteeing that Q holds after $x := E$

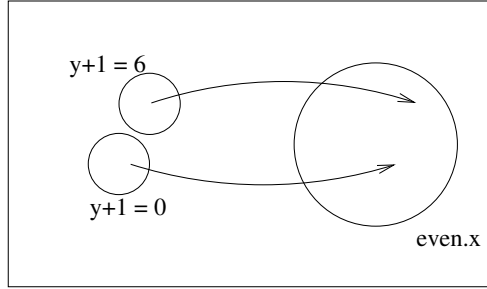
For example, consider the following triple:

$$\{P\} \ x := y + 1 \ \{even.x\}$$

In order for this to be true, $even.x$ must be true after execution of the assignment. So what must have been true before execution? It must have been the case that $y + 1$ was even! For example, $\{y + 1 = 6\} \ x := y + 1 \ \{even.x\}$ holds. As does the triple $\{y + 1 = 0\} \ x := y + 1 \ \{even.x\}$. Both of these are represented in Figure 3.2.

Indeed, saying that a triple $\{P\} \ x := y + 1 \ \{even.x\}$ holds is the same as saying:

$$[P \Rightarrow \underline{\hspace{2cm}}]$$

Figure 3.2: Two Preconditions that Guarantee the Postcondition *even.x*

In general, for the statement $x := E$, if x satisfies some predicate Q afterwards, it must be the case that E satisfied that predicate before. In other words, to prove $\{P\} \ x := E \ \{Q\}$, we must show $[P \Rightarrow Q_E^x]$. (The notation Q_b^a is used to indicate writing expression Q with all occurrences of a replaced by b).

For example, consider the triple: $\{x \geq -2\} \ x := x - y + 3 \ \{x + y \geq 0\}$. Does this triple hold? In order to prove that it holds, we must prove:

$$[x \geq -2 \Rightarrow (x - y + 3) + y \geq 0]$$

Proof. $(x - y + 3) + y \geq 0$

\equiv

\equiv

\Leftarrow

□

3.3.3 Guarded Actions

If the action is guarded, the corresponding proof obligation is slightly different from the straight assignment action. To prove the triple $\{P\} \ g \longrightarrow x := E \ \{Q\}$, we must consider two cases: the guard g may either be true or false at the start of execution. In these two cases, the action does different things (an assignment in the first case and nothing in the second). Either way, the postcondition Q must be satisfied.

To prove this triple, we must discharge the following proof obligation:

$$[(P \wedge g \Rightarrow Q_E^x) \wedge (P \wedge \neg g \Rightarrow Q)]$$

For example, consider the following triple:

$$\{x > y = 7\} \ x > y \longrightarrow x, y := y, x \ \{x > 3\}$$

We must prove:

$$[(x > y = 7 \wedge x > y \Rightarrow y > 3) \wedge (x > y = 7 \wedge \neg(x > y) \Rightarrow x > 3)]$$

$$\begin{aligned}
\text{Proof. } & (x > y = 7 \wedge x > y \Rightarrow y > 3) \wedge (x > y = 7 \wedge \neg(x > y) \Rightarrow x > 3) \\
\Leftarrow & \{ \text{antecedent strengthening of } \Rightarrow : [(X \Rightarrow Z) \Rightarrow (X \wedge Y \Rightarrow Z)] \} \\
& (y = 7 \Rightarrow y > 3) \wedge (x > y = 7 \wedge \neg(x > y) \Rightarrow x > 3) \\
\equiv & \{ 7 > 3 \} \\
& x > y = 7 \wedge \neg(x > y) \Rightarrow x > 3 \\
\equiv & \{ \text{definition of } \neg \} \\
& x > y = 7 \wedge x \leq y \Rightarrow x > 3 \\
\Leftarrow & \\
& x > y \wedge x \leq y \Rightarrow x > 3 \\
\equiv & \\
& \mathbf{false} \Rightarrow x > 3 \\
\equiv & \{ \text{property of } \Rightarrow : [\mathbf{false} \Rightarrow X \equiv \mathbf{true}] \} \\
& \mathbf{true}
\end{aligned}$$

□

As another exercise, consider the following triple:

$$\{j \neq k\} \quad i = k \longrightarrow j := k \quad \{i = j\}$$

We must prove:

$$[(j \neq k \wedge i = k \Rightarrow i = k) \wedge (j \neq k \wedge \neg(i = k) \Rightarrow i = j)]$$

Proof. We attempt this proof by first simplifying:

$$\begin{aligned}
& (j \neq k \wedge i = k \Rightarrow i = k) \wedge (j \neq k \wedge \neg(i = k) \Rightarrow i = j) \\
\Leftarrow & \\
& (i = k \Rightarrow i = k) \wedge (j \neq k \wedge \neg(i = k) \Rightarrow i = j) \\
\equiv & \\
& j \neq k \wedge \neg(i = k) \Rightarrow i = j \\
\equiv & \\
& j \neq k \wedge i \neq k \Rightarrow i = j
\end{aligned}$$

The resulting expression is clearly not, however, equivalent to **true**! Indeed, for i, j, k with values 1, 2, and 3, this expression is false. Therefore, the triple does not hold. □

We have seen how to reason about the correctness of individual assignments, executed once. Next, we consider reasoning about a computation, that is, an infinite sequence of selections and executions of individual assignment actions.

3.4 Safety

Informally, a safety property says that “nothing bad happens”. One litmus test for a safety property is that it is a property that can be violated by a *finite* computation.

Example 1. “This pen does not explode”. If we watch the pen for 1 minute, and it does not explode, safety has not been violated. Conversely, if the pen were to violate this safety property and explode, could we conclude that it had exploded after a finite computation? Certainly.

Example 2. Consider a light switch with the property: “When the switch is off, the light is off too”. Again, if a switch were to violate this property (*i.e.*, the light comes on while the switch is off), we could observe this violation after a finite computation.

Example 3. In the FindMax program above, the variable *result* is always equal to an element of the array *A*.

3.4.1 Next

Our most basic operator for safety will be **next**. A **next** property (*i.e.*, a predicate on programs) is written:

$$P \text{ next } Q$$

where *P* and *Q* are predicates on *states* in the program. To make explicit the program to which this property refers, we could write $(P \text{ next } Q).G$, where *G* is a program. Typically, however, the program is understood from context and is omitted.

Informally, the property $P \text{ next } Q$ means that if a program is in a state satisfying predicate *P*, its very next state (*i.e.*, after choosing and executing exactly one action) must satisfy *Q*.

How could we prove that a program satisfies such a property? Since *any* action could be chosen as the next one to be executed, we must show that *every* action, if it begins in *P*, must terminate in *Q*.

Proof Rule. To prove

$$(P \text{ next } Q).G$$

we must show

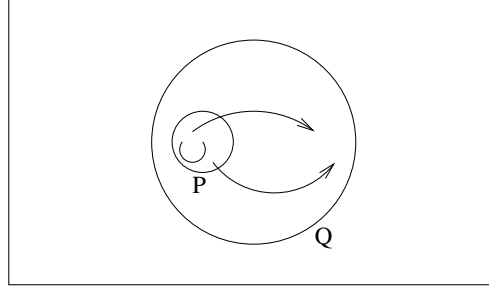
$$(\forall a : a \in G : \{P\} \text{ a } \{Q\})$$

One subtlety is that skip is always part of any program. So, in order to have $P \text{ next } Q$ one of the things we will have to show is $\{P\} \text{ skip } \{Q\}$. What does this mean about how *P* and *Q* are related? _____.

Figure 3.3 is a graphical representation of a **next** property.

The predicate *Q* can be seen as a “constraint” on how far the program can move out of *P* in one step. Of course, once outside of *P* (but possibly still in *Q*) this property says nothing about what must happen next.

As an exercise, consider which of the following theorems are true for **next**.

Figure 3.3: Graphical Representation of $P \text{ next } Q$

Constants.

false	next	Q
P	next	true
true	next	false

Junctivity.

$$\begin{aligned} (P_1 \text{ next } Q_1) \wedge (P_2 \text{ next } Q_2) &\Rightarrow (P_1 \wedge P_2) \text{ next } (Q_1 \wedge Q_2) \\ (P_1 \text{ next } Q_1) \wedge (P_2 \text{ next } Q_2) &\Rightarrow (P_1 \vee P_2) \text{ next } (Q_1 \vee Q_2) \end{aligned}$$

Weakening the RHS.

$$(P \text{ next } Q) \wedge [Q \Rightarrow Q'] \Rightarrow (P \text{ next } Q')$$

Strengthening the LHS.

$$(P \text{ next } Q) \wedge [P' \Rightarrow P] \Rightarrow (P' \text{ next } Q)$$

As a simple example, consider a program with a single assignment statement, $x := x + 1$. For this program, the **next** property we can write is:

$$(\forall k :: x = k \text{ next } \underline{\hspace{2cm}})$$

As a short-hand, we will frequently leave the universal quantification as implicit. So we will write:

$$x = k \text{ next } \underline{\hspace{2cm}}$$

3.4.2 Stable

Stability means that once something becomes true, it remains true. We write **stable**. P , where P is a predicate, to indicate that P is stable. If this is a property of program G , this would be written **stable**. $P.G$ (recall that function application associates to the left). Often G is understood from context.

Proof Rule. Formally, stability is defined by:

$$\mathbf{stable}.P \equiv P \mathbf{next} P$$

Again, as an exercise, decide which of the following theorems are valid for **stable**.

$$\begin{aligned} &\mathbf{stable.true} \\ &\mathbf{stable.false} \\ &\mathbf{stable}.P \wedge \mathbf{stable}.Q \Rightarrow \mathbf{stable}.(P \wedge Q) \\ &\mathbf{stable}.P \wedge \mathbf{stable}.Q \Rightarrow \mathbf{stable}.(P \vee Q) \\ &\mathbf{stable}.P \wedge [P \Rightarrow P'] \Rightarrow \mathbf{stable}.P' \\ &\mathbf{stable}.P \wedge [P' \Rightarrow P] \Rightarrow \mathbf{stable}.P' \end{aligned}$$

For our simple example of a program with a single assignment, $x := x + 1$, what is a stable property?

3.4.3 Invariant

A stable predicate that holds initially is said to be an *invariant*. That is, this predicate is true for the entire computation.

Proof Rule. Formally, then, the definition of invariance is:

$$\mathbf{invariant}.P \equiv \mathbf{initially}.P \wedge \mathbf{stable}.P$$

For example, consider the FindMax example. Let M be the maximum value in the array. That is:

$$M = (\mathbf{Max} x : 0 \leq x < N : A[x])$$

The following is a property of Findmax:

$$\mathbf{invariant}.(r \leq M)$$

Proof. First observe the following (from the properties of maximum):

$$(\forall x : 0 \leq x < N : A[x] \leq M)$$

There are two proof obligations:

1. **initially**.($r \leq M$)

$$\begin{aligned} &r = A[0] \\ \Rightarrow &\{ A[0] \leq M \} \\ &r \leq M \end{aligned}$$

2. **stable**.($r \leq M$)

$$\begin{aligned}
 & \mathbf{stable}.(r \leq M) \\
 \equiv & \\
 & (r \leq M) \mathbf{next} (r \leq M) \\
 \equiv & \\
 & (\forall a :: \{r \leq M\} \quad a \quad \{r \leq M\}) \\
 \equiv & \quad \{ \text{definition of program} \} \\
 & (\forall x : 0 \leq x < N : \{r \leq M\} \quad r := \max(r, A[x]) \quad \{r \leq M\}) \\
 \equiv & \quad \{ \text{assignment axiom} \} \\
 & (\forall x : 0 \leq x < N : \underline{\hspace{10em}}) \\
 \equiv & \quad \{ \text{lemma} \} \\
 & (\forall x : 0 \leq x < N : r \leq M \Rightarrow r \leq M) \\
 \equiv & \quad \{ \text{predicate calculus} \} \\
 & \mathbf{true}
 \end{aligned}$$

□

3.4.4 Unless

Informally, P **unless** Q means that if P is true at some point, it remains true unless Q becomes true, at which point all bets are off. Conceptually, Q is like a “gate” through which the program must pass.

Consider Figure 3.4 with predicates P and Q . Complete this figure with possible state transition arrows. There are 3 areas of the figure which can be considered separately:

1. $P \wedge \neg Q \longrightarrow \underline{\hspace{2cm}}$
2. $\neg P \wedge Q \longrightarrow \underline{\hspace{2cm}}$
3. $P \wedge Q \longrightarrow \underline{\hspace{2cm}}$

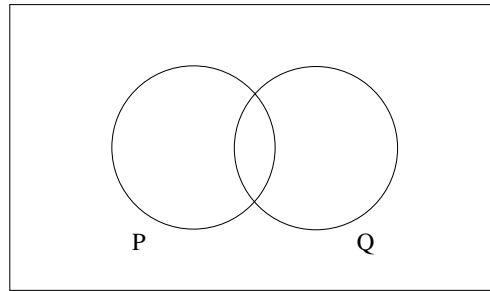


Figure 3.4: Graphical Representation of P **unless** Q

As an exercise, write down the definition of **unless** in terms of **next**.

Safety properties form an important class of program properties. We will use them all the time. Safety properties alone, however, are not enough. This is because they can only *prevent* things from happening. A trivial program, therefore, that does nothing, can vacuously satisfy a safety property.

Therefore, we need something more. This “something more” is progress.

3.5 Progress

Informally, a progress property says that “something good happens eventually”. It can be viewed as a predicate on possible computation suffixes. Unlike safety, it can *not* be violated by a finite execution. (Progress is also referred to as “liveness”).

Example 1. “Eventually, this pen will levitate”. We can watch the pen for some amount of time, say 1 minute. Is it possible to detect after that time whether or not the property has been violated? No. This is an example of a progress property.

Example 2. “When the switch is turned on, eventually the light comes on too.”

We saw that safety properties alone are not enough. We have now introduced a new class of properties, progress properties. A natural question now might be, are these two classes of program properties enough? It was shown by Alpern and Schneider (in *Information Processing Letters*, 1985) that all program properties of interest can indeed be expressed as a conjunction of safety and progress [AS85].

3.5.1 Transient

Our fundamental operator for progress properties is **transient**. Informally, **transient**. P means that if P becomes true at some point in the computation, it is guaranteed to become false at some later point. (This informal rule is not quite accurate, as we shall see shortly, but it serves to introduce the notion of transience.) Thus, if P is transient, it must be false infinitely often in a computation. (But not vice versa, again as will be seen later.)

Proof Rule. To prove

$$\mathbf{transient}.P.G$$

we must show

$$(\exists a : a \in G : \{P\} \quad a \quad \{\neg P\})$$

Notice that we must find *one* action in the program that is guaranteed to falsify P . Notice the difference between the proof rules for **transient** and **next**. The latter has universal quantification, while the former has only existential quantification.

Why is it sufficient to find a single action that falsifies P ? Our nondeterministic model of computation allows the selection of this action to be deferred for quite a while. Before it is selected, lots of things could happen in the computation, so how do we know that our transient behavior is guaranteed? Intuitively, at any point in the computation where P holds, we know that there is a future point in the computation when the one action guaranteed to falsify it will be selected (by fairness). Between these two points in the computation one of two things can happen: either some other action falsifies P (in which case the transience is satisfied) or no other action falsifies P in which case the one action guaranteed to falsify it will execute when P is true and will result in $\neg P$.

Pictorially, for **transient**. P to be a property of a program, there must be an action that maps out of P from all points inside it. Complete Figure 3.5 in a manner that expresses this intuition.

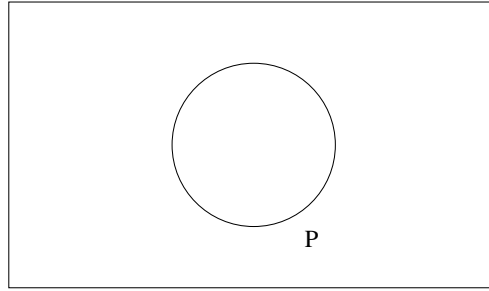


Figure 3.5: Graphical Representation of **transient**. P

As an exercise, which of the following theorems do you think apply to **transient**?

Strengthening.

$$\mathbf{transient}.P \wedge [P' \Rightarrow P] \Rightarrow \mathbf{transient}.P'$$

Weakening.

$$\mathbf{transient}.P \wedge [P \Rightarrow P'] \Rightarrow \mathbf{transient}.P'$$

As an example, consider the program with a single action, $even.x \longrightarrow x := x + 1$. Prove that **transient**. $(x = 2)$ is a property of this program.

$$\begin{aligned}
 & \text{Proof.} \quad \mathbf{transient}.(x = 2) \\
 & \equiv \quad \{ \text{definition of } \mathbf{transient} \} \\
 & \quad \{x = 2\} \quad even.x \longrightarrow x := x + 1 \quad \{x \neq 2\} \\
 & \equiv \quad \{ \text{assignment axiom} \} \\
 & \quad (x = 2 \wedge even.x \Rightarrow (x + 1) \neq 2) \wedge (x = 2 \wedge \neg even.x \Rightarrow x \neq 2) \\
 & \equiv
 \end{aligned}$$

$$\begin{aligned}
& (x = 2 \wedge \text{even}.x \Rightarrow x \neq 1) \wedge (x = 2 \wedge \neg \text{even}.x \Rightarrow x \neq 2) \\
\equiv & \\
& x = 2 \wedge \neg \text{even}.x \Rightarrow x \neq 2 \\
\equiv & \\
& \mathbf{false} \Rightarrow x \neq 2 \\
\equiv & \{ \text{property of } \Rightarrow \} \\
& \mathbf{true}
\end{aligned}$$

□

As another example, consider the program with a single action, $n \leq 2 \longrightarrow n := n + 1$. Is the following a property of the program: **transient**.($n = 0 \vee n = 1$) .

$$\begin{aligned}
& \mathbf{transient}.(n = 0 \vee n = 1) \\
\equiv & \{ \text{definition of } \mathbf{transient} \} \\
& \{n = 0 \vee n = 1\} \quad n \leq 2 \longrightarrow n := n + 1 \quad \{\neg(n = 0 \vee n = 1)\} \\
\equiv & \{ \text{distribution of } \neg \text{ over } \vee \} \\
& \{n = 0 \vee n = 1\} \quad n \leq 2 \longrightarrow n := n + 1 \quad \{n \neq 0 \wedge n \neq 1\} \\
\equiv & \{ \text{assignment axiom} \} \\
\equiv &
\end{aligned}$$

Can you reduce this to **true**?

3.5.2 Ensures

Informally, P **ensures** Q means that if P holds, it will continue to hold so long as Q does not hold, and eventually Q does hold. This last requirement means there is (at least) one action that establishes Q starting from any P state.

Again, complete Figure 3.6 to capture the intuition behind this operator.

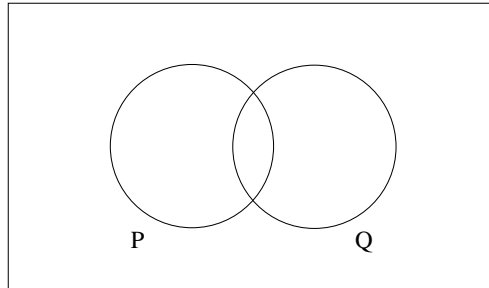


Figure 3.6: Graphical Representation of P **ensures** Q

Proof Rule. Formally **ensures** is defined by

$$P \text{ ensures } Q \equiv ((P \wedge \neg Q) \text{ next } (P \vee Q)) \wedge \text{transient}.(P \wedge \neg Q)$$

The **ensures** operator is slightly more general than **transient**. For example, consider the program with a single action, $\text{even}.x \longrightarrow x := x + 1$. Prove that the following is a property of the program:

$$(x = 2 \vee x = 6) \text{ ensures } (x = 3 \vee x = 7)$$

Proof. There are two proof obligations.

1. **next** property

$$\begin{aligned} & (x = 2 \vee x = 6) \wedge x \neq 3 \wedge x \neq 7 \text{ next } x \in \{2, 3, 6, 7\} \\ \equiv & \\ & x = 2 \vee x = 6 \text{ next } x \in \{2, 3, 6, 7\} \\ \equiv & \quad \{ \text{definition of next and Assignment Axiom} \} \\ & x = 2 \vee x = 6 \Rightarrow x + 1 \in \{2, 3, 6, 7\} \\ \equiv & \\ & \text{true} \end{aligned}$$

2. **transient** property

$$\begin{aligned} & \text{transient}.(x = 2 \vee x = 6) \\ \equiv & \quad \{ \text{definition of transient} \} \\ & \{x = 2 \vee x = 6\} \text{ even}.x \longrightarrow x := x + 1 \quad \{x \neq 2 \wedge x \neq 6\} \end{aligned}$$

(This calculation is left as an exercise.)

□

Although **ensures** is slightly higher level than **transient**, it is still quite low-level. For example, consider the program with a single action, $n \leq 2 \longrightarrow n := n + 1$. Does this program satisfy the property:

$$n = 1 \text{ ensures } n = 3$$

The answer is no. Make sure you understand why not.

3.5.3 Leads-to

Leads-to is perhaps the most commonly used operator in expressing progress properties. Informally, $P \rightsquigarrow Q$ means that if P is true at some point, Q will be true (at that same or a later point) in the computation. This is quite different than **ensures**, which restricts what the computation can do before establishing Q . With leads-to, the computation can meander in all sorts of directions, so long as it ends up in Q . Also, notice that if *both* P and Q are true at a point in the computation, the leads-to property has been satisfied.

Again, use Figure 3.7 to draw an intuitive representation of the important aspects of leads-to.

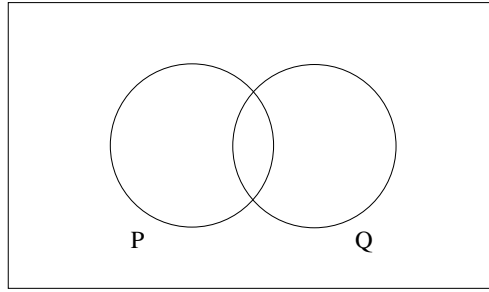


Figure 3.7: Graphical Representation of $P \rightsquigarrow Q$

As an exercise, examine the following theorems about leads-to and determine which are valid.

Constants.

$$\begin{aligned} P &\rightsquigarrow \text{true} \\ \text{false} &\rightsquigarrow P \\ P &\rightsquigarrow P \end{aligned}$$

Weakening the RHS.

$$(P \rightsquigarrow Q) \wedge [Q \Rightarrow Q'] \Rightarrow P \rightsquigarrow Q'$$

Strengthening the LHS.

$$(P \rightsquigarrow Q) \wedge [P' \Rightarrow P] \Rightarrow P' \rightsquigarrow Q$$

If those were too easy for you, think about the following ones...

Stable Strengthening.

$$\text{stable}.P \wedge \text{transient}.(P \wedge \neg Q) \Rightarrow P \rightsquigarrow (P \wedge Q)$$

Progress-Safety-Progress (PSP).

$$(P \leadsto Q) \wedge (R \text{ next } S) \Rightarrow (P \wedge R) \leadsto ((R \wedge Q) \vee (\neg R \wedge S))$$

Conjunctivity.

$$(P \leadsto Q) \wedge (P' \leadsto Q') \Rightarrow (P \wedge P') \leadsto (Q \wedge Q')$$

Notice the difference between leads-to and implication with respect to conjunctivity.

Formally, the definition of leads-to is given by:

$$\begin{aligned} P \text{ ensures } Q &\Rightarrow P \leadsto Q \\ (P \leadsto Q) \wedge (Q \leadsto R) &\Rightarrow P \leadsto R \\ (\forall i :: P_i \leadsto Q) &\Rightarrow (\exists i :: P_i) \leadsto Q \end{aligned}$$

It is critical to understand the difference between the following two properties:

$$\begin{aligned} &\mathbf{transient}.P \\ &P \leadsto \neg P \end{aligned}$$

It is possible to write a program that satisfies latter property but does not satisfy the former! You should find such an example to be sure you understand this distinction. This distinction means that the two expressions are related by _____ (equivalence or implication). In particular, the relationship is:

$$\mathbf{transient}.P \text{ --- } P \leadsto \neg P$$

As an example, consider the program with a single action, $n \leq 2 \longrightarrow n := n + 1$. Prove that this program satisfies the property $n = 1 \leadsto n = 3$.

Proof. We carry out this proof in two stages. In the first stage, we show $n = k \wedge n \leq 2 \text{ ensures } n = k + 1$. To complete this stage, there are two proof obligations.

1. $(n = k \wedge n \leq 2 \wedge n \neq k + 1) \text{ next } ((n = k \wedge n \leq 2) \vee n = k + 1)$

$$\begin{aligned} &\{n = k \wedge n \leq 2\} \quad n \leq 2 \longrightarrow n := n + 1 \quad \{(n = k \wedge n \leq 2) \vee n = k + 1\} \\ &\equiv \\ &n = k \wedge n \leq 2 \Rightarrow n + 1 = k + 1 \\ &\equiv \\ &\mathbf{true} \end{aligned}$$

2. $\mathbf{transient}.(n = k \wedge n \leq 2)$

$$\begin{aligned}
& \{n = k \wedge n \leq 2\} \quad n \leq 2 \longrightarrow n := n + 1 \quad \{n \neq k \vee n > 2\} \\
\equiv & \\
& n = k \wedge n \leq 2 \Rightarrow n + 1 \neq k \vee n + 1 > 2 \\
\Leftarrow & \\
& n = k \Rightarrow n + 1 \neq k \\
\equiv & \\
& \mathbf{true}
\end{aligned}$$

We now proceed with the second stage of the proof, in which we use the result established above to calculate:

$$\begin{aligned}
& n = k \wedge n \leq 2 \textbf{ ensures } n = k + 1 \\
\Rightarrow & \quad \{ \text{definition of } \rightsquigarrow \} \\
& n = k \wedge n \leq 2 \rightsquigarrow n = k + 1 \\
\Rightarrow & \quad \{ \text{one-point rule, for } n = 1 \text{ and } n = 2 \} \\
& (n = 1 \rightsquigarrow n = 2) \wedge (n = 2 \rightsquigarrow n = 3) \\
\Rightarrow & \quad \{ \text{transitivity of } \rightsquigarrow \} \\
& n = 1 \rightsquigarrow n = 3
\end{aligned}$$

□

3.5.4 Induction

In practice, proving a leads-to property often involves an inductive argument. That is, it may not be easy to prove directly that $P \rightsquigarrow Q$, so instead we show that $P \rightsquigarrow P'$, where P' is “closer” to the goal, Q . The inductive theorem for leads-to makes this intuition precise. The inductive theorem is based on a concept that you have seen before in sequential programming: metrics.

Definition 1 (Metric). A metric (or “variant function”) is a function from the state space to a well-founded set (e.g., the natural numbers).

The well-foundedness of the range means that the value of the function is bounded below (i.e., can only decrease a finite number of times).

Theorem 10 (Induction for \rightsquigarrow). For a metric M ,

$$(\forall m :: P \wedge M = m \rightsquigarrow (P \wedge M < m) \vee Q) \Rightarrow P \rightsquigarrow Q$$

The metric can be seen as a quantification of how close the program is to the goal, “Q”. A better way, perhaps, to think of a metric is as a series of gates through which the computation must pass. Once passing through the gate labelled m , it must eventually pass through a gate with a label smaller than m , or establish Q . Note that this rule does allow the metric to go up as well as down. The leads-to property simply says that *eventually* the metric has a value less than m .

A slightly restricted form of a metric is one that does not increase without establishing Q . In this case, the gates are one-way. Although this form is less general, its use is common enough to warrant listing it explicitly.

Theorem 11 (Restricted Form of Induction for \leadsto). *For a metric M*

$$\begin{aligned} & (\forall m :: P \wedge M = m \text{ next } (P \wedge M \leq m) \vee Q) \\ & \wedge (\forall m :: \text{transient}.(P \wedge M = m)) \\ \Rightarrow & P \leadsto Q \end{aligned}$$

As an exercise, prove that this restricted induction rule is a corollary of the previous one.

The informal argument based on the application of this induction rule is frequently divided into three separate proof obligations:

- The type of the metric is a well-founded set. (It is usually sufficient to show that the set is bounded below.)
- The value of the metric can not increase.
- The value of the metric must change eventually.

(Note: sometimes metrics are used in the other “direction”. That is, a metric is found that can not *decrease* and is bounded *above*.)

An alternate formulation of the induction rule is to show that *every enabled action* decreases the metric. This formulation is frequently used to show termination as it guarantees, quite directly, that the program reaches a fixed point. This alternate formulation is given as the next theorem.

Theorem 12 (Induction for \leadsto). *For a metric M ,*

$$\begin{aligned} & (\forall i, m :: \{P \wedge M = m \wedge g_i\} \quad g_i \longrightarrow a_i \quad \{(P \wedge M < m) \vee Q\}) \\ & \wedge (\forall i :: \neg g_i) \Rightarrow Q \\ \Rightarrow & P \leadsto Q \end{aligned}$$

As an exercise, prove that this restricted induction rule is a corollary of the original induction theorem.

An important (and common) example of the application of the inductive theorem for leads-to is in establishing the termination of programs. In that case, P is some invariant property of the program (for example, you might be able to use **true**), and Q is FP . In this situation, the theorem can be stated more simply, since $(\forall i :: \neg g_i) \Rightarrow FP$ holds:

$$\begin{aligned} & (\forall i, m :: \{P \wedge M = m \wedge g_i\} \quad g_i \longrightarrow a_i \quad \{(P \wedge M < m) \vee FP\}) \\ \Rightarrow & P \leadsto FP \end{aligned}$$

As an illustration of how induction is used in conjunction with leads-to, as well as of the other theorems and properties in this chapter, the next chapter presents a collection of small examples.

Chapter 4

Small Example Programs

In this chapter, we present a collection of small examples. Each example is introduced informally, then given a formal specification. The program is then given, followed by a proof of correctness. The proofs of correctness will be given with less formality than the treatment of previous chapters.

4.1 Proof Structure

All these examples have specifications and proofs that follow a similar structure. The first step of the proof involves calculating the fixed point and an invariant for the program. Then, if the program terminates (ie reaches the fixed point), we know that the conjunction of these two things is true. The second step of the proof is to establish that the program does indeed terminate. This step involves finding a metric. Recall that a metric must be nonincreasing and bounded below (or nondecreasing and bounded above). It must also be guaranteed to change eventually.

4.2 FindMax

We have already seen the FindMax example in earlier chapters. We now present a formal argument that our intuition about this program is right: the program is “correct”.

Recall the FindMax program calculates the maximum element in an array of integers, assigning this maximum value to a variable r . To simplify the exposition, we define a constant M :

$$M = (\mathbf{Max} \, x : 0 \leq x \leq N - 1 : A[x])$$

As is typical in these programs, the specification is given by a safety and a progress property.

$$\mathbf{true} \leadsto r = M$$

stable. $(r = M)$

The program, which has been given previously, is repeated here.

```

Program      FindMax
var           A : array 0..N - 1 of int,
                r : int
initially     r = A[0]
assign
  (  $\parallel x : 0 \leq x \leq N - 1 : r := \max(r, A[x])$  )

```

This program satisfies the specification.

Proof. **Fixed Point.** We have already calculated the fixed point of this program as being:

$$\begin{aligned}
 FP &\equiv (\forall x : 0 \leq x \leq N - 1 : r = \mathbf{max}(r, A[x])) \\
 &\equiv r \geq (\mathbf{Max} x : 0 \leq x \leq N - 1 : A[x]) \\
 &\equiv r \geq M
 \end{aligned}$$

Invariant. We also showed earlier that the following is a property of the program:

invariant. $(r \leq M)$

Therefore, at termination, we have the conjunction of FP and the invariant, that is: $r \geq M \wedge r \leq M$, or $r = M$.

Metric. It remains to be shown, however, that the program does indeed terminate. For this we must find a metric. A good choice for a metric is r . This is indeed a metric for the following reasons:

1. It is guaranteed not to decrease (*i.e.*, the **next** property for metrics). That is, we prove $r = k \mathbf{next} r \geq k$.

$$\begin{aligned}
 &\{r = k\} \quad r := \max(r, A[x]) \quad \{r \geq k\} \\
 \equiv &\quad \{ \text{assignment axiom} \} \\
 &r = k \Rightarrow \mathbf{max}(r, A[x]) \geq k \\
 \equiv &\quad \{ \text{property of } \mathbf{max} \} \\
 &r = k \Rightarrow r \geq k \vee A[x] \geq k \\
 \equiv & \\
 &\mathbf{true}
 \end{aligned}$$

2. It is bounded above. This follows directly from the invariant property that shows that it is bounded above by M .
3. If it is below M , there is an action guaranteed to increase it. In other words, **transient.** $(r = k \wedge r < M)$. To see this, let m be the index in the array such that $A[m] = M$.

$$\begin{aligned}
& \mathbf{transient}.(r = k \wedge r < M) \\
\equiv & \quad \{ \text{definition of } \mathbf{transient} \} \\
& (\exists a :: \{r = k \wedge r < M\} \quad a \quad \{r \neq k \vee r \geq M\}) \\
\Leftarrow & \quad \{ \text{definition of program} \} \\
& \{r = k \wedge r < M\} \quad r := \max(r, A[m]) \quad \{r \neq k \vee r \geq M\} \\
\equiv & \quad \{ \text{assignment axiom} \} \\
& r = k \wedge r < M \Rightarrow \max(r, A[m]) \neq k \vee \max(r, A[m]) \geq M \\
\Leftarrow & \quad \{ \text{weakening antecedent} \} \\
& r = k \wedge r < M \Rightarrow \max(r, A[m]) \neq k \\
\equiv & \quad \{ \text{definition of } m \} \\
& r = k \wedge r < M \Rightarrow \max(r, M) \neq k \\
\Leftarrow & \\
& M > k \Rightarrow \max(r, M) \neq k \\
\Leftarrow & \quad \{ \text{weakening antecedent} \} \\
& M > k \Rightarrow \max(r, M) > k \\
\equiv & \quad \{ \text{property of } \mathbf{max} \} \\
& \mathbf{true}
\end{aligned}$$

With this metric, we can apply induction to establish that the program terminates. That is:

$$\begin{aligned}
& \mathbf{true} \\
\equiv & \quad \{ \mathbf{transient} \text{property established above} \} \\
& \mathbf{transient}.(r = k \wedge r < M) \\
\Rightarrow & \quad \{ \mathbf{transient}.P \Rightarrow (P \rightsquigarrow \neg P) \} \\
& r = k \wedge r < M \rightsquigarrow r \neq k \vee r \geq M \\
\Rightarrow & \quad \{ \mathbf{stable}.(r \geq k) \} \\
& r = k \wedge r < M \rightsquigarrow r > k \vee r \geq M \\
\equiv & \quad \{ [X \vee Y \equiv (\neg Y \wedge X) \vee Y] \} \\
& r < M \wedge r = k \rightsquigarrow (r < M \wedge r > k) \vee r \geq M \\
\Rightarrow & \quad \{ \text{induction} \} \\
& r < M \rightsquigarrow r \geq M \\
\equiv & \quad \{ \text{definition of } FP \} \\
& r < M \rightsquigarrow FP \\
\equiv & \quad \{ \mathbf{initially}.(r < M) \} \\
& \mathbf{true} \rightsquigarrow FP
\end{aligned}$$

□

In the remaining examples, we will be less thorough in the development of the proof. We will focus on the three main stages: calculating the fixed point, finding a useful invariant, and finding an appropriate metric.

4.3 Sorting

In this example, we consider sorting an array of integers. The specification for this program says that at termination, the array is in nondescending order and is

a permutation of the original array. It also requires that the program eventually terminate.

Consider the following program:

```

Program      Sort
var          A : array 0..N - 1 of int
assign
  (  $\parallel i, j : 0 \leq i < j < N : A[i] > A[j] \longrightarrow \text{swap}(A, i, j)$  )

```

As an exercise, it is worthwhile to try and convince yourself that this program is indeed correct, before reading on to the proof.

Proof. Fixed Point. The fixed point of this program is given by:

$$\begin{aligned}
 & (\forall i, j : 0 \leq i < j < N : A[i] > A[j] \Rightarrow A[i] = A[j]) \\
 \equiv & \{ \text{property of } \Rightarrow : [X \Rightarrow Y \equiv \neg X \vee Y] \} \\
 & (\forall i, j : 0 \leq i < j < N : A[i] \leq A[j] \vee A[i] = A[j]) \\
 \equiv & \{ \text{property of } \Rightarrow : [X \vee Y \wedge (Y \Rightarrow X) \Rightarrow X] \} \\
 & (\forall i, j : 0 \leq i < j < N : A[i] \leq A[j])
 \end{aligned}$$

That is, the array in nondescending order.

Invariant. An invariant of the program is that the array is a permutation of the original array. This property is true initially trivially. Since each action can only swap two elements of the array, each action preserves this property.

Therefore, at termination, we have that the array is in nondescending order and is a permutation of the original array. That is, if the program terminates, it has sorted the array.

Metric. As a metric for this program, we use the number of out-of-order pairs. That is,

$$\left(\sum i, j : 0 \leq i < j < N \wedge A[i] > A[j] : 1 \right)$$

Clearly this metric is bounded below (by 0). It is not clear, however, that the metric can only decrease. To see this, consider the effect of a single swap that swaps elements X and Y . These two elements divide the array into thirds:

$$\underbrace{a \ a \ a \ \dots \ a}_X \ \underbrace{b \ b \ \dots \ b}_Y \ \underbrace{c \ c \ \dots \ c}$$

Since the action swaps elements X and Y , it must be the case that $X > Y$. Therefore, after the swap, this pair is no longer out-of-order. But is it possible for the swap to cause *other* pairs, that used to be in order to now be out of order?

Consider the different cases for other pairs that are in order before the swap. We show that they are still in order after the swap.

1. Neither element of the pair is X or Y . The swap does not affect this pair.

2. One element of the pair is in part a of the array. Again, the swap does not affect the number of out-of-order pairs for this element (since both swapped elements are to its right.)
3. One element of the pair is in part c of the array. The swap does not affect the number of out of order pairs for this element for the same reason as the previous case.
4. One element of the pair is in part b . If it was in-order with respect to X before the swap (*i.e.*, $c > X$), it will be in-order with respect to Y after the swap, since $X > Y$. Similarly, if it was in-order with respect to Y before the swap, it will be in-order with respect to X after the swap.

Therefore the number of out-of-order pairs can never increase.

It is also easy to see that any action that performs a swap decreases the metric by at least one (the pair that was swapped is now in-order). Therefore, every enabled action decreases the metric. Therefore, the program terminates. \square

4.4 Earliest Meeting Time

The Earliest Meeting Time problem consists of finding the first time at which three professors are all simultaneously available. The three professors are named F , G , and H . With each professor, we associate a function (f , g , and h respectively). These functions map time to time and represent the earliest time that professor is available. For example, $f.t$ is the earliest time at or after time t at which professor F is available. Therefore, $f.t = t$ corresponds to F being available at time t .

We are given that there is a time at which all three are simultaneously available. We define M to be the earliest such time.

$$M = (\mathbf{Min} \, t : f.t = g.t = h.t : t)$$

We are to calculate this minimum time.

$$\begin{aligned} \mathbf{true} &\leadsto r = M \\ \mathbf{stable}.(r = M) \end{aligned}$$

```

Program      EMT
var           $r : \text{time}$ 
initially     $r = 0$ 
assign
     $r := f.r$ 
     $\parallel$   $r := g.r$ 
     $\parallel$   $r := h.r$ 

```

This program satisfies the specification.

Proof. Fixed Point.

$$FP \equiv r = f.r = g.r = h.r$$

Therefore, $r \geq M$.

Invariant.

$$\text{invariant.}(r \leq M)$$

Therefore, at termination, we have $r = M$.

Metric. As a metric, we use r . It is guaranteed not to decrease, since $f.t \geq t$, by definition of f . It is also bounded above by M (as given by the invariant). To see that it is guaranteed to change if it is below M , consider any value for r , where $r < M$. For this value, there is a professor (say F) who is not available (otherwise all professors would be available and this would be the minimum time!) Therefore $f.r > r$ so the action $r := f.r$ increases the metric. Therefore, the program terminates. \square

4.5 Greatest Common Divisor

Consider a program to calculate the greatest common divisor (GCD) of two integers, X and Y . The specification for this program is given by:

$$\begin{aligned} \text{true} &\leadsto x = y = \text{gcd}(X, Y) \\ \text{stable.}(x = y = \text{gcd}(X, Y)) \end{aligned}$$

Program *GCD*
var $x, y : \text{int}$
initially $x > 0 \wedge y > 0 \wedge x = X \wedge y = Y$
assign
 $x > y \longrightarrow x := x - y$
 $\parallel y > x \longrightarrow y := y - x$

This program satisfies the specification.

Proof. Fixed Point.

$$\begin{aligned} &FP \\ \equiv & \\ &(x > y \Rightarrow x = x - y) \wedge (y > x \Rightarrow y = y - x) \\ \equiv & \\ &(x > y \Rightarrow y = 0) \wedge (y > x \Rightarrow x = 0) \end{aligned}$$

Invariant.

$$\text{invariant.}(x > 0 \wedge y > 0 \wedge \text{gcd}(x, y) = \text{gcd}(X, Y))$$

Therefore, at termination, we have FP and the invariant, which we can calculate to be:

$$x = y = \gcd(X, Y)$$

Metric. As a metric, we choose $x + y$. This value is bounded below, since $x > 0$ and $y > 0$ by the invariant. Also, it never increases since an action only subtracts a positive value from either x or y . Finally, if x and y differ (*i.e.*, the program has not terminated), the enabled action (there is only one) decreases their sum.

Therefore, this program terminates.

□

Chapter 5

Time, Clocks, and Synchronization

5.1 References

1. Logical time (and the “happens-before” relation) is defined in Lamport’s article “Time, Clocks, and the Ordering of Events in Distributed Systems”, CACM 21(7) p.558–565, 1978 [Lam78]. This article also discusses the synchronization of physical clocks and presents an algorithm for accomplishing this synchronization and the bound on the resulting error, assuming message delay is bounded.
2. Vector clocks were developed independently by Fidge [Fid88] and Mattern [Mat89]. The original references are (1) “Timestamps in Message-Passing Systems that Preserve the Partial Ordering”, by C. J. Fidge in Proceedings of the 11th Australian Computer Science Conference, 10(1), Feb 1988, p. 56–66, and (2) “Virtual Time and Global States of Distributed Systems”, by F. Mattern in Parallel and Distributed Algorithms, Cosnard et al. editors, Elsevier Science, North-Holland, 1989, p.215–226.
3. The Singhal and Shivaratri book contains a good discussion of both logical time and vector clocks [SS94].

5.2 Introduction

The concept of time is pervasive. When we think of an event, we typically think of it having occurred at a particular time. The granularity with which we view time might vary depending on the event. For example, Christofero Colombo sailed across the Atlantic in 1492. Paul Henderson scored the winning goal for Canada in the Canada Cup hockey tournament on September 28, 1972. The

midterm was written last Friday at 9:30am. Regardless, whether explicit or implicit, we associate specific events with the time at which they occurred.

The concept of time is pervasive because of the fundamental utility in such an implicit ordering. Consider the following examples:

- A bell is used to signal the end of class. We know we will have enough time to get to the next class because it won't begin for another 12 minutes.
- "Meet me at 7pm in the SEL". Having just the *location* of the meeting is not enough information.
- In a criminal court case, the witness is asked: "Where were you at 10:30pm on the night of April 17th?"

Notice, however, that this use of time only makes sense if both parties have the *same* notion of time.

On the surface, time is simply a monotonic "counter". To a large extent, we make use of this notion of time by assuming that all parties have access to the same counter. In a distributed system, however, there is no shared state. In particular, then, there is no shared counter (or any other kind of global clock). So we can not implicitly associate an event with its time, because one process may have a different clock than another.

Rather than trying to implement some kind of shared counter, we begin this chapter by stepping back and examining what is really important about time. What we frequently care about is the *ordering* of events. That is, we want to know when one event happened before, or after, or concurrently with another. The formalization of this relation is the basis for logical time.

5.3 Logical Time

5.3.1 Happens Before

When does event A "happen before" event B ? The easy answer, of course, is "when the time of A is earlier than the time of B "! But this is circular, since it is precisely the notion of time that we are trying to define.

Instead, we turn the question around and ask: why would anyone possibly care? It may seem we are dodging the question, but the reason we turn the question around in this way is to understand what is fundamental about "happening before". For example, your professor might ask: "Did you submit your homework before you received the solution set?" Why would the professor care? Because _____.

This suggests a definition for "happens before": Event A happens before event B if and only if _____.

In a distributed system, there are 3 ways in which event A can affect another event B :

1. A and B are on the same processors and A is earlier in computation than B .

2. A and B are on different processors and A is the send of a message and B is the receive event for that same message.
3. There is some third event C , which A can affect, which in turn can affect B .

For notation, we write $A \longrightarrow B$ to mean “event A happens before event B ”.

Concurrency. This definition of “happens before” suggests a definition for concurrency. In particular, events A and B are concurrent (written $A \parallel B$) exactly when A can *not* affect B and vice versa. That is:

$$A \parallel B = \text{_____} \wedge \text{_____}$$

Is concurrency transitive? That is, does the following property hold?

$$(A \parallel B) \wedge (B \parallel C) \Rightarrow (A \parallel C)$$

5.3.2 Timelines

During a computation, events occur on different processes at various times. One way to visualize a computation is to draw what is sometimes known as a “time-line” or “space-time” diagram. In such a diagram, the events are represented by vertices and the “happens-before” relation is included as directed edges between the vertices. Furthermore, the diagram is structured so that all events occurring on process P_i are arranged in a horizontal line, with the computation proceeding towards the right.

For example, Figure 5.1 shows a possible timeline for a computation with four processes. Notice how the labels for the individual events are omitted and the happens-before relation on the same process is understood. The edges *between* horizontal lines represent _____.

Notice that in this diagram, all the arrows point towards the right. That is, a horizontal axis of “true time” is used and events are placed according to when they actually occur. An important consequence of this observation is that there are no cycles in the directed graph.

More generally, if “true time” is not used to place the events (since there is no way to calculate what the “true time” of an event is anyways!), a directed graph results. Such a graph still has horizontal lines of events that occur on the same process, but now arrows between horizontal lines can point to the left. For example, the timeline given in Figure 5.2 is equivalent to the one given previously in Figure 5.1. The *key property* of timelines, therefore, is that they do not contain _____. Indeed, the happens-before relation forms a *partial order* on the events of the computation.

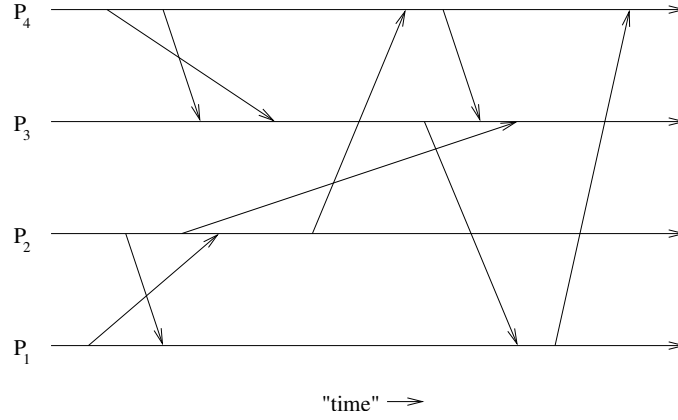


Figure 5.1: Timeline for a computation with four processes.

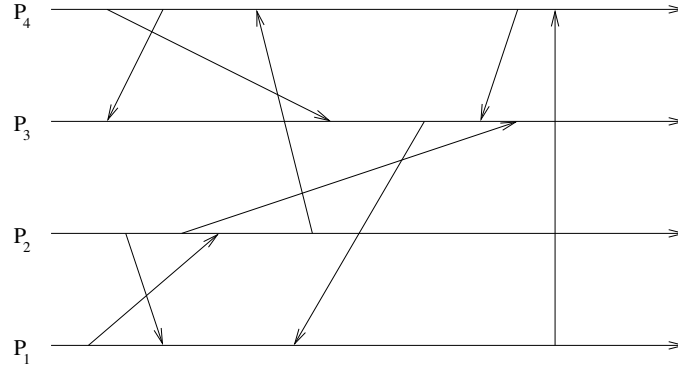


Figure 5.2: Equivalent timeline for a computation with four processes.

5.3.3 Logical Clock

The abstraction of a monotonic counter is used for a clock. The value of this counter can be used to assign a value (a “timestamp”) to an event. We will write the timestamp of an event A as $time.A$. Of course, there is no global logical clock: Each process must maintain its own.

In order for these clocks to be useful, we want to be sure that they order events in a manner that is consistent with the happens-before relation defined above. In particular, we would like:

$$A \longrightarrow B \Rightarrow time.A < time.B$$

As an aside, note that the converse might not hold.

5.3.4 Algorithm

The desired property suggests the algorithm. It is an invariant we wish to maintain as new events are generated. That is, for each new event that occurs, we must assign it a timestamp such that its timestamp is guaranteed to be greater than the timestamps of any events that “happened-before”. There are 3 cases to consider for possible events:

1. **Local event.** The clock is incremented and this updated value is the timestamp of the new event.

$$\begin{aligned} & \text{clock} := \text{clock} + 1 \\ & ; \text{time.A} := \text{clock} \end{aligned}$$

2. **Send event.** The clock is incremented and this updated value is the timestamp of the new event and of the message being sent.

$$\begin{aligned} & \text{clock} := \text{clock} + 1 \\ & ; \text{time.A}, \text{time.m} := \text{clock}, \text{clock} \end{aligned}$$

3. **Receive event.** The clock is updated by taking the maximum of the current clock (time of last event on this process) and the timestamp of the message (time of the corresponding send event). This maximum must be incremented to ensure that the new clock value is strictly larger than *both* previous events (which “happened-before” the current event). Again, the updated value is the timestamp of the new event.

$$\begin{aligned} & \text{clock} := \mathbf{max}(\text{time.m}, \text{clock}) + 1 \\ & ; \text{time.A} := \text{clock} \end{aligned}$$

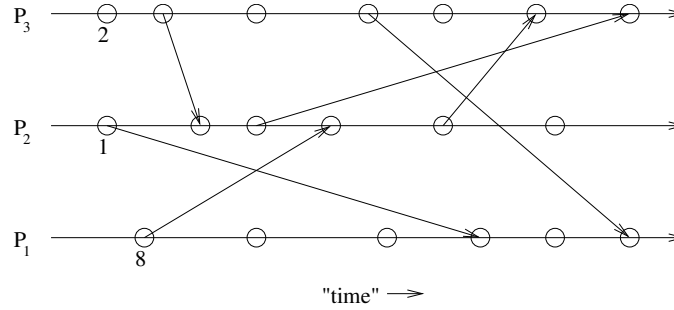
Notice that the invariant maintained with respect to *clock* is that it is equal to the most recently assigned timestamp.

(*Aside:* There are many equivalent ways of writing these assignments. The sequential composition of assignments above with *;* is not strictly necessary. It is possible to rewrite each of these sequences of assignments as a single multiple assignment. Such a rewrite is not simply a matter of replacing the *;* with a *||*, but it is possible. This presentation was chosen for clarity and convenience.)

As an exercise, consider the timeline in Figure 5.3. Assign logical timestamps to the events (indicated by circles) in accordance with the algorithm above. Initially, the clocks begin at the values indicated.

The algorithm can be written as a program in our action system model. For a single process (*j*) the algorithm is:

Program	<i>LogicalClock j</i>
var	<i>j, k</i> : processes,
	<i>ch.j.k</i> : channel from <i>j</i> to <i>k</i> ,


$$\begin{array}{ll}
m : \text{message}, & \\
A, B : \text{events}, & \\
\text{clock}.j : \text{logical time of } j, & \\
\text{time}.A : \text{logical time of } A, & \\
\text{initially} & \text{clock}.j = 0 \\
\text{assign} & \\
\quad \text{local event } A \longrightarrow & \text{clock}.j := \text{clock}.j + 1 \\
& ; \quad \text{time}.A := \text{clock}.j \\
\parallel \text{ send event } A \longrightarrow & \text{clock}.j := \text{clock}.j + 1 \\
\quad (\text{to } k) & ; \quad \text{time}.A, \text{time}.m := \text{clock}.j, \text{clock}.j \\
& ; \quad \text{ch}.j.k := \text{ch}.j.k \mid m \\
\parallel \text{ rcv event } A \longrightarrow & \text{clock}.j := \max(\text{time}.m, \text{clock}.j) + 1 \\
\quad (m \text{ from } k) & ; \quad \text{time}.A, \text{ch}.j.k := \text{clock}.j, \text{tail}(\text{ch}.j.k)
\end{array}$$

The following property can be established as an invariant of this program:

$$\begin{aligned} & (\forall A, j : A \text{ occurs at } j : \text{time}.A \leq \text{clock}.j) \\ \wedge & (\forall m, j, k : m \in \text{ch}.j.k : (\exists A : A \text{ occurs at } j : \text{time}.A = \text{time}.m)) \\ \wedge & (\forall A, B :: A \longrightarrow B \Rightarrow \text{time}.A < \text{time}.B) \end{aligned}$$

5.3.5 Total Ordering

One property of any partial order is that there exists at least one way to *totally* order the elements so that the partial order is satisfied. Indeed, there may exist several such orderings. That is, we can write an ordered sequence of all the elements in the partial order:

$$x_1, x_2, x_3, \dots$$

such that $x_i \leq x_j \Rightarrow x_i$ is listed before x_j .

In particular, for events and the “happens-before” relation, we can write the list of events as

$$e_1, e_2, e_3, \dots, e_i, \dots$$

such that for all i, j , if e_i “happens-before” e_j , e_i comes before e_j in the list.

The timeline diagrams represent the partial order of events and the assignment of logical times to events gives us one way to globally order them. We can write a totally ordered sequence of events using logical time by writing the events in _____.

5.4 Vector Clocks

5.4.1 Motivation

With logical time, we made the observation that the fundamental property we cared about was that future events can not affect events in the past. Put another way, events with larger timestamps can not “happen-before” events with smaller timestamps. More formally:

$$A \longrightarrow B \Rightarrow \text{time}.A < \text{time}.B$$

We observed that with the timestamping of Lamport’s logical clocks, the relation above is really implication and not equivalence (*i.e.*, the converse does not hold). This means we can not determine whether one event “happened-before” another event simply by examining their logical timestamps.

For example, consider the incomplete timeline given in Figure 5.4. The

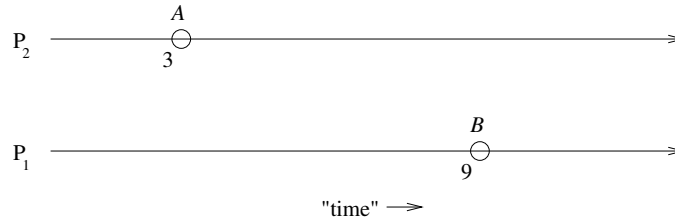


Figure 5.4: A partial timeline with two events: A and B .

timestamp of event A , 3 , is less than the timestamp of event B , 9 . But these two events are on different processes. How are these two events related? We know that $\neg(B \longrightarrow A)$ (by the contrapositive of the property above). But do we know that $A \longrightarrow B$? The answer is no. Without further information (*i.e.*, the messages sent and received), we don’t know whether this relation holds. (Notice that even being told that process P_1 does not send any messages to P_2 is not enough, since \longrightarrow is transitive and so there could exist a chain of \longrightarrow dependencies through other processes.)

The source of the problem is that a single integer does not capture enough information. We do not know whether event B received its timestamp because of a chain of local events or because of a chain of messages after event A . In general, an event can receive a timestamp n either because the previous local event had timestamp $n - 1$ or because it is a receive event for a message with timestamp $n - 1$. (Of course both reasons could be true). Just the value of the integer alone, however, does not capture which “happens-before” relation forced the assignment of this timestamp to the event.

The solution, then, is to not throw this information away. This leads to the development of a new timestamping scheme, one known as vector clocks. To distinguish vector and logical clocks, we will use *vtime* and *vclock* for the vector quantities. The benefit of the vector clock scheme is that it has the property:

$$A \longrightarrow B \equiv vtime.A < vtime.B$$

5.4.2 Central Idea

The key idea behind vector clocks is that instead of each process keeping a single counter for its clock (and for timestamping events on that process), each process keeps a *list* of counters, one for each process in the computation. For example, in Figure 5.5, event A receives a timestamp consisting of a vector of four clocks. (There are therefore a total of four processes in the computation.)

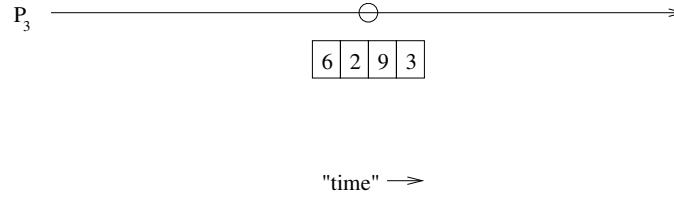


Figure 5.5: Timestamping an event with a vector clock.

The algorithm for timestamping events and updating the local clock remains, in essence, the same:

- A clock is maintained by each process, representing the timestamp last assigned to an event occurring at that process.
- Each message sent is timestamped with that process' current time (*i.e.*, a vector of integers).
- When a local event occurs, the clock must be incremented to assign a time to this local event. (Question: How?)
- When a receive event occurs, it must be timestamped appropriately (*i.e.*, with a value that is greater than the last local event and greater than the corresponding send event). (Question: How?)

The only issue to be resolved is how to perform the clock updates. Consider the update that must occur when a receive occurs. In Figure 5.6 event C is a receive event and event A is the corresponding send.

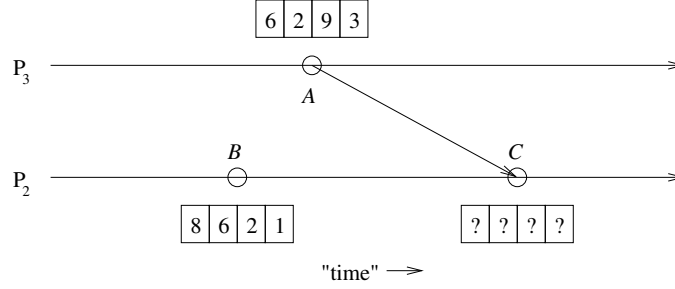


Figure 5.6: Timestamping a receive event with a vector clock.

The question is: what values do we assign to the components of $time.C$? In order to maintain the property we had with logical clocks (*i.e.*, $A \rightarrow B \Rightarrow vtime.A < vtime.B$), we must assign $vtime.C$ in such a way that we have both:

$$\begin{aligned} vtime.A &< vtime.C \\ vtime.B &< vtime.C \end{aligned}$$

Therefore, we need to decide what it means for one vector to be less than ($<$) another. We define this as the element-wise maximum of the two vectors is greater than the local clock value. That is, the formal definition of the ordering of two vectors $time.A$ and $time.B$ is given by:

$$\begin{aligned} vtime.A = vtime.B &\equiv (\forall i :: vtime.A.i = vtime.B.i) \\ vtime.A \leq vtime.B &\equiv (\forall i :: vtime.A.i \leq vtime.B.i) \\ vtime.A < vtime.B &\equiv vtime.A \leq vtime.B \wedge vtime.A \neq vtime.B \end{aligned}$$

Equivalently, the $<$ operator can be defined directly:

$$vtime.A < vtime.B \equiv (\forall i :: vtime.A.i \leq vtime.B.i) \wedge (\exists i :: vtime.A.i < vtime.B.i)$$

In other words, to assign a timestamp to a receive event, the element-wise maximum of the timestamp of the message and the local clock is taken. In order to ensure that this is strictly greater than *both*, we increment the component corresponding to the local process.

Notice that with vector clocks, we have to be careful how we increment a clock. With logical clocks, we simply wanted to ensure that the assigned time is larger than the previous time. We are free to increment the counters by any positive amount (we chose 1 every time in the previous section). With vector clocks, however, we have several choices for making a time value larger. For

example, we could increase only one component or increase all the components. Intuitively, if we are haphazard with how the vector is increased (*e.g.*, increasing all the components by various amounts) there is the same loss of information as we saw with logical clocks and we won't have the desired equivalence property.

One way to think of process j 's vector clock is as a list of the most recent news from all the processes in the computation (including itself). It follows that we expect an invariant of the implementation to be:

$$(\forall j, k :: vclock.k.j \leq vclock.j.j)$$

5.4.3 Algorithm

The algorithm for updating vector clocks is analagous to that for logical clocks. The only part to which we have to pay special attention is how to increment the clock when processing a receive event. Again, there are 3 cases to consider for possible events:

1. **Local event.** The clock component corresponding to the process is incremented and the updated clock value is the timestamp of the new event.

$$\begin{aligned} vclock.j &:= vclock.j + 1 \\ ; \quad vtime.A &:= vclock \end{aligned}$$

2. **Send event.** The clock component corresponding to the process is incremented and the updated clock value is the timestamp of the new event and of the message being sent.

$$\begin{aligned} vclock.j &:= vclock.j + 1 \\ ; \quad vtime.A, vtime.m &:= vclock, vclock \end{aligned}$$

3. **Receive event.** The clock is updated by taking the component-wise maximum of the current clock (vector time of last event on this process) and the timestamp of the message (vector time of the corresponding send event). This maximum must be incremented to ensure that the new clock value is strictly larger than *both* previous events (which “happened-before” the current event). It is incremented by increasing *only the local component* of the vector. Again, the updated value is the timestamp of the new event.

$$\begin{aligned} vclock &:= \mathbf{max}(vtime.m, vclock) \\ ; \quad vclock.j &:= vclock.j + 1 \\ ; \quad vtime.A &:= vclock \end{aligned}$$

Again, the invariant maintained with respect to $vclock$ is that it is equal to the most recently assigned timestamp.

As an exercise, consider the timeline in Figure 5.7. Assign vector timestamps to the events (indicated by circles) in accordance with the algorithm above. Initially, the clocks begin at the values indicated.

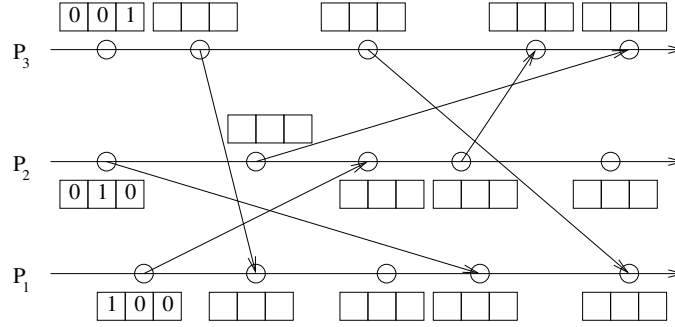


Figure 5.7: Assigning timestamps using vector clocks.

The algorithm can be written as a program in our action system model. For a single process (j) the algorithm is:

```

Program      VectorClock.j
var            $j, k$  : processes,
                $ch.j.k$  : channel from  $j$  to  $k$ ,
                $m$  : message,
                $A$  : event,
                $vclock.j$  : vector time of  $j$ ,
                $vtime.A$  : vector time of  $A$ ,

initially     $vclock.j.j = 1$ 
                $\wedge (\forall k : k \neq j : vclock.j.k = 0)$ 

assign
  local event  $A \longrightarrow$     $vclock.j.j := clock.j.j + 1$ 
                                   ;  $vtime.A := vclock.j$ 
  || send event  $A \longrightarrow$   $vclock.j.j := vclock.j.j + 1$ 
    (to  $k$ )                       ;  $vtime.A, vtime.m := vclock.j, vclock.j$ 
                                   ;  $ch.j.k := ch.j.k \mid m$ 
  || rcv event  $A \longrightarrow$   $vclock.j := \max(vtime.m, vclock.j)$ 
    ( $m$  from  $k$ )                   ;  $vclock.j.j := vclock.j.j + 1$ 
                                   ;  $vtime.A, ch.j.k := vclock.j, tail(ch.j.k)$ 

```

The following property can be established as an invariant of this program (where A_j denotes event A that occurred at process j , and m_j denotes a message m sent by process j):

$$\begin{aligned} & (\forall j, k :: vclock.k.j \leq vclock.j.j) \\ \wedge & (\forall j, k, m_j :: vtime.m_j.k \leq vclock.j.k) \\ \wedge & (\forall A_j, B_k :: A_j \longrightarrow B_k \equiv vtime.A_j < vtime.B_k) \\ \wedge & (\forall A_j, B_k :: A_j \longrightarrow B_k \Leftarrow vtime.A_j.j \leq vtime.B_k.j) \end{aligned}$$

The second last conjunct in this invariant is the desired equivalence property that we set out to establish. By examining the timestamps of two events, we can determine whether one happened before the other (or whether they were concurrent).

Notice also the last conjunct in the invariant. It gives a “short-cut” for determining whether there is a happens before relation between two events. It states that the entire vectors do not need to be compared. The only way for the j component of event B_k ’s timestamp to be greater than the j component of event A_j ’s timestamp is for there to be a message chain from A_j to B_k . That is, $A_j \rightarrow B_k$.

For example, in Figure 5.8 the partial vector clocks for two events are given. Event A is on process 2, while event B is on process 3. Because $vtime.B.2$ is greater than (or equal to) $vtime.A.2$, there must be a chain of messages from A to B . That is, we can conclude from only this partial information that $A \rightarrow B$. The clock updating scheme ensures that these partial values are only possible when $A \rightarrow B$. (Which, in turn, means that the other entries of these vector timestamps must satisfy the \leq ordering.)

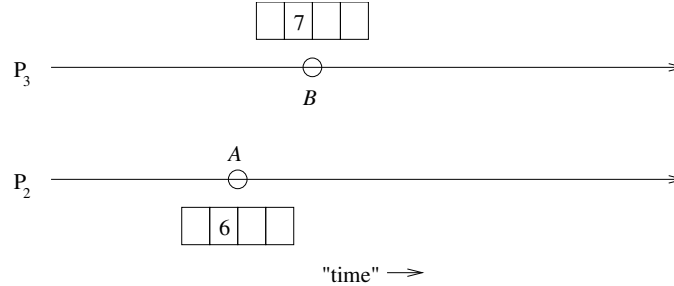


Figure 5.8: Using vector clocks to determine whether $A \rightarrow B$.

5.5 Synchronization of Physical Clocks

Virtual clocks and logical time is fine for capturing the partial order of events in a distributed system. But what if we want each process to have something like a “real” clock? Such a clock could be used to report the time-of-day, or even to timestamp events. Of course, the lack of shared state makes the direct implementation of such a shared clock impossible. Each process can, however, maintain its own timer that advances at the appropriate rate (*e.g.*, one tick per millisecond).

This is analogous to a person wearing a wrist watch. The current time displayed by that watch is not shared state. We believe in the accuracy of the watch however and consult it as if it were “true time”.

Unfortunately, a watch that ticks accurately is not enough. It can be used to accurately time the duration of some interval (*e.g.*, a stop-watch at a race),

but it can not be used to tell time (*e.g.*, is it 11:30 yet?) until it has been *synchronized* with “true time”. Even after having been synchronized, real clocks do not tick perfectly accurately and can “drift” over time. To correct this, the clock must be synchronized again. (The more accurate the ticking of the clock, the less often such resynchronizations are needed.)

In this section, we discuss clock synchronization schemes first under the assumption that messages are transmitted with arbitrary delay, then under the assumption that message delay is bounded by some fixed value.

5.5.1 Messages with Unbounded Delay

Consider a process receiving a time notification from another process. In Figure 5.9, process P receives a message from T indicating that the current time (at T) is 3:00pm. In this situation, P knows at point x of its computation

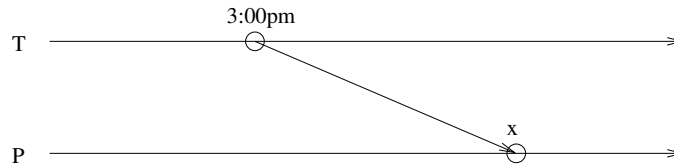


Figure 5.9: Single time message with unbounded delay.

only that it is currently _____ at T .

Request-Reply

The reason that the assertion at point x is so weak is that P has no way of knowing the delay in the transmission of the message from T . To improve on this situation, P must be able to bound this delay. One solution, is to introduce a protocol between P and T whereby P first sends a message to T , then waits for the reply (containing the current time at T). Since P can time the interval between its first message and receiving the reply, an upper bound is known on the delay incurred by T 's message. This is illustrated in Figure 5.10.

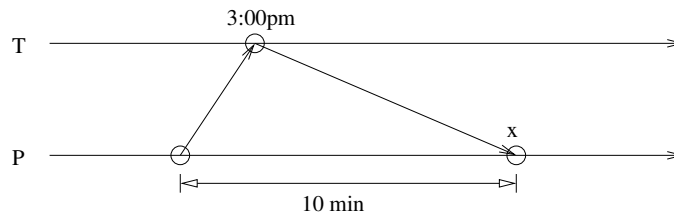


Figure 5.10: Request and reply protocol for obtaining current time.

In this figure, say the interval at P measures 10 minutes. Then, at point x in the computation, P knows that the time message from T was sent (at some point) in the last 10 minutes! Therefore, at x , P can infer that the *current* time at T is in the interval [_____ .. _____].

Multiple Requests

If more accuracy is required, the protocol can be repeated in the hope of narrowing the interval. Of course, if the delays on the messages (both outgoing and incoming) are exactly the same, there is no improvement. If, however, the delays vary, the possible interval narrows.

For example, consider P immediately sending another request to T . This time the round-trip delay for a reply (received at point y is 12 minutes). P would expect the *current* time at T to be in the interval $[3:12 \text{ .. } 3:22]$. The

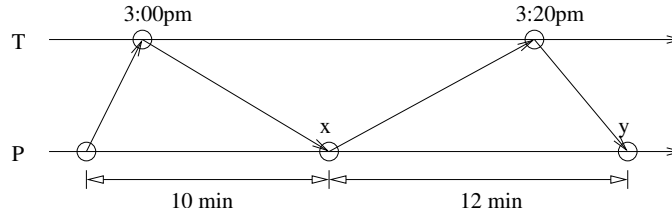


Figure 5.11: Multiple requests and replies for obtaining current time.

time value received in this second reply, however, is 3:20. Therefore, the current time at T must be in the interval $[3:20 \text{ .. } 3:32]$ (by the same calculation as before). In order for both of these to be correct, the current time at T (at point y) must be in their intersection, or the interval [_____ .. _____]. Notice that this resulting interval is tighter than either of the previous two!

Another way to get a similar effect is to simultaneously send requests for current time to multiple servers. For example, P might send requests to servers T_1 , T_2 , T_3 , and T_4 . Once a reply has been received from all of these servers, the appropriate intervals for each are known. Assuming all servers are accurate (*i.e.*, are synchronized with “true time”), then the intersection of these intervals can be taken as before. See Figure 5.12.

Again, notice in Figure 5.12 that the resultant interval is narrower (*i.e.*, higher accuracy) than any one individual interval.

This analysis is based on the assumption that all T_i are accurate. This is a big assumption! In general, the T_i will not themselves be perfectly synchronized with “true time”. This means that the intersection of all the intervals might be empty.

Therefore, instead of taking the intersection over all intervals, we can view each interval as a vote for the possibility of the true time lying in that interval. These votes can be tallied to determine a likely interval (or set of intervals).

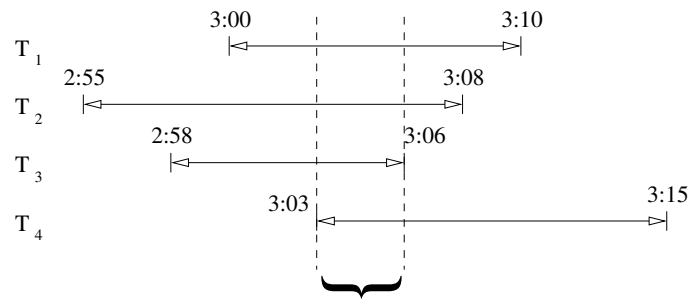


Figure 5.12: Multiple simultaneous requests for obtaining current time.

Algorithm. One way to compute such a tally is as follows:

- Sort all the interval endpoints.
- Initialize a counter to 0.
- Traverse the sorted list of endpoints
 - For a minimum endpoint, increment the counter
 - For a maximum endpoint, decrement the counter

This algorithm calculates how many intervals are satisfied by a given time. For example, in Figure 5.13 you should draw in the calculated tally as a function of possible current time.

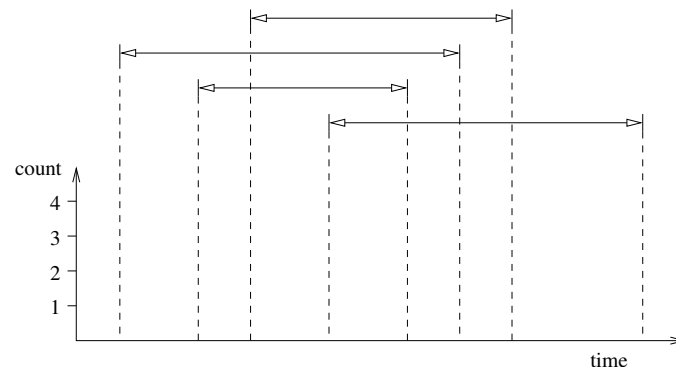


Figure 5.13: Tallying counts for candidate current times.

In general, this graph (*i.e.*, the calculated tally) could have virtually any pattern of increments and decrements to the count. (Subject to the constraint, of course, that the count is always non-negative and is 0 to the left and right.) For example, Figure 5.14 is a possible result of the collection of several intervals.

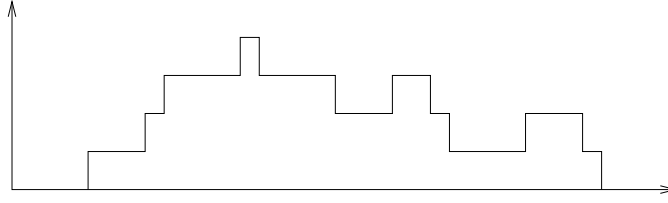


Figure 5.14: Possible result of tallying several intervals.

The information contained in such a tally can be used in various ways. For example, the interval(s) with the highest count could be used. Alternatively, an apriori decision might be made on the greatest number of faulty servers. Then, any interval that exceeds the corresponding threshold (*e.g.*, has a count of at least 3) is part of the solution interval.

5.5.2 Messages with Bounded Delay

If an upper (and lower) bound is known on the delay a message can incur, this information can be used to further restrict the interval. For example, in the Figure 5.10 we saw that process P could deduce the time at T was in the interval $[3:00 \dots 3:10]$. If it was known, however, that the minimum message delay was 2 minutes, this interval could be restricted to $[3:02 \dots 3:08]$.

Lamport's paper "Time, Clocks, and the Ordering of Events in a Distributed System" also analyzes the synchronization of physical clocks following the logical clock scheme (*i.e.*, a clock can only be increased to bring it back into synchrony, never decreased).

In this analysis, a clock is viewed as a continuous, differentiable function of time, $C.t$ (and $C_i.t$ for process i 's clock). An assumption is made that clocks run at approximately the correct rate. That is, the derivative of the clock with respect to time is about 1. More precisely, the error in rate is bounded by:

$$\left| \frac{dC_i.t}{dt} - 1 \right| < \kappa$$

The bound κ is very small (certainly much less than 1) and represents an upper bound on the amount of "wobble".

This bound means that after a period of time, τ , two clocks (each drifting in opposite directions) can be out of synchrony with each other by a total of $2\kappa\tau$.

A further assumption is made that message delay is known to lie within a particular interval. If the minimum delay is n and the maximum delay is m , then this helps bound the size of the uncertainty interval when a timestamped message is received. For example, a process receiving a message from T timestamped with a value t knows that the current time at T is in the interval $[t + n \dots t + m]$. The size of this interval, then, is given by $m - n$. Call this

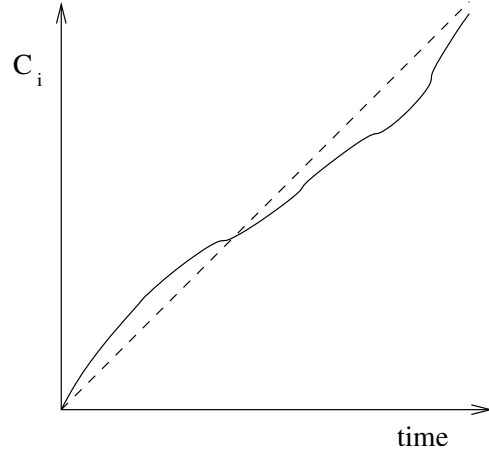


Figure 5.15: A physical clock with drift.

size ζ . Finally, if the diameter of the graph (*i.e.*, the minimum number of hops required for any process to communicate with any other) is d , then the synchronization error (ϵ) is bounded by

$$\epsilon \leq d(2\kappa\tau + \zeta)$$

Chapter 6

Diffusing Computations (Gossip)

6.1 References

A nice, brief, and seminal reference for synchronization using diffusing computations is Dijkstra and Scholten's paper "Termination Detection for Diffusing Computations", in *Information Processing Letters*, 11(1), p. 1–4, 1980 [DS80].

6.2 Introduction

The term "diffusing computation" refers to a computation that begins at a single process and spreads out to include the entire set of processes. This is also known as "gossip" because the way the diffusion occurs is similar to the way gossip travels among people: one person tells everyone they know, then these people tell everyone they know, and so on, and so on, ...

In this chapter, we will look at a particular instance of this larger class of algorithms. We will examine how gossip can be used for barrier synchronization of a collection of processes.

One application for barrier synchronization of this kind is in rolling over sequence numbers. For example, with logical time the counters used to timestamp events must constantly increase. In a real implementation, however, these counters are bounded by some maximum value. It is therefore necessary to periodically rollover and start reusing small values of the counter. In order to start using the low counter values, a process must notify all other processes of its intention to rollover the counter and it must know that these processes have received this notification.

In general, a barrier synchronization is a point at which processes wait until all processes have reached the barrier before proceeding. See Figure 6.1. The key property is that after the synchronization point, a process knows that all

other processes have (at least) reached the synchronization point.

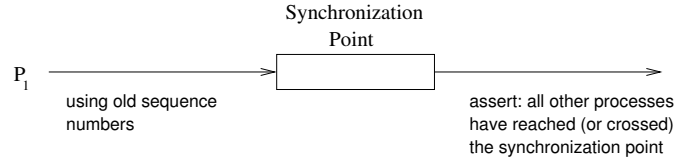


Figure 6.1: Barrier synchronization

6.3 Operational View

We are given some fixed topology of nodes (processes) connected by undirected edges (channels). The graph is finite and connected. For example, see Figure 6.2. One node is distinguished as the “initiator” (call it I).

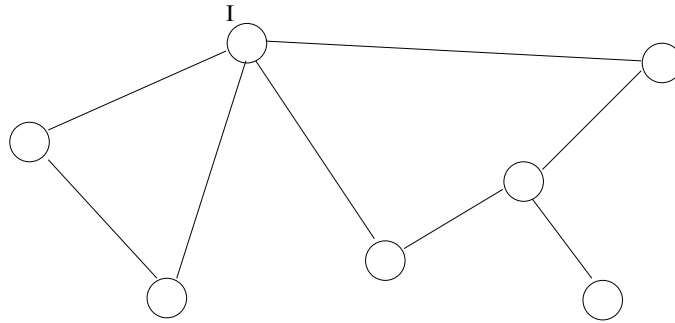


Figure 6.2: Example of a given topology for diffusing computation

The computation spreads by “diffusion” (if you are a chemist) or “gossip” (if you are sociologist). When a process first hears the gossip it must:

1. pass the gossip on to all its other neighbors, and
2. remember its source of the gossip (its “parent”)

This is a spreading phase, in which more and more processes are included in the gossip. It is easy to see that eventually all processes will have heard the gossip (assuming the graph is connected).

This, however, is only half the problem. In order to have a barrier synchronization, the initiator must know when all the processes have been included in the gossip. This suggests that we need not only an expanding phase, but a constricting phase as well!

Indeed, we add the following rule: When a process hears an acknowledgement back from *all* of its “other neighbors”, it passes an acknowledgement back to its parent. What about a leaf node (*i.e.*, a node with a single neighbor)? When it receives the gossip (from its one neighbor) it can immediately _____.

Notice that we actually only need a *single* kind of message! That is, we do not need separate gossip messages and acknowledgements. Instead, there is a single kind of message. The *first* arrival of such a message is considered the gossip (and sets the parent for this node). Subsequent arrivals of messages (*i.e.*, from children) are considered acknowledgements.

If we try to reason about this algorithm operationally, it quickly becomes a muddle. The difficulty is that there are many possible ordering of events, and they seem to have different implications. In particular, when a process x receives a message from a process y , it seems that x might not be able to tell whether it is receiving the *gossip* from y or an *acknowledgement* from y . This situation is depicted in Figure 6.3.

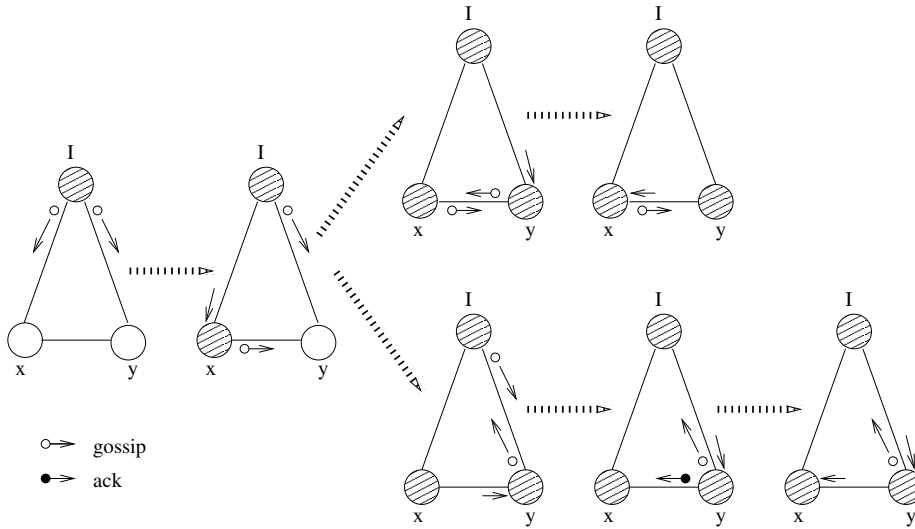


Figure 6.3: Possible race condition in gossip algorithm

From x 's point of view, both cases are identical. Yet in one case the message it is receiving is a gossip (*i.e.*, y has set x to be its child), while in the other case it is receiving an acknowledgement (*i.e.*, y has set x to be its parent).

Put another way, the “problem” is that processes that x considers to be its children might not consider x to be their parent!

Convincing ourselves that this algorithm is correct, therefore, requires some careful thought. An operational view might give us some intuition, but it is hard to argue convincingly about the correctness of this algorithm with operational traces. Instead, we take a more assertional approach.

6.4 Specification

The first part of any argument of correctness must be a specification. For this purpose, we begin with some definitions:

$$\begin{aligned}
 x \text{ nbr } y &\equiv x \text{ and } y \text{ are neighbors} \\
 msg(x, y) &\equiv \text{there's a message in the channel from } x \text{ to } y \\
 done &\equiv (\forall v : v \text{ nbr } I : msg(v, I))
 \end{aligned}$$

Now the specification of the algorithm is given in terms of a safety and a progress requirement:

safety: invariant. $(done \Rightarrow (\forall u :: u \text{ has completed gossip}))$

progress: $(\forall v : v \text{ nbr } I : msg(I, v)) \rightsquigarrow done$

6.5 Algorithm

A process can be in one of three states: *idle*, *active*, and *complete*. Initially processes are *idle*. Once they hear the gossip (and pass it along to their children), they become *active*. Once they hear acknowledgements from their children (and acknowledge their parent), they become *complete*.

For a (non-initiator) process u :

```

Program      Gossip  $u$ 
var            $parent_u$  : process,
                 $state_u$  : {idle, active, complete},
                 $msg(a, b)$  : channel from  $a$  to  $b$ ,
initially     idle
                 $\wedge (\forall v : u \text{ nbr } v : \neg msg(u, v))$ 
assign
  (  $\parallel v : v \text{ nbr } u : idle \wedge msg(v, u) \longrightarrow$ 
     $parent_u := v$ 
     $\parallel (\parallel w : w \text{ nbr } u \wedge w \neq v : msg(u, w) := \mathbf{true})$ 
     $\parallel state_u := active$  )
   $\parallel active \wedge (\forall v : v \text{ nbr } u \wedge v \neq parent_u : msg(v, u)) \longrightarrow$ 
     $msg(u, parent_u) := \mathbf{true}$ 
     $\parallel state_u := complete$ 

```

We will refer to these two actions as A_1 and A_2 respectively.

6.6 Proof of Correctness

We reason about the two properties separately.

6.6.1 Safety

First, define two directed graphs. Each graph is defined by its set of vertices and its set of edges.

$T_1 \stackrel{\text{def}}{=} \begin{array}{l} \text{vertices: set of active or complete nodes} + \text{Initiator} \\ \text{edges: } \langle u, \text{parent}_u \rangle \text{ for all nodes in graph} \end{array}$

$T_2 \stackrel{\text{def}}{=} \begin{array}{l} \text{vertices: set of active nodes} + \text{Initiator} \\ \text{edges: } \langle u, \text{parent}_u \rangle \text{ for all nodes in graph} \end{array}$

As an aside, note that $T_2 \subseteq T_1$.

There are two key invariants:

1. T_1 is a _____. (It only grows.)
2. T_2 is a _____. (It grows then shrinks.)

Other invariants that are helpful include:

$$\begin{aligned} \text{msg}(u, v) &\Rightarrow u \in T_1 \\ u.\text{complete} &\Rightarrow (\forall v : \langle v, u \rangle \in T_1 : v.\text{complete}) \end{aligned}$$

Now we show that the two key invariants hold for this program.

T_1 is a tree.

Proof. First observe that it is clearly true initially, since initially T_1 consists of a single node, the Initiator, and no edges.

Next we observe that edges and vertices are never removed from T_1 . There is a single action that can add edges and vertices, and that is action A_1 . Consider when A_1 adds a vertex u to the graph. If action A_1 is enabled, it means $\text{idle}.u \wedge \text{msg}(v, u)$. This means that v is in T_1 (from the helpful invariant). So adding a vertex u and an edge $\langle u, v \rangle$ is ok (*i.e.*, we are adding a leaf, which keeps it a tree). \square

T_2 is a tree.

Proof. The proof is similar to the above argument for why it is true initially and why it is preserved by action A_1 .

The difference here is that T_2 can shrink. That is, vertices can be deleted (by action A_2). In order for such a deletion to preserve the invariant, we must ensure that only _____ are removed.

We argue this by contradiction. That is, assume that the u that is removed (by becoming *complete*) is *not* a leaf. Therefore, there exists an n in T_2 that has u as its parent. But for A_2 to be enabled, there must be $\text{msg}(n, u)$ (since n can not be u 's parent). Therefore, n has sent a message to its parent and so must be *complete*. This means, however, that n is *not* in T_2 as assumed. This is a contradiction, and hence u must be a leaf. \square

6.6.2 Progress

We show progress by finding a metric (in this case, an increasing metric).

Proof. As a metric, we choose $(\#complete, \#active)$. This is a pair whose first element is the size of $T_1 - T_2$ and whose second element is the size of T_2 . This metric is ordered *lexicographically*.¹

This metric is bounded above (since the graph is finite). It is also guaranteed not to decrease, since **stable**.(*complete.u*) and a node stops being *active* (*i.e.*, the second component decreases) only by becoming *complete* (*i.e.*, the first component increases).

Now it remains to show that the metric increases eventually. That is,

$$(\exists v :: \neg complete.v) \wedge M = m \rightsquigarrow M > m$$

Consider a v that is not *complete*. There are 2 cases:

1. v is *idle*. Therefore, v is not in T_1 . Now consider any path from v to I (such a path exists because the graph is connected). There is some node n on this path that is the “last node” in T_1 . So n must have neighbors that are not in T_1 . Therefore, there is an action that increases the metric (one of these neighbors receiving the gossip from n and becoming active).
2. v is *active*. You should be able to complete this part of the argument...

□

¹For lexicographic ordering: $(a_1, b_1) \leq (a_2, b_2) \equiv a_1 \leq a_2 \vee (a_1 = a_2 \wedge b_1 \leq b_2)$

Chapter 7

Mutual Exclusion

7.1 Reference

Chapter 6 of the Singhal and Shivaratri book [SS94] is an excellent reference for this material.

7.2 Introduction

A distributed collection of processes share some common resource. They require, however, mutually exclusive access to this shared resource. Our task is to design a program that acts as a “mutual exclusion layer” that resolves the conflict between processes competing for the shared resource. Figure 7.1 is a basic sketch of this architecture.

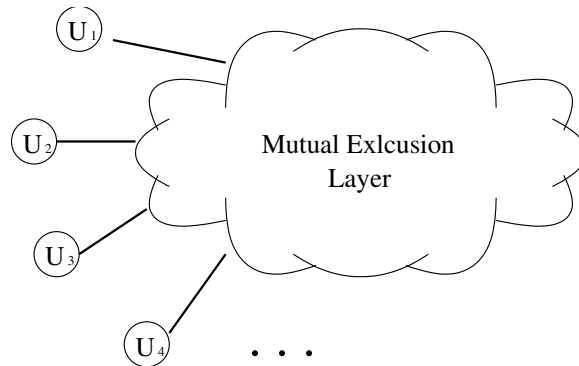


Figure 7.1: Mutual exclusion layer arbitrates user process conflicts

Each user process cycles between three states: noncritical (NC), try (TRY), and critical section (CS). Some of these transitions, in particular from NC to

TRY and from CS to NC , are controlled by the user process. The remaining transition, from TRY to CS , is controlled by the mutual exclusion layer. The state-transition is given in Figure 7.2

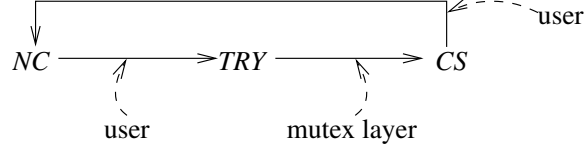


Figure 7.2: State transition diagram for user processes

The requirement that the user process conform to this state transition diagram can be stated more formally as:

$$\begin{aligned}
 &NC \text{ next } NC \vee TRY \\
 &\text{stable}.TRY \\
 &CS \text{ next } CS \vee NC
 \end{aligned}$$

Note that these are requirements on the *user process* only. They will not necessarily be properties of the mutual exclusion layer and hence not be properties of the final, composed, system.

In addition, we place one more important requirement on the user process: the critical section must be finite. That is:

$$\text{transient}.CS$$

We also fix a protocol to be used between the user processes and the mutual exclusion layer. The user process sends a message “try” when it wishes to access the shared resource (corresponding to entering its “try” state). The mutual exclusion layer replies with a message “grant” when access to the shared resource has been given. Finally, the user process sends a message “exit” when it is releasing the shared resource and returning to the noncritical state. This protocol is summarized in Figure 7.3

Our task is to design a mutual exclusion layer that does this job. The first step, of course, is to formalize what we mean by “this job”.

7.3 Specification

Safety. The safety requirement states that no 2 users are in their critical section at the same time.

Progress. There are two kinds of progress requirement we could make:

1. *Weak.* If some user process is in TRY , eventually some user process enters CS .

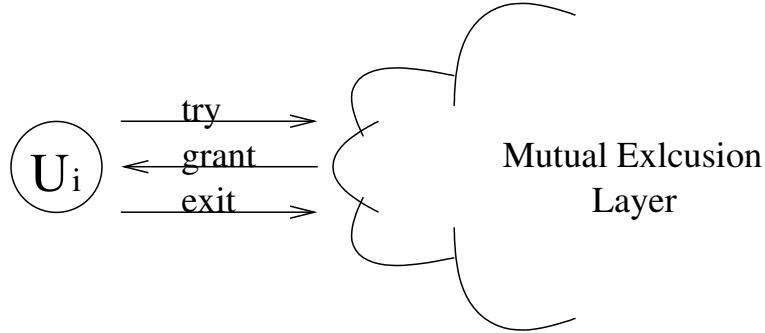


Figure 7.3: Protocol between user process and mutual exclusion layer

2. *Strong*. If user process i is in TRY , eventually user process i enters CS .

Typically we are interested in solutions that give strong progress.

7.3.1 A Trivial Solution

Perhaps the simplest solution is to implement the mutual exclusion layer as a single process. This process maintains a queue of outstanding requests. Requests are granted one at a time, in a first-come first-server manner. Safety and progress are obviously satisfied, so this solution is correct.

On the other hand, this centralized solution creates a bottleneck. We would prefer our solution to be decentralized (*i.e.*, it should itself be a distributed system). For example, as shown in Figure 7.4, the layer might consist of a collection of system processes (the P_i 's), one for each user process U_i .

At this point, it is helpful to step back and consider a more general problem. Mutual exclusion can be viewed as an instance of the problem of maintaining a distributed variable value.

7.4 Distributed Atomic Variables

7.4.1 Introduction

Consider having a single variable, call it x , that multiple processes wish to update. Of course, the value of this variable could be stored in a single location and every update to this variable would consist of a message to (and reply from) that location. This solution is quite inefficient, however. Instead, we would like to *distribute* the management of this single variable, x .

One natural solution might be to have all processes keep a copy of the current value of the variable. To update the value, a process changes its local copy and broadcasts the update.

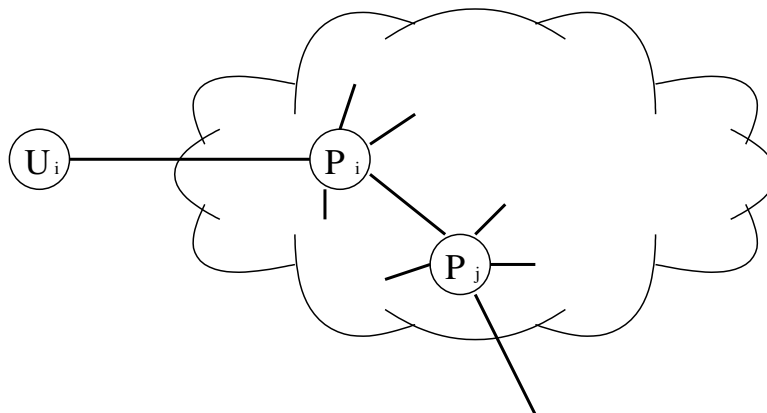


Figure 7.4: Mutual exclusion layer as a distributed system

There is a problem with this solution, however. Different processes could have different views of the order of updates on x . For example, in Figure 7.5, x begins with value 0. Process 1 increments x (new value is 1), while process 2 doubles x (new value is 0). What is the final value of x ?

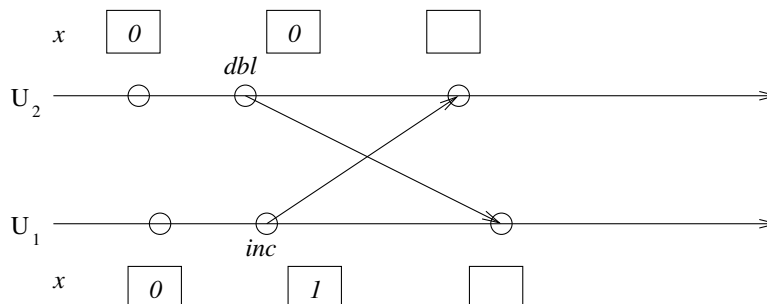


Figure 7.5: Updates to a distributed copies of a shared variable

The solution is that we need to order events so that there is no ambiguity. That is, we need a total ordering on events that all processes agree on. We've already seen such an ordering using _____. That is, every process executes the updates on its copy of x in _____ order of _____.

Now consider a process U_i that wishes to modify, say increment, the variable x . Question: When is such an increment operation “dangerous” to perform? Answer: _____. Put another way, when is it “safe” to perform? Answer: _____.

7.4.2 Algorithm

Each process keeps as part of its state the following:

- copy of x ,
- logical clock,
- queue of “modify requests” (with their logical time stamps), and
- list of “known times”, one for each other process.

A process executes a modification request when:

- the request has the minimum logical time of all requests, *and*
- all “known times” are later than the time of the request.

The example given earlier in Figure 7.5 is repeated in Figure 7.6 with this algorithm. The value of x is initially 0 and the logical clock of U_1 is 1 while the logical clock of U_2 is 6.

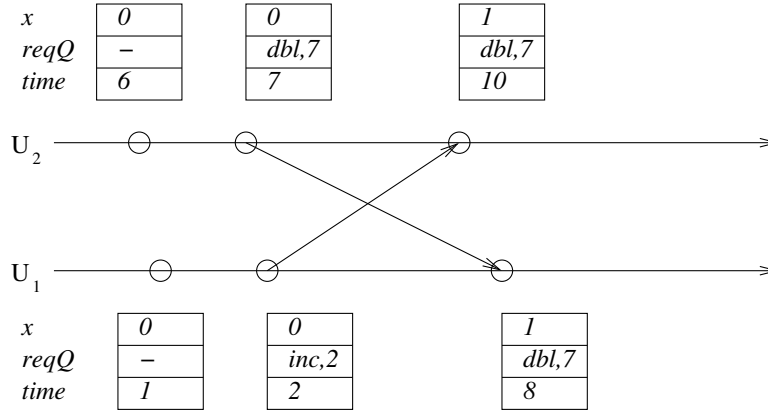


Figure 7.6: Request queues for sorting update requests

One potential problem with this approach is obtaining a “known time” from all the other processes. What if there is no “known time” for some U_i ? For example, what if U_i never issues a request to update x ? To solve this difficulty, we need a protocol whereby the time from U_i is obtained (regardless of whether it wishes to issue its own update request for x). Either the time can be explicitly requested from U_i or we can use a system of acknowledgements to make sure that recent time values are known.

As an aside, this approach is based on the assumption that logical time can indeed be *totally* ordered. That is, there are no “ties”. This can be done by making use of a common assumption, namely that processes have unique id’s, to break any ties.

7.5 Nontoken-Based Solutions

7.5.1 Lamport's Algorithm

In the trivial solution we first postulated for this problem, there was a single, centralized, “pending request queue”. This queue resolved conflicts by allowing processes to enter their critical section in a first-come, first-serve manner.

This queue is just a data structure like any other and in the previous section we saw how to support a *distributed* version of a data structure (in that case an integer with increment and double operations). We can use this same strategy to support a distribution of the pending request queue data structure.

In this algorithm, each process keeps:

$reqQ$: queue of timestamped requests for CS (sorted in _____ order), and
 $knownT$: list of last “known times” for all other processes.

To request entry to its critical section: P_i broadcasts $\langle req_i, t_i \rangle$ to all other processes

When P_j receives a request to enter a critical section, a (timestamped) acknowledgement is returned.

Now, to enter CS , P_i must have:

1. req_i at the head of $reqQ$, and
2. for all other j , $knownT[j]$ must be larger than the timestamp of req_i .

On the other hand, to release the CS , P_i does the following:

1. removes req_i from its $reqQ$, and
2. broadcast a $\langle release_i \rangle$ message to all other processes.

When a process receives a $\langle release_i \rangle$ message, it removes P_i 's request from its $reqQ$. Note that this may cause this process to enter its CS (since it may now have the request with the minimum timestamp)!

Proof of Correctness

Safety.

Proof. The proof proceeds by contradiction. Assume P_a and P_b are both in their CS . Therefore, both P_a and P_b have their own requests at the head of their (sorted) $reqQ$. So the head of $P_a.reqQ$ is $\langle req_a, t_a \rangle$, while the head of $P_b.reqQ$ is $\langle req_b, t_b \rangle$.

Assume, wlog¹, that $t_a < t_b$. But, since P_b is in its CS , it must be the case that $t_b < P_b.knownT[a]$. Hence, req_a (with its timestamp of t_a) must be in $P_b.reqQ$ (assuming messages are FIFO). Hence req_b is not at the head of $P_b.reqQ$. \square

¹without loss of generality

Progress.

Proof. As a metric for a particular process P_i , use the number of entries in $P_i.knownT$ that are less than its request time ($req_i.t$). Clearly this is bounded below (by 0). Also, since logical time is monotonically increasing, it never increases.

To see that this metric is guaranteed to decrease, consider a process P_j with an entry less than P_i 's request time. That is:

$$P_i.knownT[j] < req_i.t$$

Process P_j 's logical time is guaranteed to increase beyond P_i 's (since the request message, $\langle req_i, req_i.t \rangle$ has been sent to P_j). Also, an acknowledgement with this increased logical time will be returned to P_i , removing P_j from the list of processes with a known time less than $req_i.t$. \square

7.5.2 Optimization #1

As an optimization, notice that not all acknowledgements are required. In particular, if a request has *already* been sent with a *later* timestamp than a received request, the request received does not need to be acknowledged. The acknowledgement is required to guarantee a known time for this process that is greater than the request time. But the request already sent can serve as such an acknowledgement!

For example, in Figure 7.7, process P_i does not need to send an acknowledgement for the most recent request (from process P_j).

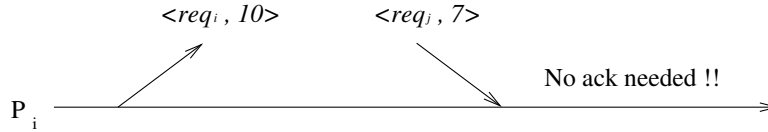


Figure 7.7: First optimization: reducing the number of acknowledgements

7.5.3 Optimization #2: Ricart-Agrawala

As a further optimization, we can eliminate acknowledgements for even more requests. In particular, if a request has already been sent with an *earlier* timestamp than a received request *and that request is still pending*, there is no need to acknowledge the received request. Eventually, the pending request will be granted and we will send a $\langle release \rangle$ message, which will serve as the acknowledgement!

For example, in Figure 7.8, process P_i does not need to send an acknowledgement for the most recent request (from process P_j).

The modifications of the original algorithm are the following:

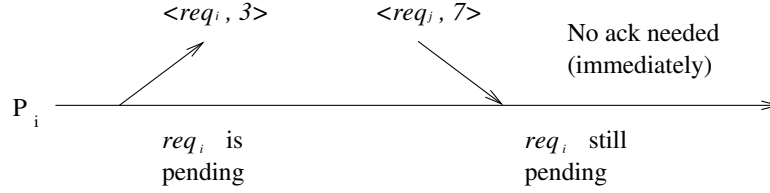


Figure 7.8: Ricart-Agrawala optimization: deferring acknowledgements

1. When P_i receives $\langle req_j, t_j \rangle$, it defers its acknowledgement if:

- (a) P_i is in CS , or
- (b) P_i is in TRY and $t_j > t_i$ of req_i .

Otherwise, the acknowledgement is sent immediately.

2. When P_i exits CS , it sends all deferred acknowledgements.

7.6 Token-Based Solutions

7.6.1 Introduction

Another category of solutions to the mutual exclusion problem can be categorized as token-based algorithms. The problem specification is similar, although with token-based algorithms there is typically a fixed (finite and connected) topology of processes and channels that connect them. We also assume there are no self-loops. The safety and progress properties required of the solution, however, are unchanged.

The key idea in all of these algorithms is the use of a single, indivisible *token*. A token is a very useful concept that recurs frequently in the design of distributed algorithms. It is defined by a simple property:

Key Idea. Tokens can be neither created nor destroyed.

A process is allowed to enter the critical section *only if* it holds the token. This simple rule guarantees the safety part of the specification!

For progress, we must guarantee that every process that enters TRY eventually receives the token. There are several different algorithms to guarantee this. We examine them in increasing order of generality.

7.6.2 Simple Token Ring

Consider the graph where processes are arranged as a ring. (If the given topology is not a ring, any subset of edges that form a ring can be used. This algorithm can only be applied if there exists such a subset of edges.) See Figure 7.9.

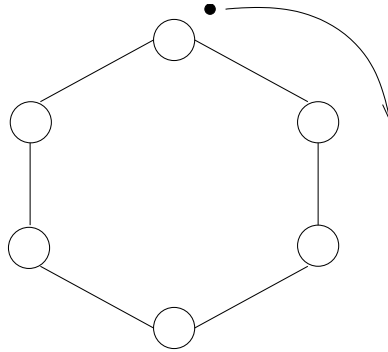


Figure 7.9: Simple token ring

The token is made to constantly circulate in a clockwise (CW) direction around the ring. If a process wishes to enter CS , it simply waits for the token to come around.

Algorithm

To use resource:

$hungry := \mathbf{true}$

When token arrives:

if $hungry$

[use resource]

send token on (CW)

Proof of Correctness The proof of correctness of this algorithm is trivial.

Concerns There is an efficiency concern, however. Regardless of the demand for the shared resource, the token continues to circulate.

7.6.3 Token Ring with Requests

Our second token-based scheme addresses this efficiency concern by circulating the token only in response to requests to enter CS . In addition to the token, we add a second kind of message: *request*.

Key point: Tokens circulate one way and requests circulate in the *opposite* direction.

In Figure 7.10, tokens circulate CW while requests circulate counter-clockwise (CCW).

Notice that after a process has sent a request (either on its own user process's behalf or forwarded from a neighbor) there is no need for it to send any subsequent requests!

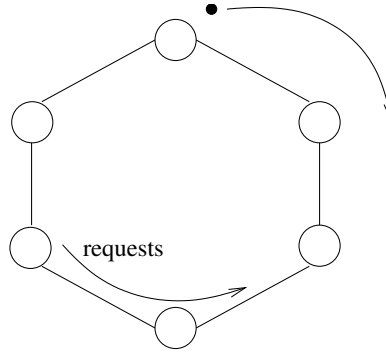


Figure 7.10: Token ring with requests

Algorithm

To use the resource:

```

    if holder  $\neq$  self
        hungry := true
        if  $\neg$ asked
            send request (CCW)
            asked := true
        wait until using
    else
        using := true
        [use the resource]
        using := false
        if pending_requests
            send token on (CW)
            pending_requests := false

```

When a request is received:

```

    if holder = self  $\wedge$   $\neg$ using
        send token on (CW)
    else
        pending_requests := true
        if holder  $\neq$  self  $\wedge$   $\neg$ asked
            send request (CCW)
            asked := true

```

When the token is received:

```

    asked := false
    if hungry
        using := true
        hungry := false

```

```

else
    send token on (CW)
    pending_requests := false

```

Some key points to keep in mind with respect to this algorithm are the following.

- A process forwards *at most one* request.
- Every process “*knows*” where the token is in the system (*i.e.*, somewhere to its right).
- Requests are sent *towards* the token.
- The token travels along the same path as requests, but in *the opposite direction*.

Proof of Correctness The proof of safety remains trivial. The proof of progress is not difficult but requires (of course) finding an appropriate metric.

Concerns Although this algorithm addresses the concern of the previous solution (*i.e.*, when there is no demand the token does not circulate), it does have one serious limitation. Namely, it requires _____.

7.6.4 Token Tree (Raymond)

A more general token-based approach uses a *spanning tree* of the given topology to pass the token. Since every connected graph has a spanning tree, this algorithm can be applied to any connected topology.

The solution is similar to the token ring in the following key ways:

- Every process “*knows*” where the token is.
- Requests are sent towards the token.
- The token travels along the same path as requests, but in the opposite direction.

How can we ensure that all of these properties are true? The key idea is:

So how is this key invariant maintained? Answer:

There is one potential problem to deal with. In the token ring, a process with *pending_requests* knew where to send the token once it arrived. That is, it would simply send it along CW. With a tree, however, when a process receives the token, to which neighbor (child) should it be sent? Infact, there could be *pending_requests* from multiple children!

The solution to this problem is to maintain _____.

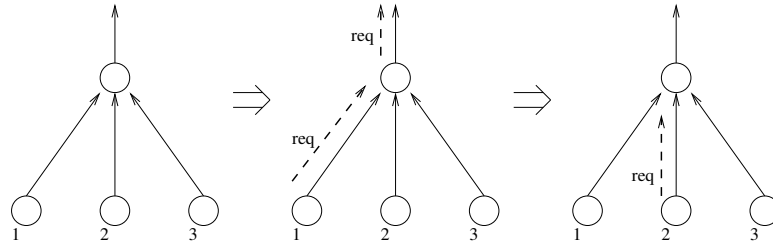


Figure 7.11: Multiple pending requests

Proof of Correctness The proof of correctness again hinges on finding a good variant. In this case, the argument is more subtle than for the token ring since the token does not appear to be getting “closer” in an obvious way!

Concern One potential concern with this algorithm is that the path followed by the token (and requests) is limited to the edges of a spanning tree. The actual graph, however, may contain considerably more edges that could shorten the distance needed for the token to travel to reach its destination. The next solution allows us to exploit these edges by making use of the entire topology.

7.6.5 Token Graph

Raymond’s algorithm can be generalized to arbitrary connected topologies in a very natural way. Again, we require that all processes know where the token is. this can be accomplished by pointing all edges “towards” the token.

Notice that sometimes there is a choice as to what “towards” means. In particular, we must be careful that no _____ are formed!

Key Idea. The graph is acyclic (*i.e.*, it is a partial order).

Question: how do we maintain this invariant?

Answer:

7.6.6 Summary of Key Ideas for Token-based Solutions

- Indivisible token can be neither created nor destroyed (and this guarantees safety).
- Tree is directed and token is always at the root.
- Process sends only one request (may need to maintain a list of pending requests).

- Generalization of tree to partial order.
- To maintain a partial order, make all edges incoming for node with token.

Chapter 8

Dining Philosophers

8.1 References

1. The dining philosophers problem first appeared in a paper by Dijkstra in 1971 [Dij71]. It has become a classic synchronization problem and its many solutions are discussed in many references.
2. This presentation of the hygienic solution follows chapter 12 of the UNITY book [CM88] rather closely.

8.2 Introduction

In the initial formulation of the dining philosophers problem, five philosophers are sitting around a table. Between each philosopher is a single fork and, in order to eat, a philosopher must hold both forks.

As in the formulation of the mutual exclusion problem, philosophers cycle between 3 states: thinking, hungry, and eating. Again, the philosopher controls its own transitions from thinking to hungry and from eating to thinking. On the other hand, the conflict-resolution layer controls transitions from hungry to eating.

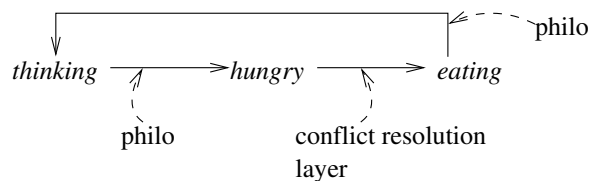


Figure 8.1: State transitions for a philosopher

Again we require that philosophers eat for a finite time. Philosophers may think for an arbitrary amount of time (potentially unbounded).

As a generalization of the classic formulation, we extend the notion of neighbor. Instead of five philosophers around a table, we consider an arbitrary (but finite) undirected graph. Vertices in the graph represent philosophers and edges define neighbors (we require that the graph not have any self loops). The classic formulation can be seen as an instance of this generalization with _____. Similarly, the mutual exclusion problem considered in the previous chapter can be seen as an instance of the generalized dining philosophers problem with _____.

8.3 Specification

We wish to design a “conflict resolution layer” that guarantees the required specification:

safety: (mutual exclusion) neighbors do not eat at the same time.

progress: (starvation freedom) every hungry philosopher gets to eat eventually.

To help formalize this specification, we introduce some notation. For a philosopher u , we denote its state by three predicates:

$u.t$: u is thinking

$u.h$: u is hungry

$u.e$: u is eating

For each pair of philosophers, u and v , we also have a boolean $E(u, v)$. This boolean is true exactly when there is an edge between u and v . That is:

$$E(u, v) \equiv u \text{ and } v \text{ are neighbors}$$

We can now write the specification more formally:

safety: _____

progress: _____

8.4 Naive Solutions

The first naive solution is to simply ask for permission to eat from all of one’s neighbors. The hungry process then waits to receive permission from all these neighbors. This is equivalent (in the anthropomorphic formulation) to trying to grab all available forks. The problem with this solution is that deadlock is possible. That is, a cycle of dependencies can form where each process is waiting for the next one in the cycle.

As an attempt to patch this naive solution, we could prevent the formation of cycles by requiring a process to acquire all forks at once. If this is not successful (*i.e.*, the philosopher fails to acquire permission from even one neighbor) the philosopher releases any held forks and tries again. This algorithm, however, is still not correct. The problem is that _____.

8.5 Hygienic Solution

As a general principle in conflict resolution, it is necessary to “break the symmetry”. As long as one philosopher looks exactly like any other, the problem is hopeless. In the original formulation of the problem, with 5 philosophers around a table, one solution is to have all philosophers grab their right fork first, *except for one*, which grabs its left fork first. This is an example of how symmetry can be broken.

In our general formulation, we need to break the symmetry in an undirected graph. One obvious way to do this is to give edges a _____. Of course, when we do this, we must be careful not to introduce a _____, since this would defeat the purpose (*i.e.*, of breaking the symmetry).

Therefore, a structure is imposed on the graph, as in Figure 8.2, such that it is acyclic. That is, the graph is a _____! As usual, partial orders can be drawn so that all the edges point the same way. With logical time, we drew the timelines so that all edges were directed towards the right. Here, we typically draw the partial order so that edges are directed up.¹

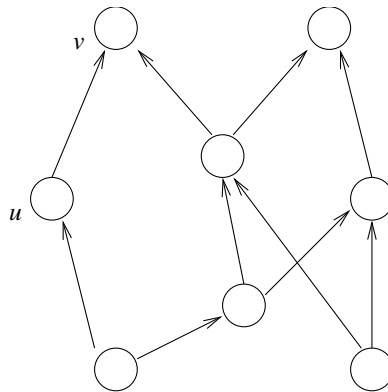


Figure 8.2: Partial order of philosophers

A directed edge represents “priority”. That is, in Figure 8.2, philosopher v is higher in the partial order than philosopher u , and therefore v has higher priority than u .

¹With the convention that edges always point up, the arrows can even be omitted. This is usually called a “Hasse diagram”.

There are two important rules:

1. Conflicts for a shared resource (*i.e.*, the fork) are resolved in favor of the process with higher priority.
2. After a process “wins” a conflict (and gets to eat), its neighbors should be given a chance to win the next round. So the priority of the winner is *lowered*.

Recall that the key property of this graph is that it is _____. How do we maintain this invariant property? That is, when we change the graph (*i.e.*, by lowering the priority of a philosopher), how do we make sure that the new graph is still acyclic? Answer: _____. Such a modification is guaranteed not to create new cycles. (Why?) Hence the resulting graph is still a partial order.

8.6 Refinement of Specification

The solution is developed as a series of refinements of the original specification. Each refinement can be proven to imply the previous one. In addition, each refinement introduces new constructs that suggest how to implement the solution.

8.6.1 Safety (Forks)

The first refinement suggests how to implement the original safety property, *i.e.*:

$$(\forall u, v :: \textbf{invariant} . (\neg(E(u, v) \wedge u.e \wedge v.e)))$$

In order to ensure this mutual exclusion between neighbors, we use the notion of a token. We introduce one token for each edge (and call this token a “fork”). To eat, a philosopher must hold all of its forks.

$$(\forall u, v :: \textbf{invariant} . (u.e \wedge E(u, v) \Rightarrow \textit{fork}(u, v) = u))$$

We can prove that this property implies the original specification, so this is indeed a refinement.

This refinement does not, however, address the issue of priority. So we refine the specification further.

8.6.2 Priority (Clean vs. Dirty)

A fork can be held by either neighbor. That is, it can be held by either the low priority or high priority philosopher of a neighbor pair. Priority is used to arbitrate in the case that *both* philosophers want the fork, but if only the low priority philosopher wants the fork it can certainly acquire it.

Hence, forks alone do not tell us which philosopher is higher priority. To encode priority, we add state to the fork, allowing it to be “clean” or “dirty”. Now the priority of an edge can be defined in terms of this state.

$$u \leq v \quad \equiv \quad \begin{aligned} &\text{shared fork at } v \text{ and clean} \\ &\vee \text{ shared fork at } u \text{ and dirty} \end{aligned}$$

This can be rewritten as:

$$u \leq v \quad \equiv \quad \begin{aligned} &(fork(u, v) = v \wedge clean(u, v)) \\ &\vee (fork(u, v) = u \wedge \neg clean(u, v)) \end{aligned}$$

Complete Figure 8.3 to illustrate the two scenarios in which v could have higher priority than u .

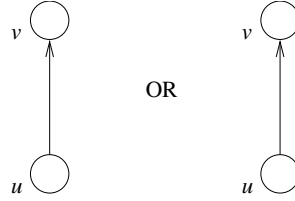


Figure 8.3: Possible position and state of a shared fork, given $u \leq v$

Some important properties of clean and dirty forks are the following:

1. An eating process holds all its forks and the forks are dirty.
2. A process holding a clean fork continues to hold it (and it remains clean) until the process eats.
3. A dirty fork remains dirty until it is sent from one process to another (at which point it is cleaned).²
4. Clean forks are held only by hungry philosophers.

As an aside, why is it important that clean forks not be held by thinking philosophers? Because _____.

The result of these properties is that a process yields priority only when it eats! Again, all of these properties together can be shown to imply the previous refinement of the specification. In particular, this refinement still gives the desired progress property. The proof of this implication, of course, uses a metric.

Under what conditions does a process u send a fork to a neighbor v ?

²This property is the reason for the name of this algorithm: “hygienic dining philosophers”. A philosopher cleans a dirty fork before giving it to its neighbor.

1. the fork is at u
2. _____
3. _____
4. _____

In a distributed system, however, how can process u tell whether v is hungry? This issue motivates the next refinement.

8.6.3 Neighbor Hunger (Request Tokens)

We introduce another token for each edge of the graph. This token is called a “request token” and is used to request a fork when the process becomes hungry. The key invariant for a request token is: if u holds *both* the fork and the request token shared with v , then v is hungry.

Some important properties of this refinement are the following:

1. u sends a request token to v if
 - (a) u holds the request token,
 - (b) _____, *and*
 - (c) _____.
2. u sends the fork to v if
 - (a) u holds both the fork and the request token,
 - (b) _____, *and*
 - (c) _____.
3. a hungry process eats when
 - (a) it holds all its forks, *and*
 - (b) for all its neighbors, either the fork is clean or it does not hold the request token.

8.7 Algorithm

In the previous section, the original specification was refined with a series of new specifications. Note that each specification in the series was *tighter* or *stronger* than the previous. With each successive refinement, a new mechanism was introduced that was more easily implementable. With the final refinement, it is easy to see how to write a program that satisfies this specification (and hence also satisfies the original).

Program *hygienic*
initially $p.state = thinking$
 $(\forall q : E(p, q) : clean(p, q) = \mathbf{false})$
 Priorities form a partial order
always $p.t \equiv p.state = thinking$
 $p.h \equiv p.state = hungry$
 $p.e \equiv p.state = eating$
assign
 $[H_p]$ $p.h \wedge fork(p, q) = q$
 $\longrightarrow req(p, q) := q;$
 $[E_p]$ $p.h \wedge (\forall q : E(p, q) : fork(p, q) = p \wedge (clean(p, q) \vee req(p, q) = q))$
 $\longrightarrow p.state := eating;$
 $clean(p, q) := \mathbf{false};$
 $[R_p]$ $req(p, q) = p \wedge fork(p, q) = p \wedge \neg clean(p, q) \wedge \neg p.e$
 $\longrightarrow fork(p, q) := q;$
 $clean(p, q) := \neg clean(p, q);$

8.7.1 Message-Passing Pseudocode

As an exercise in understanding the relationship between our programming notation and real distributed programs written in some message-passing notation, you should try to write the pseudocode for the dining philosophers solution given above. After you have gone through this exercise, compare your solution to the pseudocode given in this section.

```
bool fork[i]: T iff this philo holds ith fork
bool clean[i]: T iff fork is clean
bool request[i]: T iff you have the request token

int num_neigh: number of neighbors this philosopher has
int fork_ct:   number of forks held

m represents incoming messages

philosopher {

  while (true)
    /* state == thinking */
    repeat
      /* thinking philo holds only dirty forks
      recv (m)
      switch (m)
        case (req from neighbor i)
          /*assert fork[i] = T; clean[i] = F*/
          fork[i] = F
```

```

        fork_ct --
        send (fork) to neighbor i
    case (became_hungry)
        state = hungry
until (state == hungry)

for (i < num_neigh; i = 0; i ++)
    if (fork[i] == F)
        /*assert request[i] = T*/
        send (req) to neighbor i
        request[i] = F

while (fork_ct < num_neigh)
    recv (m)
    switch(m)
        case (req from neighbor i)
            /*assert fork[i] = T*/
            request[i] = T
            if (clean[i] = F)
                send (fork) to neighbor i
                fork[i] = F
                fork_ct --
                send (req) to neighbor i
                request[i] = F
        case (fork from neighbor i)
            /* fork arrives clean */
            fork[i] = T
            clean[i] = T
            fork_ct ++
    endwhile

state = eating
for (i < num_neigh; i = 0; i ++)
    clean[i] = F

repeat
    recv(m)
    switch (m)
        case (req from neighbor i)
            request[i] = T
        case (done_eating)
            state = thinking
until (state == thinking)

for (i < num_neigh; i = 0; i ++)
    if (request[i] = T)

```

```

fork[i] = F
send (fork) to neighbor i

end while

```

8.8 Proof of Correctness

Overview The proof of correctness can be done in stages corresponding the series of refinements presented above. At each stage, the fact that the refined specification implies the previous specification is shown. Finally, the proof that the program text (which has not been given here) satisfies the last, most refined, specification is given.

The most important (or perhaps subtle) stage of this proof is the refinement that introduces priority (using clean and dirty forks) and uses this notion to establish progress for individual philosophers (as required by the previous refinement).

Informal Sketch Informally, the rules for changing priority guarantee that a hungry process does not go “down” in the partial order, unless it eats. That is, any changes made to the partial order do not increase the number of paths to a hungry process.

Also, a hungry process—if it does not eat—must rise in the ordering. This follows from the observation that a hungry process does not remain at the top indefinitely. When such a hungry process eats, it falls to the bottom and other hungry processes go “up”.

Slightly More Formal Sketch As a more formal treatment of this proof of correctness, we should propose a metric and establish that it satisfies the required properties. One good metric for the progress of a hungry process u is the sum of the number of processes above u and the number of *thinking* processes above u . Let this sum be $u.m$. This metric is bounded below (by 0). It is also guaranteed not to increase (by the argument given above that (i) the partial order is modified by making a vertex a sink, which can not increase the number of processes above u , and (ii) for a process to become thinking it must have been eating, and hence becomes a sink). That is:

$$u.m = k \text{ unless } u.e \vee u.m < k$$

Finally, we can show that the metric is guaranteed to decrease. That is:

$$u.h \wedge u.m = k \rightsquigarrow u.e \vee (u.h \wedge u.m < k)$$

Establishing this final leads-to property is really the crux of the proof. First, define what is meant by being at the “top” of the partial order. We say a process is at the “top” when it has no *hungry* neighbors of higher priority.

$$u.top \equiv (\forall v : E(u, v) \wedge v.h : v \leq u)$$

We can then show that a hungry process does not remain at the top indefinitely:

$$u.h \wedge u.top \rightsquigarrow \neg u.h \vee \neg u.top$$

From this we can establish two things:

1. A process at the top eats eventually, since:

$$u.h \wedge u.top \wedge u.m = k \rightsquigarrow u.e \vee u.m < k$$

2. If a hungry process is *not* at the top, there is a hungry process *above* it that *is* at the top!

$$\textbf{invariant}.(\neg u.top \Rightarrow (\exists v : u < v : v.h \wedge v.top))$$

These two properties together give us the desired progress property for the metric:

$$u.h \wedge u.m = k \rightsquigarrow u.e \vee (u.h \wedge u.m < k)$$

And this metric property (using induction) gives the need result (*i.e.*, $u.h \rightsquigarrow u.e$).

For a complete treatment, refer to the Unity book.

8.9 Summary of Key Points

1. The graph is initialized to break the symmetry (by forming a *partial order*).
2. To lower a philosopher in the partial order, make *all* the edges outgoing.
3. Tokens called *forks* are used for mutual exclusion.
4. Priority is encoded in *clean* and *dirty* forks.
5. Request tokens are used to determine whether a neighbor is hungry.

Chapter 9

Snapshots

9.1 References

1. The seminal paper on snapshots was written in 1985 by Chandy and Lamport [CL85].
2. The Singhal and Shivaratri text book [SS94] contains a good description of snapshots in Sections 5.6 and 5.7.
3. The Unity book presents snapshots in Chapter 10 (but for an informal presentation, the previous reference is better).

9.2 Problem Description

We are given a (fixed) set of processes and FIFO channels that connect them (*i.e.*, a finite, connected topology). A process can do one of three things:

1. change internal (local) state,
2. send a message, *or*
3. receive a message.

The global state of such a system is defined as the union of all the local states of the individual processes *and* the state of the channels. Some examples of global state include:

- The number of eating philosophers.

$$(\sum p : p.e : 1)$$

- The number of tokens in the system.

$$(\sum v : v.holding : 1) + (\sum u, v : E[u, v].holding : 1)$$

Our task is to determine this global system state. This is also known as “taking a snapshot”. It is important to do this *without stopping the underlying computation*. That is, as this snapshot is being taken, the state can be changing!

Recall that there is no global clock in our model of computation. If there were, the problem would be easy. We would simply require all processes to record their local state at a particular time, say 12:00. What is the state of each channel in such a snapshot? The messages in transit. Question: how can this state be calculated? Answer:_____.

As a crude intuitive analogy, consider an army of ants, each with their own local view of the world. They want to coordinate their behavior so as to form a complete picture of an elephant! A single ant takes a small picture, and together the collage has to form a coherent picture the entire elephant. The challenge lies in the facts that (i) the ants do not have a synchronized notion of time and (ii) the elephant may be moving!

9.3 The Naive Approach

Recall the timeline-based representation of the actions (we called them events) in a computation. In Figure 9.1, the x’s indicate when the individual processes record their local state.

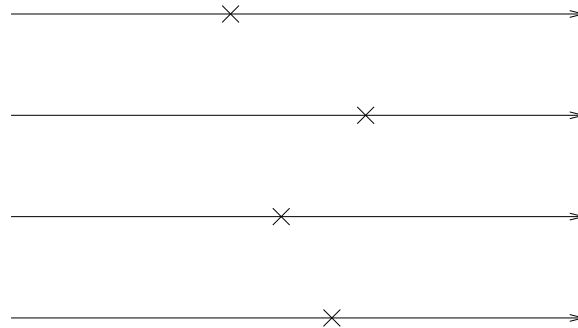


Figure 9.1: Simplified timeline indicating recording of local state

These individual views are joined to form a snapshot of the global state. A snapshot, then, is like a “wavey cut” through a timeline, as shown in Figure 9.2.

Notice, however, that not all cuts make sense! For example, consider a distributed network containing bank account information. Initially, the balance at bank node 1 is \$1000, while the balance at bank node 2 is \$0. A transfer of \$500 is requested from node 1 to node 2. This computation is represented in Figure 9.3.

Now, if a snapshot is taken and the bank nodes record their local state at the x’s, the net worth recorded by the snapshot is _____. Therefore, this snapshot has failed to capture an accurate view of the global system state.

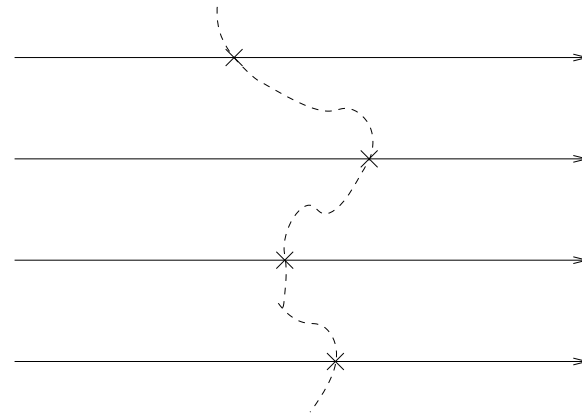


Figure 9.2: Wavey cut through a timeline

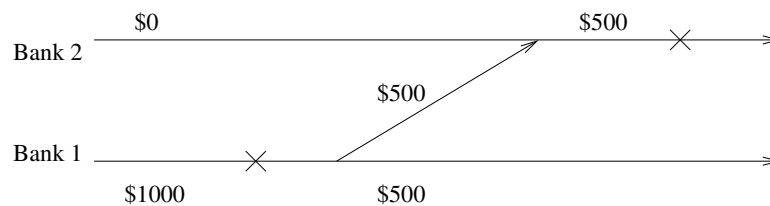


Figure 9.3: Transfer of funds in a distributed bank

9.4 Consistent Cuts

The root of the problem described above is that the \$500 being transferred is counted twice (once at node 1 and then again at node 2). We said that the global state was the union of the local process states and the channel states. But in the previous example, what is the state of the channel between the two nodes?

If the arrow were going from “inside” the cut to “outside”, then the state of the channel at the cut would simply be _____. The other direction, however, makes no sense. Graphically, a cut such as illustrated in Figure 9.4 makes sense while a cut such as given in the example (and repeated in Figure 9.5) does not.

A cut that “makes sense” is called *consistent*. The following are all equivalent:

- A cut is a valid snapshot.
- A cut is consistent.
- A cut has no incoming edges.

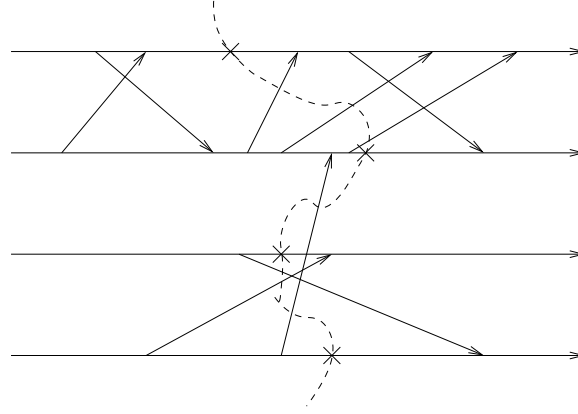


Figure 9.4: Consistent cut

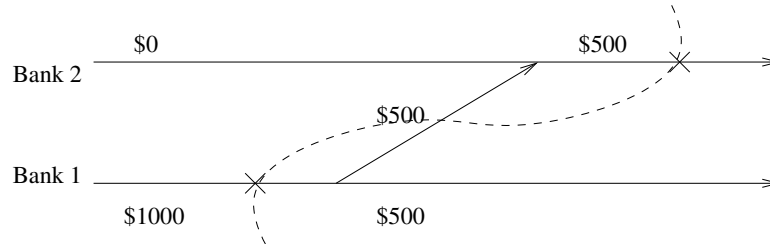


Figure 9.5: Inconsistent cut

- A cut is input closed.
- $(\forall c : c : \text{channel} : \#sent.c \geq \#rcv.c)$

9.5 Solution #1: Logical Time

We observed that if there were a shared clock, taking a snapshot would be easy. It is natural to ask, then, whether logical time can be used in the determination of a valid snapshot.

In order for a cut to *not* be consistent, it must be the case that there is a

_____ relationship between the local snapshots at two processes.

$$\neg(\text{cut is consistent}) \Rightarrow (\exists p, q :: \text{snap}.p \longrightarrow \text{snap}.q)$$

$$\Rightarrow (\exists p, q :: \text{ltime}.p \text{ --- } \text{ltime}.q)$$

Therefore, one way to guarantee that a cut *is* consistent is to use the con-

trapositive. That is:

$$\begin{aligned} \text{cut is consistent} &\Leftarrow \neg(\exists p, q :: \text{itime.snap.p} \text{ ___ } \text{itime.snap.q}) \\ &\equiv \underline{\hspace{10em}} \end{aligned}$$

In other words, every process records its state at the same *logical* time. Question: how is the state of each channel calculated? Answer: _____.

9.6 Utility of Snapshots

Surprising Observation. The “wavey cut” corresponding to a snapshot may never have occurred as an *actual* global state during the computation! For example, consider the bank application from earlier and the following sequence of events:

1. Initially, node 1 has \$1000 and node 2 has \$0.
2. The balance in node 2 is increased to \$200 (bank error in your favor!)
3. A transfer of \$500 from node 1 to node 2 is made.

The timeline for this computation, along with a consistent cut is shown in Figure 9.6.

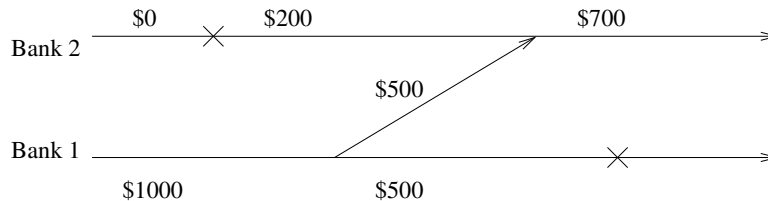


Figure 9.6: A possible consistent cut

Now notice the global state recorded by this consistent cut. The net worth of the account is \$1000 (\$500 at node 1 and \$500) in transit. But this global state *never occurred* in the actual computation! The net worth of the account was \$1200 during the transfer.

Nevertheless, we consider such a cut consistent (*i.e.*, valid). Why would having a global state that may never have occurred be useful?

Put another way, even though the snapshot state *is not* guaranteed to have occurred, what *is* guaranteed? Answer: _____.

View the collection of snapshot information as a computation (superimposed on the original computation) that takes some time (*i.e.*, some number of actions). It begins in state S_b and ends in a state S_e . This is represented in Figure 9.7.

What we require of the snapshot state, S_{snap} , is:

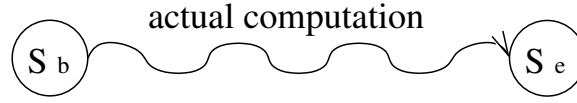
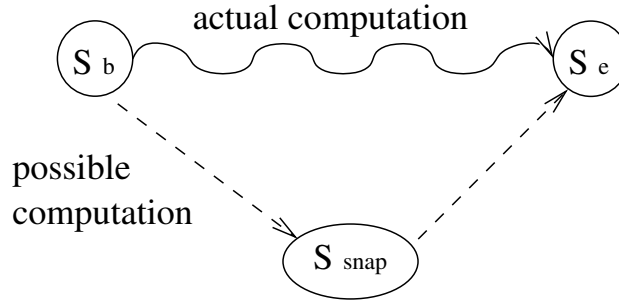






Figure 9.7: Computation during a snapshot

1. there exists a path from S_b to S_{snap} , and
2. there exists a path from S_{snap} to S_e .

Figure 9.8: Reachability of S_{snap}

One (main) use of snapshots, therefore, is in the detection of *stable* predicates! This utility is due to the following two key properties for a predicate P that is stable:

1.  \Rightarrow 
2.  \Rightarrow 

9.7 Solution #2: Marker Algorithm

9.7.1 Algorithm

Key Idea. Markers are sent along channels to “flush out” messages that are in transit when snapshot is taken. (Of course, this requires FIFO channels.) To begin a snapshot, the initiator records its local state and sends a marker along each outgoing channel.

When a process receives a marker for the first time, it:

- records its local state,
- records the incoming channel along which the marker was received as being empty, and

- sends markers along each outgoing channel.

When a process receives a subsequent marker, it:

- records the state of that incoming channel as _____.

The snapshot is complete when all processes have received markers along all incoming channels.

9.7.2 Proof of Correctness

The progress part of this algorithm (*i.e.*, that eventually a snapshot is gathered) is easy. The markers spread from the initiator until a marker has been sent on every channel. This algorithm eventually records the state of every process and every channel.

But is this recorded global state a valid snapshot? In other words, *could* the recorded state have occurred?

Label each action in the computation as either “pre” or “post”. A “pre” action is an action at some process p that occurs *before* p records its local state for the snapshot. Conversely, a “post” action is an action that occurs *after* that process records its local state. The actual computation consists of a sequence of “pre” actions and “post” actions (interleaved).

We prove that we can *swap* adjacent actions in the computation to form an *equivalent* computation consisting of a sequence of all the “pre” actions followed by all the “post” actions.

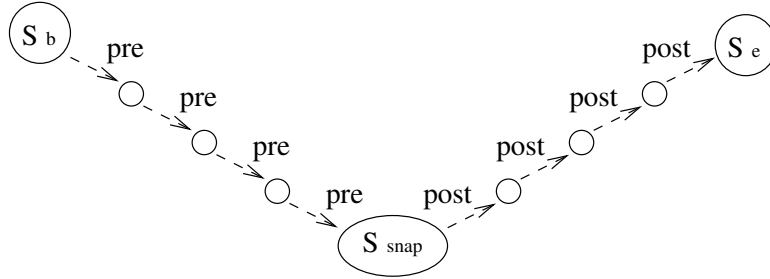


Figure 9.9: Result of swapping adjacent actions

When we do this swapping, we will preserve the order of the “pre”s with respect to each other as in the original computation. Similarly, we will preserve the order of “post”s.

Consider a pair of actions in the sequence that are out of order. Such a pair has the form $\langle a_{post}, b_{pre} \rangle$. First observe that these two actions must be on different processes. There are three cases for b_{pre} :

1. b_{pre} is a *local* action. In this case, it is easy to swap this action with a_{post} , since the two can not affect each other.

2. b_{pre} is a *send* action. Again, the swap is easy.
3. b_{pre} is a *receive* action. In this case, we can not perform the swap with a_{post} if a_{post} is the corresponding send! We show that it is impossible for a_{post} to be the corresponding send action. Since this action, a_{post} is a “post” action, a marker must have been sent in all outgoing channels. Since the channels are FIFO, this marker must be delivered *before* the message being sent in action a . This means that the b action must be a “post” action as well, contradicting the assumption that they are out of order.

This swapping of adjacent pairs is continued until we have the desired sequence.

□

Chapter 10

Termination Detection

10.1 Problem Description

We are given a fixed topology of processes and unidirectional channels. As usual, the topology must be finite. Unlike other problem formulations, we permit self-loops and we do not require the graph to be connected. An example topology is given in Figure 10.1.

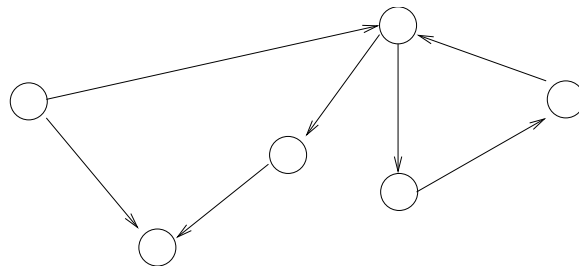


Figure 10.1: Topology with directed channels

A process can be in one of two possible states: *active* or *idle*. A process that is in the *active* state can perform the following actions:

1. send a message,
2. receive a message, *or*
3. change state to be *idle*.

An *idle* process, on the other hand, can only perform a single action: receive a message, at which point it becomes *active*. Complete Figure 10.2 to indicate these allowed state transitions.

The program for such a process is given below.



Figure 10.2: State transitions allowed for a process

```

Program       $P_i$ 
var           $state : \{active, idle\}$ 
                $out : \{P_i \text{'s outgoing channels}\}$ 
                $in : \{P_i \text{'s incoming channels}\}$ 
always       $(act \equiv state = active)$ 
                $\wedge (idl \equiv state = idle)$ 
initially    $act$ 
                $\wedge (\forall c : c \in out : empty.c)$ 
assign
  (  $\parallel c : c \in out : act \longrightarrow \text{send msg on } c$  )
   $\parallel ( \parallel c : c \in in : act \wedge \neg empty.c \longrightarrow \text{rcv msg on } c )$ 
   $\parallel act \longrightarrow state := idle$ 
   $\parallel ( \parallel c : c \in in : idl \wedge \neg empty.c \longrightarrow \quad state := active$ 
                                     ;  $\text{rcv msg on } c )$ 

```

Notice that the initial conditions indicate that all processes are active and all channels are empty.

Question: When should we consider such a system to be “terminated”? You can answer the question intuitively. You can also answer the question by calculating the *FP* for this program. Answer: the computation has terminated when:

1. _____, and
2. _____.

10.2 Algorithm

We begin by introducing a special “detector” process. All processes are assumed to have a channel to the detector. See Figure 10.3.

When a process becomes *idle*, it sends a message to the detector. So, when the detector has heard from all processes, it knows that all processes

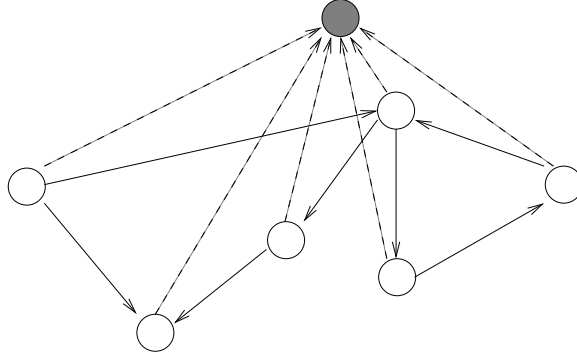


Figure 10.3: Process for detecting termination

_____. Is this enough information to conclude that the computation has terminated? Answer: _____.

For example, consider a system with two processes, p and q . There is a single channel, from p to q . Initially, both processes are active and the channel is empty. Process p sends a message to q and then becomes *idle*, notifying the detector. Process q becomes *idle* (before the message from p arrives) and notifies the detector. Both notifications arrive at the detector and it concludes that the computation has terminated. But this is not correct because there is still a message in transit from p to q .

Therefore, simply indicating a transition to *idle* is not enough. In addition to this information, a process must also send information concerning its _____. In particular, a process sends:

- The number of messages sent on each of its outgoing channels.
- The number of messages received on each of its incoming channels.

For example, a process with two incoming channels (d and e) and two outgoing channels (f and g), as illustrated in Figure 10.4, would send a message to the detector of the form:

$$< id, inputs\{(d, r.d), (e, r.e)\}, outputs\{(f, s.f), (g, s.g)\} >$$

where $r.c$ is the number of messages received on channel c and $s.c$ is the number of messages sent on channel c .

At the detector, the number of messages put *into* a channel c (*i.e.*, sent) are recorded as $in.c$, while the number of messages taken *out of* a channel c (*i.e.*, received) are recorded as $out.c$. The detector looks for the condition:

- a notification has been received from all processes, *and*
- $(\forall c :: in.c = out.c)$

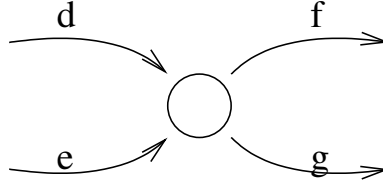


Figure 10.4: Local topology of a single process

Notice that this modification solves the problem illustrated earlier with the erroneous detection of termination for the system with 2 processes, p and q . With this augmented protocol, the detector would have received a notification from all processes (satisfying the first condition), but the value of $in.c$ would be _____ while the value of $out.c$ would be _____. Hence, it would not be the case that $in.c = out.c$ and the detector would not report termination.

10.3 Specification

We define the boolean predicate *done*, set by the detector, by:

$$done \equiv (\forall p :: notified.p) \wedge (\forall c :: in.c = out.c)$$

The specification of the algorithm can be stated by:

safety: invariant.($done \Rightarrow$ computation has terminated)

progress: computation has terminated $\leadsto done$

We can rewrite this specification using the definition of the termination of the computation:

safety: invariant.($done \Rightarrow (\forall p :: p.idl) \wedge (\forall c :: r.c = s.c)$)

progress: $(\forall p :: p.idl) \wedge (\forall c :: r.c = s.c) \leadsto done$

10.4 Proof of Correctness

The progress part of the proof is easy. If computation has terminated, all the processes have become idle and all the channels are empty. This means that every process has sent a message to the detector and furthermore the number of messages sent and received on every channel are equal. Eventually, the detector reports termination.

For safety, we must show:

$$done \Rightarrow (\forall p :: p.idl) \wedge (\forall c :: r.c = s.c)$$

That is:

$$(\forall p :: \text{notified}.p) \wedge (\forall c :: \text{in}.c = \text{out}.c) \Rightarrow (\forall p :: p.\text{idl}) \wedge (\forall c :: r.c = s.c)$$

The key insight is that termination is _____. We're hoping that if this condition holds of the detector's—possibly outdated—view of the computation, it must hold *now*. When does this work? Answer: When the view is a valid _____. So, to prove safety, we just need to show that the state seen by the detector (when it flags termination, that is, when *done* is true) is a _____).

When *done* is true, the detector has received information from all the processes. It therefore has information from some past cut of the computation. This cut is a valid snapshot exactly when _____. Formally, the condition needed for this cut to be a valid snapshot is:

$$(\forall c :: s.c \geq r.c)$$

But, since *done* is true, we know that for this particular cut we have the property:

$$(\forall c :: s.c = r.c)$$

Therefore it *is* a valid snapshot. Therefore, termination being a property of this view of the computation means that it is a property of the current computation as well (since termination is stable).

□

Chapter 11

Garbage Collection

11.1 Reference

The presentation of this material follows chapter 16 in the Unity book [CM88].

11.2 Problem Description

We are given a (finite) directed graph with:

1. A fixed set of vertices (but not edges).
2. One special vertex (the “root”).

We define the following predicates on vertices:

$$\begin{aligned} x.food &\equiv x \text{ is reachable from the root} \\ x.garbage &\equiv \neg x.food \end{aligned}$$

For example, shade in the vertices in Figure 11.1 that are food.

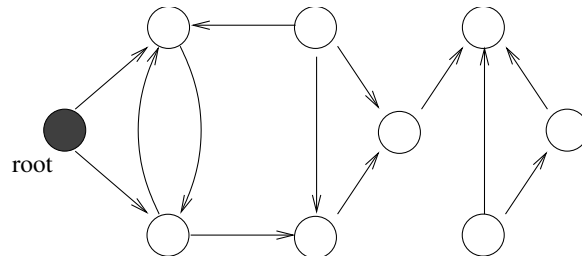


Figure 11.1: A directed graph with a designated root

A “mutator” process modifies the graph by adding and deleting edges. (Recall that the set of vertices is fixed.) The mutator makes these modifications subject to the constraint that an edge may be added only if it *points to food*.

Our task is to write a “marker” process that runs concurrently with the mutator and is responsible for marking the food. After it is done, we can be sure that *all* the food has been marked. Put another way, after it is done we know:

$$x \text{ is unmarked} \Rightarrow x \text{ is garbage}$$

Notice that this is implication and not equivalence. That is, the marker can be conservative and mark things that are not food. Why do we allow this?

While it is too much to require the marker to have marked only food when it is done, it is also too weak to require simply that the marker have marked *at least* all food when it is done. If this were the specification, an acceptable solution would simply mark all the vertices. Clearly such a solution is not interesting since we would like to collect as much garbage as possible (*i.e.*, mark as few extra, non-food, vertices as possible).

Which garbage vertices in particular is it reasonable to require the marker *not* to mark? The answer is the vertices that were _____. We call these vertices “manure”. We will settle for collecting manure (since this mark and collect procedure can be repeated to collect the garbage that was missed in the previous iteration.)

11.3 Application to Memory Storage

Although posed as an abstract problem on directed graphs, this problem has direct applicability to garbage collection for dynamically allocated memory. The vertices in the graph represent memory cells and the edges represent references (*i.e.*, one data structure contains pointers or references to another data structure). The root vertex is the segment header and it maintains references to the dynamically allocated structures as well as a “free store” list (*i.e.*, memory cells that are currently unused).

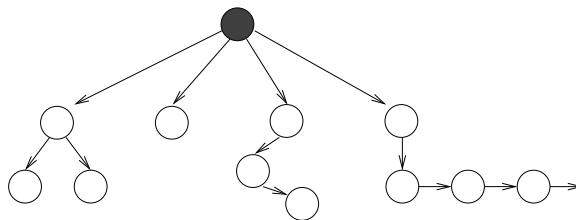


Figure 11.2: Heap of dynamically allocated storage

We want to recover useless cells and move them back to the free store list. We use the marker algorithm described above, constantly running in the background

to mark the things that are food and then everything else can be moved back to the free store list. This repeats as long as the mutator process (*i.e.*, the program) is running.

11.4 Relationship to Termination Detection

One obvious connection between the parallel garbage collection problem and the termination detection problem is that garbage collection relies on detecting that the marker process has terminated (*i.e.*, that it will not mark any more vertices).

Another, more subtle connection, is that the termination detection problem can be cast as an instance of the garbage collection problem. That is, given the task of detecting termination, we can transform the problem into one of garbage collection, then use a garbage collection scheme to answer the termination detection problem. This transformation is discussed in Tel and Mattern's 1993 TOPLAS paper ("The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes", *ACM TOPLAS*, 15(1), p.1-35).

Consider a reference-counting garbage collection scheme. Each vertex keeps track of the number of incoming pointers. When the mutator adds an edge, the reference count increases. When the mutator deletes an edge, the reference count decreases. When the reference count is 0, the vertex can be removed (*i.e.*, it is garbage).

To solve the termination detection problem, create a special node (call it a for "active"). Every active process has a reference to this node. Every time a message is sent, this reference is copied and sent in the message as well. Whenever a process goes idle, it deletes all its references to a . We can now assert that the system has terminated if and only if the reference count of a is 0.

11.5 Formal Definitions

We first introduce some notation for talking about the directed graph.

$$\begin{aligned} E[x, y] &\equiv \exists \text{ an edge from } x \text{ to } y \\ R[x, y] &\equiv \exists \text{ a path from } x \text{ to } y \end{aligned}$$

Mutator Program

Program	<i>Mutator</i>
var	x, y : vertices
initially	$(\forall x :: x.manure \equiv x.garbage)$
always	$(\forall x :: (x.food \equiv R[root, x]) \wedge (x.garbage \equiv \neg x.food))$
assign	$(\parallel x, y :: \begin{array}{l} add(x, y) \wedge y.food \longrightarrow E[x, y] := \mathbf{true} \\ \parallel del(x, y) \longrightarrow E[x, y] := \mathbf{false} \end{array})$

Derived Properties Which of the following are, in fact, properties of the mutator?

1. garbage remains garbage
2. food remains food
3. manure is garbage
4. food can be reached only from food
5. garbage can be reached only from garbage
6. manure can be reached only from manure

As a further exercise, try writing property #4 above as a formal quantification.

As another exercise, use the properties above to fill in Figure 11.3 with vertices and permitted edges.

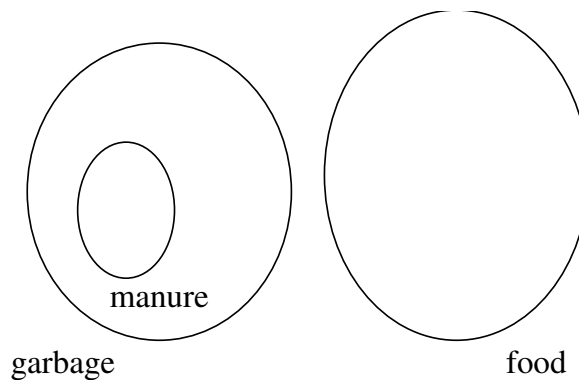


Figure 11.3: Edges between Food, Garbage, and Manure

Marker Program We begin by adding a “marked” field to each vertex (*i.e.*, $x.m$). We also add a boolean *over* that is true when the marker program has finished marking food. The specification of the marker is therefore:

safety: $\text{invariant}(\text{over} \Rightarrow \text{_____})$

progress: $\text{true} \leadsto \text{over}$

11.6 Principle of Superposition

The “Principle of Superposition” is a useful technique for structuring large algorithms. In particular, it allows us to layer one solution on top of another.

Given a program, we modify it by *adding* variables and actions in a disciplined way. In particular, the modification must conform to:

1. A new action does not affect any of the old variables, *and*
2. An existing action a can be modified to $a \parallel s$, where the new part (s) does not affect any of the old variables.

The reason that superpositioning is useful is that the new program “inherits” all of the properties of the old! That is, any property of the original program is a property of the new program as well!

In the case of the marker program, we will begin with the mutator and superimpose a new set of actions (called the “propagator”). We will then further superimpose more actions (called the “detector”). The resulting program will be the marker.

11.7 Propagator - First Attempt

Our first attempt at writing a marker is to begin with the root (and only the root) being marked and simply mark vertices that can be reached from vertices that are already marked. The mark spreads almost “gossip-like” through the directed graph, with the number of marked vertices getting larger and larger. That is, we add the following:

initially $(\forall x :: x.m \equiv (x = root))$
assign
 $(\parallel x, y :: x.m \wedge E[x, y] \longrightarrow \text{_____})$

These actions are superimposed with the mutator, so the resulting program still satisfies the behavior of a mutator. Does the resulting system, however, now satisfy the desired behavior of a marker? In other words, does this solution work for marking vertices?

The (perhaps surprising) answer is _____. This is because the actions of the mutator can work to frustrate the marker, preventing it from ever marking a particular food vertex. How can this happen?

To fix this difficulty, we have to do more than spread marks in a gossip-like fashion.

11.8 Propagator - Second Attempt

We modify the mutator propagator to the following program:

```

Program      Propagator
var           $x, y : \text{vertices}$ 
initially     $(\forall x :: x.m \equiv (x = \text{root}) )$ 
                $\wedge (\forall x :: x.\text{manure} \equiv x.\text{garbage} )$ 
always       $(\forall x :: (x.\text{food} \equiv R[\text{root}, x]) \wedge (x.\text{garbage} \equiv \neg x.\text{food}) )$ 
assign      (  $\parallel x, y ::$ 
                $\begin{array}{l} \text{add}(x, y) \wedge y.\text{food} \longrightarrow E[x, y], y.m := \text{true}, \text{true} \\ \parallel \text{del}(x, y) \longrightarrow E[x, y] := \text{false} \\ \parallel x.m \wedge E[x, y] \longrightarrow y.m := \text{true} \end{array}$ 
               )

```

Make sure you can identify the actions that were superimposed on the original mutator program to create this new version. Also make sure you can confirm that they conform with the requirement on superimposed actions.

Is this program correct?

11.9 Specification of Propagator

We begin with a definition:

$$ok[x, y] \equiv (x.m \wedge E[x, y] \Rightarrow y.m)$$

Put another way, there is only *one* way for a pair (x, y) to *not* be *ok*. Complete Figure 11.4 to illustrate the situation in which $\neg ok[x, y]$.



Figure 11.4: A pair of vertices for which $\neg ok[x, y]$

Now define a predicate that indicates the marker has terminated.

$$T \equiv \text{root}.m \wedge (\forall x, y :: ok[x, y])$$

The first claim is that the propagator program satisfies the following specification:

invariant. $(x.\text{manure} \Rightarrow \neg x.m)$

stable. T

true $\leadsto T$

The second claim is that this specification is a refinement of the original! That is, any program that satisfies this specification must satisfy the specification for the marker given at the beginning of this chapter.

11.10 Proof of Correctness

Only the proof of the first property is given here. That is, we show that the propagator program satisfies the property:

$$\mathbf{invariant}.(x.manure \Rightarrow \neg x.m)$$

Proof. The invariant (call it I) to be shown can be restated as:

$$(\forall z :: \neg z.manure \vee \neg z.m)$$

The initially part of the proof obligation can be discharged by examining the initially section of the program. (You should be able to complete this using basic predicate calculus.)

For the stable part, there are two actions that can modify $z.m$ and so must be considered.

- Action: $add(x, y) \wedge y.food \longrightarrow E[x, y], y.m := \mathbf{true}, \mathbf{true}$

$$\begin{aligned} & \{I\} \\ & \quad add(x, y) \wedge y.food \\ & \{I \wedge y.food\} \\ \Rightarrow & \\ & \{I \wedge \neg y.manure\} \\ & \quad E[x, y], y.m := \mathbf{true}, \mathbf{true} \\ & \{(\forall z : z \neq y : \neg z.manure \vee \neg z.m) \wedge \neg y.manure\} \\ \Rightarrow & \\ & \{I\} \end{aligned}$$

- Action: $x.m \wedge E[x, y] \longrightarrow y.m := \mathbf{true}$

$$\begin{aligned} & \{I\} \\ & \quad x.m \wedge E[x, y] \\ & \{I \wedge \neg x.manure \wedge E[x, y]\} \\ \Rightarrow & \\ & \{I \wedge \neg y.manure\} \\ & \quad y.m := \mathbf{true} \\ & \{I\} \end{aligned}$$

This concludes the proof. □

Chapter 12

Byzantine Agreement

12.1 References

This chapter presents three separate problems.

- The two-generals problem. This problem first appeared in a paper in 1978 by J.N. Gray [Gra78]. It is now a classic problem in networking and is treated in many standard textbooks, including Garg's [Gar02, chapter 24].
- Asynchronous consensus. The impossibility of achieving consensus in an asynchronous system with the possibility of a single failure is from Fischer, Lynch, and Paterson in a 1985 paper [FLP85]. It is also presented in Garg's book [Gar02, chapter 25] as well as Lynch's book "Distributed Algorithms" [Lyn96, chapter 12].
- Synchronous consensus with Byzantine faults. This problem was solved in [PSL80] and is presented in [Gar02, chapter 26].

12.2 Background: Two-Generals Problem

We begin with a classic problem in agreement.

Two armies are on opposite sides of a valley in which their common enemy lies. The generals of the two armies are initially either prepared to attack, or they wish to retreat. Any attack, however, must be coordinated with the other army. That is, the generals should decide to attack exactly when they are both initially willing to attack, otherwise they should decide to retreat. Unfortunately, the generals do not have a pre-arranged plan and can not communicate with each other directly. The only means of communication is by sending messengers through dangerous enemy territory. Because of the danger, there is no guarantee that a message will reach its destination.

This story is meant to model a situation where two principals are trying to agree on a single value, in this case "attack" or "retreat". The communication link between the principles, however, may experience intermittent failures.

Under these conditions, there is no deterministic algorithm for reaching agreement.

The proof is by contradiction. Assume that such an algorithm does exist and consider an execution of the algorithm that results in “attack” being decided (by both generals). Say this algorithm completes in r rounds. Call the generals P_1 and P_2 and assume that the last message received is by P_2 . Now consider the case where this last message is lost. Since P_1 has no way to know that the message is lost, P_1 must still decide on the same value. Hence P_2 must also decide on the same value. Repeat this argument again, for the new run of the algorithm, of length $r - 1$. Again, both generals must still decide on “attack”. This process of losing the last message can be repeated until the computation consists of no messages being exchanged. The two generals must still decide on “attack”. Such an algorithm is not correct since one of the generals might not be initially willing to attack.

12.3 Faults in Distributed Systems

There are many models for failures in distributed systems. The two-generals problem above considers *link failures*, in particular the case where messages can be lost. Other kinds of link failures include message duplication, message corruption, and message insertion.

In addition to links failing, one can also consider *process failures*. Some common examples of fault models include:

Fail-stop. A process fails by stopping execution. Other processes, however, are notified of this failure.

Crash. A process fails by stopping execution, and other processes are *not* notified. This is also known as fail-silent.

Byzantine. A failed process may perform arbitrary actions, including sending and receiving messages. Byzantine failure allows failed processes to act maliciously.

Designing algorithms to tolerate these faults can be difficult. Notice that Byzantine faults are strictly harder to tolerate than crash faults. Crash faults, in turn, are strictly harder to tolerate than fail-stop faults.

12.4 Binary Consensus

The two-generals problem is a specific instance of what is known as a consensus problem. Solving the consensus problem is difficult in the context of faults (*e.g.*, messages that are not delivered, as above). Its general formulation considers the case of n processes, each with an initial (binary) value v_i , a decision (binary) value d_i , and a completion (boolean) variable t_i . The initial value is constant.

constant. v_i

The completion variable represents the commitment of a process to a decision value: once true, the decision value can not be changed.

$$\mathbf{stable}.(t_i \wedge (d_i = 0)) \wedge \mathbf{stable}.(t_i \wedge (d_i = 1))$$

The requirements for binary consensus can be divided in three parts.

Agreement. Two correct processes can not commit to different decision variables.

$$(\forall i, j : t_i \wedge t_j : d_i = d_j)$$

Validity. If all initial values are equal, correct processes must decide on that value.

$$(\exists k :: (\forall i :: v_i = k)) \Rightarrow (\forall i : t_i : d_i = v_i)$$

Termination. The system eventually terminates.

$$\mathbf{true} \leadsto (\forall i :: t_i)$$

Which of these parts are safety properties and which are progress?

12.5 Asynchronous Consensus with Process Failures

Solving the consensus problem is hard in the presence of faults. Indeed, it has been shown that it is *impossible* to solve this problem in an asynchronous system in the presence of just 1 process crash failure! This result is disturbing for 3 reasons: (i) the asynchronous model is so pervasive for reasoning about and designing distributed systems, (ii) the fault class is so benign (*i.e.*, crash failures), and (iii) the number of faults needed to foil consensus is so small (*i.e.*, just 1). The paper establishing this famous impossibility result (published in 1985) is known as “FLP” after the initials of the authors, Fischer, Lynch, and Paterson.

Informally, the proof of correctness is based on the following two observations: (i) A correct protocol must begin in a state in which either result (0 or 1) is a possible decision value, and (ii) Any global state in which either result is possible permits a continuation of computation in which either result is still possible.

Consensus is a fundamental problem in distributed systems, arising in the context of database transactions (the commit/abort problem), group membership, leader election, and reliable multicast. For this reason, the FLP impossibility result has received considerable attention and has spawned a rich array of research. Many modifications to the model or assumptions made in FLP have been proposed. These modifications include:

Synchrony. No solution is possible in asynchronous networks—that is, networks in which there is no upper bound on message delay or differences in processing speeds. The consensus problem can, however, be solved in synchronous networks—that is, networks in which such bounds do exist.

Randomness. No deterministic solution to the consensus problem exists, but probabilistic algorithms do exist.

Failure detectors. Informally, a critical difficulty in achieving consensus in an asynchronous network is that it is impossible to distinguish between a crashed process and a slow one. Failure detectors are oracles that provide information about the (possible) failure of processes.

Approximation. The agreement condition (that all correct processes decide on the same value) can be weakened to approximate agreement: all correct processes decide on a value from a set of cardinality at most k .

Each of these subtopics is a research field in its own right. For example, there is an entire taxonomy of failure detectors (categorized according to the false positives and false negatives they can produce). In the rest of this chapter, we will examine consensus in synchronous networks.

12.6 Synchronous Agreement with Crash Faults

Consider a simple algorithm for agreement: every process broadcasts (to all other processes, including itself) its initial value v_i . In a synchronous network, this can be done in a single “round” of messages. After this round, each process decides on the minimum value it received.

If no faults occur, this algorithm is correct. In the presence of a crash fault, however, a problem can arise. In particular, a problem may occur if a process crashes _____ a round. When this happens, some processes may have received its (low) initial value, but others may not have.

To address this concern, consider the following simplifying assumption: say that at most 1 process can crash. How can the simple algorithm above be modified to handle such a failure? Answer: by using _____ rounds. In the first round, processes broadcast their own initial value. In the second round, processes broadcast _____. Each process then decides on the minimum value among all the sets of values it received in the second round.

Notice that if the one crash occurs during the first round, the second round ensures that all processes have the same set of values from which to decide. On the other hand, if the one crash occurs during the second round, the first round must have completed without a crash and hence all processes have the same set of values from which to decide.

The key observation is that if no crash occurs during a round, all processes have the same set of values from which to decide and they correctly decide on the same minimum value. Thus, to tolerate multiple crashes, say f , the

protocol is modified to have _____ rounds of synchronous communication. Of course, this requires knowing f , an upper bound on the number of possible crash faults.

12.7 Synchronous Agreement with Byzantine Faults

One behavior permitted by a byzantine failure is to simply stop executing actions; *i.e.*, precisely the behavior required by a crash failure. Therefore, Byzantine faults are strictly harder to tolerate than crash faults.

Indeed, with only 3 processes, it is not possible to tolerate a byzantine failure. Intuitively, P_1 sees two other processes, P_2 and P_3 , but which does it believe? Even adding a message from P_2 like: “ P_3 sent X ” does not help, since P_2 could be lying.

In general, there is no solution if the number of processes is 3 times the number of byzantine faults (or less). So, a lower bound on the number of processes required to tolerate f faults is _____. Perhaps surprisingly, this lower bound can be achieved.

Before presenting this solution, however, we consider a simpler version of the Byzantine agreement problem.

12.7.1 With Authenticated Signatures

Authenticated signatures allow processes to sign messages in such a way that they can not be forged by other processes (even byzantine ones). These signatures simplify the problem since, informally, they provide a way for P_2 to *convince* P_1 that P_3 did indeed send X by sending to P_1 the note it received, signed by P_3 . Since signatures can not be forged, P_1 then knows that P_3 did indeed send to P_2 message X and can compare that with what it received from P_3 . A discrepancy indicates that P_3 is faulty.

Consider the case where it is known *a priori* that at most 1 (byzantine) failure is possible. How can the algorithm used for crash failures be adapted to solve this problem? The key insight is that all signed messages sent by P_i can be compared to see if P_i is correct or faulty. So, the same algorithm, with two rounds, can be used simply by incorporating the comparison of signed messages and discarding the value proposed by a process if it did not make that same proposal to all other processes. This algorithm is summarized in Figure 12.1.

With crash failures, the informal argument of correctness was based on the fault occurring either in the first round or the second. With byzantine faults, the faulty process continues to execute so there may be faulty messages in both rounds. However, any faulty messages in the first round are recognized by all other processes as faulty after the second round, regardless of any message sent by the faulty process during the second round.

Now consider the case where more than 1 process can fail. In this case, 2 rounds do not suffice. (Why not?) In order to tolerate f faults, a *consensus tree* is constructed. In this tree, the nodes at level i represent message chains

For each process P_i :

- broadcast signed v_i to all other processes;
- broadcast set of received (signed) values;
- compare “received signed v_j ’s” for each P_j ;
- if they agree, include P_j , else exclude it;
- choose minimum v of included P_j ’s;

Figure 12.1: Byzantine Agreement with Authentication (Tolerates 1 Fault)

of length i . For example, a message chain of length 3 is a (signed) message from P_j containing a (signed) message from P_k containing a (signed) message from P_l . The root is level 0 (no message). It has N children, representing the messages received in the broadcast of the first round. Each of those nodes have $N - 1$ children, representing the messages received in the second round. Each of those nodes then have $N - 2$ children, and so on. See Figure 12.2, where a node labelled l, k, j represents a message chain of length 3 (from P_j containing message from P_k containing a message from P_l).

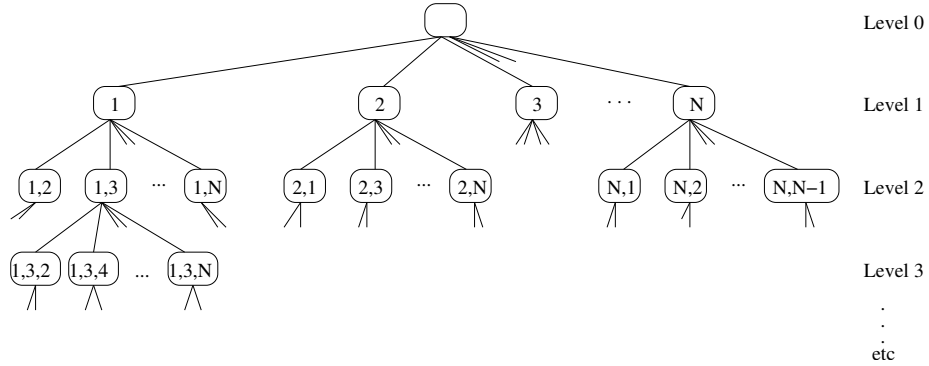


Figure 12.2: Consensus Tree of Message Chains

Each node in the tree is given a decision value, and these values bubble up the tree, from children to parents, until the root is assigned a value. The root's value is the decision value for the process. Consider the $N - 3$ children of some node (l, k, j) : $(l, k, j, 1), (l, k, j, 2), \dots, (l, k, j, N)$. When should the decision values of these children bubble up to their parent? Answer: when they _____. In this case, this process knows that P_j relayed the same information (regarding P_k 's message regarding P_l 's message) to all processes.

In order for this tree to calculate the same value for all processes, every message chain in the leaf nodes must include at least 1 non-faulty process. Thus, in order to tolerate f faults, the leaf nodes must represent message chains of length _____. That is, a consensus tree of height _____ is required.

12.7.2 Without Authenticated Signatures

Even when authenticated signatures are not available, a consensus tree can still be used to achieve Byzantine agreement. The key question to be answered, however, is when to bubble up a value from a set of children to the parent. A natural answer is to use simple majority. Indeed, if there are “enough” processes, the majority will be nonfaulty and the correct value will be bubbled up.

The following observation is central to the correctness of this algorithm: Consensus tree leaves that *end* in a non-faulty process have the same value among all processes in the system. This is because a non-faulty process sends the same message to all other processes, in every round. Thus, if there are f possible faults, a leaf must have $2f$ siblings, for a total size of the group of siblings of $2f + 1$. Since each sibling leaf in the group ends in a different process (and at most f are faulty) the majority from each group will be a correct value.

How many processes are needed in total? Recall that $f + 1$ rounds are used. That is, the leaves of the consensus tree are at level $f + 1$. From the structure of this tree, a node at level i has $N - i$ children. Thus, parents of leaves are at level f and hence have $N - f$ children. As seen above, this number of children is actually $2f + 1$. Therefore, $N = 3f + 1$.

Chapter 13

Discrete-Event Simulation

13.1 References

1. An excellent survey of the field, written in 1990, is given in the article “Parallel Discrete Event Simulation” by R. Fujimoto in *CACM*, 33(10), p. 30–53 [Fuj90].
2. Chapter 10 of the Barbosa textbook (“An Introduction to Distributed Algorithms”) also gives a nice introduction to the area [Bar96].
3. The original work on the optimistic simulation algorithm known as “Time Warp” appeared in the article “Virtual Time” by David Jefferson in *TOPLAS*, 7(3), p. 404–425, 1985 [Jef85].

13.2 Background: Sequential DES

We are given a network consisting of nodes and directed links between the nodes. There are 3 categories of nodes: sources, servers, and sinks. By convention each kind of node is represented by a different kind of symbol. In Figure 13.1 a source is shown with a link to a server which in turn has a link to a sink.

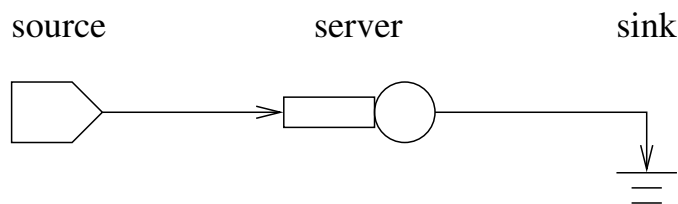


Figure 13.1: Three basic kinds of nodes

Sources generate “clients” (or “jobs”) that travel through this network along the links. The clients obtain service at the server nodes. Servers can provide service to at most a single client at a time.¹ If a client arrives at a server that is already busy with another client, it waits in a queue associated with that server. Finally, clients disappear from the system at sinks.

The movement of clients along links is considered to be instantaneous. Service, however, does require time, in addition to any time spent waiting in the associated queue. The time a client spends in the system, therefore, is the sum of the times it spent being serviced and the times it spent waiting in a queue.

For example, Figure 13.2 illustrates a system in which clients must speak with a receptionist before they are placed in one of two different queues for service. After service, the clients may have to move to the other queue, or may leave the system. In general, there can be several sources and sinks and servers with various properties.

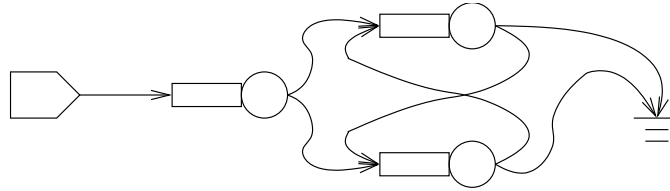


Figure 13.2: A simple simulation system with three servers

A defining aspect of discrete-event simulation is that the behavior of the system over time can be completely captured by a series of discrete events over time. In particular, there are four possible kinds of events. A client may:

1. arrive in the system,
2. arrive at a server,
3. complete service, *or*
4. leave the system.

With a precise-enough clock, we can assume that no two events happen at the same time. The behavior is therefore given by a list of time-stamped events in increasing order of time. From this list, all statistics of interest can be recovered (*e.g.*, the average length of a queue, the maximum number of clients in the system, the average time spent in the system, *etc.*).

For simple systems, closed-form analytical solutions exist for determining these statistics. For example, for a single server and single queue, Little’s law relates the average service and arrival rates to the expected queue length. For more complex systems, however, such analytical solutions can be very difficult.

¹There are many generalizations and characterizations of server behavior. We consider only a single, simple case here.

These systems can be simulated to gather the relevant statistics. In this way, the systems can be tuned to optimize various performance measures of interest. Discrete-event simulation is used to model many real-world situations such as: banks with tellers and clients; hospitals with doctors, operating rooms, nurses, and patients; and computer systems with resources and processes.

Systems are typically modeled as being “memoryless”. That is, the time taken for one thing does not depend on how long was taken to perform previous things. For example, the service time a customer experiences at a teller does not depend on the service time the previous customer experienced at that teller. Also, the time until the next customer arrives at the bank does not depend on when the previous customer arrived. This assumption reflects a view that customers are independent entities.

13.3 A Sequential Algorithm

The fundamental data structure for discrete-event simulation is an *event queue*. This queue maintains a list of scheduled future events in increasing order of time.

The basic algorithm consists of an iteration of the following steps:

1. Remove the first event from the head of the event queue.
2. Simulate this event (this may generate new future events).
3. Insert any new events generated in the previous step into the event queue.

The “current time” in the algorithm moves in discrete steps (that depend on the time stamps of dequeued events).

One subtlety in this algorithm is the way in which clients are generated at sources. This is done by “seeding” the event queue with an initial arrival event. Now simulating this arrival event not only involves inserting the new client in the system (sending them to their first node in the system) but *also* involves scheduling the next (future) arrival.

For example, consider the simple system given in Figure 13.2. Initially, the event queue contains a single event: $\langle \text{arrival } c_1, 1 \rangle$. This event is dequeued (the “current time” is now 1) and simulated. Simulating this event sends customer c_1 to a server, where it begins service immediately. This results in a future event being scheduled: the completion of service of c_1 at this server. Say this event is $\langle \text{complete } c_1, 10 \rangle$. A second event is generated: The arrival of the next customer. Say this event is $\langle \text{arrival } c_2, 4 \rangle$. Both of these events are inserted in the (now empty) event queue, in increasing order of time. This loop repeats.

Notice that the single event queue is a bottleneck. Events must wait for earlier events to be simulated, even if they are completely independent! We would like to remove this bottleneck and parallelize (and distribute) this algorithm for discrete-event simulation.

13.4 Time-Driven Simulation

As an aside, there is another class of simulation algorithms known as time-driven simulations. In discrete-event simulation, the absolute *sizes* of the time intervals between events do not affect how quickly the simulation can proceed. Only the number of events matters. This view, however, is not appropriate for continuous systems that are constantly evolving.

For example, consider a billiard table. At any given instant in time, the state of the system is given by the position and velocity of each of the balls. The next state occurs at the next instant of time, where process positions have changed. If we attempt to “discretize” the state changes by considering only the collisions, we see that it is very difficult to schedule the next event.

Instead, a different approach is taken. The simulation is advanced by small increments in physical *time*. The state of the billiard table is calculated at some time t . It is then evolved to the new state at time, say, $t + 100ms$. We must then determine whether there were any collisions in that time interval. The time interval by which the simulation advances must be chosen carefully: too large and it may be difficult to detect events of interest, too small and the simulation proceeds very slowly.

Also, notice that with time-driven simulations, the computational complexity is proportional to the amount of physical time being simulated (and the granularity of the time steps).

13.5 Conservative Approach

Consider each source and server in the simulation system to be a separate process. Clients are passed as messages between the processes. We assume FIFO channels, which guarantees that messages from a given process arrive in increasing order. Each process maintains a single event queue (representing the future events scheduled to occur at that process).

Question: When is it “safe” to process an event at the head of this queue?

Answer: When we know that _____.

13.5.1 A Naive Algorithm

A naive attempt to implement such an algorithm is to allow a process to simulate the first event in its event queue only if it has received at least one event from each of its in-neighbors. This condition guarantees that it will not receive an event time stamped earlier than the event currently at the head (and therefore it is safe to simulate this event). If a process does not have at least one event in its queue from every in-neighbor, it simply waits.

This algorithm is not correct, however. In particular, deadlock is possible. Consider the topology given in Figure 13.3.

Initially, there is a single event generated by the source (*i.e.*, the arrival of a single customer). Server s_1 cannot process this event, however, since it does

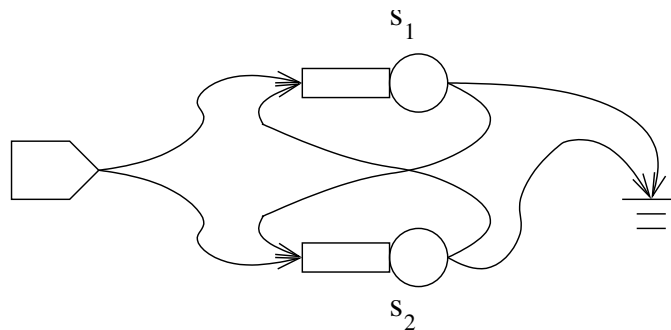


Figure 13.3: Possible deadlock with naive algorithm

not have an event from s_2 . Similarly, s_2 cannot process any events because its queue is empty. Therefore, no events can be simulated and the simulation does not make progress.

13.5.2 A Correct Algorithm

The difficulty encountered above is similar to the issue encountered in Lamport's time-based mutual exclusion scheme. In order to advance, there must be a message (event) from every neighbor, so we must guarantee that such events arrive. When a node generates an event and sends it to one of its out-neighbors, it updates the time only at that one destination node.

The solution, then, is to send update events to *all* out-neighbors. These "update" messages that do not represent actual events are called "null events".

Now consider the time stamp to use for null events. If a node simulates an event (*e.g.*, customer c_1 completes service) time stamped 23, this could cause the transmission of a message (*i.e.*, the client) to the next server in the network. This event (arrival of c_1) would be time stamped _____. What time stamp should be used for the null events sent to the other out-neighbors? Answer: _____.

Unfortunately, the intuitive answer above does not solve the deadlock problem. Consider a cycle of empty queues (*i.e.*, containing only null events). For each process in the cycle, the empty queue could contain the minimum time stamp. Because processing a null event does not advance the clock, this situation is a deadlock.

To prevent such cycles, null events must be given strictly *larger* time stamps than the current time at that process. For example, if the service time at a process was known be at least 5 minutes, a null event for the example given above could be generated with time stamp _____. This null event would be sent to *all* out-neighbors.

The key idea behind null events, then, is that they represent a promise to that neighbor to not generate an event with an earlier time stamp. The size

of the increment of null messages above the current time is known as the *look-ahead*. In order for this algorithm to be correct, processes must have positive (*i.e.*, strictly greater than 0) look-ahead.

13.6 Extensions

Several refinements of this basic algorithm have been developed, published, and implemented. Some of these are listed below.

1. **Pre-compute service times.** If the service time is independent of the particulars of the clients, the length of time required for the next service request can be calculated before the next client has even arrived. This service time becomes the look-ahead. It is guaranteed to be no less than the minimum and so can only give larger look-aheads than the original version of the algorithm.
2. **Request time-updates explicitly.** Rather than sending null events as the default behavior, processes can explicitly request null events (*i.e.*, updated time information) when needed. That is, when a queue for a particular in-neighbor becomes empty, that neighbor is queried for a null event explicitly.
3. **Allow deadlock.** Yet another approach is to not use null messages at all. Rather, the system is allowed to deadlock! One of the standard deadlock detection and recovery schemes is used to ensure that the computation is continued eventually.
4. **Increased synchronization.** A tighter synchronization between processes can be used to ensure that no process gets too far “ahead” of any other. There is a window of time in which events are “safe” (*i.e.*, can be executed unconditionally). This variant must ensure that this safe window advances.

All of these variants have different strengths that become apparent under different experimental conditions. In general, the performance of this style of conservative discrete-event simulation is highly dependent on the magnitude of the look-ahead. The magnitude of the look-ahead, in turn, is highly application-specific.

13.7 Optimistic Approach

The conservative approaches above are based on allowing only *safe* events (*i.e.*, events that cannot be affected by the arrival of future events) to be executed. A different class of algorithms, known as *optimistic* algorithms relaxes this constraint. Processes simulate the events in their queues optimistically, in the hope that no events with an earlier time stamp will arrive at some point in the future.

If an event with an earlier time stamp does arrive, the process must undo part of the simulation! This is known as roll-back.

A seminal paper on optimistic simulation appeared in 1985 (“Virtual Time” by Jefferson, in *TOPLAS* 7(3)) and introduced the algorithm known as *time warp*. This name stems from the fact that the algorithm allows time to flow backwards.

With optimistic simulation algorithms, messages can arrive with time stamps that are earlier than the current logical time. Such a message is known as a *straggler*. The main issue in these algorithms is how to deal with stragglers.

To address the problem of stragglers, a (time stamped) history of past states is maintained. When a straggler arrives with a time stamp of t , the process recovers from this history the *last valid state before t* . See Figure 13.4.

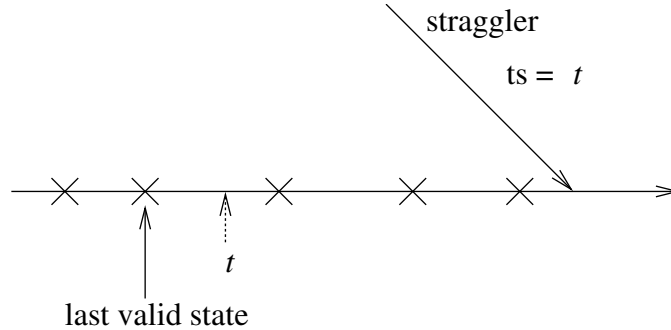


Figure 13.4: Roll-back caused by the arrival of a straggler

There is, however, a complication: This process may have already sent events to *other* processes after t ! These events must be undone. This is accomplished by sending an *anti-event*. The affect of an anti-event is to cancel the corresponding event.

There are two cases for the arrival of an anti-event. It could arrive with a time stamp greater than the current local time. In this case, the corresponding event to be cancelled has not yet been simulated. The corresponding event can therefore be deleted from the event queue. The second case is that the anti-event arrives with a time stamp *less* than the current local time. In this case, the anti-event is itself a straggler! The process receiving this straggler needs to roll-back, potentially causing other anti-events to be sent...

Clearly, we must convince ourselves that this series of cascading roll-backs can not continue indefinitely, thus preventing the simulation from advancing. The proof of correctness is based on the idea of *Global Virtual Time* (GVT). The GVT is the minimum time stamp in the system. We can show that:

- the algorithm never rolls back to before GVT, *and*
- the event scheduled at time GVT can be executed without causing a roll-back.

13.8 Extension: Lazy Cancellation

As an optimization, notice that—sometimes—incorrect simulations can produce the correct result! That is, even though a process proceeded optimistically on an assumption that later turned out to be false, it may be that the events it scheduled happen regardless! For example, in Figure 13.5 the event labeled e_1 is scheduled again, after the process rolls back to before time t .

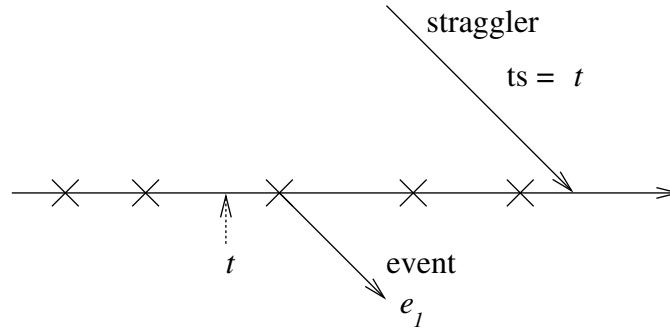


Figure 13.5: Lazy cancellation of optimistically generated events

In the lazy cancellation optimization, after a process rolls back, it resumes the simulation without sending anti-events. An anti-event is sent only when it is discovered that an anti-event is necessary (because the optimistic simulation generated an event that is not generated by the new simulation)

13.9 Summary of Key Ideas

1. Discrete-event simulation is based on an *event queue* of scheduled events that is kept sorted in increasing time. The first event in this queue is safe to simulate.
2. Conservative approaches to distributed discrete-event simulation avoid making mistakes. That is, an event is simulated only if it is *guaranteed* to be safe. *Null events* are used to guarantee no events will be generated up to a future time and positive *look-ahead* ensures there are no cycles.
3. Optimistic approaches occasionally make mistakes, simulating an event that must later be undone. These algorithms can receive *stragglers*, causing *roll-back*. The correctness of the Time Warp algorithm follows from the fact that GVT advances.

Appendix A

Order of Operations

The operators used in these notes are listed here in decreasing order of binding power.

- $.$ (function application)
- \neg (logical negation)
- $*$ / (arithmetic multiplication and division)
- $+$ $-$ (arithmetic addition and subtraction)
- $<$ $>$ \leq \geq $=$ \neq (arithmetic comparison)
- \wedge \vee (logical and and or)
- **next** **unless** **ensures** \leadsto (temporal operators)
- \Rightarrow \Leftarrow (logical implication and explication)
- \equiv $\not\equiv$ (logical equivalents and discrepance)

Bibliography

- [AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
- [Bar96] Valmir C. Barbosa. *An Introduction to Distributed Algorithms*. The MIT Press, Cambridge, Massachusetts, 1996.
- [CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.
- [DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
- [DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, 1990.
- [Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11 Australian Computer Science Conference*, pages 55–66, 1988.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [Fuj90] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.
- [Gar02] Vijay K. Garg. *Elements of Distributed Computing*. John Wiley & Sons, 2002.

- [Gra78] Jim Gray. Notes on data base operating systems. In Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors, *Operating Systems, An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer, 1978.
- [GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
- [Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., San Francisco, California, 1996.
- [Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.
- [Mis01] Jayadev Misra. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Monographs in Computer Science. Springer-Verlag, New York, New York, 2001.
- [PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.
- [SS94] Mukesh Singhal and Niranjana G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.