Science of Computer Programming 14 (1990) 159-184
North-Holland

SPECIFYING CONCURRENT OBJECTS AS COMMUNICATING PROCESSES

Jayadev MISRA*
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA

Received January 1990

1. Introduction

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 [3-5]. Another way [2] 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). The common goal of a specification, however, is to serve as a legal document that spells out a contract between a user and an implementer: The user may assume no more about an object than what is specified and the implementer must satisfy the specification through his implementation.

We view a specification not merely as a legal contract but additionally as a means to

(1) deduce properties of a specified object,
(2) deduce properties of other objects in which the specified object is a component (i.e., inheritance of properties), and
(3) implement the object by stepwise refinement 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.

In many 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

* Partially supported by the ONR Contract N00014-90-J-1640 and by the Texas Advanced Research Program under Grant No. 003658-065.

(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 paper, 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 specification mechanism is based on "UNITY logic" [1]. Auxiliary variables can be defined directly using the constructs in this logic. The inference rules of the logic can be applied to deduce various properties of an object from its specification. Also, we may use the logic to prove that one specification refines (i.e., implements) another specification and that an implementation meets a specification.

1.1. Outline of the paper

We give a brief introduction to UNITY in Section 2 including all the notations and logic to understand this paper. The main example treated in this paper is a

bounded bag (or multiset). Section 3 contains an informal description and a formal specification of this object. We demonstrate the usefulness of the specification by showing that concatenation of two bags results in a bag of the appropriate size. The specification in Section 3 is based on an auxiliary variable that encodes the internal state. We provide an alternative specification in Section 4 that uses input-output sequences as auxiliary variables; we show that this specification is a refinement of the earlier one. Section 5 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. Section 6 contains a refinement of the bag specifications of Section 4, and Section 7 gives an implementation of the specification of Section 6. The usefulness of the method is discussed in Section 8.

All the 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 paper appears in [7]. The specifications and the proofs have been considerably simplified in the current version.

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 Section 7 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 notions useful for 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 bag example 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 2.1. Throughout this section p, q and 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,

$$p \textit{ unless } q$$

denotes that once predicate p is true it remains true at least as long as q is not true. Formally (using Hoare's notation)

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

i.e., 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.

It follows from this definition 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 future states.

Notation 2.2. 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,

$$u = k \textit{ unless } u > k$$

where k is free, is a shorthand for

$$\langle \forall k :: u = k \textit{ unless } u > k\rangle.$$

Examples 2.3.

(1) Integer variable u does not decrease:

$$u = k \textit{ unless } u > k$$

or

$$u \geqslant k \textit{ unless } \textit{false}$$

(2) A message is received (i.e., predicate *rcvd* holds) only if it had been sent earlier (i.e., predicate *sent* already holds):

$$\neg \textit{rcvd unless sent}$$

Example 2.4 (*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

$$\text{initially } v = 0$$

$$u = m \ \wedge \ v = n \textit{ unless } u \neq m \ \wedge \ v = n+1$$

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; 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*, by 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,

$$p \textit{ ensures } q$$

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. Formally,

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

It follows from this definition 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 some time 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*, denotes 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.

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

Transitivity.

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

Disjunction. In the following m ranges over any arbitrary set, and m is not free in q:

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

Remark 2.5 (*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 definitions, 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 [1].

Reflexivity, antireflexivity.

$$p \text{ unless } p, \qquad p \text{ unless } \neg p$$

Consequence weakening.

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

Stable conjunction.

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

We need the following facts about constants; see [6] 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,

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

Corollary 2.6. *For any predicate p over program variables u,*

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

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.

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

2.1.6. Substitution axiom

The 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

$$p \mapsto q$$

It is important to note that when several programs are composed (see Section 2.2), the substitution axiom can be applied only with an invariant of the composite program.

2.2. Program composition through union

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

Programs F and G may be thought of as executing asynchronously in $F \square G$. 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; in this view, the bag program, the producer, and the consumer are composed through the union operator to form a system.

The following theorem is fundamental for understanding union. It says that an *unless* property holds in $F \square G$ iff it holds in both F and G; an *ensures* property holds in $F \square G$ 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.

$$
\begin{aligned}
& p \textit{ unless } q \quad \text{in } F \square G \\
\equiv\ & p \textit{ unless } q \quad \text{in } F \ \wedge\ p \textit{ unless } q \quad \text{in } G
\end{aligned}
$$

$$
\begin{aligned}
& p \textit{ ensures } q \quad \text{in } F \square G \\
\equiv\ & (p \textit{ unless } q \quad \text{in } F \ \wedge\ p \textit{ ensures } q \quad \text{in } G) \\
& \vee (p \textit{ ensures } q \quad \text{in } F \ \wedge\ p \textit{ unless } q \quad \text{in } G)
\end{aligned}
$$

Corollary 2.7.

$$
\frac{p \textit{ unless } q \quad \text{in } F,\ p \text{ stable} \quad \text{in } G}{p \textit{ unless } q \quad \text{in } F \square G}
$$

Corollary 2.8.

$$
\frac{e \text{ constant} \quad \text{in } F,\ e \text{ constant} \quad \text{in } G}{e \text{ constant} \quad \text{in } F \square G}
$$

Corollary 2.9.

$$
\frac{p \text{ stable} \quad \text{in } F,\ p \text{ invariant} \quad \text{in } G}{p \text{ invariant} \quad \text{in } F \square G}
$$

Corollary 2.10.

$$
\frac{p \textit{ ensures } q \quad \text{in } F,\ p \text{ stable} \quad \text{in } G}{p \textit{ ensures } q \quad \text{in } F \square G}.
$$

If variable x cannot be accessed by a program F (i.e., x is local to some other program), then x is constant in F; hence—using the constant formation rule—any expression, $e(x)$, over x is constant in F. This fact is expressed in Corollary 2.11. We write "x is not accessed in F" in quotes because this is not a proposition in our logic.

Corollary 2.11.

$$\frac{\text{"}x \text{ is not accessed in } F\text{"}}{e(x) \text{ constant in } F}$$

We shall also use properties of the following form in the specification of F, where P and Q are arbitrary properties. In the following, G is quantified over programs:

$$\langle \forall G :: (P \text{ in } G) \Rightarrow (Q \text{ in } F \,\Box\, G) \rangle$$

This says that if P is a property of any program G then Q is a property of $F \,\Box\, G$. This is a convenient way of specifying the properties of the environment with which F may be composed and the resulting properties of the composite program. This kind of formula will be crucial in specifications of concurrent objects, because the objects typically assume certain properties of their environments.

3. Specification methodology and bag specification

3.1. 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 will have 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 Fig. 1, 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 and 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

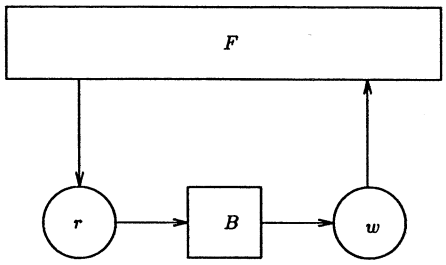

Fig. 1. A program B and its environment F.

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.

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

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

Then it follows that

$$X \preccurlyeq Y \equiv X = \phi \vee X = Y$$

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 and W are free variables (of type, data or ϕ).

P1. $r \preccurlyeq R$ stable in B.

P2. $W \preccurlyeq w$ 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.

3.2. 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. 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 r, b and 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 3.1. All bags in this paper are finite. We treat r and w to be bags of size at most 1. Union of bags u and v is written as $u + v$; the bag $u - v$ is the bag obtained by removing as many items of v as possible from u. The value ϕ is treated as an empty bag.

Combining with P0, P1, P2 defined previously, we obtain the following specification.

3.3. Specification of a concurrent bag of size N, $N > 0$

There exists a variable b, local to B, of type bag such that:

P0. initially $b + w = \phi$ in B

P1. $r \preccurlyeq R$ stable in B

P2. $W \preccurlyeq w$ stable in B

P3. $|b| \leq N$ invariant in B

P4. $r + b + w$ constant in B

P5. $\langle \forall F ::$

(P5.1) $|b| < N \mapsto r = \phi$ in $B \,[]\, F$

(P5.2) $|b| > 0 \mapsto w \neq \phi$ in $B \,[]\, F$

$\rangle$

Remark 3.2. A program that implements B may not have a variable named b. It is merely required that b be computable from the internal variables of B; in this sense, b is an auxiliary variable of B.

Remark 3.3. Since b is a local variable of B, it is not accessed by the environment F. Using Corollary 2.11, $|b| \leq N$ stable in F. Since $|b| \leq N$ is invariant in B, using Corollary 2.9:

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 and w. A different bag specification is given in Section 4.2 where F is assumed to access r and w appropriately.

3.4. Deducing properties from the specification

We deduce one safety and one progress property from the specification given in Section 3.3.

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

Proof. 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 Section 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 (Section 2.1.5) applied to the above two
$w = \phi \ \wedge \ |r+b| \leq N+1$ *unless* $|r+b| \leq N$
, rewriting the left-hand side and weakening the consequence (Section 2.1.5)
$w = \phi$ *unless* $|r+b| \leq N$
, substitution axiom applied to the left-hand side with invariant $|r+b| \leq N+1$ (see P6)
$w = \phi$ *unless* $|b| < N \ \vee \ r = \phi$
, consequence weakening (Section 2.1.5) □

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

Proof. In the following proof all properties are in $B \,\square\, F$ 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 $B \,\square\, F$
, Corollary 2.7
$|b| > 0 \ \mapsto \ w \neq \phi$
, from P5.2
$|b| > 0 \ \wedge \ w = \phi \ \mapsto \ |b| < N \ \vee \ r = \phi$
, 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 □

3.5. Bag concatenation

Let B_1 implement a bag of size M, $M > 0$, with input and output variables, r and v, respectively, and B_2 implement a bag of size N, $N > 0$, with input and output

variables v and w, respectively. We show that $B = B_1 \Box B_2$ implements a bag of size $M+N+1$ with input and output variables r and w, respectively. The arrangement is shown pictorially in Fig. 2. Observe that r and w are different variables (i.e., B_1, B_2 are not connected cyclically) and hence, r cannot be accessed by B_2 nor can w be accessed by B_1.

We first rewrite the specification from Section 3.3 for B_1 and B_2. In the following, R, V and W are free variables.

There exists b_1, local to B_1, such that

P0. initially $b_1 + v = \phi$ in B_1

P1. $r \preccurlyeq R$ stable in B_1

P2. $V \preccurlyeq v$ stable in B_1

P3. $|b_1| \leq M$ invariant in B_1

P4. $r + b_1 + v$ constant in B_1

P5. $\langle \forall G ::$
(P5.1) $|b_1| < M \mapsto r = \phi$ in $B_1 \Box G$
(P5.2) $|b_1| > 0 \mapsto v \neq \phi$ in $B_1 \Box G$
$\rangle$

There exists b_2, local to B_2, such that

P0. initially $b_2 + w = \phi$ in B_2

P1. $v \preccurlyeq V$ stable in B_2

P2. $W \preccurlyeq w$ stable in B_2

P3. $|b_2| \leq N$ invariant in B_2

P4. $v + b_2 + w$ constant in B_2

P5. $\langle \forall H ::$
(P5.1) $|b_2| < N \mapsto v = \phi$ in $B_2 \Box H$
(P5.2) $|b_2| > 0 \mapsto w \neq \phi$ in $B_2 \Box H$
$\rangle$

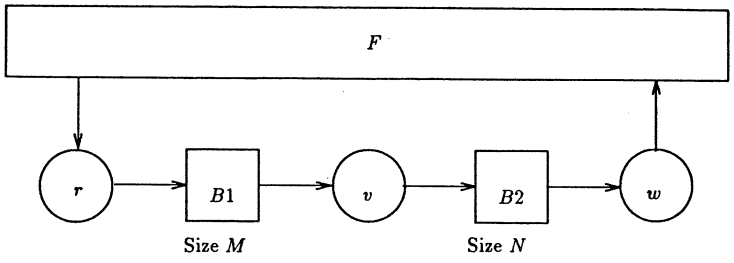

Fig. 2. Concatenation of bags B_1, B_2. $B = B_1 \Box B_2$ has size $M+N+1$.

The properties of B, to be proven, are (again from Section 3.3):

There exists b, local to B, such that

P0. initially $b+w=\phi$ in B

P1. $r \preccurlyeq R$ stable in B

P2. $W \preccurlyeq w$ 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 $B \,[]\, F$
(P5.2) $|b| > 0 \mapsto w \neq \phi$ in $B \,[]\, F$
$\rangle$

We start the proof by defining b to be b_1+v+b_2. To show that b is local to B observe that b_1+v+b_2 is constant in the environment F since none of the variables, b_1, v or b_2, can be accessed by F.

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

initially $b_1+v=\phi$
, from P0 of B_1
initially $b_2+w=\phi$
, from P0 of B_2
initially $b_1+v+b_2+w=\phi$
, from the above two
initially $b+w=\phi$
, definition of b □

Proof of P1.

r constant in B_2
, r cannot be accessed by B_2
$r \preccurlyeq R$ stable in B_2
, constant formation; a constant predicate is stable
$r \preccurlyeq R$ stable in B_1
, from P1 of B_1
$r \preccurlyeq R$ stable in $B_1 \,[]\, B_2$
, from Corollary 2.7 □

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

Proof of P3.

$|b_1| \leq M$ invariant in $B_1 \square B_2$
, from P6 applied to B_1 (with $F = B_2$)
$|v| \leq 1$ invariant in $B_1 \square B_2$
, definition of v
$|b_2| \leq N$ invariant in $B_1 \square B_2$
, from P6 applied to B_2 (with $F = B_1$)
$|b_1 + v + b_2| \leq M + N + 1$ invariant in $B_1 \square B_2$
, from the above three $\square$

Proof of P4. $r + b + w = r + b_1 + v + b_2 + w$:

$r + b_1 + v$ constant in B_1
, from P4 of B_1
$b_2 + w$ constant in B_1
, b_2 and w cannot be accessed by B_1
$r + b_1 + v + b_2 + w$ constant in B_1
, constant formation rule
$r + b_1 + v + b_2 + w$ constant in B_2
, similarly
$r + b_1 + v + b_2 + w$ constant in $B_1 \square B_2$
, Corollary 2.8 $\square$

Proof of P5. Consider any arbitrary F. Setting G to $B_2 \square F$ in P5 for B_1 we get:

$$|b_1| < M \mapsto r = \phi \quad \text{in } B_1 \square B_2 \square F \tag{1}$$

$$|b_1| > 0 \mapsto v \neq \phi \quad \text{in } B_1 \square B_2 \square F \tag{2}$$

Similarly, setting H to $B_1 \square F$ in P5 for B_2 we get:

$$|b_2| < N \mapsto v = \phi \quad \text{in } B_1 \square B_2 \square F \tag{3}$$

$$|b_2| > 0 \mapsto w \neq \phi \quad \text{in } B_1 \square B_2 \square F \tag{4}$$

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

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

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 $B \square F$, i.e., $B_1 \square B_2 \square F$:

$$
\begin{array}{ll}
 & |b| < M+N+1 \\
\Rightarrow & |b_1| < M \ \vee \ v = \phi \ \vee \ |b_2| < N \\
 & \quad \text{, since } b = b_1 + v + b_2 \\
\mapsto & |b_1| < M \ \vee \ v = \phi \\
 & \quad \text{, above fact about } \mapsto \text{ and (3)} \\
\mapsto & |b_1| < M \ \vee \ r = \phi \\
 & \quad \text{, above fact about } \mapsto \text{ and P8 for } B_1 \ (v = \phi \text{ stable in } B_2 \,\Box\, F) \\
\mapsto & r = \phi \\
 & \quad \text{, above fact about } \mapsto \text{ and (1)} \qquad \Box
\end{array}
$$

3.6. A note on the bag specification

It is interesting to note that the specification of Section 3.3 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,

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

Since $N = 0$, it follows (from P6 and the definition of b) that $|b| = 0$ is an invariant in $B \,\Box\, F$. 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. It may seem that the following strengthening of the progress specification could generalize the specification to $N \geq 0$.

$$
\begin{array}{l}
\langle \forall F :: \\
\quad |b| < N \ \vee \ w = \phi \mapsto r = \phi \quad \text{in } B \,\Box\, F \\
\quad |b| > 0 \ \vee \ r \neq \phi \mapsto w \neq \phi \quad \text{in } B \,\Box\, F \\
\rangle
\end{array}
$$

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 B to implement the first progress condition (while satisfying P4: $r + b + w$ is constant in B). 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.

4. Alternative specification of a bag

4.1. Communication sequences as auxiliary variables

We propose another bag specification in this section. We take as auxiliary variables

the bags $\hat{r}$ and $\hat{w}$, where $\hat{r}$ is the bag of data items written into r (and similarly $\hat{w}$). Such a bag is typically defined by augmenting the program text appropriately: whenever r is changed, $\hat{r}$ is altered by adding the new value of r to it. As shown in Example 2.4, our logic provides a direct way of defining $\hat{r}$ and $\hat{w}$ without appealing to the program text. In the following, R and W are free variables (of the same type as data) and X and Y are free variables that are of type bag:

A0. initially $(\hat{r}, \hat{w}) = (r, w)$ in $B \square F$

A1. $\hat{r} = X \wedge r = R$ *unless* $\hat{r} = X + r \wedge r \neq R$ in $B \square F$

A2. $\hat{w} = Y \wedge w = W$ *unless* $\hat{w} = Y + w \wedge w \neq W$ in $B \square F$

To understand the relationship between $\hat{r}$ and r and (similarly for $\hat{w}$ and w), note that $\hat{r}$ remains unchanged as long as r is unchanged and $\hat{r}$ may be 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, $\hat{r}$ remains unchanged.)

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

A3. $r \subseteq \hat{r}$ invariant in $B \square F$

A4. $w \subseteq \hat{w}$ invariant in $B \square F$

We prove A3; proof of A4 is similar.

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

$\hat{r} = X \wedge r = R$ *unless* $\hat{r} = X + r \wedge r \neq R$
, from A1
$(\hat{r}, r) = (X, R)$ *unless* $(\hat{r}, r) \neq (X, R) \wedge r \subseteq \hat{r}$
, consequence weakening
$r \subseteq \hat{r}$ stable
, Corollary 2.6

Initially $r \subseteq \hat{r}$ because $r = \hat{r}$. Hence $r \subseteq \hat{r}$ invariant. □

4.2. Bag specification

The following specification is for a bag of size N, $N \geq 0$. Properties R1 and R2 are the same as P1 and P2. In order to overcome the problems described in Section 3.6, we make some assumptions about the environment F. Specifically, we require F to obey a similar access protocol to w and r as the ones obeyed by B for access to r and w: Environment F may write into r only if $r = \phi$ and it may change w, to ϕ, only if $w \neq \phi$. Property R3 states that under these conditions on F, the bag $\hat{w}$ is a subbag of $\hat{r} - r$ (Property R3.1); these two bags differ by no more than N (Property

R3.2); if $\hat{r}$ and $\hat{w}$ differ by no more than N, then r is or will be set to ϕ (Property R3.3); if $|\hat{r}|$ exceeds $|\hat{w}|$, then w is or will be set to non-ϕ (Property R3.4).

R0. initially $w = \phi$ in B

R1. $r \leqslant R$ stable in B

R2. $W \leqslant w$ stable in B

R3. $\langle \forall F :: (B.hypo \text{ in } F) \Rightarrow (B.conc \text{ in } B \,[\!]\, F) \rangle$

where

$B.hypo :: R \leqslant r$ stable, $w \leqslant W$ stable
$B.conc ::$
(R3.1) $\hat{w} \subseteq \hat{r} - r$ invariant
(R3.2) $|\hat{r} - r - \hat{w}| \leqslant N$ invariant
(R3.3) $|\hat{r} - \hat{w}| \leqslant N \mapsto r = \phi$
(R3.4) $|\hat{r}| > |\hat{w}| \mapsto w \neq \phi$

We show, in Section 4.3, that under certain conditions the proposed specification is a refinement of the specification in Section 3.3. Now we deduce a few properties from the given specification. Assuming *B.hypo* in F

R4. $\hat{r} - \hat{w} + w$ constant in B

R5. $\hat{r} - \hat{w} - r$ constant in F.

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

Proof of R4. All properties in the following proof are in B.

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

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

4.3. Proof of refinement

We show that for $N>0$, the specification of Section 4.2 is a refinement of the specification of Section 3.3, *provided that B.hypo holds in the environment F.* In the absence of such a requirement on F, the specification of Section 4.2 does not prescribe B's behavior (that is when B is composed with an arbitrary F) whereas the specification of Section 3.3 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

B.hypo in F, $N>0$, and the properties R0–R3 of Section 4.2

there exists b, local to B, satisfying properties P0–P5 of Section 3.3. We let

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

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

Proof of P0. $b+w=\hat{r}-\hat{w}-r+w$

initially $(\hat{r}, \hat{w})=(r, w)$.

Hence

initially $b+w=\phi$. □

Proofs of P1 and P2. Directly from R1 and R2. □

Proof of P3.

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

Proof of P4. $r+b+w=\hat{r}-\hat{w}+w$

$\hat{r}-\hat{w}+w$ constant
, from R4 □

Proof of P5.1.

$|b|<N \ \wedge \ r\neq\phi$
$\Rightarrow|\hat{r}-\hat{w}-r|<N \ \wedge \ r\neq\phi$
, using the definition of b
$\Rightarrow|\hat{r}-\hat{w}|\leq N$
, $r\subseteq\hat{r}$ (A3)
$\mapsto r=\phi$
, from R3.3

Hence

$$|b| < N \ \wedge \ r \neq \phi \mapsto r = \phi$$

also

$$|b| < N \ \wedge \ r = \phi \mapsto r = \phi$$

, implication

The result follows using disjunction on the above two. □

Proof of P5.2. Similar to that of P5.1. □

5. Specifications of concurrent queue and concurrent stack

We consider two other concurrent data objects—queue and stack—in this section. Each has variables r and 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).

5.1. 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 Section 3.3. In the following "uv" denotes concatenation of sequences u and v and ϕ denotes the null sequence. A queue of size N, $N > 0$, is defined by a program Q where:

There exists a variable q, local to Q, of type sequence such that:

Q0. initially $q = \phi \ \wedge \ w = \phi$ in Q

Q1. $r \preccurlyeq R$ stable in Q

Q2. $W \preccurlyeq w$ stable in Q

Q3. $|q| \leq N$ invariant in Q

Q4. rqw constant in Q

Q5. $\langle \forall F ::$

(Q5.1) $|q| < N \mapsto r = \phi$ in $Q \,[]\, F$

(Q5.2) $|q| > 0 \mapsto w \neq \phi$ in $Q \,[]\, F$

$\rangle$

5.1.1. Queue implements bag

We show that from the specification of Section 5.1 we can deduce the specification in Section 3.3 for an appropriate b. Denote the bag of items in q by $[q]$. Let $b = [q]$.

Proofs of P0, P1 and 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 B—we observe (below all properties are in B):

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

5.2. 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 outcome of the reads.

We propose a specification, analogous to that of Section 4.2, for a program S that implements a stack of size N, $N>0$. Let $\bar{r}$ and $\bar{w}$ denote the sequences of data items written into r and w respectively. Analogous to the definitions of $\hat{r}$ and $\hat{w}$ by (A0, A1 and A2) we define $\bar{r}$ and $\bar{w}$ in $S\,\square\,F$ (where F is any arbitrary environment of S) by (A0′, A1′, A2′) below. As in Section 5.1, concatenations of sequences are shown by juxtapositions and ϕ denotes the null sequence.

A0′. initially $(\bar{r}, \bar{w})=(r, w)$ in $S\,\square\,F$

A1′. $\bar{r}=X \wedge r=R$ *unless* $\bar{r}=Xr \wedge r\neq R$ in $S\,\square\,F$

A2′. $\bar{w}=Y \wedge w=W$ *unless* $\bar{w}=Yw \wedge w\neq W$ in $S\,\square\,F$

Now, we define $X\,N\,Y$—a binary relation between sequences X and Y expressing that Y is the complete output sequence of a stack of size N given X as the input sequence. For $N\geqslant 0$, it is the strongest relation satisfying

$$\phi\,N\,\phi$$

and

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

where a is any arbitrary data item. 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 output of a, 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, 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—using induction on the length of X—can be proven about this relation. In particular,

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

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

Now we specify program S that implements a concurrent stack of size N, $N>0$, with shared variables r and w as before. Note that S3.1 is the only property that differs substantially from the corresponding property of the bag in Section 4.2.

S0. initally $w=\phi$ in S

S1. $r \preccurlyeq R$ stable in S

S2. $W \preccurlyeq w$ stable in S

S3. $\langle \forall F::(B.hypo \text{ in } S \Rightarrow (B.conc \text{ in } S \,[]\, F)\rangle$

where

$B.hypo::R \preccurlyeq r$ stable, $w \preccurlyeq W$ stable

$B.conc::$ {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

(S3.2) $|\bar{r}| \leqslant |r|+|\bar{w}|+N$ invariant

(S3.3) $|\bar{r}| \leqslant |\bar{w}|+N \mapsto r=\phi$

(S3.4) $|\bar{r}| > |\bar{w}| \mapsto w \neq \phi$

5.2.1. Stack implements a bag

We show that from the specification of a concurrent stack we can deduce the bag specification in Section 4.2 for $N>0$. Note that $\hat{r}=[\bar{r}]$, $\hat{w}=[\bar{w}]$. It follows that $\hat{r}-r=[\hat{r}-r]$. Proofs of Properties R0–R3 are entirely straightforward, except for R3.1: $\hat{w} \subseteq \hat{r}-r$.

$(\bar{r}-r)\ N\,\bar{z}$
$\Rightarrow [\bar{r}-r]=[\bar{z}]$
, 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
$\langle \exists \bar{z}::(\bar{r}-r)\ N\,\bar{z} \wedge \bar{w} \sqsubseteq \bar{z}\rangle$
$\Rightarrow \hat{w} \subseteq \hat{r}-r$
, from the above using $[\bar{w}]=\hat{w}$ and $[\bar{r}-r]=\hat{r}-r$.

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.

6. Refinement of the bag specification

We refine the specification of Section 4.2 as a step toward implementing a bag. It can be shown from that specification that concatenation of two bags of size M and N, where M and N are nonnegative, results in a bag of size $M+N+1$; the proof is similar in structure to the proof in Section 3.5; also, a similar proof for a queue appears in [7]. 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. 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 $(r \neq \phi) \wedge (w = \phi)$ resulting in $w \neq \phi$ (Property T4).

T0. initially $w = \phi$ in B

T1. $r \preccurlyeq R$ stable in B

T2. $W \preccurlyeq w$ stable in B

T3. $r+w$ constant in B

T4. $r \neq \phi$ ensures $w \neq \phi$ in B

6.1. Proof of the refinement

We show that Properties R0–R3 of Section 4.2, for $N=0$, can be deduced from T0–T4. Proofs of R0, R1 and R2 are immediate from T0, T1 and T2. Property R3 is of the form,

$$\langle \forall F :: (B.hypo \text{ in } F) \Rightarrow (B.conc \text{ in } B \,\square\, F) \rangle$$

We first show that

T5: $\langle \forall F ::$
$(B.hypo \text{ in } F) \wedge (\text{T0} \wedge \text{T3}) \Rightarrow (\hat{r} = \hat{w} + r \text{ invariant in } B \,\square\, F) \rangle$

Proof.

$\hat{r} - \hat{w} + w$ constant in B
, from R4
$r+w$ constant in B
, from T3
$\hat{r} - \hat{w} + w - (r+w)$ constant in B
, constant formation
$\hat{r} - \hat{w} - r$ constant in B
, simplifying the above expression
$\hat{r} - \hat{w} - r$ constant in F
, from R5 assuming $B.hypo$ in F
$\hat{r} - \hat{w} - r$ constant in $B \,\square\, F$
, Corollary 2.8

Initially

$\hat{r} - \hat{w} - r = \phi$ in $B \,\square\, F$

, from $\hat{r} = r$ and $\hat{w} = w = \phi$

Hence

$\hat{r} - \hat{w} - r = \phi$ invariant in $B \,\square\, F$,

i.e.,

$\hat{r} = \hat{w} + r$ invariant in $B \,\square\, F$. □

Next we prove the four properties in *B.conc*, from *B.hypo* in F, T1, T2, and $\hat{r} = \hat{w} + r$ invariant in $B \,\square\, F$.

Proof of R3. Consider an arbitrary F in which *B.hypo* holds. The proof of R3.1 is immediate given T5. The proof of R3.2 is immediate given T5 and that $N = 0$. To prove R3.3 we have to show that

$|\hat{r}| \leqslant |\hat{w}| \mapsto r = \phi$ in $B \,\square\, F$.

In the following all properties are in $B \,\square\, F$:

$|\hat{r}| \leqslant |\hat{w}| \Rightarrow r = \phi$

, from T5

$|\hat{r}| \leqslant |\hat{w}| \mapsto r = \phi$

, using implication rule on the above

To prove R3.4 we have to show that $(|\hat{r}| > |\hat{w}| \mapsto w \neq \phi)$ 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 $R \leqslant r$ stable (from *B.hypo*)

$r \neq \phi$ stable in F

, Corollary 2.6

$r \neq \phi$ *ensures* $w \neq \phi$ in B

, from T4

$r \neq \phi$ *ensures* $w \neq \phi$ in $B \,\square\, F$

, using Corollary 2.10 on the above two

$r \neq \phi \mapsto w \neq \phi$ in $B \,\square\, F$

, definition of $\mapsto$

$|\hat{r}| > |\hat{w}| \mapsto w \neq \phi$ in $B \,\square\, F$

, from T5 $|\hat{r}| > |\hat{w}| \equiv r \neq \phi$ □

7. An implementation

The specification of Section 6 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):

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

The proof that this fragment has 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 $\langle \Box i : 0 < i \leq N+1 :: t(i) \rangle$ as a shorthand for $t(1) \Box t(2) \Box \cdots \Box t(N+1)$, 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 after **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 $\langle \Box i : 0 < i \leq N+1 :: b[i] = \phi \rangle$
  assign $\langle \Box i : 0 < i \leq N+1 :: b[i-1], b[i] := \phi, b[i-1] \quad \text{if } b[i] = \phi \rangle$
end {*bag*}

8. Evaluation of the proposed methodology

Equational notation, as in [2], is generally preferred for specifications because it enables a programmer to separate two concerns: deducing facts about an object independent of its implementation and implementing the given equations to minimize time or storage. As we have argued, the objects that are concurrently accessed are, as yet, not amenable to equational specifications. An important research problem is to determine the conditions under which an object can be specified equationally, and yet accessed concurrently.

Our specification of the bag illustrates how some of the requirements described in the introduction are met. The specification is concise even though it makes concurrent access explicit. The notation admits of efficient manipulation—see, for instance, the lengths of the refinement proofs in Sections 4, 5 and 6. We believe that one of the most important applications of formal specifications is reasoning about an object; a cumbersome notation—such as a programming notation—defeats the main purpose of specification.

It may seem from our specification that we have artificially simplified the problem by introducing shared variables—r and w in case of bag, queue, and stack—through which the object communicates with this environment. How about multiple processes

accessing an object through remote procedure calls, for instance? First, remote procedure call by a process is equivalent to the process storing the procedure parameters in certain (shared) variables which the object can later access; in this sense, we have not introduced any artificial simplicity. Second, accesses by multiple processes (for instance, multiple producers writing into shared variables $r.j, 0 \leq j < L$) can be treated within our theory as follows. Imagine that a *sequencer* process copies some $r.j$ into r whenever $r = \phi$; the different $r.j$ are chosen fairly during the course of the program execution. The index j could be part of the data written into r, if, for instance, the object needs to respond to a request in $r.j$ by writing into $w.j$. (Similarly, outputs for different consumers may be sequenced.) The entire concurrent object may be viewed as a union of an input sequencer, data manager (the program that actually implements the data structures) and the output sequencer, and each of these may be specified individually. We have shown how to specify the data manager; specifications of input/output sequences are entirely straightforward. A direct specification of a queue with multiple input/output variables appears in [8, Section 3.4].

Acknowledgement

It was a suggestion from Leslie Lamport which led me to study this problem and to understand how auxiliary variables can be defined and effectively used. I am indebted to Ambuj Singh for suggesting simplifications for my original specification; his work clearly reveals the importance of formal reasoning as I was unable to justify his simplications using intuitive arguments alone. Comments from the participants of the Ninth International Summer School in Marktoberdorf, FRG, were most helpful; particular thanks go to Jan L.A. van de Snepscheut. Thorough and critical readings of different versions of the manuscript by Mani Chandy, Edgar Knapp and J.R. Rao have improved the presentation.

References

[1] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation* (Addison-Wesley, Reading, MA, 1988).
[2] J. Guttag, Abstract data types and the development of data structures, *Comm. ACM* **20** (1977) 396–404.
[3] B. Hailpern, Verifying concurrent processes using temporal logic, Lecture Notes in Computer Science **129** (Springer, Berlin, 1982).
[4] I. Hayes, ed., *Specification Case Studies* (Prentice-Hall, Englewood Cliffs, NJ, 1987).
[5] L. Lamport, A simple approach to specifying concurrent systems, Digital Systems Research Center Rep. 15, 130 Lytton Avenue, Palo Alto, CA (1986).
[6] J. Misra, Monotonicity, stability and constants, Notes on Unity 10-89, University of Texas at Austin, Austin, TX (1989).
[7] J. Misra, Specifications of concurrently accessed data in: J. van de Snepscheut, ed., *Mathematics of Program Construction*, Lecture Notes in Computer Science **375** (Springer, Berlin, 1989).
[8] A. Singh, Ranking in distributed systems, Ph.D. Thesis, University of Texas at Austin, Austin, TX (1989).