Chapter 5
Asynchronous Compositions of Programs

Jayadev Misra[1]
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

April 5, 1994

[1]This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658–219 and by the National Science Foundation Award CCR–9111912.

Contents

5.1 Introduction

A founding principle of software engineering is that large systems should be built out of components linked together in some fashion. We begin our study of program composition by considering the *union* operator (written as

$\|$); this operator is used for compositions of asynchronously operating components. In subsequent chapters we consider other operators for program composition.

A central goal of program composition theory is to answer questions of the following form:

- Given a program consisting of one or more components, how do we deduce the properties of the program from the specifications of its components?
- Given the specification of a program, how do we partition its design among its components?
- When does a program "inherit" a property of one of its components?

The first question is fundamental for program analysis. The second question is identical to the first except that it involves a design question: *how* to partition a specification. We ignore the heuristic aspects of design in this chapter. Therefore, our answer to the first two questions is given by inference rules that permit the deductions of program properties from those of its components; these rules can also provide guidelines for postulating specifications of components from a program specification. The third question is motivated by the possibility of adopting the following "ideal" methodology for program design: Partition the specification of the program (to be designed) so that each part in the partition is the specification of a component. If each component meets its specification and the program inherits the properties of its components then the program meets its specification. Each component can then be designed in a similar manner until the specifications are so trivial that they can be implemented directly. Such a methodology, though attractive, poses a number of formidable technical challenges. We develop a theory in Chapter 6 that qualifies the kinds of properties that can be so inherited.

We have been intentionally vague about what constitutes a component of a system. Traditionally, a component represents a function or procedure which, upon invocation, returns a result; or, it includes the representation of a data object and the methods by which the representation may be manipulated; or, it is a process that carries out a portion of the overall computation, autonomously. We make no specific assumptions about the nature of a component; therefore, our theory includes each of these interpretations as a special case. We call each component a *program* and we expect the programs to operate concurrently on global as well as their local data. Formal definition of union is given later in this chapter.

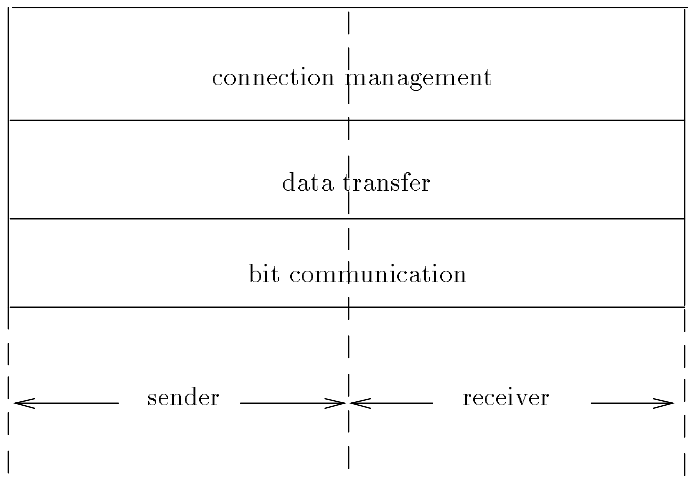


Figure 5.1: Interface between F, G and sem. The label on an edge is the value to which the shared variable is set by the corresponding program.

A common misconception in program structuring is that a *process*—whose code can be executed on a single processor, or that can be viewed as a unit of computation, as in a transaction processing system, for instance—constitutes a "natural" decomposition of a system; therefore, it is argued that a system should be understood (i.e., specified) process by process. In many cases, it is best to understand a system in a manner orthogonal to the way it is structured as a set of processes. As an example, Figure 5.1 shows a communication protocol structured as a hierarchy of three components: bit communication, data transfer and connection management. Each component addresses a specific concern in the given application program. Now, there are two processes—a sender and a receiver—in this system. Each of the components is partitioned between the sender and the receiver. However, the protocol is best understood component by component, not process by process. It is easier to apply the theory if the interactions among the components are "loose" in the sense that only a few variables are shared, and there is a strict discipline in the manner in which the shared variables are accessed (read or written). In such cases, concise specifications can be written for the components. In a later chapter, we propose a definition of loose-coupling. As an example of "tight" coupling, a mutual exclusion algorithm, typically, includes processes whose interactions cannot be expressed succinctly, process by process. It is possible, though not useful, to design or verify such systems by considering one process at a time.

5.2 Composition by Union

An informal description of union of programs F, G—denoted by $F \parallel G$—is as follows; $F \parallel G$ is a program in which

- initially, the initial conditions of both F, G hold,
- an execution is an interleaving of actions from F and G, and
- if at any point an action of F (or G) is ready to execute then an action of F (or G) executes eventually.

Union may be applied to any finite set of programs; for technical reasons, we require that this set of programs be nonempty.

Restriction: Throughout this chapter, we assume weak fairness in program execution. □

5.2.1 Definition of Union

A formal definition of union depends on how the component programs—F, G—are specified. We define union of Unity programs.

For program composition, it is particularly important to specify which variables named in a program are *local* to that program, i.e., those that can neither be read nor written by any other program; the remaining variables named in that program are *global* (i.e., they may, possibly, be read or written by other programs). We will prefix the declaration of a variable in a program by *local* or *global*.

Note on Name Clashes: Local variables of different programs may have identical names. Similarly, a local variable of one program may have the same name as a global variable declared in another program. It simplifies matters to assume that the local variable of different programs are renamed, if necessary, to avoid such name clashes. □

The local (global) variables of $F \parallel G$ are the ones that are declared local (global) in F or in G. We require that the declarations of the same global variable in two different programs be not conflicting, e.g., if global variable x is declared integer in F, it is either undeclared (and hence, not accessed) in G or declared integer in G.

Similarly, the initial conditions of F, G should not conflict in order for their union to be defined. The initial condition of $F \parallel G$ is the conjunction of the initial conditions of F and G. (It is useful for theoretical purposes to define $F \, [\!] \, G$ even when the initial conditions of F, G are conflicting; we do not pursue this possibility for the moment.)

The body of $F \, [\!] \, G$ consists of all the actions from F and from G.

Example: We show two programs—F, G—and their union.

Program F
  declare
    global b: boolean,
    local nf: integer
  initially $nf, b = 0, true$
  assign
    $nf, b \;:=\; nf + 1, false \quad \text{if} \quad b$
end $\{F\}$

Program G
  declare
    global b: boolean,
    local ng: integer
  initially $ng, b = 0, true$
  assign
    $ng, b \;:=\; ng + 1, true \quad \text{if} \quad \neg b$
end $\{G\}$

Program $F \, [\!] \, G$
  declare
    global b: boolean,
    local nf, ng: integer
  initially $nf, ng, b = 0, 0, true$
  assign
    $nf, b \;:=\; nf + 1, false \quad \text{if} \quad b$
  $[\!] \; ng, b \;:=\; ng + 1, true \quad \text{if} \quad \neg b$
end $\{F \, [\!] \, G\}$ □

Notation: So far, we have been writing properties such as p **co** q, p transient, p **en** q or $p \mapsto q$, fixed point predicates (FP) and initial condition (IC) without explicit mention of the program to which these apply. This is because we always had a single program under consideration. Now, we will be explicit with program names; we write, for instance,

$$p \ \textbf{co} \ q \quad \text{in } F$$

to denote that p **co** q holds in program F and

$$F.FP \text{ and } F.IC$$

to denote the fixed point predicate and the initial condition, respectively, of program F. □

5.2.2 Locality Axiom

A program can modify only the variables declared in it; this is the essence of the locality axiom, given below. We write "x outside F" to mean that x is not declared (as a local or a global variable) in F.

(Locality) $$\frac{x \text{ outside } F}{x \text{ constant in } F}$$

Recall, from the constant formation rule (Section 3.3.2), that any expression consisting of constants is constant. Therefore, an expression that names no variable declared in F is constant in F. In particular, an expression of free variables is constant.

5.2.3 Hierarchical Program Structures

From the definition of union, each global variable of F and of G is global in $F \parallel G$. It is often necessary to declare that a variable shared between F, G cannot be accessed by any other program. For instance, if F and G communicate by messages then the channels between them are local to any system of which both F, G are components. For the example given in Section 5.2.1, we may wish to specify that b cannot be accessed by any program other than F and G.

We propose a notation that permits us to hide variables. It also allows hierarchical program construction using union. Consider the following example.

Program H
  declare global x: integer,
          local y: integer
  Program F
  declare global x, y: integer,
          local z: integer
  $\langle\langle$body of $F\rangle\rangle$
  end $\{F\}$
    declare global x, y: integer,
  $\parallel$ **Program** G
    declare global x, y: integer,
            local u: integer
    $\langle\langle$body of $G\rangle\rangle$
    end $\{G\}$
end $\{H\}$

Here H is $F \ [] \ G$ with the restriction that the set of local *and* global variables of H is the set of global variables of $F \ [] \ G$. Thus, variable x could be accessed by programs other than F, G whereas y cannot be accessed outside H. Additionally, we allow H to initialize its local and global variables as long as it does not conflict with the initial condition of $F \ [] \ G$.

Properties of H can be deduced from the properties of $F \ [] \ G$; any property of $F \ [] \ G$ that names only its global variables, i.e., the (local and global) variables of H, is a property of H. Furthermore, we may use the initial condition of H to deduce additional properties, such as invariants, that do not hold in $F \ [] \ G$.

The declaration mechanism we have proposed hides variables. Hence, it can be used for hierarchical program constructions. For instance, programs F, G may themselves hide certain variables of the components out of which they are built. In this form of hierarchy, only the access-rights are constrained; unlike sequential program-hierarchy, flow of control is unaffected.

5.2.4 Union Theorem

Fundamental to our study of union is the union theorem. It permits us to deduce certain properties of a system from those of its components. The union theorem can be proven only when we have a formal definition of union, as is the case for the union of Unity programs. For programs expressed in other notations, the union theorem may be regarded as a set of axioms defining the meaning of asynchronous composition.

We give the theorem for the union of a nonempty finite set of programs, $\langle [] \ i \ :: \ F_i \rangle$ (index i assumes values from a nonempty finite set).

Theorem (Union Theorem)

1. $\langle [] \ i \ :: \ F_i \rangle.IC = \langle \forall \ i \ :: \ F_i.IC \rangle$
   $\langle [] \ i \ :: \ F_i \rangle.FP = \langle \forall \ i \ :: \ F_i.FP \rangle$

2. p **co** q in $\langle [] \ i \ :: \ F_i \rangle = \langle \forall \ i \ :: \ p$ **co** q in $F_i \rangle$

3. p transient in $\langle [] \ i \ :: \ F_i \rangle = \langle \exists \ i \ :: \ p$ transient in $F_i \rangle$ □

Condition (1) gives a method to compute the initial conditions and fixed point predicates of a system from those of its components. From condition (2), a **co**-property of a system is a **co**-property of each component and vice versa. Condition (3) is the counterpart of condition (2) for transient predicates.

Corollaries In the following, corollaries (2,3,4) are the counterparts of the union theorem for the various special cases. Observe that we have an equivalence (rather than an implication) in each case. Corollaries (5,7) show that certain properties of a component—**co**, **en**—are inherited by a system provided the predicate in the lhs is stable in the other components; a similar rule applies to invariant (Corollary 6).

1. $$\frac{\begin{array}{c} p \ \mathbf{co}\ q \quad \text{in } F \\ p' \ \mathbf{co}\ q' \quad \text{in } G \end{array}}{p \ \wedge\ p' \ \mathbf{co}\ q \ \vee\ q' \quad \text{in } F \ [\!]\ G}$$

2. $$p \text{ stable} \quad \text{in } \langle [\!]\ i \ :: \ F_i \rangle = \langle \forall\ i \ :: \ p \text{ stable} \quad \text{in } F_i \rangle$$

3. $$e \text{ constant} \quad \text{in } \langle [\!]\ i \ :: \ F_i \rangle = \langle \forall\ i \ :: \ e \text{ constant} \quad \text{in } F_i \rangle$$

4. $$\begin{array}{l} p \ \mathbf{en}\ q \quad \text{in } \langle [\!]\ i \ :: \ F_i \rangle = \langle \forall\ i \ :: \ p \ \wedge\ \neg q \ \mathbf{co}\ p \ \vee\ q \quad \text{in } F_i \rangle \ \wedge \\ \qquad\qquad\qquad\qquad\qquad\quad \langle \exists\ i \ :: \ p \ \mathbf{en}\ q \quad \text{in } F_i \rangle \end{array}$$

5. $$\frac{\begin{array}{c} p \text{ stable} \quad \text{in } F \\ p \ \mathbf{co}\ q \quad \text{in } G \end{array}}{p \ \mathbf{co}\ q \quad \text{in } F \ [\!]\ G}$$

6. $$\frac{\begin{array}{c} p \text{ stable} \quad \text{in } F \\ p \text{ invariant} \quad \text{in } G \end{array}}{p \text{ invariant} \quad \text{in } F \ [\!]\ G}$$

7. $$\frac{\begin{array}{c} p \text{ stable} \quad \text{in } F \\ p \ \mathbf{en}\ q \quad \text{in } G \end{array}}{p \ \mathbf{en}\ q \quad \text{in } F \ [\!]\ G}$$

Proof of the Union Theorem and Its Corollaries

Proof of the Union Theorem

part (1) The predicate $\langle [\!] \; i \; :: \; F_i \rangle .IC$, the initial condition of $\langle [\!] \; i \; :: \; F_i \rangle$, is, by definition, the conjunction of the initial conditions of the components, F_is. Similarly,

$\langle [\!] \; i \; :: \; F_i \rangle .FP$
$\equiv$ {Let $s.FP$, for an assignment statement s, be the predicate obtained by replacing ":=" by "=" in s. From the definition of FP}
$\langle \forall \; s \; : \; s \; \in \; \langle [\!] \; i \; :: \; F_i \rangle \; : \; s.FP \rangle$
$\equiv$ {The set of statements in $\langle [\!] \; i \; :: \; F_i \rangle$ is the union of the statements in the components, i.e., F_is}
$\langle \forall \; i \; :: \; \langle \forall \; s \; : \; s \; \in \; F_i \; : \; s.FP \rangle \; \rangle$
$\equiv$ {using the definition of $F_i.FP$}
$\langle \forall \; i \; :: \; F_i.FP \rangle$

part (2)

p **co** q in $\langle [\!] \; i \; :: \; F_i \rangle$
$\equiv$ {definition of **co**}
$\langle \forall \; s \; : \; s \; \in \; \langle [\!] \; i \; :: \; F_i \rangle \; : \; \{p\} \;\; s \;\; \{q\} \rangle$
$\equiv$ {The set of statements in $\langle [\!] \; i \; :: \; F_i \rangle$ is the union of the statements in the components, i.e., F_is}
$\langle \forall \; i \; :: \; \langle \forall \; s \; : \; s \; \in \; F_i \; : \; \{p\} \;\; s \;\; \{q\} \rangle$
$\equiv$ {definition of **co**}
$\langle \forall \; i \; :: \; p$ **co** q in $F_i \rangle$

part (3)

p transient in $\langle [\!] \; i \; :: \; F_i \rangle$
$\equiv$ {definition of transient}
$\langle \exists \; s \; : \; s \; \in \; \langle [\!] \; i \; :: \; F_i \rangle \; : \; \{p\} \;\; s \;\; \{\neg p\} \rangle$
$\equiv$ {Statements in $\langle [\!] \; i \; :: \; F_i \rangle$ is the union of the statements in the components, i.e., F_is.}
$\langle \exists \; i \; :: \; \langle \exists \; s \; : \; s \; \in \; F_i \; : \; \{p\} \;\; s \;\; \{\neg p\} \rangle \; \rangle$
$\equiv$ {definition of transient}
$\langle \exists \; i \; :: \; p$ transient in $F_i \rangle$

Proofs of the Corollaries

1. p **co** q in F , premise
   $p \wedge p'$ **co** $q \vee q'$ in F , strengthening the lhs and weakening the rhs
   $p \wedge p'$ **co** $q \vee q'$ in G , similarly
   $p \wedge p'$ **co** $q \vee q'$ in $F \; [\!] \; G$, union theorem (part 2) □

2. Set q to p in part (2) of the union theorem. $\Box$

3. e constant in $\langle [\!] \; i \;\; :: \;\; F_i \rangle$
$= \;\; \{$definition of constant$\}$
$\langle \forall \; m \;\; :: \;\; e = m$ stable in $\langle [\!] \; i \;\; :: \;\; F_i \rangle \; \rangle$
$= \;\; \{$Corollary 2$\}$
$\langle \forall \; m \;\; :: \;\; \langle \forall \; i \;\; :: \;\; e = m$ stable in $F_i \rangle \; \rangle$
$= \;\; \{$predicate calculus$\}$
$\langle \forall \; i \;\; :: \;\; \langle \forall \; m \;\; :: \;\; e = m$ stable in $F_i \rangle \; \rangle$
$= \;\; \{$definition of constant$\}$
$\langle \forall \; i \;\; :: \;\; e$ constant in $F_i \rangle$ $\Box$

4. p **en** q in $\langle [\!] \; i \;\; :: \;\; F_i \rangle$
$= \;\; \{$definition of **en**$\}$
$[p \wedge \neg q$ **co** $p \vee q$ in $\langle [\!] \; i \;\; :: \;\; F_i \rangle] \wedge$
$[p \wedge \neg q$ transient in $\langle [\!] \; i \;\; :: \;\; F_i \rangle]$
$= \;\; \{$union theorem, parts 2,3$\}$
$\langle \forall \; i \;\; :: \;\; p \wedge \neg q$ **co** $p \vee q$ in $F_i \rangle \wedge$
$\langle \exists \; i \;\; :: \;\; p \wedge \neg q$ transient in $F_i \rangle$
$= \;\; \{$definition of **en**$\}$
$\langle \exists \; i \;\; :: \;\; p$ **en** q in $F_i \rangle \wedge \langle \forall \; i \;\; :: \;\; p \wedge \neg q$ **co** $p \vee q$ in $F_i \rangle$ $\Box$

5. p **co** p in F , premise
p **co** q in G , premise
p **co** $p \vee q$ in $F \; [\!] \; G$, Corollary (1)
p **co** q in $F \; [\!] \; G$, $p \Rightarrow q$ from p **co** q in G $\Box$

6. p invariant in G , premise
$G.IC \Rightarrow p$, definition of invariant
$(F \; [\!] \; G).IC \Rightarrow G.IC$, part (1) of the union theorem
$(F \; [\!] \; G).IC \Rightarrow p$, from the above two

Also,
p stable in G , from the premise, p invariant in G
p stable in F , premise
p stable in $F \; [\!] \; G$, from Corollary (2)

Hence, from $(F \; [\!] \; G).IC \Rightarrow p$ and p stable in $F \; [\!] \; G$,
p invariant in $F \; [\!] \; G$ $\Box$

7. p **co** p in F , premise

$p \wedge \neg q$ **co** $p \vee q$ in G	, from p **en** q in G	
$p \wedge \neg q$ **co** $p \vee q$ in $F \, [\!]\, G$	, Corollary 1 on the above two	(*)
$p \wedge \neg q$ transient in G	, from p **en** q in G	
$p \wedge \neg q$ transient in $F \, [\!]\, G$	, union theorem (part 3)	
p **en** q in $F \, [\!]\, G$	, from the above and (*), using the definition of **en**	□

5.3 Examples

Program union is a central concept in our theory. So, we consider a few examples involving union in some depth. We show the component programs and derive some of their properties employing the union theorem and its corollaries.

The treatment of progress properties in these examples will not be entirely satisfactory because the union theorem provides a direct method for proving transient predicates (and *ensures*-properties) only; we have, as yet, no way to deduce a *leads-to* property of a system from those of its components. We return to this question in Section 5.4.2.

5.3.1 Parallel Search

The following program contains the essence of parallel state-space search. In the search problem, a given state-space is partitioned and a process assigned to search each part (in the partition) independently. The processes accumulate their results in certain global variables.

A simple example is to count the number of zeroes in an array, $A[0..N]$, $N \geq 0$. We partition the elements of A into those with even indices and those with odd indices. A separate process, *even* or *odd*, counts the number of zeroes in the appropriate part; the result is stored in a global variable, z.

Program *even*
  declare
    global z : natural, A : array$[0..N]$ of integer,
    local i : natural
  initially $i, z = 0, 0$
  assign
    $i, z \;:=\; i+2, z+1 \quad$ if $\; A[i] = 0 \;\wedge\; i \leq N$
    $\qquad\quad \sim i+2, z \qquad$ if $\; A[i] \neq 0 \;\wedge\; i \leq N$
end $\{even\}$

Program *odd*
  declare
    global z : natural, A : array[0..N] of integer,
    local j : natural
  initially $j, z = 1, 0$
  assign
    $j, z = \quad j+2, z+1 \quad \text{if} \quad A[j] = 0 \ \wedge \ j \leq N$
    $\qquad\quad \sim j+2, z \quad\quad \text{if} \quad A[j] \neq 0 \ \wedge \ j \leq N$
end $\{odd\}$

It is straightforward to see that z is the number of zeroes in A when the program state satisfies the fixed point predicates of both processes (further, both processes will eventually satisfy their FPs). Observe that each action in F or G is an "atomic" action in $F \ [\!] \ G$. Thus, assignments to i and z in *even* are carried out together, without preemption by any action of *odd*. Also, note that A is a read-only variable though this fact is not expressed in the declaration.

Note: A better methodology for parallel search is for each process to store the result of its search in a local variable. Upon completion of its search it updates the global variable, in which the result of the entire search is stored, using the value of its local variable. Thus, *even* and *odd* may store their zero-counts in local variables ez and oz, respectively, and upon completion of their searches, they may increase z by ez and oz, respectively. □

Derivations of Program Properties We prove the following properties of *even* $[\!]$ *odd*.

1. $z \geq m$ stable , for any integer m.

2. $even.i \ \wedge \ i \leq N+2$ invariant $\{even.i$ is *true* iff i is *even*; similarly, $odd.j\}$
   $odd.j \ \wedge \ j \leq N+2$ invariant

3. $i > N$ stable
   $j > N$ stable

4. $z = \langle + \ k \ : \ 0 \leq k < i \ \wedge \ even.k \ \wedge \ A[k] = 0 \ : \ 1 \rangle$
   $+ \langle + \ k \ : \ 0 \leq k < j \ \wedge \ odd.k \ \wedge \ A[k] = 0 \ : \ 1 \rangle$ invariant

5. $true \ \mapsto \ i > N$,
   $true \ \mapsto \ j > N$

6. $true \;\mapsto\; z = \langle + \; k \;:\; 0 \le k \le N \;\wedge\; A[k] = 0 \;:\; 1 \rangle$

In the following proofs, we do not explicitly show the proofs of **co**- or **en**-properties in the components—*even* or *odd*. We have seen in Chapters 3 and 4 how such properties are deduced from program texts.

Proof of (1), $z \ge m$ stable in $even \;[\!]\; odd$

$z \ge m$ stable in $even$
, from text of $even$
$z \ge m$ stable in odd
, from text of odd
$z \ge m$ stable in $even \;[\!]\; odd$
, union-theorem corollary on the above two

Proof of (2), $even.i \;\wedge\; i \le N+2$ invariant in $even \;[\!]\; odd$

i constant in odd , locality axiom (see Section 5.2.2)
$even.i \;\wedge\; i \le N+2$ stable in odd
, from above (a constant predicate is stable)
$even.i \;\wedge\; i \le N+2$ invariant in $even$
, from text of $even$
$even.i \;\wedge\; i \le N+2$ invariant in $even \;[\!]\; odd$
, union-theorem corollary on the above two

Proof of $odd.j \;\wedge\; j \le N+2$ invariant: similar.

Proof of (3), similar to proof of 2.

Proof of (4), $z = \langle + \; k \;:\; 0 \le k < i \;\wedge\; even.k \;\wedge\; A[k] = 0 \;:\; 1 \rangle +$
$\langle + \; k \;:\; 0 \le k < j \;\wedge\; odd.k \;\wedge\; A[k] = 0 \;:\; 1 \rangle$ in $even \;[\!]\; odd$

initially
{union theorem}
$i, j, z = 0, 1, 0$ in $even \;[\!]\; odd$
$\Rightarrow$ {predicate calculus}
$z = \langle + \; k \;:\; 0 \le k < i \;\wedge\; even.k \;\wedge\; A[k] = 0 \;:\; 1 \rangle + \langle + \; k \;:\; 0 \le k <$
$j \;\wedge\; odd.k \;\wedge\; A[k] = 0 \;:\; 1 \rangle$

The remaining proof obligation is to show that the above expression is stable in $even \;[\!]\; odd$. From the union-theorem corollary, we have to show that this expression is stable in both *even* and *odd*; these follow, directly, from the texts of these program components.

Proof of (5), $true \;\mapsto\; i > N \quad \text{in } even \,[\!]\, odd$

$i = k$ **en** $i > k \quad \text{in } even$	, from text of *even*
$i = k$ stable in *odd*	, locality axiom
$i = k$ **en** $i > k \quad \text{in } even \,[\!]\, odd$	, union-theorem corollary
$i = k \;\mapsto\; i > k \quad \text{in } even \,[\!]\, odd$	, definition of $\mapsto$
$true \;\mapsto\; i > N \quad \text{in } even \,[\!]\, odd$	, induction on integers

Proof of $true \;\mapsto\; j > N$ is similar.

Note the structure of the above progress proof: An *ensures*-property is first established for the system using the union theorem; then, the *leads-to*-property is established and induction applied. Conversely, if we had first established

$$true \;\mapsto\; i > N \quad \text{in } even$$

then the union theorem could not have been used in proving the desired result.

Proof of (6), $true \;\mapsto$
$$z = \langle + \; k \;:\; 0 \le k \le N \;\wedge\; A[k] = 0 \;:\; 1 \rangle \quad \text{in } even \,[\!]\, odd$$

In $even \,[\!]\, odd$

$true$

$\mapsto$ {completion rule using (3) and (5)}

$i > N \;\wedge\; j > N$

$\Rightarrow$ {using (2)}

$i > N \;\wedge\; j > N \;\wedge\; even.i \;\wedge\; i \le N + 2 \;\wedge\; odd.j \;\wedge\; j \le N + 2$

$\Rightarrow$ {using (4)}

$z = \langle +k \;:\; 0 \le k \le N \;\wedge\; A[k] = 0 \;:\; 1 \rangle$

The proof of this program is given in excruciating detail to highlight the application of the union theorem. In each case, the proof is carried out at two levels: (1) properties that are deduced directly from the text of a program component, and (2) properties that are deduced by applying the union theorem and the other inference rules. We have emphasized the second level of the proof. This is because the properties of the components may be derived in different ways for programs written in different languages. In particular, this proof structure permits us to carry out deductions where the components are specified by their properties and not by their codes. We see such an example in Section 5.3.3.

5.3.2 Handshake Protocol

We show a protocol where two programs—F, G—enter their critical sections (in order to use a shared resource, such as a printer or a transmission medium) alternately. A shared boolean variable, b, determines which process enters its critical section next: If b holds then F can enter its critical section and if $\neg b$ holds then G can enter its critical section. Upon completion of its critical section each program inverts b to allow the other program to enter its critical section.

The programs are shown in the example in Section 5.2.1. We have not shown the entry or exit from the critical sections, explicitly; variables nf and ng count the number of entries into their critical sections by F and G, respectively.

Note: A simple variation of this solution is to introduce two boolean variables, bf and bg, to represent b. The representation is based on the invariant

$$b \equiv (bf = bg)$$

Therefore, b can be inverted by inverting either bf or bg. This observation allows us to restrict the accesses of the programs to the global variables; while both programs may read the values of both bf and bg, only program F writes into bf and G into bg. □

Derivations of Program Properties We consider the basic version of the handshake protocol, given as an example in Section 5.2.1. We show that, in $F \,[\!]\, G$

$0 \leq nf - ng \leq 1$ and
$true \;\mapsto\; nf > m$, for any integer m

(These properties imply that each program—F, G—enters its critical section arbitrarily often, and the programs enter their critical sections alternately.) These facts can be established directly from the code of $F \,[\!]\, G$. Our interest, however, is in deducing these facts *without* constructing $F \,[\!]\, G$. We prove

1. $0 \leq nf - ng \leq 1 \;\wedge\; [b \;\equiv\; (nf = ng)]$ invariant in $F \,[\!]\, G$
2. $true \;\mapsto\; nf > m$ in $F \,[\!]\, G$

Employing the substitution axiom on (1),

$0 \le nf - ng \le 1$ invariant in $F \,[\!]\, G$, can be established.

Proof of 1:

$$
\begin{array}{ll}
 & (F \,[\!]\, G).IC \\
\Rightarrow & \{\text{union theorem}\} \\
 & b, nf, ng = true, 0, 0 \\
\Rightarrow & \{\text{predicate calculus}\} \\
 & 0 \le nf - ng \le 1 \ \wedge\ [b \ \equiv\ (nf = ng)]
\end{array}
$$

The remaining task is to show that
$0 \le nf - ng \le 1 \ \wedge\ [b \ \equiv\ (nf = ng)]$ stable in $F \,[\!]\, G$.

The above predicate is stable in F and in G, from program text.
Hence, the result. $\square$

Proof of 2: In the following proof, k, m are arbitrary integers. We base the proof on two simple properties, 2.1 and 2.2.

2.1 $nf = k \ \wedge\ \neg b \mapsto\ nf = k \ \wedge\ b$ in $F \,[\!]\, G$, proof given below

$$
\begin{array}{llll}
2.2 & nf = k \ \wedge\ b & \mapsto\ nf > k & \text{in } F \,[\!]\, G \text{ , proof given below} \\
 & nf = k \ \wedge\ \neg b & \mapsto\ nf > k & \text{in } F \,[\!]\, G \text{ , transitivity on the above two} \\
 & nf = k & \mapsto\ nf > k & \text{in } F \,[\!]\, G \text{ , disjunction on the above two} \\
 & true & \mapsto\ nf > m & \text{in } F \,[\!]\, G \text{ , induction on integers} \quad \square
\end{array}
$$

Proof of 2.1, $nf = k \ \wedge\ \neg b \ \mapsto\ nf = k \ \wedge\ b$ in $F \,[\!]\, G$

$$
\begin{array}{ll}
nf = k \ \wedge\ \neg b \ \mathbf{en}\ nf = k \ \wedge\ b & \text{in } G \text{ , from the text of } G \\
nf = k \ \wedge\ \neg b \ \text{stable} & \text{in } F \text{ , from the text of } F \\
nf = k \ \wedge\ \neg b \ \mathbf{en}\ nf = k \ \wedge\ b & \text{in } F \,[\!]\, G \text{ , union-theorem corollary} \\
nf = k \ \wedge\ \neg b \mapsto\ nf = k \ \wedge\ b & \text{in } F \,[\!]\, G \text{ , definition of } \mapsto \quad \square
\end{array}
$$

Proof of 2.2 {similar to that of 2.1} $nf = k \ \wedge\ b \ \mapsto\ nf > k$ in $F \,[\!]\, G$

$$
\begin{array}{lll}
nf = k \ \wedge\ b & \mathbf{en}\ nf > k & \text{in } F \text{ , from the text of } F \\
nf = k \ \wedge\ b & \text{stable} & \text{in } G \text{ , from the text of } G \\
nf = k \ \wedge\ b & \mathbf{en}\ nf > k & \text{in } F \,[\!]\, G \text{ , union-theorem corollary} \\
nf = k \ \wedge\ b & \mapsto nf > k & \text{in } F \,[\!]\, G \text{ , definition of } \mapsto \quad \square
\end{array}
$$

5.3.3 Semaphore

We show a program, *sem*, that implements a binary semaphore. Programs F and G operate on the semaphore by requesting a P-operation or a V-operation. Program F (or G) requests a P-operation by setting a global boolean variable pf (or pg) to *true*. The program *sem* responds to a request for P-operation by granting the semaphore to F or G (if the semaphore is available); the response is indicated by changing the corresponding variable—pf or pg—to *false* from *true*. (Thus, as long as pf is *true* F is "waiting" for the semaphore.) Similarly, F and G request V-operations by setting global variables vf and vg, respectively, to *true* and the *sem* program sets them *false* after completions of the respective V-operations. It is readily seen that program *sem* can be regarded as the union of two programs, *semf* and *semg*, responding to requests from F and G, respectively. Now, the value of the semaphore has to be shared between *semf* and *semg*; boolean variable s holds this value. Figure 5.2 shows the decomposition of *sem*. The codes for *semf* and *semg* are as follows.

Program *semf*
  declare global pf, vf, s: boolean
  initially $pf, vf, s = \mathit{false}, \mathit{false}, \mathit{true}$
  assign
    $pf, s := \mathit{false}, \mathit{false}$ if $pf \wedge s$
  $[\!]$ $vf, s := \mathit{false}, \mathit{true}$ if vf
end $\{semf\}$

Program *semg*
  declare global pg, vg, s: boolean
  initially $pg, vg, s = \mathit{false}, \mathit{false}, \mathit{true}$
  assign
    $pg, s := \mathit{false}, \mathit{false}$ if $pg \wedge s$
  $[\!]$ $vg, s := \mathit{false}, \mathit{true}$ if vg
end $\{semg\}$

Program *sem* is $semf \; [\!] \; semg$ in which variables pf, pg are global and s is local.

The protocol for changing pf, vf, pg, vg that we have described is too liberal; it permits program F to simultaneously set pf *and* vf to *true*, and this is illegal according to our interpretation. We can restrict the behavior of F by imposing explicit constraints in its specification. A better strategy is to syntactically restrict the interface in such a way that few illegal actions are permitted. We illustrate this for the semaphore problem.

Introduce a global boolean variable rf which F sets to *true* to request a P-operation or a V-operation. The ambiguity—whether the requested operation is a P- or a V-operation—can be resolved by the semaphore program because it can retain the information regarding which program

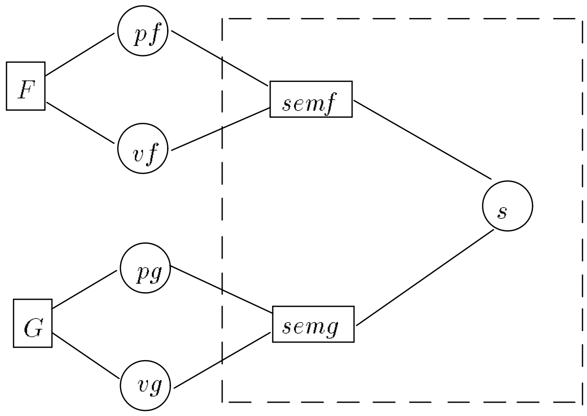


Figure 5.2: Decomposition of program *sem* into *semf* and *semg*. The semaphore, s, is shared between *semf* and *semg*.

"holds" the semaphore; if program F "holds" the semaphore then rf being *true* signals a V-operation and if F does not "hold" the semaphore then rf being *true* signals a request for a P-operation. The semaphore program sets rf *false* to indicate completion of the requested operation. Similarly, program G communicates its intention using a global boolean variable, rg.

The above proposal simplifies the interface considerably. But, the semaphore program now has to retain the information regarding who "holds" the semaphore. Let sh be a variable that assumes the following values:

F: program F holds the semaphore

G: program G holds the semaphore

N: Neither program holds the semaphore

Now, we no longer need the boolean variable s (indicating the semaphore value) because

$$s \equiv (sh = \mathrm{N})$$

Furthermore, introduction of sh prevents both programs from holding the semaphore simultaneously; thus, the proof obligation for mutual exclusion has been discharged by syntactic means.

As before, we implement the semaphore program as the union of programs $semf1$ and $semg1$, hiding sh as a local variable.

Program $semf1$
  declare global rf: boolean,
          sh: (F,G,N)
  initially sh =N
  assign

$$
\begin{array}{lrcll}
 & rf, sh & := & \mathit{false}, \mathrm{F} & \text{if } \; rf \wedge sh = \mathrm{N} \\
[\!] & rf, sh & := & \mathit{false}, \mathrm{N} & \text{if } \; rf \wedge sh = \mathrm{F}
\end{array}
$$

end $\{semf1\}$

Program $semg1$
  declare global rg: boolean,
          sh: (F,G,N)
  initially sh =N
  assign

$$
\begin{array}{lrcll}
 & rg, sh & := & \mathit{false}, \mathrm{G} & \text{if } \; rg \wedge sh = \mathrm{N} \\
[\!] & rg, sh & := & \mathit{false}, \mathrm{N} & \text{if } \; rg \wedge sh = \mathrm{G}
\end{array}
$$

end $\{semg1\}$

The program $semf1 \; [\!] \; semg1$ implements a "weak" semaphore; program F may request the semaphore but it may never hold it because program G could be granted the semaphore arbitrarily many times (and G releases the semaphore arbitrarily many times). A progress property of this program is

$$
\begin{array}{lll}
rf \mapsto sh \neq \mathrm{N} & \text{in } F \; [\!] \; G \; [\!] \; semf1 \; [\!] \; semg1 \\
rg \mapsto sh \neq \mathrm{N} & \text{in } F \; [\!] \; G \; [\!] \; semf1 \; [\!] \; semg1
\end{array}
$$

A "strong" semaphore is one in which every request for a P-operation is eventually honored, provided every program holding the semaphore releases it, eventually.

In order to implement a strong semaphore, we introduce a boolean variable pr that indicates the priority in granting the semaphore; variable pr is *true* iff F is to be given priority whenever there is a contention for the semaphore. The initial value of pr is immaterial. Whenever F is granted the semaphore pr is set *false*—thus giving priority to G in future contentions—and similarly, pr is set *true* whenever G is granted the semaphore. Analogous to variable sh, variable pr is shared between $semf$ and $semg$. The initial value of pr is immaterial.

Program $semf2$
  declare global rf, pr : boolean,
          sh : (F,G,N)
  initially sh =N

assign

rf, sh, pr := $false$, F, $false$ if $rf \wedge sh =$N $\wedge$ $(pr \vee \neg rg)$
{P-operation}

⫿ rf, sh := $false$, N if $rf \wedge sh =$F
{V-operation}

end {$semf2$}

Program $semg2$

declare global rg, pr : boolean,
sh : (F,G,N)

initially $sh =$N

assign

rg, sh, pr := $false$, G, $true$ if $rg \wedge sh =$N $\wedge$ $(\neg pr \vee \neg rf)$
{P-operation}

⫿ rg, sh := $false$, N if $rg \wedge sh =$G
{V-operation}

end {$semg2$}

We show below that

$rf \mapsto sh =$F in F ⫿ G ⫿ $semf2$ ⫿ $semg2$ and,
$rg \mapsto sh =$G in F ⫿ G ⫿ $semf2$ ⫿ $semg2$

provided the holder of the semaphore releases it, eventually.

Derivations of Program Properties We consider the last version of the semaphore program, consisting of $semf2$ and $semg2$, implementing a strong semaphore. We have four programs under consideration: F, G, $semf2$ and $semg2$. This is an example where texts of some of the components—F, G—are unavailable. We will prove properties of the system assuming certain properties of F and G.

For the composite program F ⫿ G ⫿ $semf2$ ⫿ $semg2$ we use the abbreviation sys. The safety property of sys is that at most one program—F or G—"holds" the semaphore at any time; this is vacuously established because sh identifies the holder of the semaphore, if any. The progress property of sys is that any program, F or G, requesting the semaphore is granted the semaphore, eventually, provided every semaphore holder requests a V-operation, eventually. Specifically, our goal is to establish

$rf \mapsto sh =$ F in sys
(and, analogously,
$rg \mapsto sh =$ G in sys)

assuming

$sh = \mathrm{F} \mapsto rf \quad \text{in } sys$ (s0)

$sh = \mathrm{G} \mapsto rg \quad \text{in } sys$ (s1)

We assume that programs F, G obey the following protocol in modifying the global variables.

rf stable in F (s2)

rg stable in G (s3)

Also, all global variables other than rf are outside F; similarly, rf, pr, sh are outside G. Therefore, from the locality axiom,

rg, pr, sh constant in F (s4)

rf, pr, sh constant in G (s5)

We can establish the following safety properties of sys.

$sh = \mathrm{F} \Rightarrow \neg pr$ invariant (s6)

$sh = \mathrm{G} \Rightarrow pr$ invariant (s7)

$rf \wedge sh \neq \mathrm{F} \ \mathbf{co} \ rf \vee sh = \mathrm{F}$ (s8)

$\neg rg \wedge sh = \mathrm{G} \ \mathbf{co} \ sh = \mathrm{G}$ (s9)

The proofs of these properties are obtained by applying the union theorem. We prove (s9).

Proof of (s9), $\neg rg \wedge sh = \mathrm{G} \ \mathbf{co} \ sh = \mathrm{G} \quad \text{in } sys$

$sh = \mathrm{G}$ stable in G , from s5

$\neg rg \wedge sh = \mathrm{G} \ \mathbf{co} \ sh = \mathrm{G} \quad \text{in } G$, strengthening the lhs of the above

$\neg rg \wedge sh = \mathrm{G}$ stable in F , from s4

$\neg rg \wedge sh = \mathrm{G}$ stable in $semf2$, text of $semf2$

$\neg rg \wedge sh = \mathrm{G}$ stable in $semg2$, text of $semg2$

$\neg rg \wedge sh = \mathrm{G} \ \mathbf{co} \ sh = \mathrm{G} \quad \text{in } sys$, union-theorem corollary on the above four □

We can establish the following progress properties, again, applying the union theorem (and its corollaries). These progress properties capture the essence of the system operation: (s10) says that a request for a V-operation is eventually honored, and then the program other than the requester has priority; (s11) says that if a requesting program has the priority and the semaphore is available then the program is, eventually, granted the semaphore; (s12) says that a request for a P-operation cannot be permanently ignored if the semaphore is available (either the requester is granted the semaphore or the other program requests the semaphore).

In *sys*:

$$rg \ \wedge \ sh = \mathrm{G} \mapsto \ pr \ \wedge \ sh = \mathrm{N} \tag{s10}$$
$$rf \ \wedge \ pr \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F} \tag{s11}$$
$$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \mapsto \ (sh = \mathrm{F}) \vee (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N}) \tag{s12}$$

Note: Each of the properties (s8–s12) has a dual property obtained by replacing one predicate in the following pairs by the other: (rf, rg), $(pr, \neg pr)$, $(sh = \mathrm{F},\ sh = \mathrm{G})$, $(sh \neq \mathrm{F},\ sh \neq \mathrm{G})$ and $(sh = \mathrm{N},\ sh = \mathrm{N})$. For instance, the dual of (s11) is $rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{G}$. □

The proofs of all the progress properties are similar. In each case, we establish the property as an *ensures* property in *sys*. For (s10), an appropriate property is transient in $semg2$, for (s11) in $semf2$ and for (s12) in $semf2$. According to the union-theorem corollary, we have to show that a corresponding safety property holds in the other components; these properties are derived from (s2–s5) and the texts of $semf2$ and $semg2$. A typical proof is given below.

Proof of (s12), $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \mapsto \ (sh = \mathrm{F}) \ \vee$
$(rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N})$ in *sys*

We will show

$$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \ \mathbf{en} \ (sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N}) \text{ in } sys$$

by applying the union-theorem corollary to the following safety property that holds in each component,

$$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \ \mathbf{co} \ (sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$$

and

$$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \text{ transient in } semf2.$$

In the following proof, for F and $semg2$ we discharge the proof obligation by showing that the lhs, $rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$, is stable.

$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ stable in F , from (s2,s4)
$rf \ \wedge \ sh = \mathrm{N}$ stable in G , from s5
$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \ \mathbf{co} \ (sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$ in G
, strengthening the lhs and weakening the rhs

$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ stable in $semg2$, text of $semg2$
$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ **co** $(sh = \mathrm{F}) \ \vee \ (rf \ \wedge \ sh = \mathrm{N})$ in $semf2$
, text of $semf2$
$rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N}$ transient in $semf2$, text of $semf2$ □

The properties (s2–s12) have all the information we need to establish the desired progress property. We don't need to look at program texts any more; all further manipulations are based on the derived rules. This is a typical strategy that we advocate: Establish certain elementary properties from the program text (or the specifications of the components) and then, apply the derived rules to establish more complex properties.

Proof of $rf \ \mapsto \ sh = \mathrm{F}$ in sys

In the following proof all properties are in sys. The result follows by taking disjunction of the following three.

$rf \ \wedge \ sh = \mathrm{G} \mapsto \ sh = \mathrm{F}$, proven below $(s13)$
$rf \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F}$, proven below $(s14)$
$rf \ \wedge \ sh = \mathrm{F} \mapsto \ sh = \mathrm{F}$, implication □

Proof of (s13), $rf \ \wedge \ sh = \mathrm{G} \mapsto \ sh = \mathrm{F}$

$\neg rg \ \wedge \ sh = \mathrm{G} \mapsto \ rg \ \wedge \ sh = \mathrm{G}$, PSP on (s1,s9)
$\neg rg \ \wedge \ sh = \mathrm{G} \mapsto \ pr \ \wedge \ sh = \mathrm{N}$, transitivity on above and s10
$sh = \mathrm{G} \mapsto \ pr \ \wedge \ sh = \mathrm{N}$, disjunction on above and s10
$rf \ \wedge \ sh = \mathrm{G} \mapsto \ (rf \ \wedge \ pr \ \wedge \ sh = \mathrm{N}) \ \vee \ (sh = \mathrm{F})$
, PSP on (above, s8)
$rf \ \wedge \ sh = \mathrm{G} \mapsto \ sh = \mathrm{F}$, cancellation on (above, s11) □

Proof of (s14), $rf \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F}$

$rg \ \wedge \ sh = \mathrm{N} \mapsto \ (rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N}) \ \vee \ (rg \ \wedge \ pr \ \wedge \ sh = \mathrm{N})$
, implication
$rg \ \wedge \ \neg pr \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{G}$, dual of (s11)
$rg \ \wedge \ sh = \mathrm{N} \mapsto \ (sh = \mathrm{G}) \ \vee \ (rg \ \wedge \ pr \ \wedge \ sh = \mathrm{N})$
, cancellation on the above two
$rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N} \mapsto \ (rf \ \wedge \ sh = \mathrm{G}) \ \vee$
$(rf \ \wedge \ rg \ \wedge \ pr \ \wedge \ sh = \mathrm{N}) \ \vee \ (sh = \mathrm{F})$
, PSP on (above, s8)
$rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N} \mapsto \ (rf \ \wedge \ pr \ \wedge \ sh = \mathrm{N}) \ \vee \ (sh = \mathrm{F})$
, cancellation with (s13) and
weakening the second term in the

$$
\begin{array}{ll}
 & \text{above rhs} \\
rf \ \wedge \ rg \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F} & \text{, cancellation on (above, s11)} \\
rf \ \wedge \ \neg rg \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F} & \text{, cancellation on (s12, above)} \\
rf \ \wedge \ sh = \mathrm{N} \mapsto \ sh = \mathrm{F} & \text{, disjunction of the above two}
\end{array}
\qquad \Box
$$

5.3.4 Vending Machine

This example is inspired by Hoare[5]. A vending machine has a coin slot that accepts either a small coin or a large coin from a user. The vending machine has two buttons, one for requesting a small chocolate and the other for a large chocolate. If a button is pressed after depositing an adequate amount of money—a small coin for a small chocolate or a large coin for a large chocolate—then the machine dispenses the appropriate kind of chocolate. If a large coin has been deposited and the button is pressed for a small chocolate then the machine dispenses a small chocolate and returns the appropriate change.

The behavior of the machine can be described by the finite-state diagram of Figure 5.3. Each node represents a state and each edge a transition; the initial state is denoted by a circle. The label on an edge is the name of the event that causes the transition. The events could be caused by the user—*uds*, for instance, denotes the event of the user depositing a small coin—or, the event could be generated by the vending machine—*mdl*, for instance, denotes the event of the machine dispensing a large chocolate. Figure 5.3 also gives the meaning of each event in the diagram.

This specification provides a succinct description of the allowable execution sequences. Thus,

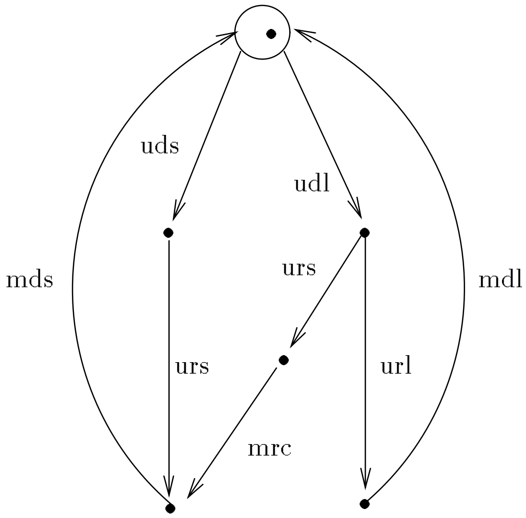
$$udl \ url \ mdl$$

is an allowable execution sequence whereas

$$uds \ mdl$$

is not an allowable execution sequence. The small size of the specification makes it possible to trace all individual execution sequences and be assured of the "correctness of the specification."

The given specification should be regarded as a specification of the entire system, consisting of *both* the user and the vending machine. Whenever components of a system are "tightly-coupled"—an action in one component is followed by an action in another component, for instance—their behaviors are often best described by state diagrams. In such cases, there is little advantage in separating the specifications of the individual components. However, in this approach, for systems with even a few components the



uds: user deposits small coin
udl: user deposits large coin
urs: user requests small chocolate
url: user requests large chocolate
mrc: machine returns change
mds: machine dispenses small chocolate
mdl: machine dispenses large chocolate

Figure 5.3: The state transitions in the Vending Machine.

state space could become extraordinarily large. It is then imperative to specify each component individually. For this example, we specify the user and the vending machine, individually.

The interface between the user and the vending machine (VM, for short) consists of the following variables.

cin: (s, l, ϕ) The value of *cin* is the amount of coin in the input box of the machine. Here, s denotes a small coin, l a large coin and ϕ, the absence of any coin. This variable can be increased (from ϕ to s or l) by the user alone; it may be decreased (to ϕ) by VM alone.

but: (s, l, ϕ) The value of *but* is the outstanding request for a chocolate; s is for a small chocolate and l for a large chocolate; the value of *but* is ϕ when there is no outstanding request for a chocolate. As for *cin*, the value of *but* can be increased only by the user and decreased only by VM.

Note: We have abstracted from our original description by replacing two chocolate buttons by one. This abstraction prevents requesting a small and a large chocolate simultaneously. □

dis: (s, l, ϕ) The value of *dis* is the size of the chocolate in the dispenser-tray, s for small, l for large and ϕ for the absence of a chocolate. This variable is increased (from ϕ to s or l) by the VM and decreased (to ϕ) by the user removing the dispensed chocolate.

cout: (boolean) The value of *cout* indicates if there is money in the coin-return box. This variable may be set *true* only by VM and *false* only by the user.

For the specification, we use the following partial order over $\{\phi, s, l\}$:

$$\phi < s \quad \text{and} \quad \phi < l$$

Note: This partial order is often called a "flat order"; here, ϕ is the "bottom" element and it is "smaller" than every other element; the remaining elements are unrelated by the partial order. □

In the following, variables m, n take arbitrary values from $\{\phi, s, l\}$. The specification for USER merely states how each of the interface variables—*cin*, *but*, *dis*, *cout*—can be changed by USER. There is no progress requirement for USER.

In USER ::
$cin \geq m$ stable
$but \geq m$ stable
$dis \leq m$ stable
$\neg cout$ stable

The safety specification of VM is obtained similarly.

In VM ::
$cin \leq m$ stable
$but \leq m$ stable
$dis \geq m$ stable
$cout$ stable

Additionally, we have the progress requirement for VM that it has to dispense the appropriate chocolate and change. For all C, B from $\{\phi, s, l\}$

$$cin, but, dis, cout = C, B, \phi, false \ \wedge \ B \neq \phi \ \wedge \ C \geq B \ \mapsto$$
$$cin, but, dis, cout = \phi, \phi, B, C > B \quad \text{in VM}$$

Note on the Specification Our proposed specification is quite crude. The progress property of VM, as described, poses considerable technical difficulties; we explain the difficulties and propose a different specification later in this section. We have modelled the chocolate dispenser tray by the variable *dis*; this disallows VM from dispensing another chocolate until the previous one has been removed. A general dispenser-tray can be modelled by a bag whose elements are s or l. The user simply removes an element from the bag and VM adds to the bag. Similar remarks apply to *cout*. Our specification allows the user to set the interface variables in arbitrary order. In particular, the user may first set *but* and then *cin*. The specification in Figure 5.3 disallows this possibility.

Derivations of Program Properties A secondary purpose of this subsection is to derive certain properties of the vending machine system by applying the union theorem. The main purpose is to show that certain intuitively plausible properties, justified by the informal description, cannot be established from the given formal specification.

First, observe that the progress property of VM—a *leads-to* property—is in a form that is not amenable to compositional deduction by the union theorem. (See remarks made in Section 2.6.3 just prior to the introduction of the properties, s0 and s1.) We strengthen the property to an *ensures* property: For all C, B from $\{\phi, s, l\}$

$$cin, but, dis, cout = C, B, \phi, false \ \wedge \ B \neq \phi \ \wedge \ C \geq B \ \textbf{en}$$
$$cin, but, dis, cout = \phi, \phi, B, C > B \quad \text{in VM}$$

We can now establish certain simple properties in VM [] USER. The following property states that a small chocolate will be dispensed provided a small coin is inserted, a small chocolate is requested and both the chocolate dispenser-tray and the output coin-box are empty.

$$cin, but, dis, cout = s, s, \phi, false \ \mapsto \ dis = s \quad \text{in VM [] USER}$$

Proof:

$cin = s$ stable in USER , from USER specification
$but = s$ stable in USER , similarly
$dis = \phi$ stable in USER , similarly
$\neg cout$ stable in USER , similarly
$cin, but, dis, cout = s, s, \phi, false$ stable in USER
, conjunction of above four
$cin, but, dis, cout = s, s, \phi, false$ **en**
$cin, but, dis, cout = \phi, \phi, s, false$ in VM
, instantiating the progress property of VM with $C, B \ := \ s, s$
$cin, but, dis, cout = s, s, \phi, false$ **en**
$cin, but, dis, cout = \phi, \phi, s, false$ in VM [] USER
, union-theorem corollary, applied to the above two
$cin, but, dis, cout = s, s, \phi, false \ \mapsto \ dis = s$ in VM [] USER
, definition of $\mapsto$ and weakening the rhs □

An intuitively plausible property of VM [] USER is

$$cin, but = s, s \ \mapsto \ dis = s.$$

That is, a small chocolate will be dispensed if a small coin is inserted and a small chocolate is requested. This property, however, does not hold in our system. In particular, if the chocolate-dispenser tray already contains a large chocolate, i.e., $dis = l$, then VM is prevented from dispensing a small chocolate, i.e., setting dis to s, because

$$dis \geq m \text{ stable}$$

for all m in $\{\phi, s, l\}$. Thus, the USER can break the system by not removing a single chocolate. We have discussed how to model the dispenser-tray by a bag (see Note on the Specification); the reader may wish to prove this progress property in the revised model.

A more involved property is

$$cin, but = s, l \text{ stable}$$

That is, if the user inserts a small coin and requests a large chocolate the system is deadlocked! This is plausible because: (1) the user cannot add additional coins nor can he cancel his request and (2) the vending machine cannot honor this request. However, this property is not provable. The specification of VM does not prevent it from removing an inserted coin or cancelling a button-push. If desired, VM's specification may be modified to state that *cin* (or *but*) is not modified until *but* (or *cin*) has the appropriate value.

5.3.5 Message Communication

An asynchronous message communicating system is a program where the components are the individual processes and the global variables represent the contents of the channels along which communications take place. For instance, if process F sends messages to process G along the FIFO channel C, then our system is $F \parallel G$ and the global variable (declared in both F, G) is a sequence (of messages) C. Program F "sends" by appending messages to the end of C and program G "receives" by removing the message at the front of C, provided C is nonnull. A typical example of message communication is a sequence of programs $F_0, \ldots, F_N$, through which data is pipelined. Let F_i send messages in FIFO order to F_{i+1}, $0 \le i < N$. Let C_i denote the channel directed from F_i to F_{i+1}. Then C_i is a sequence (of messages) and C_i is declared as a global variable in F_i and F_{i+1}. The system is $\langle \parallel i \ :: \ F_i \rangle$.

Channels between the processes need not be FIFO. An unordered channel—see Section 4.5.3—may be implemented by a bag of messages. Also, a bounded FIFO channel, C, is implemented by a bounded sequence; the sender can append a message to C only if the length of C is lower than the bound. Other forms of asynchronous communication—for instance, where each transmission is acknowledged—can be, similarly, modelled. Of particular interest is the transmission of "signals"; a signal has no associated data and a new signal can be sent only after the previous one has been acknowledged. Signal transmission and acknowledgement can be modelled using a global boolean variable. The variable is set *true* to indicate signal transmission and *false* to indicate acknowledgement. We have used this scheme in the semaphore program—Section 5.3.3—where variable rf, for instance, was set *true* by program F to request an operation and set *false* by program $semf$ to indicate the completion of that operation.

Synchronous message communication, where the send and the receive happen simultaneously[5], cannot be easily modelled using the union operator on the sending and receiving processes.

A more elaborate example of message communication is treated next.

A Client-Server Network A set of "client" processes—$CL_0, \ldots, CL_J$—request service from a set of server processes—$SR_0, \ldots, SR_K$—periodically during a computation. Any server can serve any client. A typical example is a set of user processes (clients) connected to a set of printers (servers) where the print-request of any user can be satisfied by any printer.

We propose a communication network connecting the clients to the servers. Each channel in the network carries requests; furthermore, a channel contains at most one request at any time. Therefore, a channel can be modelled by a (global) variable whose value is either a request or ϕ, denoting that the corresponding channel is empty. The protocol for accessing such a variable is as follows: A "send" on the channel is simulated by writing into the corresponding variable provided its current value is ϕ; a "receive" is simulated by reading the corresponding variable value and then setting its value to ϕ.

The components of the system are the following. The interconnection network is shown in Figure 5.4.

Clients, $CL_j, 0 \leq j \leq J$: Client CL_j sends its requests using global variable r_j; as described above, CL_j may store a request in r_j provided the value of r_j is ϕ.

Client-Manager, *CM*: This process merges the requests of the individual clients and issues one request at a time to the servers using a variable r. This process reads r_js (as described above, reading an r_j is accompanied by setting it to ϕ) in some order and writes into variable r. The policy for merging the requests is internal to this process. In many cases, a fair-merge of the requests is implemented, in the sense that every request in every r_i appears, eventually, in r.

Servers, $SR_k, 0 \leq k \leq K$: Server SR_k receives requests from the global variable w_k (as described above, SR_k sets w_k to ϕ after reading a request).

Server-Manager, *SM*: This process receives a stream of requests, one at a time, from a global variable w and it distributes the requests among the w_ks. The policy for distribution need not be fair among the

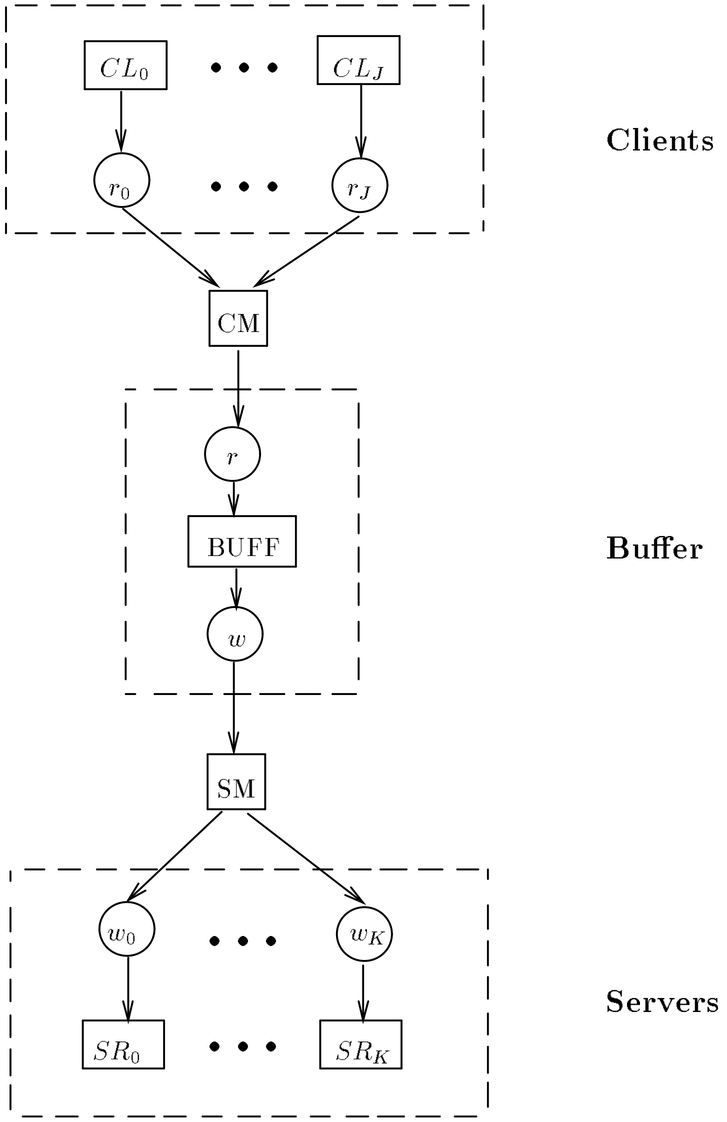


Figure 5.4: A client-server interconnection network: CM is client-manager and SM is server-manager.

servers; an unfair policy does not employ all the servers on an equitable basis.

Buffer, *BUFF*: This process implements a buffer of N requests, $N \geq 0$, in order to smooth-out the speed variations of the requests and the services. The process reads from r and writes into w (where r is written by *CM* and w is read by *SM*).

The entire system is

$$\langle [\!]\ j\ :\ 0 \leq j \leq J\ :\ CL_j \rangle\ [\!]\ \mathrm{CM}\ [\!]\ \mathrm{BUFF}\ [\!]\ \mathrm{SM}\ [\!]\ \langle [\!]\ k\ :\ 0 \leq k \leq K\ :\ SR_k \rangle$$

and the variables—$r_j s, r, w, w_k s$—may be declared local to this system.

5.4 Shortcomings of the Union Theorem

5.4.1 Substitution Axiom

Substitution axiom, described in Section 3.3, allows us to replace an invariant by *true*, and vice versa, in any property of a program. This axiom has proved to be extremely useful in practice. However, it poses certain methodological difficulties when applied to the union of programs.

Any invariant of $F\ [\!]\ G$ can still be replaced by *true* (and vice versa) in any property of $F\ [\!]\ G$; in fact, any such invariant can be used in the substitution axiom on a property of any component, F or G. However, it is not permissible to use "local" invariants of components in applying the substitution axiom. To see this, consider programs F, G where the programs share two global variables p, q, and F has:

initially $p, q\ :=\ true, true$
assign $q\ :=\ p$

Then,

$p \wedge q$ invariant in F	, text of F
q invariant in F	, substitution axiom in F

Now, suppose G does not modify q. Then

q stable in G	
q invariant in $F\ [\!]\ G$	corollary of the union theorem

This conclusion is invalid as can be seen by considering the program G that has

assign $p := \neg p$

The invalid step in the above argument is to apply the substitution axiom in F using a predicate, $p \wedge q$, that is invariant in F alone. A more general version of the substitution axiom that removes this methodological difficulty is given in Section ????.

5.4.2 Progress Properties under Union

The union theorem permits us to deduce transient predicates of a system from the transient predicates of any component. Our ultimate goal is to deduce the *leads-to* properties in a similar fashion. Unfortunately, there is no analogous result for the *leads-to* properties. Plausible inference rules—if $p \mapsto q$ holds in both F and G then $p \mapsto q$ in $F \,[\!]\, G$, or if p stable in F and $p \mapsto q$ in G then $p \mapsto q$ in $F \,[\!]\, G$—are all invalid. (It may be possible to prove a metatheorem that there can be no such inference rule for *leads-to*.) The following example illustrates the difficulties.

Example: Let F, G share a global integer variable x, which is incremented continually by F and decremented continually by G. For any k, we can show

$$\begin{array}{ll} & \mathit{true} \mapsto |x| > k \quad \text{in } F \\ \text{and} & \mathit{true} \mapsto |x| > k \text{ in } G \end{array}$$

However, $\mathit{true} \mapsto |x| > k$ in $F \,[\!]\, G$ does not hold because alternate steps by F and G starting in state $x = 0$, say, has $|x| \leq 1$ at all times. Observe in this example that *true* is stable in both F and G. □

The above example lends new meaning to the proverb, "Too many cooks spoil the broth": Even though either one of F or G could establish a property, their interleaved execution may not. In general, a proof of $p \mapsto q$ in F may consist of many proof steps, some of which could be falsified by G. For instance, $p \mapsto q$ may have been established in F by

$$p \mapsto r \quad \text{in } F \quad \text{and} \quad r \mapsto q \quad \text{in } F$$

Suppose that program G always falsifies r, eventually. Then, the following segment of an execution of $F \,[\!]\, G$ starting in a state that satisfies p

$$\{p\} \text{ execution of } F \quad \{r\} \text{ execution of } G \quad \{\neg r\}$$

eventually establishes $\neg r$, and there is no guarantee that q will ever be established later during this execution. Program G is said to have *interfered* with F's execution.

We may restrict the behavior of program G such that it never *interferes* with F's execution; unfortunately, this amounts to prohibiting G from writing into a variable that F reads. Selective prohibition, such as requiring that G may not falsify any of the proof steps in $p \mapsto q$, is impossible without exhibiting the structure of the proof; and this defeats the whole purpose of modular composition, which is to hide information, such as a local proof structure, within a component.

We believe that the absence of an appropriate inference rule for *leads-to* is a major impediment to compositional design. Program components cannot be designed in isolation; their interactions have to be taken into account. This problem is so central that we address it in a number of different ways. In all cases, we impose restrictions on the specifications and/or the compositions of programs.

In one approach (see Chapter 6), instead of specifying a traditional progress property of a component F, such as

$$p \mapsto q \quad \text{in } F$$

we specify that

$$p \mapsto q \quad \text{in } F \,[\!]\, G$$

for *any* G that satisfies a given condition. This is a property of program F. For example, if program F continually increments global integer variable x, then we can show, for any k

$$true \mapsto |x| > k \quad \text{in } F \,[\!]\, G$$

provided G never decreases x.

In another approach (see Chapter 6), we state the assumptions about the other components explicitly in each program. For example, program F may declare x to be an integer variable and then state that x can only be read or incremented in the other components of the system. Such declarations can be checked by a linker program that constructs the union of the components; moreover, certain properties of the entire system such as

$$true \mapsto |x| > k$$

can be proven from the text of F alone.

Yet another approach is to restrict the ways in which the components may modify the global variables. We show that if the actions of different programs are commuting—the exact definition of commuting has to wait until Chapter **????**—then certain progress properties of a component also hold at the system level.

5.5 Theoretical Issues

In this section, we propose certain axioms that any asynchronous composition operator should meet. Also, we study what it means for one program to refine another. Two programs are taken to be equivalent if they refine each other. Our approach is based on identifying a program with its properties; a refined program, simply, has all the properties of the program it refines. We propose two different definitions of refinement. We show that only one of the notions of refinement (and equivalence) is preserved under union, i.e., if F, G are equivalent then so are $F \; [\!] \; H$ and $G \; [\!] \; H$, for any H. This demonstration shows why "transient" (and *ensures*) is an important concept, particularly, in the study of asynchronous composition.

5.5.1 Axioms of Union

It is clear from our description of union (for Unity programs) that union is commutative and associative; further, there is a program—which has no variable and whose only action is *skip*—that serves as the "identity" element of union. Formally, we take the following as axioms for the asynchronous composition operator. For arbitrary programs F, G, H

- (Commutativity) $F \; [\!] \; G = G \; [\!] \; F$, if either side is defined,
- (Associativity) $(F \; [\!] \; G) \; [\!] \; H = F \; [\!] \; (G \; [\!] \; H)$, if either side is defined.
- (Existence of *id*) There is a program, *id*, such that $F = F \; [\!] \; id$

We have not yet defined the meaning of equality of programs. We take up this question next and study its relationship to union.

5.5.2 A Definition of Refinement

Notation: For a program F, let $F.\mathbf{co}$ be the set of pairs (p, q) such that

$$p \;\mathbf{co}\; q \quad \text{in } F.$$

Let F.transient be the set of predicates p such that p transient in F. The following are restatements of the appropriate parts of the union theorem, for programs F, G.

$$(F \; [\!] \; G).\mathbf{co} = F.\mathbf{co} \;\cap\; G.\mathbf{co}$$
$$(F \; [\!] \; G).\text{transient} = F.\text{transient} \cup G.\text{transient} \qquad \square$$

For programs F, G, we say that F is *refined* by G—written as $F \Rightarrow G$—if

$$G.IC \Rightarrow F.IC \ , \ F.\mathbf{co} \subseteq G.\mathbf{co} \ , \ F.\text{transient} \subseteq G.\text{transient}.$$

That is, G's initial condition is stronger than F's and G has all the **co**-properties and the transient predicates of F. Define $F = G$ to mean that $F \Rightarrow G$ and $G \Rightarrow F$.

Given $F \Rightarrow G$, we can show that any invariant of F is an invariant of G, as follows.

$$\begin{array}{ll}
 & p \text{ invariant} \quad \text{in } F \\
= & \{\text{definition of invariant}\} \\
 & F.IC \Rightarrow p \ \wedge \ p \text{ stable} \quad \text{in } F \\
\Rightarrow & \{\text{from } F \Rightarrow G, \ G.IC \Rightarrow F.IC \text{ and } p \text{ stable in } G\} \\
 & G.IC \Rightarrow p \ \wedge \ p \text{ stable} \quad \text{in } G \\
= & \{\text{definition of invariant}\} \\
 & p \text{ invariant} \quad \text{in } G
\end{array}$$

From the definition, "$\Rightarrow$" is reflexive and transitive. Furthermore, $\Rightarrow$ is preserved under union as shown in the following theorem.

Theorem: If $(F \Rightarrow G)$ then $(F \parallel H \Rightarrow G \parallel H)$ provided $F \parallel H$ and $G \parallel H$ are defined.

Proof:

1. We show $(G \parallel H).IC \Rightarrow (F \parallel H).IC$

$$\begin{array}{ll}
 & (G \parallel H).IC \\
= & \{\text{union theorem, condition (1)}\} \\
 & G.IC \ \wedge \ H.IC \\
\Rightarrow & \{G.IC \Rightarrow F.IC, \text{ from } F \Rightarrow G\} \\
 & F.IC \ \wedge \ H.IC \\
= & \{\text{union theorem, condition (1)}\} \\
 & (F \parallel H).IC
\end{array}$$

2. We show $(F \parallel H).\mathbf{co} \subseteq (G \parallel H).\mathbf{co}$

$$\begin{array}{ll}
 & (F \parallel H).\mathbf{co} \\
= & \{\text{union theorem, condition (2)}\} \\
 & F.\mathbf{co} \ \cap \ H.\mathbf{co} \\
\subseteq & \{F.\mathbf{co} \subseteq G.\mathbf{co}, \text{ from } F \Rightarrow G\} \\
 & G.\mathbf{co} \ \cap \ H.\mathbf{co} \\
= & \{\text{union theorem, condition (2)}\} \\
 & (G \parallel H).\mathbf{co}
\end{array}$$

3. We show $(F \,[\!]\, H)$.transient $\subseteq$ $(G \,[\!]\, H)$.transient

$$
\begin{array}{ll}
 & (F \,[\!]\, H).\text{transient} \\
= & \quad \{\text{union theorem, condition (3)}\} \\
 & F.\text{transient} \cup H.\text{transient} \\
\subseteq & \quad \{F.\text{transient} \subseteq G.\text{transient} \qquad , \text{from } F \Rightarrow G\} \\
 & G.\text{transient} \cup H.\text{transient} \\
= & \quad \{\text{union theorem, condition (3)}\} \\
 & (G \,[\!]\, H).\text{transient}
\end{array}
$$
□

Corollary: If $F = G$ then $F \,[\!]\, H = G \,[\!]\, H$.

Note: Our definition of refinement requires, implicitly, that a program name the same variables as the program it refines. A more general approach is to permit "data refinements" in which a group of variables is replaced by another group; the relation between the two groups of variables could be given by an invariant relation. See Knapp[6] for a discussion of refinement along these lines. □

5.5.3 Alternative Definition of Refinement

We propose an alternative definition of refinement that is weaker (more general) than the previous one. For programs F, G define $F \Rightarrow G$ by

$$G.IC \;\Rightarrow\; F.IC \;,\; F.\mathbf{co} \subseteq G.\mathbf{co} \;,\; F.lt \subseteq G.lt$$

where $F.lt$ is the set of pairs (p, q) where

$$p \;\mapsto\; q \quad \text{in } F.$$

As before, $F = G$ means $F \Rightarrow G$ and $G \Rightarrow F$. Clearly, "$\Rightarrow$" is reflexive and transitive and "=" is an equivalence relation. However, it is not the case that for any H,

if $(F \Rightarrow G)$ then $(F \,[\!]\, H \Rightarrow G \,[\!]\, H)$
or even,
if $(F = G)$ then $(F \,[\!]\, H = G \,[\!]\, H)$

We give a counterexample by showing three programs F, G, H where $F = G$ but $F \,[\!]\, H \neq G \,[\!]\, H$. Let F, G, H access a common integer variable, x, that takes on three possible values: 0, 1 or 2. Initially $x = 0$ in F, G and H. Program F has two transitions, α and β, where the values of $\alpha.x, \beta.x$ are

$x =$	0	1	2
$\alpha.x =$	2	1	2
$\beta.x =$	1	2	2

Table 5.1: Transitions of Program F

$x =$	0	1	2
$\alpha'.x =$	2	2	2
$\beta'.x =$	1	1	2

Table 5.2: Transitions of Program G

shown in Table 1. Similarly, program G has two transitions—α', β'—whose values are given in Table 2.

Programs F, G have the same set of **co**-properties because for any state, the set of possible next states in F, G are identical; this can be seen by comparing Tables 1 and 2, column by column. Now, the fair executions of F, G yield identical sequences of states; in both cases, the possible sequences of states are given by $0\ 1^*\ 2^\infty$; that is, initially $x = 0$, x is 1 for a finite number of transitions, x becomes 2 eventually, and it remains 2 forever. Since the sequence of states in fair executions of F, G are identical, $F.lt = G.lt$. Therefore, $F = G$, according to the proposed definition.

Now, consider a program H that has one transition, γ, as defined in Table 3. We will show that $(F \parallel H).lt \neq (G \parallel H).lt$. Therefore, $(F \parallel H) \neq (G \parallel H)$.

$x = 0 \ \mapsto\ x = 2$ in $G \parallel H$:

$x = 0 \ \vee\ x = 1$ transient	in G	, from Table 2
$x = 0 \ \vee\ x = 1$ transient	in $G \parallel H$	, union theorem (Condition 3)
$x = 0 \ \vee\ x = 1$ **en** $x = 2$	in $G \parallel H$	, definition of **en**
$x = 0 \ \vee\ x = 1 \ \mapsto\ x = 2$	in $G \parallel H$	, definition of $\mapsto$

$x =$	0	1	2
$\gamma =$	0	0	2

Table 5.3: Transitions of Program H

$x = 0 \not\mapsto\ x = 2 \quad \text{in } F \,[\!]\, H$:

We display an execution of $F \,[\!]\, H$ in which $x \neq 2$ holds at all times. Consider the fair execution sequence $(\beta\alpha\gamma)^{\infty}$ starting in $x = 0$; the sequence of states is $(0\ 1\ 1)^{\infty}$.

Discussion The notion of transient predicate encapsulates "atomic transitions." Our definition in Section 5.5.2 suggests that any refinement of a component that preserves atomicity also refines an asynchronous system. The weaker notion of refinement, as given in Section 5.5.3, is not required to preserve atomicity and the counterexample shows that the refinement of a component may not refine the system. Thus, the notion of atomicity—as embodied in transient and *ensures*—seems crucial in refinements of asynchronous systems.

There are occasions where we prefer to use the weaker notion of refinement (given in Section 5.5.3); in such cases, we are careful to note that the entire system should be refined as a whole, and not component by component.

5.6 Conditional Properties

5.6.1 Introduction

The union theorem, introduced in Section 5.2.4, is our main tool for the study of asynchronous compositions of programs. The major virtue of this theorem is that it provides a simple rule for deducing the **co**-properties and transient predicates of a system from those of its components. The major shortcoming of this theorem is that it does not provide a rule for deducing the *leads-to* properties of a system from those of its components (see Section 5.3.2). The only way we can apply the union theorem to $p \mapsto q$ in F, say, is to exhibit the proof of $p \mapsto q$ and then show that none of the other components falsify the proof steps. Such a strategy defeats the whole purpose of modular program construction. (Yet, this may be the only strategy when the components are "tightly-coupled," as is typically the case for examples such as mutual-exclusion. It is best to treat such a tightly-coupled system as a single component, in which all transitions of the components appear explicitly. Alternatively, **co**-properties and transient predicates of the components may first be deduced and the union theorem may then be applied.)

The difficulty that we have encountered is not merely due to the formalism we have adopted. The problem—how to design components of a system so that only limited knowledge of the other components is required during the design process—is fundamental. There is empirical evidence to support the view that such designs are possible; application programs are routinely developed for execution under operating systems given only the specification of the latter, in the form: "messages are eventually delivered," or "each resource request is eventually met." We propose a theory to facilitate such designs. The proposed theory builds only upon the logical operators introduced so far; there is no new basic concept. In the rest of this chapter, we treat "conditional properties," where the specification of a component F may include a property of $F \,[\!]\, G$ assuming certain properties of the "environment" G. We introduce "closure properties" in the next chapter, where we extend the observation that any proof of a component established through manipulations of local variables only, is a property of the system.

5.6.2 Conditional Properties

A property of program F of the form

$$\langle \forall\, G \;:\; \Pi \quad \text{in}\; G \;:\; \Pi' \quad \text{in}\; F \,[\!]\, G \rangle$$

is called a conditional property. Here, Π, Π' are the usual (unconditional) properties. The property Π is what F assumes about its environment G, and Π' is what it guarantees about the system, $F \,[\!]\, G$, provided the assumption is met. Note that G is bound (through universal quantification) in the above expression denoting that *any* program satisfying Π is a possible environment of F.

Given the code/specification of F, such a conditional property can be proven by assuming

$$\Pi \quad \text{in}\; G$$

for any arbitrary G, and proving Π' for $F \,[\!]\, G$. The main tool in such a proof is the union theorem. A small example is given later in this section.

A conditional property can be used in conjunction with properties of the other components to establish system properties: given

$$\Pi \quad \text{in}\; H$$

we can deduce

$$\Pi' \quad \text{in}\; F \,[\!]\, H$$

for the above conditional property.

Example 1 (Handshake Protocol): Program *send* sends messages to its environment, sending a subsequent message only after receiving acknowledgements for the previous one. We focus our attention on the number of messages sent, st, and the number acknowledged, ak.

Program *send*
  declare global st, ak : integer
  initially $st, ak = 0, 0$
  assign $st \ := \ st + 1 \quad \text{if } ak \leq st$
end

We claim that the following is a property of *send*. In the following, m, n are arbitrary integers.

$$\langle \forall\ G : \ ak, st = m, n \ \mathbf{co}\ ak, st = m, n \ \vee \ (ak, st = m+1, n \ \wedge \ m < n) \quad \text{in } G$$
$$: \ ak \leq st \leq ak + 1 \text{ invariant in } send \ [\!]\ G\rangle$$

Consider any G satisfying the premise. Then,

initially $ak \leq st$	, initially $ak, st = 0, 0$
$ak \leq st$ stable in *send*	, text of *send*
$ak \leq st$ stable in G	, from the premise (apply elimination theorem)
$ak \leq st$ invariant in *send* $[\!]$ G	
	, from the above three

The proof of $st \leq ak + 1$ in *send* $[\!]$ G is similar. A property that is more difficult to prove is

$$\langle \forall\ G \ : \ true \ \mapsto \ ak \leq st \quad \text{in } G \ : \ true \ \mapsto \ ak \leq st \quad \text{in } send \ [\!]\ G\rangle$$

We appeal to the following result, where x is the set of global variables declared in both *send* and G, p names only the (local and global) variables of G and b is an arbitrary predicate.

$$\frac{\begin{array}{c} p \ \mapsto \ q \quad \text{in } G \\ \neg b \ \wedge \ x = m \ \mathbf{co}\ x = m \ \vee \ b \quad \text{in } send \end{array}}{p \ \mapsto \ q \ \vee \ b \quad \text{in } send \ [\!]\ G}$$

Let
  $p \ \equiv \ true \quad q \ \equiv \ ak \leq st \quad b \ \equiv \ ak \leq st.$

consider any G in which $p \ \mapsto \ q$ holds.

$$
\begin{array}{ll}
ak, st = m, n \ \mathbf{co}\ ak, st = m, n \ \vee \ b \quad \text{in } \mathit{send} & \text{, text of } \mathit{send} \\
\neg b \ \wedge \ ak, st = m, n \ \mathbf{co}\ ak, st = m, n \ \vee \ b \quad \text{in } \mathit{send} & \text{, strengthening lhs} \\
p \ \mapsto \ q \quad \text{in } G & \text{, premise} \\
p \ \mapsto \ q \ \vee \ b \quad \text{in } \mathit{send} \ [\!]\ G & \text{, the above rule} \\
\mathit{true} \ \mapsto \ ak \leq st \quad \text{in } \mathit{send} \ [\!]\ G & \text{, rewriting } p, q, b \quad \Box
\end{array}
$$

The next example shows a progress proof in a network. The example is worth studying because it illustrates how the conditional properties of the different components are exploited to arrive at a conclusion about the system.

Example 2: A program *copy* has two variables: global x and local *out*. The job of *copy* is to take the value of x and append it to the sequence *out*; variable x has a special value, ϕ, to denote that x holds no useful value, i.e., its most recent value has been appended to *out*. It is expected that some other program writes into x. It is, however, crucial that the other program not change x as long as $x \neq \phi$; otherwise, a value in x will be overwritten without being copied into *out*.

In the following, m, n are of the same type as values written into x (including ϕ); B is a free variable of type sequence. We write $B; m$ to denote the sequence obtained by appending m to the back of B. Assume that $B; \phi = B$. Property C1 specifies how x and *out* are modified. Property C2 assures us that every item from x will be appended to *out* provided the environment of *copy* obeys the desired protocol for writing into x.

Specification of *copy* ::

$$x, \mathit{out} = m, B \ \mathbf{co}\ x, \mathit{out} = m, B \ \vee \ x, \mathit{out} = \phi, B; m \quad \text{in } \mathit{copy} \qquad \text{(C1)}$$

$$
\begin{array}{l}
\langle \forall\ G \ :\ \langle \forall\ n \ :\ n \neq \phi \ :\ x = n \text{ stable in } G \rangle \ : \\
\quad x, \mathit{out} = m, B \ \mapsto \ x, \mathit{out} = \phi, B; m \quad \text{in } \mathit{copy} \ [\!]\ G \\
\rangle
\end{array} \qquad \text{(C2)}
$$

Now, consider a program *pr* (for producer) that writes successive values of a local sequence *in* into x. This program obeys the expected protocol of writing into x: it stores the next item into x only if $x = \phi$. The program has a local variable, p, (of type natural) that is the number of values from *in* that have been written into x. The conditional property, $p2$, assumes that the environment of *pr* not write (non-ϕ value) into x, i.e., *pr* is the only producer for x; otherwise, the environment could always store some non-ϕ value into x, preventing *pr* from writing its next value. In the following, i is a free variable of type natural and $in[i]$ is the i^{th} item of *in*.

Specification of *pr*

$$p, x = i, m \textbf{ co } p, x = i, m \ \vee \ (m = \phi \ \wedge \ p, x = i+1, in[i]) \quad \text{in } pr \qquad (p1)$$

$$\langle \forall\ F \ : \ x = \phi \text{ stable in } F \ : \ p, x = i, \phi \ \mapsto \ p, x = i+1, in[i] \quad \text{in } pr \ [\!]\ F \rangle \qquad (p2)$$

We show, for any natural number j

$$p, x, out = 0, \phi, \langle\ \rangle \ \mapsto \ out = in^j \quad \text{in } copy\ [\!]\ pr \qquad (cp1)$$

where $\langle\ \rangle$ denotes the empty sequence, and in^j is the sequence of items $in[i], 0 \le i < j$, in the increasing order of i.

As would be expected, the proof of $cp1$ requires us to show that (1) the next item from in will be stored in x if $x = \phi$, from the specification of pr, and (2) the value in x will be appended to out and x set to ϕ, from the specification of *copy*. First, we prove certain aditional properties in order to exploit the conditional properties, $(c2, p2)$.

$(c3)$ $x = \phi$ stable in *copy*:
$x, out = \phi, B$ stable in *copy* , let $m = \phi$ in $(c1)$
$x = \phi$ stable in *copy* , disjunction over all B

$(cp2)$ $p, x = i, \phi \ \mapsto \ p, x = i+1, in[i]$ in $copy\ [\!]\ pr$:
From ($p2$ and $c3$) using *copy* for F.

$(p3)$ $\langle \forall\ n \ : \ n \neq \phi \ : \ x = n \text{ stable in } pr \rangle$:
$\langle \forall\ i, n \ : \ n \neq \phi \ : \ p, x = i, n \text{ stable in } pr \rangle$
, setting m to n and $n \neq \phi$ in $(p1)$
$\langle \forall\ n \ : \ n \neq \phi \ : \ \langle \exists\ i \ :: \ p, x = i, n \rangle \text{ stable in } pr \rangle$
, disjunction of above over all i
$\langle \forall\ n \ : \ n \neq \phi \ : \ x = n \text{ stable in } pr \rangle$
, simplifying

$(cp3)$ $x, out = m, B \ \mapsto \ x, out = \phi, B; m$ in $copy\ [\!]\ pr$:
From ($c2$ and $p3$) using pr for G.

$(cp4)$ $p, x, out = i, m, B \quad \textbf{co} \quad p, x, out = i, m, B \ \vee$
$(m = \phi \wedge p, x, out = i+1, in[i], B) \ \vee$
$p, x, out = i, \phi, B; m$ in $copy\ [\!]\ pr$:

From the union theorem, the above **co**-property has to be proven in each component, *copy* and *pr*.

In *copy* ::
$x, out = m, B \textbf{ co } x, out = m, B \ \vee \ x, out = \phi, B; m$

, $c1$

$p = i$ stable , p constant, from locality

$p, x, out = i, m, B$ **co** $p, x, out = i, m, B \ \vee \ p, x, out = i, \phi, B; m$

, stable conjunction

The required result follows by weakening the rhs.

In pr ::

$p, x = i, m$ **co** $p, x = i, m \ \vee \ (m = \phi \ \wedge \ p, x = i + 1, in[i])$

, $p1$

$out = B$ stable , out constant, from locality

$p, x, out = i, m, B$ **co** $p, x, out = i, m, B \ \vee$

$(m = \phi \ \wedge \ p, x, out = i + 1, in[i], B)$

, stable conjunction

The required result follows by weakening the rhs.

$(cp5)$ $p, x, out = i, \phi, B \ \mapsto \ p, x, out = i + 1, in[i], B$ in $copy \ [\!] \ pr$:

In $copy \ [\!] \ pr$ ::

$p, x, out = i, \phi, B$ **co** $p, x, out = i, \phi, B \ \vee \ p, x, out = i + 1, in[i], B$

, set m to ϕ in $(cp4)$

The result follows by applying PSP to above and $cp2$.

$(cp6)$ For $m \neq \phi$,

$p, x, out = i, m, B \ \mapsto \ p, x, out = i, \phi, B; m$ in $copy \ [\!] \ pr$:

In $copy \ [\!] \ pr$ ::

$p, x, out = i, m, B$ **co** $p, x, out = i, m, B \ \vee \ p, x, out = i, \phi, B; m$

, from $(cp4)$ using $m \neq \phi$

The result follows by applying PSP to above and $cp3$.

We are now ready to prove $cp1$. For any natural number j,

$(cp1)$ $p, x, out = 0, \phi, \langle \ \rangle \ \mapsto \ out = in^j$:

$p, x, out = i, \phi, in^j \ \mapsto \ p, x, out = i + 1, in[i], in^i$

, from $cp5$ setting B to in^i

$p, x, out = i + 1, in[i], in^i \ \mapsto \ p, x, out = i + 1, \phi, in^{i+1}$

, from $cp6$ setting i, m, B to $i + 1, in[i], in^i$

$p, x, out = i, \phi, in^j \ \mapsto \ p, x, out = i + 1, \phi, in^{i+1}$

, transitivity on the above two

$p, x, out = 0, \phi, \langle \ \rangle \ \mapsto \ p, x, out = j, \phi, in^j$

, induction □

Linear Networks

Properties of a regular network—a systolic array[7, 1], for instance—can often be established easily if each component of the network is specified by a suitable conditional property. The following rule applies to a linear network; here, for $0 \leq i \leq N$, F_is are programs and Π_is are properties.

$$\frac{\begin{array}{l} \Pi_0 \quad \text{in } F_0 \\ \langle \forall\, i \ :\ 0 \leq i < N \ : \\ \qquad \langle \forall\, G \ :\ \Pi_i \quad \text{in } G \ :\ \Pi_{i+1} \quad \text{in } F_{i+1} \,[\!]\, G \rangle \\ \rangle \end{array}}{\Pi_N \quad \text{in } \langle [\!]\, i \ :\ 0 \leq i \leq N \ :\ F_i \rangle}$$

To see this rule, deduce from the premises

$$\langle \forall\, j \ :\ 0 \leq j \leq N \ :\ \Pi_j \quad \text{in } \langle [\!]\, i \ :\ 0 \leq i \leq j \ :\ F_i \rangle \ \rangle$$

The conclusion follows by instantiating j by N. The proof of the above is by induction on j.

$j = 0 \ :\ \Pi_0 \quad \text{in } F_0$, premise

$j + 1, 0 \leq j < N \ :$

$\Pi_j \quad \text{in } \langle [\!]\, i \ :\ 0 \leq i \leq j \ :\ F_i \rangle$, induction hypothesis

$\Pi_{j+1} \quad \text{in } F_{j+1} \,[\!]\, \langle [\!]\, i \ :\ 0 \leq i \leq j \ :\ F_i \rangle$

, from the above and substituting $\langle [\!]\, i \ :\ 0 \leq i \leq j \ :\ F_j \rangle$ for G in the premise

$\Pi_{j+1} \quad \text{in } \langle [\!]\, i \ :\ 0 \leq i \leq j+1 \ :\ F_i \rangle$

, rewriting the above

The linear network rule imposes no constraints on the direction of "data-flow"; any program F_i may read/write variables changed by any other program. In the example below, program F_i communicates with both F_{i-1} and F_{i+1}, $0 < i < N$; yet, the apparent circularity is resolved by judicious use of conditional properties.

Example (Factorial Network): This example is from Misra and Chandy[2].

A network to compute factorials of a sequence of natural numbers is shown in Figure 5.5. Here, each program F_i receives a sequence of natural numbers in a global variable x_i (of type, sequence of natural); the program produces the sequence of their factorials (maintaining their proper order) in a global variable y_i. The computation strategy is as follows. In program

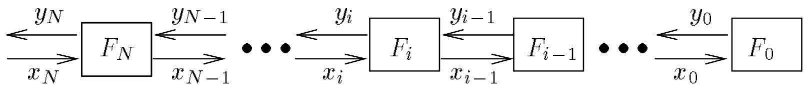


Figure 5.5: The structure of a factorial network.

F_i, $0 < i \leq N$, if a received number n is positive then $(n-1)$ is sent to F_{i-1} by appending n to x_{i-1}. Assuming that F_{i-1} delivers $(n-1)!$ eventually in y_{i-1}, program F_i then computes $n! = n \times (n-1)!$ and sends $n!$ in y_i. To maintain the proper order of outputs, F_i stores the sequence of all received numbers whose factorials have yet to be computed; if the head of the sequence is 0 then a 1 is appended to y_i; if the head of the sequence is nonzero then it is multiplied with the next received item from y_{i-1} and the result is appended to y_i. The last program, F_0, computes the factorial of each received number and appends it to y_0. (If the numbers sent to F_N are at most N then the numbers sent to F_0 are zeroes, and, hence, their factorials are easily computed.)

Notation: Let x_i denote the sequence of all numbers appended to x_i. For a sequence of natural numbers u, we write $u!$ for the sequence of their factorials (in the given order). For sequences u, v, $u \sqsubseteq v$ denotes that u is a prefix of v.

Consider the following specifications of F_i, $0 \leq i \leq N$; each predicate is an invariant of the appropriate program. The conditional property, below, states that F_{i+1} works appropriately: if it receives a prefix of $x_i!$ in y_i, it produces a prefix of $x_{i+1}!$ in y_{i+1}.

$$
\begin{array}{l}
y_0 \;\sqsubseteq\; x_0! \quad \text{in } F_0 \\
\langle \forall\, i \;:\; 0 \leq i < N \;: \\
\qquad\qquad \langle \forall\, G \;:\; y_i \;\sqsubseteq\; x_i! \quad \text{in } G \;:\; y_{i+1} \;\sqsubseteq\; x_{i+1}! \quad \text{in } F_{i+1} \,[\!]\, G \rangle \\
\rangle
\end{array}
$$

Then, applying the rule for linear networks we get

$$y_N \;\sqsubseteq\; x_N! \quad \text{in } \langle [\!]\; i \;:\; 0 \leq i \leq N \;:\; F_i \rangle$$

Similarly, consider the following progress properties (for any integer m, n, where $m, n \geq 0$). The conditional progress property states that if eventually there is an output in y_i for each input in x_i then eventually there is an output in y_{i+1} for each input in x_{i+1}.

$$|x_0| = m \;\mapsto\; |y_0| \geq m \quad \text{in } F_0$$

$$
\begin{array}{l}
\langle \forall\ i\ :\ 0 \le i < N\ :\langle \forall\ G\ : |x_i| = m\ \mapsto\ |y_i| \ge m \quad \text{in } G\ : \\
\qquad\qquad\qquad\qquad\qquad |x_{i+1}| = n\ \mapsto\ |y_{i+1}| \ge n \quad \text{in } F_{i+1} \, [\!] \, G \\
\qquad\qquad\qquad\quad \rangle \\
\rangle
\end{array}
$$

We conclude, for any m, $m \ge 0$

$$|x_N| = m\ \mapsto\ |y_N| \ge m \quad \text{in } \langle [\!]\ i\ :\ 0 \le i \le N\ :\ F_i \rangle$$

Concurrent Data Objects

A concurrent data object[11] is, merely, a global variable shared among several programs. Sometimes, it is useful to focus attention on a single variable (or a group of variables) and specify the manner in which it may be accessed (read or written). For instance, a producer and a consumer typically share a global variable called a buffer (or a queue). We may specify that (1) the queue length never exceeds a given implementation constraint, N, (2) the producer has to wait if it is attempting to add an item to a full queue, and (3) the consumer has to wait if it is attempting to remove items from an empty queue. It may be useful to hide the internal structure of the data object in its specification; thus "full" or "empty" may not appear in a specification of a queue.

A data object can be specified succinctly[4] by writing a set of equations that relate the effects of the various operations on this object. For instance, let *empty* denote an empty queue, $add(q, x)$ is the queue obtained by adding item x to the back of queue q and $remove(q)$ is the queue obtained by removing the first item of a nonempty queue q. Then, we have

$$
\begin{array}{l}
remove(add(empty, x)) = empty \quad \text{and,} \\
\text{for } q \ne empty,\ remove(add(q, x)) = add(remove(q), x)
\end{array}
$$

The implicit assumption in such a specification is that each operation represents a total function. Therefore, the operations have to be deterministic; the unordered channel (Section 4.5.3) where the message delivery order is arbitrary cannot be expressed in this formalism. Furthermore, each operation has to terminate; the fact that a producer has to wait for the queue to become nonfull (and hence, it may wait forever), is difficult to state. Finally, the above formalism assumes that the operations are performed in a strict order, i.e., the durations of two operations may not overlap; this contradicts the assumption that the operations may be concurrent, such as, appending to a full queue may overlap with a subsequent removal from the queue.

We specify a concurrent data object by treating it as a program. Other programs request access to the object by communicating with this program through certain global variables. For instance, a queue object may be represented by a program with the following global variables.

in: a program requests adding an item to the queue by storing its value in this variable; variable *in* has a special value when there is no outstanding request for addition

remove (type boolean): a program requests removal of the first item of the queue by setting *remove* to *true*; there is no request for removal if *remove* is *false*.

out: this is the value of the item last removed (this variable has a special value if no item has ever been removed).

Such a representation allows us to specify data objects in exactly the same way as programs. These specifications invariably include assumptions about the other programs; in the above example the programs must obey certain protocols in changing the variable *in*, for instance. Therefore, specifications typically include conditional properties.

A Note on Methodology: The protocols by which programs request access to a data object can be encoded as safety properties. Therefore, the conditional properties in the specification of a concurrent object are, invariably, of the form where the assumption is a safety property and the guarantee is a progress property. $\square$

In the rest of this section, we specify a concurrent bag that is shared among a group of producers and consumers. The schematic of the interaction between the programs is given in Figure 5.6. The set of producers and consumers, collectively, constitute Program F, the environment of B. Producers add items to the bag by successively storing them in r; consumers remove successive items from w. There may be several producers and consumers, or even a single program that is both the producer and the consumer; the exact number is irrelevant for the specification. We have seen such an interaction topology for client-servers in Section 5.3.5; here B is analogous to the program *Buff*. (Also, see Exercise 10.)

As in the Example 2 of Section 5.6.2, we postulate a special value, ϕ, for a variable to denote that its value is "empty," i.e., the variable contains no useful data. The protocol for reading, writing is as follows. Program F writes into r only if $r = \phi$; program B reads from r only if $r \neq \phi$ and it may set r to ϕ sometime after reading the value. Program B stores a value in w only if $w = \phi$; program F reads from w and it may set w to ϕ sometime

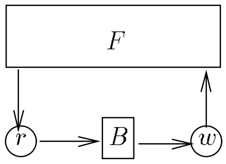


Figure 5.6: B is a concurrent bag that reads from r and writes into w. The environment F of B writes into r and reads from w.

after reading a non-ϕ value, to denote that it is ready to consume the next data item. Thus, the accesses of F, B are symmetric with respect to r, w.

Notation: All bags are finite. We treat r, w also as bags that may contain at most one item and ϕ is treated as an empty bag. Thus, $r = \phi$ denotes that r is an empty bag. Union of bags u, v is written as $u + v$. Note that $|u + v| = |u| + |v|$. □

The specification is simplified by introducing an order relation, $\prec$, over all the data values and ϕ, as follows.

$x \prec y \;\equiv\; x = \phi \;\wedge\; y \neq \phi$
Hence, $x \preceq y \;\equiv\; \{x = y \;\vee\; x \prec y \;\equiv\;\} \;\; x = \phi \;\vee\; x = y$

The following specification is of a bag B that can internally hold at most N items, $N \geq 0$. Let $\top$ denote the types of the elements of bag; variables R, W are free and they are of type $\top$.

Specification B
  declare local b: bag of $\top$,
    global r, w: $\top \cup \{\phi\}$
  initially $b + w = \phi$ in B $(B0)$
  $r \preceq R$ stable in B $(B1)$
  $W \preceq w$ stable in B $(B2)$
  $|b| \leq N$ invariant in B $(B3)$
  $r + b + w$ constant in B $(B4)$
  $\langle \forall\ F\ :\ R \preceq r$ stable in F, $w \preceq W$ stable in F:
    $|b + w| \leq N \;\mapsto\; r = \phi$ in $B \;[\!]\; F$ $(B5)$
    $|b + r| > 0 \;\mapsto\; w \neq \phi$ in $B \;[\!]\; F$ $(B6)$
  $\rangle$
end $\{B\}$

Bag b is the bag of data items internally stored by B. Property $B0$ says that both b and w are initially empty. Property $B1$ states that B can set r only to ϕ; similarly, $B2$ says that B can write only non-ϕ values into w

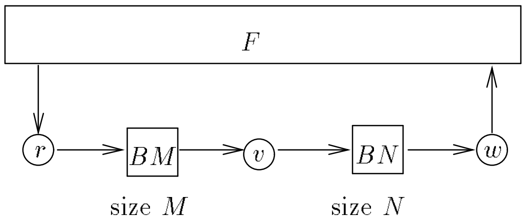


Figure 5.7: Concatenation of bags BM, BN of sizes M, N.

provided $w = \phi$. Observe that $(B2)$ prevents B from overwriting a non-ϕ value in w by either ϕ or another non-ϕ value. Property $(B3)$ specifies the size constraint on b. $(B4)$ is a conservation law: program B can neither create nor destroy any data in $r + b + w$. This allows B to freely move data between r, b, w; in particular, the order in which items are received in r may be different from the order in which they are sent via w. Note that $(B4)$ is symmetric in r, b, w, allowing data movement from w to b or r, for instance. However, $(B1, B2)$ constrain the directions of data movement. The conditional progress properties $(B5, B6)$ assume the environment to obey the access protocols for r, w; property $(B5)$ states that if there is room in b or if w is empty then the data in r, if any, will be eventually removed; property $(B6)$ is the counterpart $(B5)$ for writing into w.

From the locality of b, for the environment F
  $|b| \leq N$ stable in F
and hence, using $(B3)$
  $\langle \forall\ F\ \ ::\ \ |b| \leq N$ invariant in $B \parallel F \rangle$ $\qquad (B7)$

Bag Concatenation

We show that concatenations of two bags of sizes M, N, as shown in Figure 5.7, implements a bag of size $M + N + 1$. This result can be used to implement a bag of any size, k, $k > 0$, by concatenating $(k+1)$ bags of size 0.

Let BM, BN be programs implementing bags of size M, N, respectively. They can be concatenated provided the output global variable of one is the input of the other; Figure 5.7 shows the concatenation where r, v are the input, output variables of BM and v, w are the corresponding variables of BN. We define BMN to be $BM \parallel BN$ where v is a local variable of BMN. We show that BMN satisfies the specification B, with size $M + N + 1$.

First, we write the specifications of BM and BN. These are obtained from the specification B by substituting the appropratie input, output variables and size constraints. The following specification for BM is obtained from B by substituting bm, BM, v, V, M for b, B, w, W, N, respectively.

Specification BM
declare local bm: bag of $\top$,
global r, v: $\top \cup \{\phi\}$
initially $bm + v = \phi$ in BM $(BM0)$
$r \preceq R$ stable in BM $(BM1)$
$V \preceq v$ stable in BM $(BM2)$
$|bm| \leq M$ invariant in BM $(BM3)$
$r + bm + v$ constant in BM $(BM4)$
$\langle \forall\ G\ :\ R \preceq r$ stable in G, $v \preceq V$ stable in G:
$|bm + v| \leq M \ \mapsto\ r = \phi$ in $BM \ [\!]\ G$ $(BM5)$
$|bm + r| > 0 \ \mapsto\ v \neq \phi$ in $BM \ [\!]\ G$ $(BM6)$
$\rangle$
end $\{BM\}$

Similarly, we have the specification of BN obtained from ($B0$–$B6$) by substituting bn, BN, v, V for b, B, r, R.

Specification BN
declare local bn: bag of $\top$,
global v, w: $\top \cup \{\phi\}$
initially $bn + w = \phi$ in BN $(BN0)$
$v \preceq V$ stable in BN $(BN1)$
$W \preceq w$ stable in BN $(BN2)$
$|bn| \leq N$ invariant in BN $(BN3)$
$v + bn + w$ constant in BN $(BN4)$
$\langle \forall\ H\ :\ V \preceq v$ stable in H, $w \preceq W$ stable in H:
$|bn + w| \leq N \ \mapsto\ v = \phi$ in $BN \ [\!]\ H$ $(BN5)$
$|bn + v| > 0 \ \mapsto\ w \neq \phi$ in $BN \ [\!]\ H$ $(BN6)$
$\rangle$
end $\{BN\}$

Similar to ($B7$), we have

$\langle \forall\ G\ :\ |bm| \leq M$ invariant in $BM \ [\!]\ G \rangle$ $(BM7)$
$\langle \forall\ H\ :\ |bn| \leq N$ invariant in $BN \ [\!]\ H \rangle$ $(BN7)$

We have to show that BMN satisfies specification B, with N replaced by $M + N + 1$ (in $B3$ and $B6$). We define the local variable b to be

$$b \ \equiv\ bm + v + bn \qquad \text{(D)}$$

Proof of $(B0)$ $\quad$ **initially** $b + w = \phi$ $\quad$ in BMN:
All properties in this proof are in B.

initially $bm + v = \phi$	, from $BM0$
initially $bn + w = \phi$	, from $BN0$
initially $b + w = \phi$	, from the above two using (D)

Proof of $(B1)$ $\quad r \preceq R$ stable in BMN:

r constant in BN	, locality
$r \preceq R$ stable in BN	, from the above
$r \preceq R$ stable in BM	, $BM1$
$r \preceq R$ stable in BMN	, union-theorem corollary

Proof of $(B2)$ $\quad W \preceq w$ stable in BMN: similar to the proof of $B1$.

Proof of $(B3)$ $\quad |b| \leq M + N + 1$ invariant in BMN:

$\|bm\| \leq M$ invariant in $BM \, [\!] \, BN$	, from $BM7$ with BN for G
$\|bn\| \leq N$ invariant in $BM \, [\!] \, BN$	, from $BN7$ with BM for H
$\|b\| \leq M + N + 1$ invariant in B	, using (D)

Proof of $(B4)$ $\quad r + b + w$ constant in BMN:

$r + bm + v$ constant in BM	, $BM4$
$bn + w$ constant in BM	, locality
$r + b + w$ constant in BM	, from the above two using (D)
$r + b + w$ constant in BN	, similarly
$r + b + w$ constant in BMN	, union-theorem corollary

Proof of $(B5)$ $\quad \langle \forall\ F\ :\ R \preceq\ r$ stable in F, $w \preceq W$ stable in F $:$
$|b + w| \leq M + N + 1 \ \mapsto\ r = \phi \quad$ in $BMN \, [\!] \, F \rangle$ $:$

Consider any F satisfying the assumptions of $(B5)$.

From,

$R \preceq r$ stable in F	, assumption
$R \preceq r$ stable in BN	, locality

We have $R \preceq r$ stable in $BN \, [\!] \, F$. Also, $v \preceq V$ stable in $BN \, [\!] \, F$ from $(BN1)$ and the locality of v. Similarly, $V \preceq v$ stable in $BM \, [\!] \, F$ and $w \preceq W$ stable in $BM \, [\!] \, F$. Thus,

$BN \, [\!] \, F$ meets the assumptions for G in $(BM5, BM6)$, $\quad$ and
$BM \, [\!] \, F$ meets the assumptions for H in $(BN5, BN6)$.

In the following proof all properties are in $BM \, [\!] \, BN \, [\!] \, F$.

$$
\begin{array}{ll}
 & |b+w| \le M+N+1 \\
\Rightarrow & \{\text{from (D)}, b = bm+v+bn\} \\
 & |bm+v+bn+w| \le M+N+1 \\
\Rightarrow & \{\text{arithmetic}\} \\
 & |bm+v| \le M \ \lor \ |bn+w| \le N \\
\mapsto & \{\text{from } BN5, \text{ using } BM \ [\!] \ F \text{ for } H, \ |bn+w| \le N \ \mapsto \ v = \phi\} \\
 & |bm+v| \le M \ \lor \ v = \phi \\
\Rightarrow & \{v = \phi \ \Rightarrow \ \{\text{from } BM7\} \ (|bm+v| = |bM| \le M)\} \\
 & |bm+v| \le M \\
\mapsto & \{\text{from } BM5, \text{ using } BN \ [\!] \ F \text{ for } G\} \\
 & r = \phi
\end{array}
$$

Proof of $(B6)$: similar to that of $(B5)$ □

Data Refinement

Refinement of specifications will be studied in detail in Chapter ????. Here, we propose a refinement of Specification B. We prove the correctness of the refinement by showing that the original specification can be derived from the refined specification. We restrict attention to the case where the bag size, N, is zero. (As we have noted, any finite bag can be implemented by concatenating several zero-size bags.) The refined specification, given below, only permits movement of data from r to w provided $w = \phi$; furthermore, $r \neq \phi \ \land \ w = \phi$ cannot persist forever. In this specification, x, y are free variables of the same type as r and w.

Specification *Ref*
  declare global r, w: $\top \ \cup \ \{\phi\}$
  initially $w = \phi$ in B $(R0)$
  $r, w = X, Y$ **co** $r, w = X, Y \ \lor \ (Y = \phi \ \land \ r, w = \phi, X)$ in B $(R1)$
  $r \neq \phi \ \land \ w = \phi$ transient in B $(R2)$
end {*Ref*}

We have to prove $(B0$–$B6)$ given $N = 0$ (i.e., $b = \phi$), from $(R0$–$R2)$. Proof of $(B0)$ from $(R0)$ and $b = \phi$ is trivial. Proofs of $(B1, B2, B4)$ follow from $(R1)$ by employing the elimination theorem. We show one proof. (Such a specification, for a buffer, is given in the solution to Exercise 10.)

Proof of $(B4)$ $r + w$ constant {note: $b = \phi$}
For any z, a bag of type $\top$,

$$
\begin{array}{ll}
 & r + w = z \\
\textbf{co} & \{\text{elimination theorem on } R1\} \\
 & \langle \exists\, x, y \ : \ x + y = z \ : \ r, w = x, y \ \lor \ (y = \phi \ \land \ r, w = \phi, x)\rangle
\end{array}
$$

$\Rightarrow$ {predicate calculus}
$\langle \exists\ x, y\ ::\ x + y = z\ \wedge\ r, w = x, y \rangle$
$\vee\ \langle \exists\ x, y\ ::\ x + y = z\ \wedge\ y = \phi\ \wedge\ r, w = \phi, x \rangle$
$\Rightarrow$ {predicate calculus}
$r + w = z$

Therefore, $r + w$ constant. $\Box$

Proofs of $(B5, B6)$ appear as solution to Exercise 10.

Discussion

Equational notation, as in [4], is attractive for specifying data objects because it separates the concern of implementation from deductions of properties of the object. Concurrently accessed objects do not seem amenable to equational specification. We have proposed to specify such objects using traditional safety and progress properties. The specification of the bag $(B0$–$B6)$ is concise, and it makes concurrent access explicit. We have shown how to use such specifications to deduce properties of the object and carry out refinements of specifications.

The specification of the bag includes the declarations of the global variables—r, w—through which this program communicates with its environment. If multiple producers/consumers interact with the bag, they have to interleave their accesses to r and w. As we have shown in Section 5.????, a client-manager, CM, and a service-manager, SM, can be introduced to merge the various requests from producers into r and distribute the output from w to the consumers. Programs CM and SM can be specified independently.

Bibliographic Notes

The notion of union is from Chandy and Misra[3]; similar operators had been introduced earlier in CSP[5] and CCS[8]. Some of the difficulties regarding application of the substitution axiom have been noted in Sanders[12], Knapp[6] and Misra[10]. The example in Section 5.5.3 is based on Misra[9].

5.7 Exercises

1. A program F consists of the following statements

$$\begin{array}{llllll} & x & := & 0 & \text{if} & b \\ [] & x & := & 1 & \text{if} & \neg b \end{array}$$

where x is local to F.

Prove or disprove the following, where G is a program other than F.

(a) $(b \text{ stable} \quad \text{in } G) \;\Rightarrow\; (b \,\wedge\, x = 0 \text{ stable} \quad \text{in } F \,[\!]\, G)$

(b) $(b \text{ stable} \quad \text{in } G) \;\Rightarrow\; (x = 0 \text{ stable} \quad \text{in } F \,[\!]\, G)$

(c) $(b \text{ stable} \quad \text{in } G) \;\Rightarrow\; (b \;\mapsto\; x = 0 \quad \text{in } F \,[\!]\, G)$

(d) $(\mathit{true} \;\mapsto\; b \quad \text{in } G) \;\Rightarrow\; (\mathit{true} \;\mapsto\; x = 0 \quad \text{in } F \,[\!]\, G)$

2. Let program F be

Program F
  declare local x : integer,
  global y : integer
  assign
    $x := x - 1$
  $[\!]$ $y := -y \quad \text{if} \quad x \le 0 \,\wedge\, y > 0$
end $\{F\}$

Suppose that for a program G,

$$y \neq 0 \text{ stable} \quad \text{in } G.$$

Prove that

$$y \neq 0 \;\mapsto\; y < 0 \quad \text{in } F \,[\!]\, G.$$

3. (Modeling) The states of a client process, C, are either: *computing* (sometimes called *thinking*), *waiting* for a resource (called *hungry*) or *using* the resource (called *eating*). The behavior of the client is given by, (1) a computing process may only transit to waiting, (2) a waiting process remains waiting, (3) a process using the resource transits to computing eventually, in one atomic step. A server process, S, can affect the states of a client as follows: It can change only a waiting client to one using the resource; it can cause no other state transition.

State the given properties formally. Show that in $C \,[\!]\, S$ a process (1) remains computing unless it starts waiting, (2) remains waiting unless it starts using the resource, (3) continues using the resource unless it starts computing, and (4) the last transition happens eventually, in one atomic step.

4. Let programs F, G share only a single global variable, x. Suppose that G can only read the value of x but not write into it. Which of the following holds? Here, p, q name only the variables of F and r, s the variables of G.

$$\frac{p \textbf{ co } q \quad \text{in } F}{p \textbf{ co } q \quad \text{in } F \,[\!]\, G}\ ,\quad \frac{p \textbf{ en } q \quad \text{in } F}{p \textbf{ en } q \quad \text{in } F \,[\!]\, G}\ ,\quad \frac{p \mapsto q \quad \text{in } F}{p \mapsto q \quad \text{in } F \,[\!]\, G}$$

$$\frac{r \textbf{ co } s \quad \text{in } G}{r \textbf{ co } s \quad \text{in } F \,[\!]\, G}\ ,\quad \frac{r \textbf{ en } s \quad \text{in } G}{r \textbf{ en } s \quad \text{in } F \,[\!]\, G}\ ,\quad \frac{r \mapsto s \quad \text{in } G}{r \mapsto s \quad \text{in } F \,[\!]\, G}$$

Suppose further that F cannot modify x either. Which of the above holds?

5. Show that ($G.FP$ is the fixed point predicate of G)

(a)
$$\frac{p \textbf{ co } q \quad \text{in } F}{p \wedge G.FP \textbf{ co } q \quad \text{in } F \,[\!]\, G}$$

(b)
$$\frac{p \textbf{ en } q \quad \text{in } F}{p \textbf{ en } q \vee \neg G.FP \quad \text{in } F \,[\!]\, G}$$

(c)
$$\frac{p \mapsto q \quad \text{in } F}{p \mapsto q \vee \neg G.FP \quad \text{in } F \,[\!]\, G}$$

Hint: Use stability at fixed point rule (see Section 3.2.2).

Using the above show that

(d)
$$\frac{\begin{array}{c} p \mapsto q \quad \text{in } F \\ r \Rightarrow G.FP \\ r \text{ stable} \quad \text{in } F \end{array}}{p \wedge r \mapsto q \wedge r \quad \text{in } F \,[\!]\, G}$$

6. Let x be the set of variables declared global in both F, G. Let p be a predicate that names variables of F only (as well as free and bound variables). Show that,

$$\frac{p \ \circ \ q \quad \text{in } F}{p \ \wedge \ x = m \ \circ \ q \ \vee \ x \neq m \quad \text{in } F \parallel G}$$

where $\circ$ is **co**, **en** or $\mapsto$.

Hint: Use the following fact that appears in Chapter 3.

With p defined as above and y denoting the set of local and global variables of F

$$p \ \wedge \ y = n \ \mathbf{co} \ p \ \vee \ y \neq n \quad \text{in } F$$

7. The following rule can be used to establish a *leads-to* property of $F \parallel G$ from a *leads-to* property in component F and certain other safety properties. Prove the rule using the results in Exercise 6.

$$\frac{\begin{array}{c} p \ \mapsto \ q \quad \text{in } F \\ p \ \wedge \ x = m \ \mathbf{co} \ (p \ \wedge \ x \preceq \ m) \ \vee \ q \quad \text{in } F \parallel G \end{array}}{p \ \mapsto \ q \quad \text{in } F \parallel G}$$

where m is free, x denotes the variables declared global in both F and G, p names variables of F only (as well as free and bound variables), and the values assumed by x are well-founded under "$\prec$."

Give counterexample to the following conjecture.

$$\frac{\begin{array}{lll} p \ \wedge \ x = m & \mapsto & (p \ \wedge \ x \prec m) \ \vee \ q \quad \text{in } F \\ p \ \wedge \ x = m & \mathbf{co} & (p \ \wedge \ x \preceq m) \ \vee \ q \quad \text{in } G \end{array}}{p \ \mapsto \ q \quad \text{in } F \parallel G}$$

8. Consider the programs $semf1, semg1$ of Section 5.3.3. Show that

$$rf \ \vee \ rg \ \mapsto \ sh \neq N \quad \text{in } semf1 \parallel semg1$$

9. (Semaphore) It is required to implement a strong semaphore to be shared among N processes. Sketch a generalized version of the program given in Section 5.3.5.

10. In this exercise, we elaborate on the interface of program $BUFF$ described as a part of the client-server network in Section 5.3.5. The program in this exercise implements a one-place buffer as follows. $BUFF$ reads data from a variable r and writes the date into a variable w, all in one step (it has no additional internal storage). A variable, r or w, has a value ϕ when it holds no useful data. Program

BUFF obeys the following protocol: Whenever it reads any data from r it sets r to ϕ, and it writes into w only if $w = \phi$ (i.e., if w holds no data then). It is expected that the program *CM* (in the role of producer) writes into r only if $r = \phi$ and that the program *SM* (in the role of consumer) sets w to ϕ after reading data from it.

Propose a specification of *BUFF*. Assuming that

$$\begin{array}{ll} r \neq \phi \text{ stable} & \text{in } CM \,[\!]\, SM \\ \text{and } w = \phi \text{ stable} & \text{in } CM \,[\!]\, SM \end{array}$$

show that

$$\begin{array}{ll} w = \phi \;\mapsto\; r = \phi & \text{in } BUFF \,[\!]\, CM \,[\!]\, SM \\ r \neq \phi \;\mapsto\; w \neq \phi & \text{in } BUFF \,[\!]\, CM \,[\!]\, SM \end{array}$$

11. Introduce an operator, $||\!\rightarrow$, where

$$p \;||\!\rightarrow\; q \quad \text{in } F$$

means that $p \;\mapsto\; q$ in $F \,[\!]\, G$ for any G, where $F \,[\!]\, G$ is defined. Develop inference rules to prove $p \;||\!\rightarrow\; q$ in F from the text of F.

Note: Setting G to *id*, we have

$$\frac{p \;||\!\rightarrow\; q \quad \text{in } F}{p \;\mapsto\; q \quad \text{in } F}$$

12. Given programs F, G, H, J where $F \,[\!]\, H$, $G \,[\!]\, J$, $G \,[\!]\, H$ are defined, show that

$$\frac{F \;\Rightarrow\; G \;,\; H \;\Rightarrow\; J}{F \,[\!]\, H \;\Rightarrow\; G \,[\!]\, J}$$

Use the definition of "$\Rightarrow$" given in Section 5.5.2.

13. (Research) We have proposed that the variables of a program be classified local or global. Is there any advantage in refining the latter to read-only variables (i.e., a variable declared as read-only in F may only be read by programs other than F, while F may read and write into that variable) and read-write variables. Can you develop stronger inference rules for composition based on this classification?

5.8 Solutions

1. (a) The proof is as follows.

$$\begin{array}{ll}
b \text{ stable in } G & \text{, premise} \\
x = 0 \text{ stable in } G & \text{, locality axiom} \\
b \wedge x = 0 \text{ stable in } G & \text{, conjunction} \\
b \wedge x = 0 \text{ stable in } F & \text{, text of } F \\
b \wedge x = 0 \text{ stable in } F \,[\!]\, G & \text{, union-theorem corollary}
\end{array}$$

(b) The following execution sequence serves as a counterexample.

$$\{\neg b \wedge x = 0\} \;\; x := 1 \quad \text{if} \;\; \neg b \text{ (in } F) \;\; \{x = 1\}$$

(c) The proof is as follows.

$$\begin{array}{ll}
b \;\mathbf{en}\; x = 0 \text{ in } F & \text{, text of } F \\
b \text{ stable in } G & \text{, premise} \\
b \;\mathbf{en}\; x = 0 \text{ in } F \,[\!]\, G & \text{, union-theorem corollary} \\
b \mapsto x = 0 \text{ in } F \,[\!]\, G & \text{, definition of } \mapsto
\end{array}$$

(d) Let G have the statements

assign
  $b := true$
$[\!]$ $b := false$
end

Clearly, $true \mapsto b$ in G.

The following execution sequence of $F \,[\!]\, G$ serves as a counterexample to $true \mapsto x = 0$ in $F \,[\!]\, G$.

  $\{x = 1\}$
loop
  $b := true$ (in G)
; $b := false$ (in G)
; $x := 0$ if b (in F) $\{x$ is unchanged$\}$
; $x := 1$ (in F)
forever

2. Proof of $y \neq 0 \mapsto y < 0$ in $F \,[\!]\, G$:

$$\begin{array}{lll}
(1) & y \neq 0 \mapsto x \leq 0 \wedge y \neq 0 \text{ in } F \,[\!]\, G & \text{, proven below} \\
(2) & x \leq 0 \wedge y \neq 0 \mapsto y < 0 \text{ in } F \,[\!]\, G & \text{, proven below} \\
 & y \neq 0 \mapsto y < 0 \text{ in } F \,[\!]\, G & \text{, transitivity on (1,2)}
\end{array}$$

Proof of (1) $y \neq 0 \;\mapsto\; x \leq 0 \;\wedge\; y \neq 0 \quad \text{in } F \,[\!]\, G$:

$y \neq 0$ stable in F	, text of F
$y \neq 0$ stable in G	, given
$y \neq 0$ stable in $F \,[\!]\, G$	, union theorem
(1.1) $\mathit{true} \;\mapsto\; x \leq 0$ in $F \,[\!]\, G$	, proven below
$y \neq 0 \;\mapsto\; x \leq 0 \;\wedge\; y \neq 0$ in $F \,[\!]\, G$	, stable conjunction on the above two

Proof of (2) $x \leq 0 \;\wedge\; y \neq 0 \;\mapsto\; y < 0 \quad \text{in } F \,[\!]\, G$

$y \neq 0$ stable in G	, given
$x \leq 0$ stable in G	, locality axiom
$x \leq 0 \;\wedge\; y \neq 0$ stable in G	, stable conjunction
$x \leq 0 \;\wedge\; y \neq 0$ **en** $y < 0$ in F	, text of F
$x \leq 0 \;\wedge\; y \neq 0$ **en** $y < 0$ in $F \,[\!]\, G$	, union-theorem corollary
$x \leq 0 \;\wedge\; y \neq 0 \;\mapsto\; y < 0$ in $F \,[\!]\, G$	, definition of $\mapsto$

Proof of (1.1) $\mathit{true} \;\mapsto\; x \leq 0 \quad \text{in } F \,[\!]\, G$:

For any integer k,

$x = k$ **en** $x < k$ in F	, text of F
$x = k$ stable in G	, locality axiom
$x = k$ **en** $x < k$ in $F \,[\!]\, G$	, union-theorem corollary
$x = k \;\mapsto\; x < k$ in $F \,[\!]\, G$	, definition of $\mapsto$
$\mathit{true} \;\mapsto\; x \leq 0$ in $F \,[\!]\, G$	, induction on integers

3. Let t, h, e denote that the client is thinking, hungry or eating. Clearly,

$$t \;\vee\; h \;\vee\; e \;\equiv\; \mathit{true} \quad \text{and,} \tag{ST1}$$
$$\neg(t \;\wedge\; h)\;,\; \neg(h \;\wedge\; e)\;,\; \neg(t \;\wedge\; e) \tag{ST2}$$

The specification of the client is

t **co** $t \;\vee\; h$ in C (C1)
h stable in C (C2)
e **co** $e \;\vee\; t$ in C (C3)
e transient in C (C4)

The specification of the server describes all the forbidden state transitions; the only transition that is not explicitly forbidden is from h to e, and hence

t constant in S (S1)
e stable in S (S2)

Proof of t **co** $t \lor h$ in $C \,[\!]\, S$:
t stable in S , from S1
t **co** $t \lor h$ in $C \,[\!]\, S$, union-theorem corollary on (C1, above)

Proof of h **co** $h \lor e$ in $C \,[\!]\, S$:
$\neg t$ stable in S , from S1
$h \lor e$ stable in S , $\neg t \equiv h \lor e$ from (ST1, ST2)
h **co** $h \lor e$ in S , strengthening the lhs
h **co** $h \lor e$ in $C \,[\!]\, S$, union-theorem corollary on (C2, above)

Proof of e **co** $e \lor t$ in $C \,[\!]\, S$:
union-theorem corollary on (C3, S2).

Proof of e transient in $C \,[\!]\, S$:
union-theorem on (C4).

4. Since G can only read from x, predicate p is stable in G. Using the union-theorem corollary,

$$\frac{p \;\mathbf{co}\; q \quad \text{in } F}{p \;\mathbf{co}\; q \quad \text{in } F \,[\!]\, G} \quad \text{and} \quad \frac{p \;\mathbf{en}\; q \quad \text{in } F}{p \;\mathbf{en}\; q \quad \text{in } F \,[\!]\, G}$$

Also, applying induction on the structure of $p \mapsto q$ in F, we can show,

$$\frac{p \mapsto q \quad \text{in } F}{p \mapsto q \quad \text{in } F \,[\!]\, G}$$

Thus, G interferes not at all with F's execution.

However, none of the properties of G are, necessarily, inherited by $F \,[\!]\, G$. To see this, let x be a boolean variable and G consist of the assignment

$y := x$

where y is a local boolean variable. The following are easily proved from the text of G.

x stable in G
x **en** y in G
$x \mapsto y$ in G.

However, none of the above is a property of $F \,[\!]\, G$ where F consists of the assignment
$x \;:=\; \neg x$

If neither F nor G modifies x, applying the argument given above, every property of a component is a property of the system.

5. (a) p **co** q in F , premise
$p \wedge G.FP$ **co** q in F , strengthening lhs
$p \wedge G.FP$ stable in G , stability at fixed point in G
$p \wedge G.FP$ **co** q in $F \,[\!]\, G$, union-theorem corollary

(b) p **en** $q \vee \neg G.FP$ in $F \,[\!]\, G$
can be established, by using the union-theorem corollary, from

$p \wedge \neg q \wedge G.FP$ **co** $p \vee q \vee G.FP$ in F,
$p \wedge \neg q \wedge G.FP$ **co** $p \vee q \vee G.FP$ in G,
$p \wedge \neg q \wedge G.FP$ transient in F

The first property is proved by strengthening the lhs and weakening the rhs of the following (which is deduced from p **en** q in F)

$$p \wedge \neg q \textbf{ co } p \vee q.$$

The second property follows from stability at fixed point of G. The third property is proved by strengthening the following (which is deduced from p **en** q in F)

$$p \wedge \neg q \text{ transient in } F.$$

(c) Proof is by induction on the structure of $p \mapsto q$ in F.
- p **en** q in F: see solution to Exercise (5b).
- $p \mapsto r$ in F, $r \mapsto q$ in F:
  $p \mapsto r \vee \neg G.FP$ in $F \,[\!]\, G$, induction hypothesis
  $r \mapsto q \vee \neg G.FP$ in $F \,[\!]\, G$, induction hypothesis
  $p \mapsto q \vee \neg G.FP$ in $F \,[\!]\, G$, cancellation on the above two
- $p = \langle \exists\, i \;::\; p_i \rangle$ where $\langle \forall\, i \;::\; p_i \mapsto q$ in $F \rangle$:
  $\langle \forall\, i \;::\; p_i \mapsto q \vee \neg G.FP \rangle$ in $F \,[\!]\, G$
  , induction hypothesis
  $p \mapsto q \vee \neg G.FP$ in $F \,[\!]\, G$
  , disjunction on the above

(d)

r stable in F	, premise
$r \wedge G.FP$ **co** r in $F \parallel G$	, using (a) on the above
r stable in $F \parallel G$	, using $r \Rightarrow G.FP$ on the above
$p \mapsto q \vee \neg G.FP$ in $F \parallel G$	, using (c) on $p \mapsto q$ in F
$p \wedge r \mapsto q \wedge r$ in $F \parallel G$	, PSP on the above two; use $r \Rightarrow G.FP$

6. Let y be the set of local and global variables of G. Then, from the hint, for any n,

$$p \wedge y = n \ \mathbf{co}\ p \vee y \neq n \quad \text{in } G.$$

Since $x \subseteq y$, strengthening the lhs and weakening the rhs.

$$p \wedge x = m \ \mathbf{co}\ p \vee x \neq m \quad \text{in } G \tag{6.1}$$

(a) $$\frac{p \ \mathbf{co}\ q \quad \text{in } F}{p \wedge x = m \ \mathbf{co}\ q \vee x \neq m \quad \text{in } F \parallel G}$$

Proof:

$p \wedge x = m$ **co** q in F , strengthening lhs of the premise
$p \wedge x = m$ **co** $p \vee x \neq m$ in G
, from (6.1)
$p \wedge x = m$ **co** $q \vee p \vee x \neq m$ in $F \parallel G$
, union-theorem corollary
$p \wedge x = m$ **co** $q \vee x \neq m$ in $F \parallel G$
, from p **co** q in F, $p \Rightarrow q$.

(b) $$\frac{p \ \mathbf{en}\ q \quad \text{in } F}{p \wedge x = m \ \mathbf{en}\ q \vee x \neq m \quad \text{in } F \parallel G}$$

Proof:

$p \wedge \neg q$ **co** $p \vee q$ in F , from the premise
$p \wedge \neg q \wedge x = m$ **co** $p \vee q \vee x \neq m$ in $F \parallel G$ (6.2)
, applying (a) to the above

$p \wedge \neg q$ transient in F , from the premise
$p \wedge \neg q$ transient in $F \,[\!]\, G$, union theorem
$p \wedge \neg q \wedge x = m$ transient in $F \,[\!]\, G$
, strengthening
$p \wedge x = m$ **en** $q \vee x \neq m$ in $F \,[\!]\, G$
, from (6.2) and above

(c)
$$\frac{p \mapsto q \quad \text{in } F}{p \wedge x = m \mapsto q \vee x \neq m \quad \text{in } F \,[\!]\, G}$$

Apply induction on the structure on $p \mapsto q$ in F. Note that if $p \mapsto q$ in F is proved using $p \mapsto r$ in F and $r \mapsto q$ in F then predicate r mentions no variable of G other than x.

7. $p \mapsto q$ in F , premise
$p \wedge x = m \mapsto q \vee x \neq m$ in $F \,[\!]\, G$, using Exercise (6c)
$p \wedge x = m$ **co** $(p \wedge x \preceq m) \vee q$ in $F \,[\!]\, G$
, premise
$p \wedge x = m \mapsto (p \wedge x \prec m) \vee q$ in $F \,[\!]\, G$
, PSP on the above two
$p \mapsto q$ in $F \,[\!]\, G$, induction on the above

Counterexample to the conjecture: Let F, G share a global variable x that may assume values from $\{0,1,2\}$; let the well-founded order be defined by $0 \prec 1 \prec 2$. Program F consists of a single statement

$$x := (x+1) \mod 3$$

and G consists of

$$x := 1$$

With $p \equiv x = 1$ and $q \equiv x = 0$ we have, for all m, $m \in \{0,1,2\}$

$p \wedge x = m \mapsto (p \wedge x \prec m) \vee q$ in F
(i.e., $x = 1 \mapsto x = 0$ in F)
and $p \wedge x = m$ **co** $(p \wedge x \preceq m) \vee q$ in G
(i.e., $x = 1$ **co** $x = 1 \vee x = 0$ in G)
However, $p \mapsto q$ in $F \,[\!]\, G$
(i.e., $x = 1 \mapsto x = 0$ in $F \,[\!]\, G$)
does not hold, as can be seen by considering alternate executions of the statements of F, G starting at $x = 1$.

8. Show that

$$rf\ \mathbf{en}\ sh \neq N \quad \text{in } semf1\ [\!]\ semg1.$$

Prove this by applying the union theorem to

$rf \wedge sh = N$ **co** $rf \vee sh \neq N$ in $semf1$
$rf \wedge sh = N$ **co** $rf \vee sh \neq N$ in $semf2$
$rf \wedge sh = N$ transient in $semf1$

Similarly,
rg **en** $sh \neq N$ in $semf1\ [\!]\ semg1$
can be proven. The desired result follows by converting the two *ensures* propeties to *leads-to*s and taking their disjunction.

9. Let the processes be numbered 1 through N. The variable sh assumes a value between 0 and N; $sh = 0$ denotes that the semaphore has not been granted to any process; for $i \neq 0$, $sh = i$ denotes that process i holds the semaphore. Let $pr.i$, an integer, be the priority of process i. The strategy we adopt is that a process that is granted the semaphore is also given the lowest priority. Let low be an integer smaller than any $pr.i$. Boolean variable $r.i$, for process i, has the same role as rf or rg. The code for $sem.i$, $1 \leq i \leq N$, the program that interacts with the i^{th} process, is

Program *sem.i*

declare global $r.i$: boolean,
$pr.i, low, sh$: integer
initially $sh = 0, \langle \forall\ i\ ::\ pr.i = i \rangle, low = 0$
assign
$pr.i, low, r.i, sh\ :=\ low, low - 1, false, i$
if $r.i \wedge sh = 0 \wedge \langle \forall\ j\ ::\ pr.j \leq pr.i \vee \neg r.j \rangle$
$[\!]\ r.i, sh\ :=\ false, 0$ if $r.i \wedge sh = i$
end$\{sem.i\}$

10. We abbreviate *BUFF* to B and $CM\ [\!]\ SM$ to M. The safety property of B is that the only allowable changes in r, w is to transfer data from r to w setting r to ϕ; this change is allowed provided $w = \phi$ prior to the change. Thus, for any X, Y, ranging over data values and ϕ,

$$r, w = X, Y\ \mathbf{co}\ r, w = X, Y \vee (r, w = \phi, X \wedge Y = \phi) \quad \text{in } B \quad \text{(B1)}$$

(If $r = \phi$ or $w \neq \phi$ then both r, w retain their values.)

The progress property is that the state $r \neq \phi \;\wedge\; w = \phi$ does not persist because the data will be moved from r to w.

$$r \neq \phi \;\wedge\; w = \phi \text{ transient} \quad \text{in } B \tag{B2}$$

We are given

$$r \neq \phi \text{ stable} \quad \text{in } M \tag{M1}$$
$$w = \phi \text{ stable} \quad \text{in } M \tag{M2}$$

We prove below

$$r \neq \phi \;\wedge\; w = \phi \;\mathbf{co}\; r \neq \phi \;\equiv\; w = \phi \quad \text{in } B \;[\!]\; M \tag{BM1}$$
$$r \neq \phi \;\wedge\; w = \phi \text{ transient} \quad \text{in } B \;[\!]\; M \tag{BM2}$$

The desired progress properties can be proven using (BM1, BM2), as follows.

Proof of $w = \phi \;\mapsto\; r = \phi \quad \text{in } B \;[\!]\; M$:
All properties in the following proof are in $B \;[\!]\; M$.
$r \neq \phi \;\wedge\; w = \phi \;\mathbf{co}\; r = \phi \;\vee\; w = \phi$, weakening rhs of BM1
$r \neq \phi \;\wedge\; w = \phi$ transient , BM2
$w = \phi \;\mathbf{en}\; r = \phi$, definition of **en**
$w = \phi \;\mapsto\; r = \phi$, definition of $\mapsto$ □

Proof of $r \neq \phi \;\mapsto\; w \neq \phi \quad \text{in } B \;[\!]\; M$:
All properties in the following proof are in $B \;[\!]\; M$.
$r \neq \phi \;\wedge\; w = \phi \;\mathbf{co}\; r \neq \phi \;\vee\; w \neq \phi$, weakening rhs of BM1
$r \neq \phi \;\wedge\; w = \phi$ transient , BM2
$r \neq \phi \;\mathbf{en}\; w \neq \phi$, definition of **en**
$r \neq \phi \;\mapsto\; w \neq \phi$, definition of $\mapsto$ □

Proof of (BM1) $r \neq \phi \;\wedge\; w = \phi \;\mathbf{co}\; r \neq \phi \;\equiv\; w = \phi \quad \text{in } B \;[\!]\; M$:
In B

$$
\begin{array}{ll}
 & r \neq \phi \;\wedge\; w = \phi \\
\mathbf{co} & \{\text{elimination theorem on B1}\} \\
 & \langle \exists\, X, Y \;\; : (X \neq \phi \;\wedge\; Y = \phi \;\wedge\; r, w = X, Y) \\
 & \qquad\qquad \vee (X \neq \phi \;\wedge\; Y = \phi \;\wedge\; r, w = \phi, X) \\
 & \rangle \\
\Rightarrow & \{\text{predicate calculus}\}
\end{array}
$$

$$
\begin{array}{ll}
 & (r \neq \phi \ \wedge \ w = \phi) \ \vee \ (r = \phi \ \wedge \ w \neq \phi) \quad \text{in B} \\
\equiv & \quad \{\text{predicate calculus}\} \\
 & r \neq \phi \ \equiv \ w = \phi
\end{array}
$$

$$
\begin{array}{ll}
r \neq \phi \ \wedge \ w = \phi \text{ stable} \quad \text{in M} & \text{, conjunction of M1, M2} \\
r \neq \phi \ \wedge \ w = \phi \ \mathbf{co} \ r \neq \phi \ \equiv \ w = \phi & \text{in } B \ [\!] \ M \\
 & \text{, union-theorem corollary } \square
\end{array}
$$

Proof of (BM2) $r \neq \phi \ \wedge \ w = \phi$ transient in $B \ [\!] \ M$:
Applying union theorem on (B2) $\square$

11. Call a predicate *local* in a program if all program variables named in that predicate are local to the program. The proofs of the following rules are straightforward.

- $$\frac{\begin{array}{c} p \ \mathbf{en} \ q \quad \text{in } F \\ p \ , \ q \ \mathit{local} \quad \text{in } F \end{array}}{p \ || \!\rightarrow \ q \quad \text{in } F}$$

- $$\frac{p \ || \!\rightarrow \ q \quad \text{in } F \ , \ q \ || \!\rightarrow \ r \quad \text{in } F}{p \ || \!\rightarrow \ r \quad \text{in } F}$$

- $$\frac{\langle \forall \ p \ : \ p \in S \ : \ p \ || \!\rightarrow \ q \quad \text{in } F \rangle}{\langle \exists \ p \ : \ p \in S \ : \ p \rangle \ || \!\rightarrow \ q \quad \text{in } F}$$

12. $$
\begin{array}{ll}
 & F \ [\!] \ H \\
\Rightarrow & \{F \ \Rightarrow \ G. \text{ Applying the theorem in Section 5.5.2}\} \\
 & G \ [\!] \ H \\
\Rightarrow & \{H \ \Rightarrow \ J. \text{ Applying the theorem in Section 5.5.2}\} \\
 & G \ [\!] \ J
\end{array}
$$

13. See Chapter 6 for a more elaborate classification scheme and its ramifications.

Bibliography

[1] K. M. Chandy and J. Misra. Systolic algorithms as programs. *Distributed Computing*, 1:177–183, 1986.

[2] K. Mani Chandy and Jayadev Misra. Proofs of networks of processes. *IEEE*, SE,7(4):417–426, July 1981.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.

[4] John Guttag. Abstract data types and the development of data structures. *C.ACM*, 20(6):396–404, June 1977.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, London, 1984.

[6] E. Knapp. *Refinement as a Basis for Concurrent Program Design*. PhD thesis, The University of Texas at Austin, May 1992.

[7] H. T. Kung and C. E. Leiserson. Systolic arrays (for vlsi). In I. S. Duff and G. W. Stewart, editors, *Sparse Matrix Proc., 1978*, pages 256–282. Philadelphia, Pennsylvania: SIAM, 1979.

[8] R. Milner. *Communication and Concurrency*. International Series in Computer Science, C. A. R. Hoare, Series Editor. Prentice-Hall, 1989.

[9] J. Misra. The importance of ensuring. Notes on UNITY: 11–90, January 1990.

[10] J. Misra. Soundness of the substitution axiom. Notes on UNITY: 14–90, March 1990.

[11] J. Misra. Specifying concurrent objects as communicating processes. *Science of Computer Programming*, 14(10):159–184, October 1990.

[12] B. A. Sanders. Eliminating the substitution axiom from unity logic. *Formal Aspects of Computing*, 3:189–205, 1991.