Specification Structuring

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

April 26, 1999

1 Introduction

Programs that accept inputs, compute and produce outputs are specified by describing all possible inputs and the corresponding outputs. When the possible inputs are finite in number—as is the case with combinatorial circuits, for instance—the input,output pairs may be explicitly enumerated. However, for most programs it is convenient to describe input,output by logical propositions such as a precondition-postcondition pair, or the weakest precondition.

In this paper, we are interested in systems that typically run forever, responding to changes in some of their input variables by changing their internal states and/or some of their output variables. Example of such a system is a feedback controller that responds to a sensor reading by adjusting the valve opening, or a display program that responds to a keystroke by echoing the appropriate character on the screen. Such systems, called *reactive* by Amir Pnueli, are ubiquitous in all modern computer and communication systems. They cannot be specified merely by their input-output pairs, because input may be provided and output extracted on a continuing basis. Furthermore, these systems typically exhibit a high level of *nondeterminism* because the system may respond to inputs from many sources in an apriori undetermined order: A telephone switch that is connected to a number of users serves them in a seemingly random order.

We are interested in specifications of such systems for all the traditional reasons: We expect a specification to be a contract between the user and the implementer; the user may assume nothing more than what has been specified explicitly and the implementer must satisfy the specification through his implementation. This allows the user programs to change as long as they all obey the protocol set forth in the specification; similarly the implementation may change as long as it is faithful to the specification.

We view specification not merely as a legal contract but additionally as a means (1) to deduce new properties (of the "module" being specified), (2) to deduce properties of a system in which the given module is a component, and (3) to implement the module by stepwise refinements of its specification. Therefore, we require that a specification not merely be formal but also be in a form that admits of effective manipulation. This requirement rules out specification schemes in which program fragments (in some high level language) appear as part of specifications; typically, such program fragments cannot be manipulated effectively.

A specification of a large system will typically be large. Therefore, in the best traditions of software engineering, we have to *structure* the specification. Methodologies for program structuring have evolved over the years; now, we know that a program may be decomposed into modules based

*This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658–065 and by the Office of Naval Research Contract N00014-90-J-1640.

on control flow or data access or implementation of a set of related issues. We also know that the interfaces between modules should be narrow and well-defined. It is not clear how a specification ought to be structured. The structuring need not follow the program decomposition boundaries. For instance, we may view specification of user programs that access a common resource—say, a printer—as one module, and the specification that the access should be exclusive—i.e., the specification of mutual exclusion—as a separate module. However, when the user programs are implemented, the implementation of mutual exclusion may be intertwined with the code for the use of the printer.

In this paper, we explore a number of issues related to specifications and structuring of specifications. We specify a "module" by listing its properties; to describe the properties we use UNITY (Chandy and Misra [1988]). A short introduction to UNITY logic appears in Sec. 2. The logic is limited in the sense that it can express only some of the safety and progress properties. In particular, properties of the form, the program may deadlock, are not expressible within UNITY logic. Yet, the logic has been found useful in a number of application areas—from message communication to concurrent data access, from fault tolerance to systolic arrays. Sec. 3 contains specification of a finite state system—a vending machine; this specification is contrasted with an event-based specification. In Sec. 4, we show how to specify the problem of detecting termination of a message communicating system; this requires specification of the underlying message communications system as well as the requirements for termination detection. The specification problem for concurrent data objects is treated in Sec. 5; a simple example—a bag in which producers store items and from which consumers remove items—illustrates how objects, in general, may be succinctly specified. We discuss several aspects of specification structuring in Sec. 6. The material in Sec. 5 is mostly from Misra [1990].

2 A Brief Introduction to UNITY

A UNITY program consists of (1) declarations of variables, (2) a description of their initial values and (3) a finite set of statements. We shall not describe the program syntax except briefly in Sec. 5.10 of this paper because it is unnecessary for understanding this paper. However an operational description of the execution of a program is helpful in understanding the logical operators introduced later in this section.

An initial state of a program is a state in which all variables have their specified initial values (there can be several possible initial states if the initial values of some variables are not specified). In a step of program execution an arbitrary statement is selected and executed. Execution of every statement terminates in every program state. (This assumption is met in our model by restricting the statements to assignment statements only, and function calls, if any, must be guaranteed to terminate.) A program execution consists of an infinite number of steps in which each statement is executed infinitely often.

This program model captures many useful notions in programming, such as: synchrony, by allowing several variable values to be modified by a single (atomic) statement; asynchrony, by specifying little about the order in which the individual statements are executed; processes communicating through shared variables, by partitioning the statements of the program into subsets and identifying a process with a subset; processes communicating via messages, by restricting the manner in which shared variables are accessed and modified, etc. We shall not describe these aspects of the model; however, it will become apparent from the examples in this paper that process networks can be described effectively within this model.

2.1 UNITY logic

A fragment of UNITY logic, that is used in this paper, is described in this section.

Three logical operators, *unless*, *ensures*, and *leads-to*, are at the core of UNITY logic. Each of these is a binary relation on predicates; *unless* is used to describe the safety properties, and the other two to describe progress properties of a program.

Notation: Throughout this section p, q, r denote predicates which may name program variables,

bound variables and free variables (free variables are those that are neither program variables nor bound variables). □

2.1.1 unless

For a given program

$p \;\; unless \;\; q$

denotes that once predicate p is true it remains true at least as long as q is not true. Formally (using Hoare's notation) p *unless* q may be deduced given that if $p \wedge \neg q$ holds prior to execution of any statement of the program then $p \vee q$ holds following the execution of the statement.

$$\frac{\langle \text{for all statements } s \text{ of the program} \;\; :: \;\; \{p \;\wedge\; \neg q\} \;\; s \;\; \{p \;\vee\; q\}\rangle}{p \;\; unless \;\; q}$$

It follows from this inference rule that if p holds and q does not hold in a program state then in the next state either q holds, or $p \wedge \neg q$ holds; hence, by induction on the number of statement executions, $p \wedge \neg q$ continues to hold as long as q does not hold. Note that it is possible for $p \wedge \neg q$ to hold forever. Also note that if $\neg p \vee q$ holds in a state then p *unless* q does not tell us anything about the next state.

Notation: We write $\langle \forall u \;\; :: \;\; P.u \rangle$ and $\langle \exists u \;\; :: \;\; P.u \rangle$ for universal quantification of u in $P.u$ and existential quantification of u in $P.u$, respectively. The dummy u could denote a variable, a statement in a program, or even a program. Any property having a free variable (i.e., a variable that is neither bound nor a program variable) is assumed to be universally quantified over all possible values of the variable. Thus,

$u = k \;\; unless \;\; u > k$

where k is free, is a shorthand for

$\langle \forall \; k \;\; :: \;\; u = k \;\; unless \;\; u > k \rangle$. □

Examples

1. Integer variable u does not decrease.
   $u = k \quad unless \quad u > k$
   or, $u \geq k \quad unless \quad false$

2. A message is received (i.e., predicate *rcvd* holds) only if it had been sent earlier (i.e., predicate *sent* already holds).
   $\neg rcvd \quad unless \quad sent$ □

Example (Defining Auxiliary Variables)

Let u be an integer-valued variable and let v count the number of times u's value has been changed during the course of a program execution. The value of v is completely defined by

initially $\quad v = 0$
$u = m \;\wedge\; v = n \;\; unless \;\; u \neq m \;\wedge\; v = n + 1$

The traditional way to define v, given a program in which u is a variable, is to augment the program text with the assignment,

$v \;:=\; v + 1$

whenever u's value is changed. Variable v is called an *auxiliary variable*. Our way of defining v, without appealing to the program text, is preferable because it provides a direct relationship between u and v which may be exploited in specifications and proofs. □

2.1.2 Stable, Constant, Invariant

Some special cases of *unless* are of importance. The predicate p *unless* $false$, from definition, denotes that p remains true forever once it becomes true; we write, "p stable" as a shorthand for "p *unless* $false$." An expression e is *constant* means $e = x$ is stable, for all possible values x of e; then e never changes value. If p holds in every initial state and p is stable then p holds in every state during any execution; we then say that p is *invariant*.

2.1.3 ensures

For a given program,

$$p \;\; ensures \;\; q$$

implies that p *unless* q holds for the program and if p holds at any point in the execution of the program then q holds eventually. The inference rule that allows us to deduce p *ensures* q is

$$\frac{p \;\; unless \;\; q \;\wedge\; \langle \exists \text{ statement } s \;\; :: \;\; \{p \;\wedge\; \neg q\} \;\; s \;\; \{q\} \rangle}{p \;\; ensures \;\; q}$$

It follows from this inference rule that once p is true it remains true at least as long as q is not true (from p *unless* q). Furthermore, from the rules of program execution, statement s will be executed sometime after p becomes true. If q is still false prior to the execution of s then $p \wedge \neg q$ holds and the execution of s establishes q.

2.1.4 leads-to

For a given program, p *leads-to* q, abbreviated as $p \mapsto q$, denotes that once p is true, q is or becomes true. Unlike *ensures*, p may not remain true until q becomes true. The relation *leads-to* is defined inductively by the following rules. The first rule is the basis for the inductive definition of *leads-to*. The second rule states that *leads-to* is a transitive relation. In the third rule, $p.m$, for different m, denote a set of predicates. This rule states that if every predicate in a set *leads-to* q then their disjunction also *leads-to* q.

(basis)

$$\frac{p \;\; ensures \;\; q}{p \;\; \mapsto \;\; q}$$

(transitivity)

$$\frac{p \;\; \mapsto \;\; q \;,\; q \;\; \mapsto \;\; r}{p \;\; \mapsto \;\; r}$$

(disjunction) In the following m ranges over any arbitrary set, and it does not occur free in q.

$$\frac{\langle \forall \; m \;\; :: \;\; p.m \;\; \mapsto \;\; q \rangle}{\langle \exists \; m \;\; :: \;\; p.m \rangle \;\; \mapsto \;\; q}$$

Notes on Inference Rules

We have explained the meaning of each logical operator in terms of program execution. However, neither the definitions nor our proofs make any mention of program execution. We use only the inference rules, and a few rules derived from these definitions, in proofs; we believe that our proofs are succinct because we avoid operational arguments about program executions.

2.1.5 Derived Rules

The following rules for *unless* are used in this paper; for their proofs, see Chandy and Misra [1988] and Misra [1988].

(reflexivity, antireflexivity)

$$p \;\; unless \;\; p \;,\; p \;\; unless \;\; \neg p$$

(consequence weakening)

$$\frac{p \;\; unless \;\; q \;,\; q \;\Rightarrow\; r}{p \;\; unless \;\; r}$$

(general conjunction and general disjunction)

In the following, m is quantified over some arbitrary set.

$$\frac{\langle \forall\, m \;\; :: \;\; p.m \;\; unless \;\; q.m \rangle}{\begin{array}{llll} \langle \forall\, m \;::\; p.m \rangle & unless & \langle \forall\, m \;::\; p.m \;\vee\; q.m \rangle \;\; \wedge \;\; \langle \exists\, m \;::\; q.m \rangle & \{\text{general conjunction}\} \\ \langle \exists\, m \;::\; p.m \rangle & unless & \langle \forall\, m \;::\; \neg p.m \;\vee\; q.m \rangle \;\; \wedge \;\; \langle \exists\, m \;::\; q.m \rangle & \{\text{general disjunction}\} \end{array}}$$

(Corollary: free variable elimination)

In the following, x is a set of program variables and k is free.

$$\frac{p \;\wedge\; x = k \;\; unless \;\; q}{p \;\; unless \;\; q}$$

(Corollary: stable conjunction)

$$\frac{p \;\; unless \;\; q \;,\; r \text{ stable}}{p \;\wedge\; r \;\; unless \;\; q \;\wedge\; r}$$

We need the following facts about constants; see Misra [1989b] for proofs.

(constant formation)

Any expression built out of constants and free variables is a constant.

(constant introduction) For any function f over program variables u,

$$\frac{u = k \;\; unless \;\; u \neq k \;\wedge\; f(u) = f(k)}{f \text{ constant}}$$

Corollary 1: For any predicate p over program variables u,

$$\frac{u = k \;\; unless \;\; u \neq k \;\wedge\; p}{p \text{ stable}}$$

We use the following results about *leads-to*.

(Implication)

$$\frac{p \;\Rightarrow\; q}{p \;\mapsto\; q}$$

The following rule allows us to deduce a progress property from another progress property and a safety property.

(PSP)

$$\frac{p \;\mapsto\; q \;,\; r \;\; unless \;\; b}{p \;\wedge\; r \;\mapsto\; (q \;\wedge\; r) \;\vee\; b}$$

2.1.6 Substitution Axiom

Substitution axiom allows us to replace an invariant by *true* and vice versa, in any predicate. Thus, if I is an invariant and it is required to prove that

$$p \;\mapsto\; q \;\wedge\; I$$

it suffices to prove

$$p \;\mapsto\; q \;.$$

2.2 Program Composition through *union*

Given two programs F, G, their *union*, written $F \;[]\; G$, is obtained by appending their codes together: the initial conditions of both F, G are satisfied in $F \;[]\; G$ (and hence, we assume that initial conditions of F, G are not contradictory) and the set of statements of $F \;[]\; G$ is the union of the statements of F and G.

Programs F, G may be thought of as executing asynchronously in $F \;[]\; G$. The union operator is useful for understanding process networks where each process may be viewed as a program and the entire network is their union.

The following theorem is fundamental for understanding union. It says that an *unless* property holds in $F \;[]\; G$ iff it holds in both F and G; an *ensures* property holds in $F \;[]\; G$ iff the corresponding *unless* property holds in both components and the *ensures* property holds in at least one component. (When there are multiple programs we write the program name with a property, such as p *unless* q in F.)

Union Theorem:

$$\begin{array}{lll} p \;\; \textit{unless} \;\; q \;\; \text{in } F \;[]\; G & \equiv & p \;\; \textit{unless} \;\; q \;\; \text{in } F \;\wedge\; p \;\; \textit{unless} \;\; q \;\; \text{in } G \\ p \;\; \textit{ensures} \;\; q \;\; \text{in } F \;[]\; G & \equiv & (p \;\; \textit{unless} \;\; q \;\; \text{in } F \;\wedge\; p \;\; \textit{ensures} \;\; q \;\; \text{in } G) \\ & & \vee \;\; (p \;\; \textit{ensures} \;\; q \;\; \text{in } F \;\wedge\; p \;\; \textit{unless} \;\; q \;\; \text{in } G) \end{array}$$

Corollary 1:

$$\frac{p \;\; \textit{unless} \;\; q \;\; \text{in } F \;,\; p \text{ stable} \;\; \text{in } G}{p \;\; \textit{unless} \;\; q \;\; \text{in } F \;[]\; G}$$

Corollary 2:

$$\frac{e \text{ constant} \;\; \text{in } F \;,\; e \text{ constant} \;\; \text{in } G}{e \text{ constant} \;\; \text{in } F \;[]\; G}$$

Corollary 3:

$$\frac{p \text{ stable} \;\; \text{in } F \;,\; p \text{ invariant} \;\; \text{in } G}{p \text{ invariant} \;\; \text{in } F \;[]\; G}$$

Corollary 4:

$$\frac{p \;\; \textit{ensures} \;\; q \;\; \text{in } F \;,\; p \text{ stable} \;\; \text{in } G}{p \;\; \textit{ensures} \;\; q \;\; \text{in } F \;[]\; G}$$

Note: When several programs are composed, the substitution axiom can be applied only with an invariant of the composite program. Thus, in proving p *unless* q in F we cannot appeal to the substitution axiom using an invariant of F, if F has to be composed with another program.

2.2.1 Conditional Properties

The properties that we have described so far are of the form,

P in F

This states an *unconditional* property of F. We will find it useful to consider properties such as,

$\langle \forall\ G \ \ ::\ \ Q \text{ in } G \ \Rightarrow\ P \text{ in } F \ \ [\!]\ \ G \rangle$

This says that if Q is a property of a program G then P is a property of $F \ [\!]\ G$. Since G is universally quantified above, the given property is a *conditional* property of F. Conditional properties are used to specify the properties of the environment with which F may be composed, and the resulting properties of the composite program. Such properties will be crucial in specifications of reactive systems because reactive systems typically assume certain facts about their environments.

2.3 A Specification Format

A specification (of a program) is a list of its properties. The properties are expressed as formula in UNITY-logic. Properties may name free and bound variables as well as local and global variables of the program. We declare all these variables explicitly in a specification.

A specification has four parts:

1. The name of the specification and a list of its parameters (along with their types). This is much like a procedure name and its parameter list. The parameters are the global variables of the program, i.e., the variables that may be accessed by other programs, or the ones that are to be replaced by constants to create an instance of this specification.

2. A declaration section in which the free, bound, local and auxiliary variables are defined. Program names that appear free in the specification are declared here.

3. A hypothesis section—written as **hypo**—that states the antecedents of all the conditional properties. The assumptions about other programs—for instance, the environment of this program—as well as definitions of auxiliary variables are listed here.

4. A conclusion section—written as **conc**—that states the consequents of all the conditional properties, as well as all the unconditional properties. These are the properties that have to be established by the program given that the properties in the hypothesis section hold.

A specification does not explicitly name the program it specifies; the symbol '*' is used in the body of the specification to refer to this program.

As an example, consider the following specification.

spec $test(y$: integer)
  declare
    free G : program
  hypo
    $y \leq 0$ stable in G
  conc
    $y = 0 \ \mapsto\ y < 0$ in $* \ [\!]\ G$
end $\{test\}$

This specification says that if the environment, G, of the given program (referred to by '*') never changes y from nonpositive to positive then in the combined program (i.e., $* \ [\!]\ G$) starting in a state where $y = 0$ a state will be reached where $y < 0$. The variable y is, presumably, shared between $*$ and G. A given program satisfies this specification if we can deduce from the program code and the given hypothesis, **hypo**, that the conclusion, **conc**, holds.

Convention About the Scope of Free Variables

The free variables are quantified universally over *both* the hypothesis and the conclusion. As an example, suppose y is a parameter, k is a free variable, both of type integer, and G is a free variable of type program. The **hypo** and **conc** are as follows.

hypo $y \leq k$ stable in G
conc $y = k \mapsto y < k$ in $* \,[\!]\, G$

We interpret the specification to mean

$\langle \forall\, G\,,\ k\ ::$
$\quad y \leq k$ stable in G
$\Rightarrow\ y = k \mapsto y < k$ in $* \,[\!]\, G$
$\rangle$

If it is desired to express the following property

$\langle \forall\, G\ ::$
$\quad \langle \forall\, k\ ::\ y \leq k$ stable in $G\rangle$
$\Rightarrow\ \langle \forall\, k\ ::\ y = k \mapsto y < k$ in $* \,[\!]\, G\rangle$
$\rangle$

then k should be declared as a bound variable and the **hypo** and **conc** should read as follows.

hypo $\langle \forall\, k\ ::\ y \leq k$ stable in $G\rangle$
conc $\langle \forall\, k\ ::\ y = k \mapsto y < k$ in $* \,[\!]\, G\rangle$

Convention About Naming Programs with Properties

A property written without a program name would be taken to be a property of program '*'.

Locality Rule

A variable local to one program is constant in all other programs. More generally, an expression that names no parameter, local or auxiliary variable of program F is constant in F.

2.4 Proving That a Program Meets a Specification

We view a specification as a conditional property. To show that a given program implements this conditional property we use the technique given in Sec. 7.2.3 of Chandy and Misra [1988]. Briefly, we take the **hypo**-section as given and then use the program text to deduce the **conc**-section. However, since the program and the specification may name different variables it is required to establish a correspondence between them; the following rules are used.

- The parameters of the program and the specification have to be identical in number and type, much like the formal and actual parameters of procedures. The program parameters are replaced by the specification parameters in order to carry out the proof. Equivalently, axioms of the form $x = y$ can be added where x, y are the corresponding program and specification parameters.
- A mapping function should be specified to map the local variables of the program to the local variables of the specification; this is in the spirit of Hoare [1972] where such functions were introduced to map concrete representations to abstract representations of data. Thus, if b is a local variable in a specification that is implemented by a pair of local variables, say y, z, of the program where

$b = y + z$

we add $b = y + z$ as an axiom.

- No special treatment is required for auxiliary variables, because these are defined by the properties in the **hypo**-section.
- No special treatment for free or bound variables are required.

The same technique applies in proving that a specification T is a refinement of specification S. Essentially, we have to show that $T \Rightarrow S$. Therefore, as before, from the **hypo**-section of S and T we deduce the **conc**-section of S. Variable correspondence has to be established as given above.

3 Specification of a Vending Machine

The following example has been inspired by Hoare [1984].

A vending machine has a coin slot into which a user can put one or two small coins or one large coin. Two small coins are equal in value to a large coin. The vending machine has two buttons, for requesting a small or a large chocolate. If the 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 small chocolate is desired and a large coin (or two small coins) have been deposited then the machine dispenses the chocolate as well as a small coin.

The behavior of the user and the vending machine can be described by the diagram in Figure 1. Each state is a vertex; edges represent state transition. The label on an edge is the name of the event that causes the state transition. The initial state—designated by a circle—is at the top.

The diagram of Fig. 1 is a succinct description of a finite state device. The transitions in the diagram are effected by the user or the machine. The specification is best understood as defining certain sequences of events to be allowable. Thus,

uds uds url mdl

is an allowable sequence whereas

uds mdl

is not an allowable sequence.

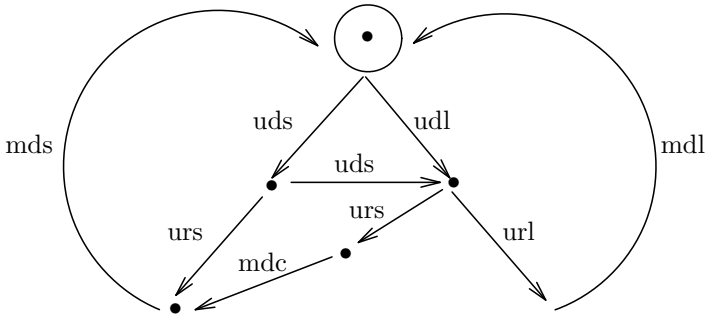
The specification constrains not only the behavior of the machine but also the behavior of user, such as: The user is not allowed to deposit a small coin and demand a large chocolate; the behavior of the machine in such a case is undefined.

3.1 A Specification in UNITY

Our approach to specification of the vending machine is to clearly separate the two components of the system: *user* and VM (for vending machine). Each of these components can be regarded as a program. These two programs interact through the following shared variables

ci {coin input} : $(\bot, s, l)$
b {button} : $(\bot, s, l)$
d {dispenser} : $(\bot, s, l)$
co {coin output} : boolean

The variable ci models the coin slot which either holds no coin ($\bot$), a small coin (s) or a large coin (l); two small coins are treated as a large coin. The two buttons are modeled by a variable b whose value is $\bot$ when no button has been pressed, s when a small chocolate has been requested and l when a large chocolate has been requested; the possibility of requesting both chocolates has



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

Figure 1: A Vending Machine

been ignored. The dispenser for chocolate is modeled by the variable d with values $\perp$ (no chocolate has been dispensed), s (a small chocolate has been dispensed) and l (a large chocolate has been dispensed). The coin dispenser, *co*, is set *true* when a coin is dispensed.

In the following description, we define $\perp < s < l$. Only the user program may increase *ci*. Also, the user program may change b from $\perp$ to s or $\perp$ to l; however, the user program may not change b from s to l or to $\perp$. The user program may set d to $\perp$ (by removing the chocolate) or *co* to *false* (by removing the dispensed coin); we do not, however, need to specify all these properties of the user in specifying the vending machine.

First, we investigate the safety properties of the vending machine; how does the machine change values of the variables ci, b, d, co? Observe that the machine may change these values only if the button is pressed (i.e., $b > \perp$) and at least the required amount of money has been deposited (i.e., $ci \geq b$). In this case the machine will set *ci* to $\perp$, b to $\perp$, d to the desired type of chocolate (the old value of b) and *co* to *true* provided the old value of *ci* exceeded that of b. Formally (using free variables CI and B)

$$(ci, b) = (CI, B) \;\; unless \;\; CI \geq B > \perp \;\wedge (ci, b, d, co) = (\perp, \perp, B, CI > B) \quad \text{in } VM \tag{1}$$

Next, we investigate the progress properties of VM. We require that the vending machine dispense the appropriate chocolate and the change, provided an adequate amount of money has been deposited and the proper button pressed, i.e.,

$$\begin{array}{ll} \langle \forall\ F & :: \quad (ci, b) = (CI, B) \;\wedge\; CI \geq B > \perp \;\mapsto \\ & \qquad (ci, b, d, co) = (\perp, \perp, B, CI > B) \quad \text{in } F \;\; \square \;\; VM \\ \rangle & \end{array} \tag{2}$$

Unfortunately, this property is impossible to implement if F operates in an arbitrary manner. For instance, consider a user that deposits a small coin—setting *ci* to s—and then requests a small chocolate—setting b to s. According to (2), it is then guaranteed that eventually ci, b will both become $\perp$. However, the following scenario is possible. Since we have not constrained the user in any way, the user may remove money from the coin slot, thereby setting *ci* to $\perp$ while b remains set to s. Under this condition, (1) guarantees that VM will change neither *ci* nor s (because the right side of the property, $CI \geq B$, is now *false*). Thus the required progress property cannot be established by any action of VM.

This situation is typical in specifications of reactive systems. Some property of the environment—in this case, the user—has to be assumed before a progress property of the entire system can be asserted. For this example, we assume that the user can only increase ci, and once the user presses a button, either s or l, it cannot change the selection. Formally, in the user program F for all k where k is $\perp, s$ or l,

$$\begin{array}{l} ci \geq k \text{ stable} \quad \text{in } F, \\ b = s \text{ stable} \quad \text{in } F, \\ b = l \text{ stable} \quad \text{in } F \end{array}$$

Under these conditions we require

$$(ci, b) = (CI, B) \ \wedge \ CI \geq B > \perp \ \mapsto \ (ci, b, d, co) = (\perp, \perp, B, CI > B) \quad \text{in } F \ [] \ VM$$

The complete specification for the vending machine is given below.

spec A $(ci, b, d \ : \ (\perp, s, l), \ co \ : \text{boolean})$
declare
free CI, B : $(\perp, s, l)$,
bound k : $(\perp, s, l)$,
free F : program
hypo
$\langle \forall \ k \ :: \ ci \geq k$ stable in $F \rangle$
$b = s$ stable in F
$b = l$ stable in F
conc
initially $(ci, b, d, co) = (\perp, \perp, \perp, \mathit{false})$
$(ci, b) = (CI, B)$ *unless* $CI \geq B > \perp \ \wedge (ci, b, d, co) = (\perp, \perp, B, CI > B)$
$(ci, b) = (CI, B) \ \wedge \ CI \geq B > \perp \ \mapsto \ (ci, b, d, co) = (\perp, \perp, B, CI > B) \quad \text{in } * \ [] \ F$
end $\{A\}$

Observe that the initial condition and the *unless* property of *, given in **conc**, have to be established from the text of * alone, whereas in proving the *leads-to* property of $* \ [] \ F$, given in **conc**, we can make use of the properties of F, given in **hypo**, as well.

3.2 Implications of the Specification

The specification A of the vending machine describes what the machine is supposed to do in certain situations for which the diagram of Fig. 1 provides no information. In particular, specification A allows the user to press a button and then deposit the appropriate coin. Certain undesirable behaviors of the user will now be treated as type violations—for instance, user depositing too many coins may cause ci to assume a value outside its type—rather than as a violation of the specification. In any case, the specification of the vending machine restricts the user behavior only minimally.

It is interesting to prove that specification A meets our intuition; for instance, we may want to show that if $ci = s$ and $b = s$ then a small chocolate will be dispensed, i.e.

$$(ci, b) = (s, s) \quad \mapsto \quad d = s \quad \text{in } F \ [] \ VM$$

provided F meets the given constraints in **hypo** and VM satisfies specification A.

From the progress property in the specification, by instantiating both CI and B to s, we get

$$(ci, b) = (s, s) \ \wedge \ s \geq s > \perp \ \mapsto \ (ci, b, d, co) = (\perp, \perp, s, CI > B) \quad \text{in } F \ [] \ VM$$

Since $s \geq s > \perp$, we simplify the lhs to $(ci, b) = (s, s)$. We weaken the rhs, using the implication rule, to $d = s$. This gives us, as desired,

$$(ci, b) = (s, s) \quad \mapsto \quad d = s \quad \text{in } F \ \square \ VM$$

It is also easy to see that program VM never increases ci, i.e.

$$ci \leq CI \text{ stable in } VM$$

To prove this property, we use Corollary 1 of Sec. 2.1.5: for variables u, predicate p over u and free variables k,

$$\frac{u = k \ \ unless \ \ u \neq k \ \wedge \ p}{p \text{ stable}}$$

From the *unless* property in **conc** of specifcation A, weakening the consequence

$$(ci, b) = (CI, B) \ \ unless \ \ (ci, b) \neq (CI, B) \ \wedge \ ci \leq CI$$

Therefore, $ci \leq CI$ stable in VM. Similarly, $b \leq B$ stable in VM—i.e., the machine never presses the button—can be established.

4 Specification of Termination Detection in a Ring

A message communicating system consists of a set of processes connected by directed channels. Each channel is directed from exactly one process to another; the messages in the channel are the ones sent by the former process to the latter which are, as yet, undelivered. A process is in one of two states: *idle* (i.e., quiescent) or *nonidle*. An idle process sends no message, and it remains idle as long as it receives no message. A nonidle process may become idle (autonomously). A message is received along a channel only if it had been sent along that channel. The system is *terminated* when all processes are idle and all channels are empty (i.e., have no messages in them) because all processes will stay idle—since the first process to become nonidle has to first receive a message—and all channels will remain empty—since idle processes send no message. It is required to develop an algorithm by which processes can determine that the system is terminated.

One of the more difficult aspects of this problem is to specify it precisely: What is given, what is to be designed, and what properties should the design satisfy. We specify the problem in this section. We do not propose algorithms for this problem; the reader may consult Misra [1989a] or Chandy and Misra [1990] for algorithms.

In this paper, we limit ourselves to a system in which the processes are connected in a unidirectional ring; the general case, described and solved in Chandy and Misra [1990], is almost analogous though more notation is needed in that case. Our purpose here is primarily to acquaint the reader with UNITY-style specification, problem decomposition and proofs, and only secondarily with the problem of termination detection.

4.1 Specification of the Underlying Message Communication System

Let there be N processes arranged in a unidirectional ring, the successor of the i^{th} process has index i'. The following variables are defined—for all i (in this problem i is quantified over all indices in the given ring):

$s.i$, the number of messages sent by i (to i')
$r.i$, the number of messages received by i (from its predecessor)
$q.i$, a boolean variable that is *true* if and only if process i is idle

There is no notion of a process in UNITY. Hence we regard the entire process network as a program, D, that manipulates the variables given above, for all i. The following properties of D have been described earlier: the number of messages sent by a process is at least the number received by its successor, and both these numbers are nonnegative ($D1$, given below); the number of messages

sent and received by a process are nondecreasing ($D2$); a process remains idle as long as it receives no message ($D3$); and an idle process has to become nonidle first in order to send a message ($D4$). In the following specification s, r, q are defined to be arrays which could be indexed by process indices.

spec *ring* (s, r : array[process] of nat, q : array [process] of boolean)
declare
free i : process,
free m, n : nat
conc
$\{D1\}$ $s.i \geq r.i' \geq 0$ invariant
$\{D2\}$ $r.i \geq m$ stable
$s.i \geq n$ stable
$\{D3\}$ $q.i \ \wedge \ r.i = m$ *unless* $r.i > m$
$\{D4\}$ $q.i \ \wedge \ s.i = n$ *unless* $\neg q.i \ \wedge \ s.i = n$
end $\{ring\}$

Assume that the given program, D, meets the specification *ring*, i.e., the properties, $D1 - D4$, are the only properties of D that we will assume. There are several aspects of the specification that are worth noting. First, we do not assume that the channels are FIFO, i.e., messages sent along a channel may be delivered in a different order from the sending order. Second, there is no guarantee that a message will be delivered. Third, several processes may receive and/or send messages in one step; however a process may not receive a message, become nonidle and send a message all in one step (from $D4$, a process can send a message in a step provided it is nonidle at the beginning of the step). The goal of the specification is to make the minimum number of assumptions about D that is required in order to detect its termination.

Note: $D3, D4$ may be replaced by their conjunction

$$q.i \ \wedge \ r.i = m \ \wedge \ s.i = n \ \textit{unless} \ r.i > m \ \wedge \ s.i = n.$$

It can be shown that the above property is equivalent to $D3$ and $D4$. □

4.2 Definition of Termination

We have informally described termination as "all processes are idle and all channels are empty (i.e., for each channel, the number of messages sent equals the number received)." We will show that none of the variables can change once the termination condition holds.

Let,

$$T \ \equiv \ \langle \forall \ i \ :: \ q.i \ \wedge \ s.i = r.i' \rangle$$

Observe that proving T to be stable only establishes that no $q.i$ will change (become *false*) once T holds; however, it is possible for $r.i, s.i$ to change while preserving $s.i = r.i'$, for all i. Hence, we prove below that $T \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$ is stable where $m.i$'s are free variables.

$D5$. $T \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$ is stable in D.

Proof of $D5$: The result is proven by taking the conjunction of $D3, D4$ and then applying the general conjunction rule over all i. In the following, all properties are of D.

$q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i'$ *unless* $r.i > m.i \ \wedge \ s.i = m.i'$
, replacing m, n by $m.i, m.i'$ in $D3, D4$ and then taking their conjunction
$\langle \forall \ i \ :: \ q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i' \rangle$ *unless*
$\langle \forall \ i \ :: \ (q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i') \ \vee \ (r.i > m.i \ \wedge \ s.i = m.i') \rangle \ \wedge$
$\langle \exists \ i \ :: \ r.i > m.i \ \wedge \ s.i = m.i' \rangle$

, taking general conjunction of the above over all i (1)

The left side of (1)
$\equiv \quad \langle \forall\ i\ ::\ q.i\ \wedge\ s.i = r.i' \rangle\ \wedge\ \langle \forall\ i\ ::\ r.i = m.i \rangle$
$\equiv \quad T\ \wedge\ \langle \forall\ i\ ::\ r.i = m.i \rangle$

The first conjunct in the right side of (1) $\Rightarrow\ \langle \forall\ i\ ::\ s.i = m.i' \rangle$
The second conjunct in the right side of (1) $\Rightarrow\ \langle \exists\ i\ ::\ r.i > m.i \rangle$
Hence the right side of (1)
$\Rightarrow \quad \langle \forall\ i\ ::\ s.i = m.i' \rangle\ \wedge\ \langle \exists\ i\ ::\ r.i > m.i \rangle$
$\Rightarrow \quad \langle \exists\ i\ ::\ r.i' > m.i'\ \wedge\ m.i' = s.i \rangle$
, replacing $\langle \exists\ i\ ::\ r.i > m.i \rangle$ by $\langle \exists\ i\ ::\ r.i' > m.i' \rangle$
$\equiv \quad \mathit{false}$
, using the substitution axiom and $s.i \geq r.i'$ from $(D1)$ □

Note: In using the substitution axiom in the last step of proof of $D5$, we have anticipated that s, r will not be changed by the environment of D and hence $s.i \geq r.i'$ is an invariant of the entire program (i.e., D and its environment). □

$D6$. T is stable in D.

Proof of $D6$: Eliminate the free variables $m.i$ (using the corollary from Sec. 2.1.5) from $D5$. □

4.3 Specification of Termination Detection

Our goal is to design a program R that detects termination of D. More precisely, let VT be a variable that only R can change. We have to design R such that

$VT \Rightarrow\ T$ invariant in $R\ []\ D$
and $T \quad \mapsto\ VT$ in $R\ []\ D$

The first property says that if VT is *true* then D is terminated. The second property guarantees that VT will be set *true* eventually if T becomes *true*. More formally, the following specification (of R) states that it can access s, r, q but not modify them, and, if the specification *ring* is met by program D then the occurrence of T (in D) will be detected by VT (in $R \,[\!]\, D$). (The symbol, T, is an abbreviation for a boolean expression, as defined in Section 4.2.)

spec *detect* (s, r : array [process] of nat,
q : array [process] of boolean,
VT : boolean)
declare
free D : program
hypo
ring in D
conc
s, r, q constant
$VT\ \Rightarrow\ T$ in $* \,[\!]\, D$
$T\ \mapsto\ VT$ in $* \,[\!]\, D$
end {*detect*}

This type of specification of a program to be designed (in this case, R) is quite common: We are given a specification of an environment in which the program is to operate (in this case D), a list of properties of the program to be designed and the properties of the overall system.

5 Specifications of Concurrent Objects

An object and its associated operations may be specified in many ways. One way is to give an abstract representation of the object data structure (viz., representing a queue by a sequence) and the effects of various operations on this abstract representation (Hailpern [1982], Hayes [1987], Lamport [1989]). Another way (Guttag [1977]) is to leave the representation aspects unspecified but to give a set of equations that relate the effects of various operations (the equations define a congruence relation in a term algebra). In many such specification schemes it is assumed that (1) each operation on an object is deterministic (i.e., applying the operation to a given state of the object results in a unique next state and/or unique values being returned), (2) an operation once started always terminates in every state of the object, and (3) operations are not applied concurrently. In many cases of interest arising in applications such as operating systems, process control systems and concurrent databases, these assumptions are rarely met. For instance, a "queue object" acts as an intermediary between a producer and a consumer, temporarily storing the data items output by the producer and later delivering them to the consumer. The queue object is required to (1) deliver data in the same order to the consumer as they were received from the producer, (2) receive data from the producer provided its internal queue spaces are nonfull, and (3) upon demand, send data to the consumer provided its internal queue spaces are nonempty. Requests from the producer and the consumer may be processed concurrently: The producer is delayed until there is some space in the queue and the consumer is delayed until there is some data item in the queue. Observe that a request from the producer, to add a data item to the queue, may not terminate if the queue remains full forever, and similarly, a request from the consumer may not terminate if the queue remains empty forever.

In this section, we propose a specification scheme for concurrent objects that allows effective manipulations of specifications and admits nondeterministic, nonterminating and concurrent operations on objects. Our approach is to view a concurrent object as an asynchronous communicating process. Such a process can be specified by describing its initial state, how each communication—send or receive—alters the state, and the conditions under which a communication action is guaranteed to take place. On the other hand, since the internal state can be determined from the sequence of communications (provided the process is deterministic), the process can also be specified in terms of the sequence of communications in which it engages. The point to note is that the internal state and the sequence of communications can be viewed as *auxiliary variables* which may be altered as a result of communications. Our specification scheme allows us to define auxiliary variables and state properties using these variables. We advocate using any auxiliary variable that allows a simple and manipulable specification, leaving open the question whether it is preferable to use "observable" input-output sequences or "unobservable" internal states.

The main example treated in this paper is a bounded bag (or multiset). Sec. 5.1 contains a methodology of specification; Sections 5.2 and 5.3 contain, respectively, an informal description and a formal specification of this object. We demonstrate the usefulness of the specification by showing (in Sec. 5.5) that concatenations of two bags results in a bag of the appropriate size. The specification in Sec. 5.3 is based on an auxiliary variable that encodes the internal state. We provide an alternative specification in Sec. 5.7 that uses the sequences of input/output as auxiliary variables; we show that this specification is a refinement of the earlier one. Sec. 5.8 contains specifications for a "queue" and a "stack." We show that a queue implements a bag and a stack implements a bag, showing in each case that the specification of the former implies the specification of the latter. Sec. 5.9 contains a refinement of the bag specifications of Sec. 5.3, and Sec. 5.10 gives an implementation of the specification of Sec. 5.9.

Most proofs are given in complete detail to emphasize that such proofs need not be excessively long or tedious, as is often the case with formal proofs. A preliminary version of this work appears in Misra [1990]. The specifications and the proofs have been considerably simplified in the current version.

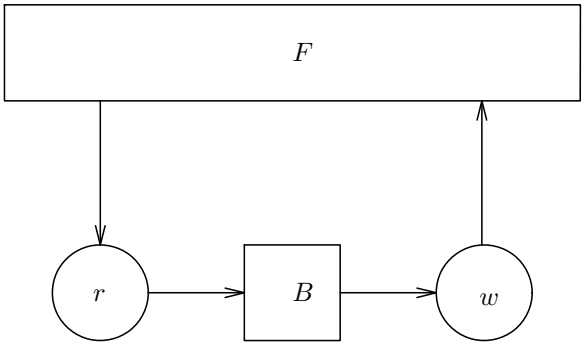


Figure 2: A Program B and its environment F

5.1 Specification Methodology

We treat a concurrent object as an asynchronous communicating process. A process and its environment communicate over channels that can typically hold several items of data. To simplify matters, we dispense with channels; we assume that communications are through certain shared variables each of which can hold at most one item of data. The access protocol to the shared variables is as follows: An "input shared variable" of the object can be written only by the environment and read by the object, an "output shared variable" of the object can be written only by the object and read by the environment. Typically, we introduce a shared variable corresponding to each operation on the object (if an operation also delivers a result, we may require two shared variables—one to request the operation and the other to store the result).

In Figure 2, B implements a bag that is shared by a group of producers and consumers. Producers add items to the bag by successively storing them in r; consumers remove successive items from w. Program F is the environment of B, representing the producers and the consumers. There might be multiple producers and consumers or even a single process that is both the producer and the consumer; the exact number is irrelevant for the specification. The values that can be written into r, w are, again, irrelevant for specification. However we do postulate a special value, ϕ, which is written into a variable to denote that it is "empty," i.e., it contains no useful data. The protocol for reading and writing is as follows. Program F writes into r only if $r = \phi$; program B reads a value from r only if $r \neq \phi$ and then it may set r to ϕ. Program B stores a value in w only if w is ϕ; program F reads from w and it may set w to ϕ to indicate that it is ready to consume the next piece of data.

Note: A more realistic model would assume that each producer and each consumer has a separate variable through which it communicates with B; i.e., the i^{th} producer writes into a variable r_i and the j^{th} consumer reads from w_j. Figure 3 shows the pattern of communication in this case. Here C stands for a "collector" program that copies some r_i into r (never ignoring any r_i forever), and D is a "distributor" program that copies w to some empty w_j (again in a fair manner).

The specification of the bag program in this case consists of the specifications of B, C and D. The specifications of C, D are quite straightforward; that is why we concentrate on the specification of B alone. □

To formalize the access protocol, we introduce an ordering relation, $\prec$, over the data values and ϕ as follows: ϕ is "smaller than" all non-ϕ values, i.e.,

$$X \prec Y \quad \equiv \quad X = \phi \ \wedge \ Y \neq \phi$$

Then it follows that

$$X \preceq Y \quad \equiv \quad X = \phi \ \vee \ X = Y$$

Property P1, below, states that B removes only non-ϕ data from r and it may set r to ϕ. Property P2 states that B writes only non-ϕ values into w provided w is ϕ. In the following, R, W are free variables (of type data; we assume that ϕ is a value of type data).

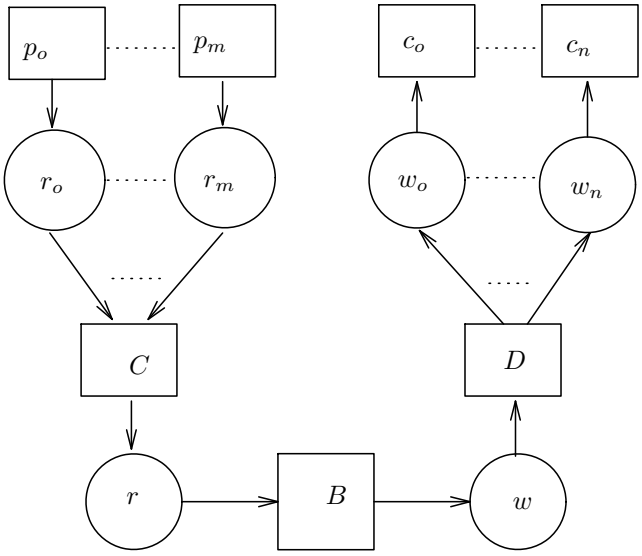


Figure 3: A bag that interacts with multiple producers and consumers

P1. $r \preceq R$ stable in B
P2. $W \preceq w$ stable in B

The access protocol specification as given by P1 and P2 is identical for all concurrent objects. The initial conditions can usually be specified for output shared variables; for this example

P0. initially $w = \phi$ in B

Next, we specify the safety properties—the effect of reading inputs or how the outputs are related to the current state—and progress properties—the conditions under which inputs are read and outputs are written. These properties are specific to each object. They are explored next for bags.

5.2 Internal State as an Auxiliary Variable (An Informal Specification)

In the following specification we introduce an auxiliary variable, b, that represents the bag of data items internally stored by B. Variable b is local to B. Properties P0, P1, P2 have been described before. The size of b does not exceed N, a given positive integer (Property P3). Property P4 states that any item read from r is added to b or stored in w (and r is then set to ϕ) and any item removed from b is stored in w (w was ϕ previous to this step); hence the union of bags r, b, w remains constant. Property P5 states that independent of the environment if the bag is nonfull ($|b| < N$) an item, if there is any, will be removed from r and analogously, if the bag is nonempty ($|b| > 0$) w is or will become non-ϕ.

Notation: All bags in this paper are finite. We treat r, w as bags of size at most 1. Union of bags u, v is written as $u + v$; the bag $u - v$ is the bag obtained by removing all common items of u and v from u. The value ϕ is treated as an empty bag.

5.3 Specification of a Concurrent Bag of Size $N, N > 0$

spec P (r, w : data, N : positive constant)
  declare
    local b : bag of data,
    free R, W : data,
    free F : program

conc
{P0} initially $b + w = \phi$
{P1} $r \preceq R$ stable
{P2} $W \preceq w$ stable
{P3} $|b| \leq N$ invariant
{P4} $r + b + w$ constant
{P5} {P5.1} $|b| < N \mapsto r = \phi$ in $* \,[\!]\, F$
{P5.2} $|b| > 0 \mapsto w \neq \phi$ in $* \,[\!]\, F$
end $\{P\}$

Program B satisfies specification P if it is possible to "construct" b from the local variables of B such that the constraints P0–P5 are met. The program may not actually have a variable named b; it is merely required that the value of b be computable from the values of the local variables of B.

Since b is a local variable of B, it is not accessed by the environment F. Using locality rule, $|b| \leq N$ stable in F. Since $|b| \leq N$ is invariant in B, using Corollary 3 of Sec. 2.2,

P6. $\langle \forall\ F \ :: \ |b| \leq N$ invariant in $B \ [\!]\ F \rangle$

Observe that P5 is a property of program B; it says that if B is composed with any program F then certain properties hold in the composite program, $B \ [\!]\ F$. In particular, P5 does not require F to obey the appropriate protocols in accessing r, w. A different bag specification is given in Sec. 5.7 where F is assumed to access r, w appropriately.

5.4 Deducing Properties from the Specification

Suppose program B satisfies specification P. We deduce one safety and one progress property.

P7. $w = \phi$ *unless* $|b| < N \ \vee \ r = \phi$ in B

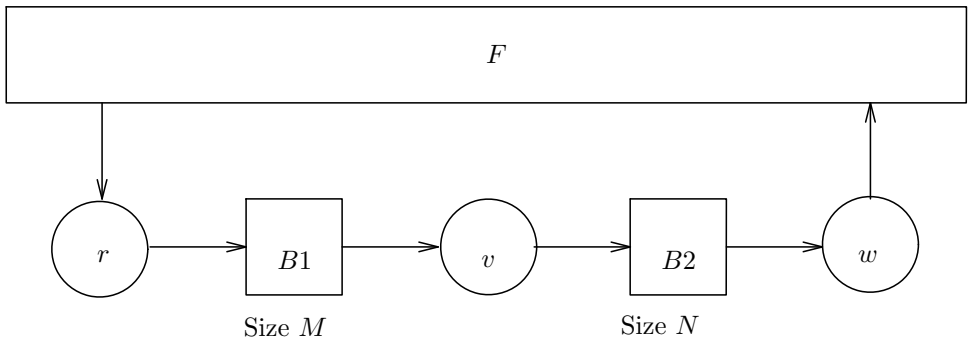
In the following proof all properties are in B.

$|r + b + w| \leq N + 1$ constant
, constant formation rule applied to P4
$|r + b + w| \leq N + 1$ stable
, a constant predicate is stable
$w = \phi$ *unless* $w \neq \phi$
, antireflexivity (see Sec. 2.1.5)
$w = \phi \ \wedge \ |r + b + w| \leq N + 1$ *unless* $w \neq \phi \ \wedge \ |r + b + w| \leq N + 1$
, stable conjunction (Sec. 2.1.5) applied to the above two
$w = \phi \ \wedge \ |r + b| \leq N + 1$ *unless* $|r + b| \leq N$
, rewriting the lhs and weakening the consequence (Sec. 2.1.5)
$w = \phi$ *unless* $|r + b| \leq N$
, substitution axiom applied to the lhs with invariant $|r + b| \leq N + 1$ (see P6)
$w = \phi$ *unless* $|b| < N \ \vee \ r = \phi$
, consequence weakening (Sec. 2.1.5) □

P8. If $w = \phi$ stable in F then
$w = \phi \ \mapsto \ |b| < N \ \vee \ r = \phi$ in $B \ [\!]\ F$

In the following proof all properties are in $B \ [\!]\ F$ unless otherwise stated.

$w = \phi$ stable in F	, given
$w = \phi$ *unless* $\|b\| < N \ \vee \ r = \phi$ in B	, from P7
$w = \phi$ *unless* $\|b\| < N \ \vee \ r = \phi$ in $B \ [\!]\ F$	, Corollary 1 of Sec. 2.2
$\|b\| > 0 \ \mapsto \ w \neq \phi$	, from P5.2



$B = B1 \parallel B2$ has Size $M + N + 1$

Figure 4: Concatenation of bags $B1, B2$

$|b| > 0 \ \wedge \ w = \phi \ \mapsto \ |b| < N \ \vee \ r = \phi$, PSP on the above two
$|b| \leq 0 \ \wedge \ w = \phi \ \mapsto \ |b| < N$, implication rule and using $N > 0$.
$w = \phi \ \mapsto \ |b| < N \ \vee \ r = \phi$, disjunction on the above two

5.5 Bag Concatenation

Let $B1$ implement a bag of size $M, M > 0$, with input, output varibles, r and v, respectively, and $B2$ implement a bag of size $N, N > 0$, with input, output variables v and w, respectively. We show that $B = B1 \ \square \ B2$ implements a bag of size $M + N + 1$ with input, output variables r and w, respectively. The arrangement is shown pictorially in Fig. 4. Observe that r and w are different variables (i.e., $B1, B2$ are not connected cyclically) and hence, r cannot be accessed by $B2$ nor can w be accessed by $B1$.

We first rewrite the properties from the specification of Sec. 5.3 for $B1, B2$. In the following, R, V, W are free variables.

There exists $b1$, local to $B1$, such that
P0. initially $b1 + v = \phi$ in $B1$
P1. $r \preceq R$ stable in $B1$
P2. $V \preceq v$ stable in $B1$
P3. $|b1| \leq M$ invariant in $B1$
P4. $r + b1 + v$ constant in $B1$
P5. $\langle \forall \ G \ ::$
(P5.1) $|b1| < M \ \mapsto \ r = \phi$ in $B1 \ \square \ G$
(P5.2) $|b1| > 0 \ \mapsto \ v \neq \phi$ in $B1 \ \square \ G$
$\rangle$

There exists $b2$, local to $B2$, such that
initially $b2 + w = \phi$ in $B2$
$v \preceq V$ stable in $B2$
$W \preceq w$ stable in $B2$
$|b2| \leq N$ invariant in $B2$
$v + b2 + w$ constant in $B2$
$\langle \forall \ H \ ::$
$|b2| < N \ \mapsto \ v = \phi$ in $B2 \ \square \ H$
$|b2| > 0 \ \mapsto \ w \neq \phi$ in $B2 \ \square \ H$
$\rangle$

The properties of B, to be proven, are (again from Sec. 5.3)

There exists b, local to B, such that
P0. initially $b + w = \phi$ in B
P1. $r \preceq R$ stable in B
P2. $W \preceq w$ stable in B
P3. $|b| \leq M + N + 1$ invariant in B
P4. $r + b + w$ constant in B
P5. $\langle \forall \ F \ ::$
(P5.1) $|b| < M + N + 1 \ \mapsto \ r = \phi$ in $B \ \square \ F$
(P5.2) $|b| > 0 \ \mapsto \ w \neq \phi$ in $B \ \square \ F$
$\rangle$

We start the proof by defining b to be $b1 + v + b2$. To show that b is local to B observe that $b1+v+b2$ is constant in the environment F since none of the variables—$b1, v$ or $b2$—can be accessed by F.

Proof of P0: All properties in this proof are in B.

initially $b1 + v = \phi$	, from P0 of $B1$
initially $b2 + w = \phi$	, from P0 of $B2$
initially $b1 + v + b2 + w = \phi$	, from the above two
initially $b + w = \phi$	, definition of b □

Proof of P1:

r constant in $B2$	, r cannot be accessed by $B2$ (locality)
$r \preceq R$ stable in $B2$	, constant formation; a constant predicate is stable
$r \preceq R$ stable in $B1$	, from P1 of $B1$
$r \preceq R$ stable in $B1 \sqcap B2$	, from Corollary 1 of Sec. 2.2 □

Proof of P2: Similar to the proof of P1. □

Proof of P3:

$\|b1\| \leq M$ invariant in $B1 \sqcap B2$	, from P6 applied to $B1$ (with $F = B2$)
$\|v\| \leq 1$ invariant in $B1 \sqcap B2$	, definition of v
$\|b2\| \leq N$ invariant in $B1 \sqcap B2$	, from P6 applied to $B2$ (with $F = B1$)
$\|b1 + v + b2\| \leq M + N + 1$ invariant in $B1 \sqcap B2$	, from the above three □

Proof of P4: $r + b + w = r + b1 + v + b2 + w$

$r + b1 + v$ constant in $B1$	, from P4 of $B1$
$b2 + w$ constant in $B1$	, $b2$ and w cannot be accessed by $B1$
$r + b1 + v + b2 + w$ constant in $B1$	, constant formation rule
$r + b1 + v + b2 + w$ constant in $B2$	, similarly
$r + b1 + v + b2 + w$ constant in $B1 \sqcap B2$	, Corollary 2 of Sec. 2.2 □

Proof of P5: Consider any arbitrary F. Setting G to $B2 \sqcap F$ in P5 for $B1$ we get

1. $|b1| < M \mapsto r = \phi$ in $B1 \sqcap B2 \sqcap F$
2. $|b1| > 0 \mapsto v \neq \phi$ in $B1 \sqcap B2 \sqcap F$

Similarly, setting H to $B1 \sqcap F$ in P5 for $B2$ we get

3. $|b2| < N \mapsto v = \phi$ in $B1 \sqcap B2 \sqcap F$
4. $|b2| > 0 \mapsto w \neq \phi$ in $B1 \sqcap B2 \sqcap F$

We will only show P5.1 for B, i.e.

$$|b| < M + N + 1 \mapsto r = \phi \quad \text{in } B \sqcap F$$

We use the following fact that is easily seen from the implication and disjunction rules for *leads-to*.

$$\frac{q \mapsto q'}{p \vee q \mapsto p \vee q'}$$

In the following proof all properties are in $B \sqcap F$, i.e., $B1 \sqcap B2 \sqcap F$.

	$\|b\| < M + N + 1$	
$\Rightarrow$	$\|b1\| < M \vee v = \phi \vee \|b2\| < N$	, since $b = b1 + v + b2$
$\mapsto$	$\|b1\| < M \vee v = \phi$	, above fact about $\mapsto$ and (3).
$\mapsto$	$\|b1\| < M \vee r = \phi$	, above fact about $\mapsto$ and P8 for $B1$ (note, $v = \phi$ stable in $B2 \sqcap F$)
$\mapsto$	$r = \phi$	, above fact about $\mapsto$ and (1).

5.6 A Note on the Bag Specification

It is interesting to note that the specification of Sec. 5.3 does not apply if $N = 0$, i.e., the internal space for storing bag items is empty. In such a case, we would expect program B to move data from r to w whenever $r \neq \phi$ and $w = \phi$. However, the progress condition P5 becomes,

$$\begin{array}{l}\langle \forall\ F \quad :: \\ \quad |b| < 0 \ \mapsto\ r = \phi \quad \text{in} * \ \square\ F \\ \quad |b| > 0 \ \mapsto\ w \neq \phi \quad \text{in} * \ \square\ F \\ \rangle\end{array}$$

Since $N = 0$, it follows (from P6 and the definition of b) that $|b| = 0$ is an invariant in $* \ \square\ F$. Hence, using the substitution axiom, the antecedent of each progress condition is *false*. In UNITY, *false* $\mapsto\ p$, for any p. Therefore, for $N = 0$ the specification provides no guarantees on progress. In particular, the specification allows a state $r \neq \phi\ \wedge\ w = \phi$ to persist forever. It may seem that the following strengthening of the progress specification could generalize the specification to $N \geq 0$.

$$\begin{array}{ll} |b| < N\ \vee\ w = \phi\ \mapsto\ r = \phi & \text{in} * \ \square\ F \\ |b| > 0\ \vee\ r \neq \phi\ \mapsto\ w \neq \phi & \text{in} * \ \square\ F \end{array}$$

However since nothing is assumed about F, program F could conceivably change $r \neq \phi \wedge w = \phi$ to $r \neq \phi\ \wedge\ w \neq \phi$ (by storing data in w), thus making it impossible for the bag program to implement the first progress condition (while satisfying P4: $r + b + w$ is constant). Thus, such a specification cannot be implemented unless we assume something more about the way F behaves, in particular that it never removes data from r nor stores into w. A specification along this line is given in the next section.

5.7 Alternative Specification of a Bag

5.7.1 Communication Sequences as Auxiliary Variables

We propose another bag specification in this section. We take as auxiliary variables the bags $\widehat{r}$ and $\widehat{w}$, where $\widehat{r}$ is the bag of data items written into r (and similarly $\widehat{w}$). Such a bag is typically defined by augmenting the program text appropriately: whenever r is changed $\widehat{r}$ is changed by adding the new value of r to it. As shown in an example in Sec. 2.1.1 our logic provides a direct way of defining $\widehat{r}$, $\widehat{w}$ without appealing to the program text. In the following, R, W are free variables (of type data) and X, Y are free variables that are of type bag of data. Program F is the environment, as before.

$$\begin{array}{ll} \text{A0.} & \text{initially } (\widehat{r},\widehat{w}) = (r, w) \quad \text{in} * \ \square\ F \\ \text{A1.} & \widehat{r} = X\ \wedge\ r = R \ \textit{unless}\ \widehat{r} = X + r\ \wedge\ r \neq R \quad \text{in} * \ \square\ F \\ \text{A2.} & \widehat{w} = Y\ \wedge\ w = W \ \textit{unless}\ \widehat{w} = Y + w\ \wedge\ w \neq W \quad \text{in} * \ \square\ F \end{array}$$

To understand the relationship between $\widehat{r}$ and r and (similarly for $\widehat{w}$ and w), note that $\widehat{r}$ remains unchanged as long as r is unchanged and $\widehat{r}$ is modified by adding the (new) value of r to it whenever r is changed. Furthermore, in our case, since the values of r will alternate between ϕ and non-ϕ, successive values assumed by r are different. (Note that whenever r is set to ϕ, $\widehat{r}$ remains unchanged.)

We deduce two simple facts. (Here "$\subseteq$" is the subbag relation.)

$$\begin{array}{ll} \text{A3.} & r\ \subseteq\ \widehat{r} \text{ invariant} \quad \text{in} * \ \square\ F \\ \text{A4.} & w\ \subseteq\ \widehat{w} \text{ invariant} \quad \text{in} * \ \square\ F \end{array}$$

We prove A3; proof of A4 is similar.

Proof of A3: All properties in the following proof are in $* \ \square\ F$.

$$\begin{array}{ll} \widehat{r} = X\ \wedge\ r = R\ \ \textit{unless}\ \ \widehat{r} = X + r\ \wedge\ r \neq R & \text{, from A1} \\ (\widehat{r}, r) = (X, R)\ \ \textit{unless}\ \ (\widehat{r}, r) \neq (X, R)\ \wedge\ r\ \subseteq\ \widehat{r} & \text{, consequence weakening} \\ r\ \subseteq\ \widehat{r} \text{ stable} & \text{, Corollary 1 of Sec. 2.1.5} \\ \text{Initially } r\ \subseteq\ \widehat{r} \text{ because } r = \widehat{r}. \text{ Hence } r\ \subseteq\ \widehat{r} \text{ invariant.} & \end{array}$$

5.7.2 Bag Specification

The following specification is for a bag of size $N, N \geq 0$. The properties R1,R2 are the same as P1,P2 of Sec. 5.3. In order to overcome the problems described in Sec. 5.6, we make some assumptions about the environment, F. Specifically, we require F to obey a similar access protocol to w, r as the ones obeyed by the bag program for access to r, w: Environment F may write into r only if $r = \phi$ and it may change w—to ϕ—only if $w \neq \phi$. Under these conditions on F, the bag $\widehat{w}$ is a subbag of $\widehat{r} - r$ (property R3.1); these two bags differ in size by no more than N (property R3.2); if $\widehat{r}, \widehat{w}$ differ by no more than N then r is or will be set to ϕ (property R3.3); if $|\widehat{r}|$ exceeds $|\widehat{w}|$ then w is or will be set to non-ϕ (property R3.4).

spec R (r, w : data, N : nat constant)
declare
free R, W : data,
auxiliary $\hat{r}, \hat{w}$: sequence of data,
free X, Y : sequence of data,
free F : program
hypo
{A0} initially $(\widehat{r},\widehat{w}) = (r, w)$ in $* \,[\!]\, F$
{A1} $\widehat{r} = X \ \wedge \ r = R$ *unless* $\widehat{r} = X + r \ \wedge \ r \neq R$ in $* \,[\!]\, F$
{A2} $\widehat{w} = Y \ \wedge \ w = W$ *unless* $\widehat{w} = Y + w \ \wedge \ w \neq W$ in $* \,[\!]\, F$
{A3} $R \preceq r$ stable in F
$w \preceq W$ stable in F
conc
{R0} initially $w = \phi$
{R1} $r \preceq R$ stable
{R2} $W \preceq w$ stable
{R3.1} $\widehat{w} \subseteq \widehat{r} - r$ invariant
{R3.2} $|\widehat{r} - r - \widehat{w}| \leq N$ invariant
{R3.3} $|\widehat{r} - \widehat{w}| \leq N \mapsto r = \phi$ in $* \,[\!]\, F$
{R3.4} $|\widehat{r}| > |\widehat{w}| \mapsto w \neq \phi$ in $* \,[\!]\, F$
end$\{R\}$

We show, in Sec. 5.7.3, that under certain conditions the proposed specification is a refinement of the specification in Sec. 5.3. Now we deduce a few properties from the given specification. Assuming **hypo**,

R4. $\widehat{r} - \widehat{w} + w$ constant in *
R5. $\widehat{r} - \widehat{w} - r$ constant in F.

The proof of R5 is similar to that of R4 by interchanging the roles of r, w. We prove R4.

Proof of R4: All properties in the following proof are in *.

$\widehat{r} = X \ \wedge \ r = R$ *unless* $\widehat{r} = X + r \ \wedge \ r \neq R$	, union theorem on A1
$r \preceq R$ stable	, from R1
$\widehat{r} = X \ \wedge \ r = R$ *unless* $\widehat{r} = X + r \ \wedge \ r \prec R$	, stable conjunction on the above two
$(\widehat{r}, r) = (X, R)$ *unless* $(\widehat{r}, r) \neq (X, R) \ \wedge \ \widehat{r} = X$	, consequence weakening
$\widehat{r}$ constant	, constant introduction

Similarly, we can show—starting from A2,R2 and applying stable conjunction—that $\widehat{w} - w$ is constant in *. Using the constant formation rule, $\widehat{r} - (\widehat{w} - w)$ is constant in *. Since $w \subseteq \widehat{w}$ (from A4) and $\widehat{w} \subseteq \widehat{r}$ (from R3.1, which can be assumed since **hypo** holds), $\widehat{r} - (\widehat{w} - w) = \widehat{r} - \widehat{w} + w$, which is then constant in *.

5.7.3 Proof of Refinement

We show that for $N > 0$, the specification of Sec. 5.7.2 is a refinement of the specification of Sec. 5.3, *provided that* **hypo** *holds*. In the absence of such a requirement on F, the specification of Sec. 5.7.2 does not prescribe B's behavior (that is, when B is composed with any arbitrary F) whereas the specification of Sec. 5.3 prescribes B's behavior even when it is composed with an arbitrary F. Hence, our proof obligation for the refinement is to show that, given

hypo, $N > 0$ and the properties R0–R3 of Sec. 5.7.2

there exists b, local to B, satisfying properties P0–P5 of Sec. 5.3. We let

$b = \widehat{r} - \widehat{w} - r$

Observe that b is local to B since b is constant in F (from R5).

Proof of P0: $b + w = \widehat{r} - \widehat{w} - r + w$
initially $(\widehat{r}, \widehat{w}) = (r, w)$. Hence initially $b + w = \phi$. □

Proofs of P1, P2: directly from R1, R2. □

Proof of P3:
$|\widehat{r} - \widehat{w} - r| \leq N$ invariant , from R3.2
$|b| \leq N$ invariant , using the definition of b □

Proof of P4: $r + b + w = \widehat{r} - \widehat{w} + w$
$\widehat{r} - \widehat{w} + w$ constant , from R4 □

Proof of P5.1: All properties in this proof are in $* \,[\!]\, F$.
$|b| < N$
$\Rightarrow$ $|\widehat{r} - \widehat{w} - r| < N$, using the definition of b
$\Rightarrow$ $|\widehat{r} - \widehat{w}| \leq N$, $|r| \leq 1$
$\mapsto$ $r = \phi$, from R3.3 □

The proof of P5.2 is similar to that of P5.1.

5.8 Specifications of Concurrent Queue and Concurrent Stack

We consider two other concurrent data objects—queue and stack—in this section. Each has variables r, w which are accessed similarly as in the case of the bag. Our interest in these specifications is mainly to show that each of these specifications refines a bag specification, i.e., each implements a bag (of the appropriate size).

5.8.1 A Specification of a Concurrent Queue

The operation of a concurrent queue is analogous to that of a concurrent bag. The important difference is that in the former case the items are written into w in the same order in which they were read from r. We propose a specification analogous to that of Sec. 5.3. In the following "uv" denotes concatenation of sequences u, v and ϕ denotes the null sequence. A queue of size N, $N > 0$, is defined by a program Q where:

spec *queue* (r, w : data, N : positive constant)
declare
local q : sequence of data,
free R, W : data,
free F : program

conc

$\{Q0\}$ initially $q = \phi \ \wedge \ w = \phi$

$\{Q1\}$ $r \preceq R$ stable

$\{Q2\}$ $W \preceq w$ stable

$\{Q3\}$ $|q| \leq N$ invariant

$\{Q4\}$ rqw constant

$\{Q5\}$ $\{Q5.1\}$ $|q| < N \ \mapsto \ r = \phi$ in $* \ [\!] \ F$

$\{Q5.2\}$ $|q| > 0 \ \mapsto \ w \neq \phi$ in $* \ [\!] \ F$

end $\{queue\}$

5.8.2 Queue Implements Bag

We show that from the specification of Sec. 5.8.1 we can deduce the specification in Sec. 5.3 for an appropriate b. Denote the bag of items in q by $[q]$. Let $b = [q]$.

Proofs of P0,P1,P2 are trivial. Property P3 follows from Q3 and P5 from Q5 by noting that

$|b| = |[q]| = |q|$.

To prove P4—that $r + b + w$ constant in *—we observe (below all properties are in *)

rqw constant	, from Q4
$[rqw]$ constant	, constant introduction
$[r] + [q] + [w]$ constant	, $[rqw] = [r] + [q] + [w]$
$r + b + w$ constant	, $[r] = r, [q] = b, [w] = w$

5.8.3 A Specification of a Concurrent Stack

The operation of a concurrent stack is distinguished from that of a concurrent bag by the requirements that the items read from r be pushed onto a stack (read is permitted only if there is room in the stack) and the item written into w at any point be the top of the stack which is then removed from the stack. A stack is seldom accessed concurrently because speed differences between the producer—process that writes into the stack—and the consumer—that reads from the stack—affects the outcomes of the reads.

We propose a specification, analogous to that of Sec. 5.7.2, for a program S that implements a stack of size N, $N > 0$. Let $\bar{r}, \bar{w}$ denote the sequences of data items written into r, w respectively. Analogous to the definitions of $\widehat{r}, \widehat{w}$ by (A0,A1,A2) we define $\bar{r}, \bar{w}$ in $S \ [\!] \ F$ (where F is any arbitrary environment of S) by $(A0', A1', A2')$ below. As in Sec. 5.8.1, concatenations of sequences are shown by juxtapositions and ϕ denotes the null sequence.

(A0′) initially $(\bar{r}, \bar{w}) = (r, w)$ in $S \ [\!] \ F$

(A1′) $\bar{r} = X \ \wedge \ r = R$ *unless* $\bar{r} = Xr \ \wedge \ r \neq R$ in $S \ [\!] \ F$

(A2′) $\bar{w} = Y \ \wedge \ w = W$ *unless* $\bar{w} = Yw \ \wedge \ w \neq W$ in $S \ [\!] \ F$

Now, we define a binary relation between sequences X, Y, written as $X \ N \ Y$, expressing that given X as an input sequence to a stack of size N sequence Y is a possible (complete) output sequence. For $N \geq 0$, it is the strongest relation satisfying

- $\phi \ N \ \phi$
- $X \ N \ X' \ , \ Y \ (N+1) \ Y' \ \Rightarrow \ aXY \ (N+1) \ X'aY'$

where a is any arbitrary data item. The first rule is trivial to see. The second rule states that given an input string aXY to a stack of size $N+1$, the item a appears in the output at some point. Prior to its output, item a is at the bottom of the stack and hence, the stack behaves as if its size is N in converting some portion of the input, say X, to X'. Following the output of a, the stack is empty and hence the remaining input sequence Y is converted to Y' using a stack of size $N+1$.

Several interesting properties can be proven about this relation using induction on the length of X. In particular,

$$X\ N\ Y\ \Rightarrow\ [X] = [Y]$$

where $[X]$ is the bag of items in X.

Now we specify a program that implements a concurrent stack of size N, $N > 0$, with shared variables r, w as before. Note that S3.1 is the only property that differs substantially from the corresponding property of the bag in Sec. 5.7.2.

spec *stack* (r, w : data, N : positive constant)
declare
free R, W : data,
auxiliary $\bar{r}, \bar{w}$: sequence of data,
bound $\bar{z}$: sequence of data,
free X, Y : sequence of data,
free F : program
hypo
$\{A0'\}$ initially $(\bar{r}, \bar{w}) = (r, w)$ in $*\ \Box\ F$
$\{A1'\}$ $\bar{r} = X\ \wedge\ r = R$ *unless* $\bar{r} = Xr\ \wedge\ r \neq R$ in $*\ \Box\ F$
$\{A2'\}$ $\bar{w} = Y\ \wedge\ w = W$ *unless* $\bar{w} = Yw\ \wedge\ w \neq W$ in $*\ \Box\ F$
$\{A3'\}$ $R\ \preceq\ r$ stable in F
$w\ \preceq\ W$ stable in F
conc
$\{S0\}$initially $w = \phi$
$\{S1\}r\ \preceq\ R$ stable
$\{S2\}W\ \preceq\ w$ stable
$\{S3\}$$\{$In S3.1, $\bar{r} - r$ is the prefix of $\bar{r}$ excluding r (if $r = \phi$, $\bar{r} - r = \bar{r}$).
Also, $\sqsubseteq$ is the prefix relation over sequences$\}$
$\{S3.1\}$ $\langle \exists\ \bar{z}\ ::\ (\bar{r} - r)\ N\ \bar{z}\ \wedge\ \bar{w}\ \sqsubseteq\ \bar{z} \rangle$ invariant in $*\ \Box\ F$
$\{S3.2\}$ $|\bar{r}| \leq |r| + |\bar{w}| + N$ invariant in $*\ \Box\ F$
$\{S3.3\}$ $|\bar{r}| \leq |\bar{w}| + N \mapsto\ r = \phi$ in $*\ \Box\ F$
$\{S3.4\}$ $|\bar{r}| > |\bar{w}| \mapsto\ w \neq \phi$ in $*\ \Box\ F$
end $\{$*stack*$\}$

Stack Implements a Bag

We show that from the specification of a concurrent stack we can deduce the bag specification in Sec. 5.7.2 for $N > 0$. Note that $\widehat{r} = [\bar{r}]$, $\widehat{w} = [\bar{w}]$. It follows that $\widehat{r} - r = [\widehat{r} - r]$. Proofs of properties R0–R3 are entirely straightforward, except for R3.1: $\widehat{w}\ \subseteq\ \widehat{r} - r$.

$(\bar{r} - r)\ N\ \bar{z}\ \Rightarrow\ [\bar{r} - r] = [\bar{z}]$
, property of the binary relation stated earlier
$\bar{w}\ \sqsubseteq\ \bar{z}\ \Rightarrow\ [\bar{w}]\ \subseteq\ [\bar{z}]$
, fact about building a bag from a sequence
$(\bar{r} - r)\ N\ \bar{z}\ \wedge\ \bar{w}\ \sqsubseteq\ \bar{z}\ \Rightarrow\ [\bar{w}]\ \subseteq\ [\bar{r} - r]$
, from the above two
$\langle \exists\ \bar{z}\ ::\ (\bar{r} - r)\ N\ \bar{z}\ \wedge\ \bar{w} \sqsubseteq \bar{z} \rangle\ \Rightarrow\ \widehat{w}\ \subseteq\ \widehat{r} - r$
, from the above using $[\bar{w}] = \widehat{w}$ and $[\bar{r} - r] = \widehat{r} - r$.

We leave it as an exercise for the reader to show that a queue of size 1 implements a stack of size 1, and vice versa.

5.9 Refinement of the Bag Specification

We refine the specification of Sec. 5.7.2 as a step toward implementing a bag. It can be shown from that specification that concatenation of two bags of size M, N, where M, N are non-negative, results in a bag of size $M + N + 1$; the proof is similar in structure to the proof in Sec. 5.5; also, a similar proof for a queue appears in Misra [1990]. Hence a bag of size N, $N > 0$, can be implemented by concatenating $(N + 1)$ bags of size 0 each. Therefore we restrict our attention to bags of size 0 and propose a refinement. As before, we have properties T0, T1, T2. If a bag has size 0, the only possible action is to move data items from r to w directly. The effect of this action is to keep $r + w$ constant (property T3), and such an action is guaranteed to take place provided $r \neq \phi \ \wedge \ w = \phi$, thus resulting in $w \neq \phi$ (property T4).

spec T $(r, w$: date)
  declare
    free R, W : data
  conc
    {T0} initially $w = \phi$
    {T1} $r \preceq R$ stable
    {T2} $W \preceq w$ stable
    {T3} $r + w$ constant
    {T4} $r \neq \phi$ *ensures* $w \neq \phi$
end $\{T\}$

5.9.1 Proof of the Refinement

Suppose that program B satisfies specification T. We show that the properties R0–R3 of Sec. 5.7.2, for $N = 0$, can be deduced from T0–T4. Proofs of R0,R1,R2 are immediate from T0,T1,T2.

T5: $\langle$**hypo** of $R \ \wedge$ T0 $\wedge$ T3 $\Rightarrow \ \widehat{r} = \widehat{w} + r$ invariant in $B \ \square \ F\rangle$

Proof:

	$\widehat{r} - \widehat{w} + w$ constant in B	, from R4 (derived from **hypo** of R)
	$r + w$ constant in B	, from T3
	$\widehat{r} - \widehat{w} + w - (r + w)$ constant in B	, constant formation
	$\widehat{r} - \widehat{w} - r$ constant in B	, simplifying the above expression
	$\widehat{r} - \widehat{w} - r$ constant in F	, from R5 (derived from **hypo** of R)
	$\widehat{r} - \widehat{w} - r$ constant in $B \ \square \ F$	, Corollary 2 of Sec. 2.2
Initially	$\widehat{r} - \widehat{w} - r = \phi$ in $B \ \square \ F$	, from $\widehat{r} = r$ and $\widehat{w} = w = \phi$ (using T0)
Hence	$\widehat{r} - \widehat{w} - r = \phi$ invariant in $B \ \square \ F$, i.e., $\widehat{r} = \widehat{w} + r$ invariant in $B \ \square \ F$.	□

Next we prove the four properties in R3.1–R3.4, from **hypo**, T1, T2, and $\widehat{r} = \widehat{w} + r$ invariant in $B \ \square \ F$.

Proof of R3: Assume **hypo** of R holds.

R3.1.	Proof is immediate given T5.	□
R3.2	Proof is immediate given T5 and that $N = 0$	□
R3.3	We have to show that $\lvert\widehat{r}\rvert \leq \lvert\widehat{w}\rvert \ \mapsto \ r = \phi$ in $B \ \square \ F$.	
	In the following all properties are in $B \ \square \ F$.	
	$\lvert\widehat{r}\rvert \leq \lvert\widehat{w}\rvert \ \Rightarrow \ r = \phi$	, from T5
	$\lvert\widehat{r}\rvert \leq \lvert\widehat{w}\rvert \ \mapsto \ r = \phi$	, using implication rule on the above □
R3.4	We have to show that $\lvert\widehat{r}\rvert > \lvert\widehat{w}\rvert \ \mapsto \ w \neq \phi$ in $B \ \square \ F$	
	$r = R$ *unless* $r \neq R$ in F	, antireflexivity of *unless*
	$r = R$ *unless* $r \neq R \ \wedge \ r \neq \phi$ in F	, stable conjunction with $R \preceq r$ stable (from **hypo**)
	$r \neq \phi$ stable in F	, Corollary 1 of Sec. 2.1.5
	$r \neq \phi$ *ensures* $w \neq \phi$ in B	, from T4
	$r \neq \phi$ *ensures* $w \neq \phi$ in $B \ \square \ F$	, using Corollary 4 of Sec. 2.2 on the above two

$$
\begin{array}{lcll}
r \neq \phi & \mapsto & w \neq \phi & \text{in } B \ \square \ F \quad \text{, definition of } \mapsto \\
|\widehat{r}| > |\widehat{w}| & \mapsto & w \neq \phi & \text{in } B \ \square \ F \quad \text{, from T5, } \ |\widehat{r}| > |\widehat{w}| \ \equiv \ r \neq \phi
\end{array}
$$

5.10 An Implementation

The specification of Sec. 5.9 can be implemented by a program whose only statement moves data from r to w provided $w = \phi$ (if $r = \phi$, the movement has no effect):

$$r, w \ := \ \phi, r \qquad \text{if} \quad w = \phi$$

The proof that this fragment has the properties T1–T4 is immediate from the definition of *unless* and *ensures*. The initial condition of this program is $w = \phi$, and hence (T0) is established.

An implementation for a bag of size N, $N > 0$, uses the union of $N + 1$ such statements: One statement each for moving data from a location to an adjacent location (closer to w) provided the latter is ϕ. We show how this program may be expressed in the UNITY programming notation.

Rename the variables r and w to be $b[0]$ and $b[N + 1]$, respectively. The internal bag words are $b[1]$ through $b[N]$. In the following program we write $\langle \square \ i \ : \ 0 < i \leq N+1 \ :: \ t(i) \rangle$ as a shorthand for $t(1) \ \square \ t(2) \ldots \square \ t(N+1)$, where $t(1)$, for instance, is obtained by replacing every occurrence of i by 1 in $t(i)$. The program specifies the initial values of $b[1]$ through $b[N + 1]$ to be ϕ (in the part followed by **initially**). The statements of the program are given after **assign**; the generic statement shown moves $b[i - 1]$ to $b[i]$ provided the latter is ϕ.

Program *bag* {of size N, $N > 0$}
  initially $\langle \square \ i \ : \ 0 < i \leq N + 1 \ :: \ b[i] = \phi \rangle$
  assign $\langle \square \ i \ : \ 0 < i \leq N + 1 \ :: \ b[i - 1], b[i] \ := \ \phi, b[i - 1] \qquad \text{if} \quad b[i] = \phi \rangle$
end {*bag*}

6 Discussion

Programmers prefer different kinds of specifications for different problems. For problems such as the vending machine, a specification using finite state automata may be satisfactory because it describes the action-sequences (or event-sequences) in a concise manner. The specification is so simple that a decomposition, for the user component and the vending machine component, is hardly required; the vending machine is expected to operate in an environment with an interface that allows very few variations. A UNITY-style specification for this problem was also quite simple.

The specification of the messsage communicating system for termination detection shows why specification boundaries need not coincide with process boundaries; in this case it proved easier to specify the message communication aspect of the entire system as a module and the termination detection specification as another module. The actual implementation will, typically, partition the termination detection algorithm among the processes.

The long example of specifying a bag illustrates most of the ideas, including introduction of auxiliary variables, in UNITY-specification. A particularly attractive idea is specification refinement that allows stepwise refinement of the specification ending in an implementation. Also, the possibility of implementing a data structure by another—such as a bag by a queue—based on their specifications alone, is a tantalizing new direction of research.

In all our specifications, the environment of the reactive system assumes no special role; it is merely treated as a program that is composed with the program to be designed. Since composition by union is commutative, the program and its environment have interchangeable roles. A component may be structured as a union of two, or more, subcomponents, one of which may serve as an environment for the other. It seems preferable to dispense with the notion of environment in formal developments.

We are only beginning to understand how specifications can be composed. For instance, we composed—by union—the specifications of two bags, one of which acted as a producer for the other (see Sec. 5.5). And we showed how a large bag can be implemented by a sequence of small bags.

However, there are many other ways of composition/decomposition that we have not addressed in the paper. One is, how to *layer* one specification on another. For instance the specification of 2-party connection in a telephone switch typically includes the specification of the connection as well as the specification for billing. We would like to specify these aspects separately even though in the final implementation code for both may be intertwined.

7 References

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

Chandy and Misra [1990] K. Mani Chandy and Jayadev Misra, "Proofs of Distributed Algorithms: An Example," (Chapter) *Proc. U.T. Year of Programming Institute on Concurrent Programming*, Addison-Wesley, 1990 (to appear).

Guttag [1977] John Guttag, "Abstract Data Types and the Development of Data Structures," *C.ACM*, 20:6, pp. 396–404, June 1977.

Hailpern [1982] B. Hailpern, "Verifying Concurrent Processes Using Temporal Logic," Springer-Verlag, *Lecture Notes in Computer Science 129*, 1982.

Hayes [1987] *Specification Case Studies*, Ian Hayes (ed.), Prentice-Hall International, Englewood Cliffs, N.J., 1987.

Hoare [1972] C. A. R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, Vol. 1, pp. 271–281, 1972.

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

Lamport [1989] L. Lamport, "A Simple Approach to Specifying Concurrent Systems," *Communications of the ACM*, **32**:1, 1989.

Misra [1988] J. Misra, "General Conjunction and Disjunction Rules for *unless*," *Notes on Unity 01-88*, University of Texas at Austin, 1988.

Misra [1989a] J. Misra, "A Foundation of Parallel Programming," *Proc. 9th International Summer School on Constructive Methods in Computer Science*, Marktoberdorf, Germany, July 24–August 5, 1988, in *NATO ASI Series, Vol. F 55*, ed. Manfred Broy, Springer-Verlag, pp. 397–433, 1989.

Misra [1989b] J. Misra, "Monotonicity, Stability and Constants," *Notes on Unity 10-89*, University of Texas at Austin, 1989.

Misra [1990] J. Misra, "Specifying Concurrent Objects as Communicating Processes," *Science of Computer Programming* (to appear). A preliminary version, "Specifications of Concurrently Accessed Data," appears in *Proc. International Conference on Mathematics of Program Construction, Groningen University, The Netherlands, June 26–30, 1989, LNCS 375*, Springer-Verlag.

Singh [1989] Ambuj Singh, "Ranking in Distributed Systems," (see Sec. 3.4, in particular), Ph.D. thesis, The University of Texas at Austin, 1989.