Modular Verification of Concurrent Programs

Brent Hailpern
Computer Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, New York 10598

Susan Owicki
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

Verifying concurrent systems can be difficult because of the complex interactions possible between system components. In this paper, we propose a technique to simplify the task: modular composition of sequential proofs. We model a parallel program as a set of modules that interact by procedure calls. The properties of each module are proved using a sequential-program verification technique. If the modules satisfy a set of constraints presented in this paper, we may compose the modules into a system and the properties of the modules into properties of the system. The constraints ensure that the specifications are *robust* for each module where they are defined or used, in the sense that they are unaffected by current actions of other modules. A specification can be guaranteed robust for module *m* by restricting it to local variables of *m*, or by using monotonic predicates, which once true remain true forever. Our technique can be used to prove safety and liveness properties of parallel programs—the liveness properties are specified using temporal logic.

The research of Susan Owicki was supported by the Defense Advanced Research Project Agency under contract MDA903-79-C-0680.

1. Introduction

Reasoning about programs is much easier if the programs are composed of modules that can be understood independently. Modularity is a central part of many programming languages [7, 20, 21, 32], and has been exploited in specification and verification [8, 9, 31]. In this paper we present a language-independent method for modular verification of parallel programs.

We view a concurrent program as a set of modules that interact through procedure calls. (Systems based on message passing [8, 15] can be modelled using this technique by treating the communication medium as a module [11].) Modules may be either simple or compound. Simple modules consist of sequential code. Examples include processes, monitors [4, 14], ADA tasks [7], and distributed processes [5]; the last two combine process activity with monitor-like procedures. A compound module is simply a set of modules that we choose to treat as an entity; its components may be either simple or compound.

A module specification consists of three components: an invariant, a commitment, and a set of service specifications. The invariant describes the *visible* states of the module; these are the states that can exist when the module is about to interact with other modules. An invariant is a *safety* specification, because it guarantees that certain *bad* states never occur. The dual of safety is *liveness*. Liveness properties guarantee that certain *good* states will occur. For sequential programs, termination is usually the only liveness property of interest. However, many concurrent programs are intended to run forever; their liveness requirements typically take the form of a guarantee that some

service is eventually provided. Liveness specifications, which are expressed using temporal logic, are given by the module commitment. Finally, the service specifications describe the safety and liveness properties of any procedures that the module makes available to its environment.

The remainder of this summary is organized as follows. In section 2 we review a set of tools that will be used in module specifications. In section 3 we discuss the specifications, and argue that the verification system is sound. Sections 4, 5, and 6 are examples that demonstrate our techniques: a bounded buffer, a distributed paging system, and a distributed registration server. Section 7 is a summary and comparison to related work.

2. *Tools*

In this section we briefly review four tools we will use in module specifications: temporal logic, auxiliary variables, histories, and private variables.

Temporal logic is an extension of ordinary logic to include an abstract notion of time. This is accomplished by adding operators for reasoning about program computations. A computation is a sequence of states that can arise during program execution. Informally, the first state in a computation represents the present, and subsequent states represent the future. Computations are not restricted to starting at the beginning of the program, so a future state in one computation can be the present state in another.

The version of temporal logic that we use was developed by Pnueli [26, 27]; it is further described by Lamport [18]. The two basic operators of temporal logic are $\Box$ (henceforth) and $\Diamond$ (eventually). The formula $\Box P$ (*henceforth P*) means that P is true for all points in the computation. The formula $\Diamond P$ (*eventually P*) means that there is some point in the computation at which P is true. When we say that a temporal formula is true for a program, we mean that it is true for all computations of that program.

If we informally interpret the first state of a computation as representing the present and the subsequent states as representing the future, then we can think of the temporal operators as quantifying over time. Under this interpretation, $\Box P$ means that P is true now and will remain true forever. The formula $\Diamond P$ states that either P is true now or it will be true at some time in the future.

Auxiliary variables are used in specifications and proofs; they do not affect the results produced by a module. They are included in the code for convenience in reasoning about the program, but they do not have to be implemented.

One class of auxiliary variables that is useful for reasoning about parallel programs is the class of history variables. A history is an unbounded sequence that records the interactions between modules. History variables have frequently been used in reasoning about concurrent systems [8, 11, 16, 23, 24]. The initial value of a history variable is the null sequence, and the only operation allowed is appending a new value.

Another class of auxiliary variables, called private variables, are useful when dealing with inter-module procedure calls. It is convenient to specify the properties of these procedures in terms of variables that are local to both the called and calling modules. A variable x declared as private [23] in a module M will have one instance for each module that calls M, and the instance corresponding to caller C can be modified only when C calls M. Thus the private variable can be treated as if local in proofs of both M and C. Module M refers to C's version of x as $x[C]$; C refers to its version as $x[*]$.

3. *Module Specifications and Verification*

We have already mentioned that module specifications contain an invariant, a commitment, and service specifications for each procedure. The service specifications express safety properties of the procedure by pre- and post-conditions, which have their usual partial-correctness interpretation. There is also a live-condition that describes liveness properties of the procedure; typically it gives conditions under which the procedure must terminate.

Ideally, the proof that a module meets its specifications should be independent of the code of other modules in the system. To achieve this, we require the assertions used in the specifications and proof of module m to be *robust* for m, meaning that no action of any other module can make the assertion false. One way of ensuring that an assertion is robust is to prove that it is invariant over every statement in the other modules of the system [22]. This approach does not meet our goals for modularity: we want assertions that can be determined to be robust without examining the code of other modules. We indicate below two kinds of assertions that satisfy this requirement. Our verification methodology depends on specifications being ro-

bust in modules where they appear, but it does not depend on the way robustness is established.

The simplest way of guaranteeing that a specification is robust for module m is to restrict it to variables that are local to m. With this approach, the invariant and commitment of a module must use only variables local to that module. The pre- and post-conditions of a service specification are a more complicated case. They must be robust for both the module being specified and the module that calls the procedure, because they will appear in proofs of both modules. For this reason, the pre- and post-conditions are restricted to variables that are local to both the calling and the called module: in effect, to procedure parameters and private variables of the called module. This makes private variables an important part of our approach.

An assertion can also be guaranteed robust for module m if it is monotonic, that is, once it becomes true it remains true forever. A monotonic predicate P is one that satisfies the temporal logic formula

$$P \supset \Box P.$$

For example, a history variable can only be extended, never shrunk, so for any history variable h

$$\forall x(x \preceq h \supset \Box x \preceq h),$$

where $x \preceq h$ means that x is an initial subsequence of h. Monotonic predicates are clearly robust for any module, because they can never be made false. More generally, an assertion is robust for m if it is monotonic with respect to any changes of variables not local to m. For example, if h is a non-local history variable and x is a local sequence, the assertion $x \preceq h$ is robust for m, because it cannot be made false by changes to h. The robustness of this assertion can be deduced immediately from the monotonic growth of history variables. In our module specifications, we will sometimes explicitly mention other monotonic properties.

Now that we have indicated how modules are specified, let us consider their proof. For a simple module, the specifications are verified directly from the code, using proof rules for the language in which the module is written. We will not attempt to give such rules here; Howard has presented safety rules for monitors [16], while liveness rules have been discussed by Pnueli, Owicki, Lamport, and Hailpern [10, 11, 25, 27]. Note that the proof of module A, which calls a procedure in module B, can use the service specifications of that procedure, without considering its code.

We require that the invariant of a simple module holds whenever the module is ready to interact with another module, either by accepting or by initiating a procedure call. At such times we will say that the module is *open*. Thus, a monitor is open when no process is executing within it, or when a process within it is ready to execute a call to a procedure in another module. Similarly, an ADA task is open when it is ready to call a procedure in another task or package, or when it is at a rendezvous point waiting for a call from another module. Requiring the invariant to hold when the module makes calls to another module is more strict than is often required; however, we will see that the result is a much more attractive interpretation of compound module invariants than would otherwise be possible. Verifying this extra requirement causes no difficulties.

For a compound module, the specifications are verified from the specifications of the components, without examining the code. The module invariant and commitment are verified by showing that they are implied by the conjunction of the component invariants and commitments. Procedures of the compound module may be procedures of some component, and in this case the service specifications of the component procedure can be carried over without change. Otherwise the service specifications are proved directly from the code.

There is an interesting point concerning the invariant of a compound module. Because it is implied by the conjunction of the component invariants, it must hold whenever all these invariants hold, that is, whenever all components of the compound module are open. Unfortunately, this may not convey much information, because it is likely that during execution of a system there will be few times when all of the modules are open. Fortunately, a much stronger interpretation of the compound module invariant is possible: at all times the variables in modules that *are* open have values that are consistent with the invariant.

Theorem: Suppose s is a state that can be reached during execution of a compound module M, and let x be a list of variables of components of M that are not open in s. Then, if the invariant of M is $M.I$, it is true in state s that $\exists x(M.I)$.

Proof sketch: Let c be a computation for M that ends in state s. (We will not be too specific about how computations are expressed, but assume that a computation is a sequence of states together with the atomic actions that cause transitions between states.) We will first derive a computation c' that ends in a state s' for which all modules are

open, and all variables not in x have the same value as in s. We derive c' in a sequence of steps. Let $c_0 = c$, and derive c_{i+1} from c_i as follows. If all modules are open in the final state of c_i, let $c' = c_i$. If not, delete the last action in c_i from a module that is not open at the end of c_i ; call this module n. Intuitively, this has the effect of *backing up* the computation by one step of n. The resulting sequence c_{i+1} is still a legitimate computation because the deleted action could affect only variables of n, and these variables are not accessible to subsequent actions of c_i. Also, any variables changed by the deleted action were in the set x, so all variables not in x have the same value in the last states of c_i and c_{i+1}.

The computation c' is thus obtained from c by backing up some modules to a point where all modules are open. In s', the final state of c', the compound invariant $M.I$ must hold because all modules are open. Moreover, $\exists x(M.I)$ holds in state s because the values of all variables not in x are the same in s and s', and $M.I$ holds in s'. □

Informally, the theorem above suggests that the module invariant is in some sense close to the visible behavior of the system it describes. More formally, this result allows us to make deductions about observable behavior of the system. In many cases we can derive useful information about the variables of a module based on the environment in which it is used. For example, suppose simple module A is part of compound module B, and that assertion P on the variables of A is implied by B's invariant. Then we can conclude that P is true whenever A is open, and not just when all modules of B are open. Note that it would be impossible to prove P's invariance directly from the code of A if it is only invariant in the environment of B.

4. Example: Bounded Buffer

For our first example consider a bounded buffer. We assume that there are p producer processes and one consumer process; thus the buffer has a global input history, as well as private input histories for each producer. To be completely rigorous, we should have two output histories: a global one for the module and a private one for the consumer. (An invariant would then state that the two histories are always equal.) Because there is only one consumer process, we will treat the output history of the module as private to the consumer when appropriate.

Variables

in: auxiliary history of item,
initially null
out: auxiliary history of item,
initially null
privIn: private auxiliary history of item,
initially null
empty: auxiliary boolean,
initially true
full: auxiliary boolean,
initially false

Invariant

$$out \preceq in \wedge isMerge(in, privIn) \wedge$$

$$(empty \supset \neg full)$$

Monotonic

History variables are monotonic as described in section 3.

Here *isMerge* is a predicate that is true if its first argument is some merge of the instances of its second argument. The first clause of the invariant states that the output history of the buffer is a prefix of the input history. The second clause states that *in* is composed of exactly those items that have been partitioned among the private histories *privIn*. The third clause asserts that *empty* and *full* are mutually exclusive conditions. The buffer has no liveness commitment that is independent of the procedures it provides to its environment. It provides two such procedures: *put* and *get*.

Operations

put (i: item)
pre: *true*
post:

$$in[*] = in'[*] \ @ \ i$$

live:

$$((at\ put \wedge \Box\Diamond\neg full) \rightsquigarrow after\ put) \wedge$$

$$(at\ put \rightsquigarrow \neg empty)$$

get (var i: item)
pre: *true*
post:

$$out = out' \ @ \ i$$

live:

$$((at\ get \wedge \Diamond\neg empty) \rightsquigarrow after\ get) \wedge$$

$$(at\ get \rightsquigarrow \neg full)$$

We use @ as the history concatenation operator; $\sim>$ indicates temporal implication ($a \sim> b$ is defined as $\Box(a \supset \Diamond b)$). The primed auxiliary variables in the post-conditions represent the values those variables had before the operation had executed (thus x' in a post-condition has the same value as x in the pre-condition). We also use the convention that private variables not mentioned by pre-/post-conditions of an operation are not changed by that operation.

The specification of *put* states that the operation can be called at any time. If *put* terminates then the private input history of the calling process has been increased by exactly the item passed to *put;* The operation will terminate if the buffer is repeatedly not full ($\Box\Diamond$ means infinitely often) — $\Diamond\neg full$ is not enough to guarantee termination because another process could use up an empty space and leave this process blocked. (Note that if only one process n can call *put* on a buffer, then $\Diamond\neg full$ does suffice, because the assertion $\neg full$ is then robust with respect to n — no process other than n can cause *full* to become true.) Furthermore, calling *put* guarantees that eventually the buffer will be empty (possibly at the time that *put* is called). The specification of *get* is similar.

We will use the bounded buffer as a building block in the next two sections.

5. *Example: Distributed Paging System*

Our second example comes from the virtual memory and network paging system of Accent [29]. Accent is a communication-oriented operating system for a number of processors (nodes) connected by a network. Many processes can exist at each node. Three goals of Accent are (1) the location of resources in the distributed system should be transparent, (2) it should be possible for any feature provided by the operating-system kernel to be provided instead by a process (for example, memory management), and (3) all services except the basic communication primitives should appear to processes as being provided through a communication (message-passing) interface.

Processes communicate through *ports*. Associated with each port is a *queue* of messages sent to that port, but not yet received. Only one process at a time can have receive access to a given port, though many processes can have send access to it. In this paper we will not deal with the issues of port (or process) creation/destruction; we will assume that the existence of ports, the existence of processes, and the binding between ports and processes are static. We will model ports by the bounded buffers of the previous section. Note that this is an oversimplification, because Accent ports permit error recovery from a send operation on a full port. Rather than model this feature, we will assume that no process allows its input ports to remain full forever:

$$\forall p \in port\ (\Box\Diamond\neg p.full). \qquad (A)$$

We must ensure that all processes that we create have property (A).

Because messages are sent to ports, rather than to processes, intermediary processes can be used to *manage* communication between distinct process groups. A prime example is a network server: if process A runs on node X and process B runs on node Y, then the network server N can provide mirror ports in X and Y so that A and B can communicate. Consider the situation of A sending a message to B. The network server N on X has an alias port B_N also on X. Process A believes that B_N belongs to B, but in fact it belongs to N. Messages sent to B_N are read by N and forwarded to the actual input port of B on Y. We will require that the network provides reliable communication between mirror ports: messages must be delivered to the destination port in the same order they were received by the network's alias port, and messages may not be lost.

In the Accent system, virtual memory is a process-provided resource. Pages (or segments) are stored on backing stores anywhere on the network, without a process knowing that its virtual memory does not resided at its node. In this system the pages are sent in messages to a paging server, which is responsible for backing up the data. In the rest of this section we will discuss the communication between a user process and such a paging system; we show that communication with such a server through a port is equivalent to communication with that server over the network.

General-Port Specification

We will define a general-port module with which a user and a paging server communicate. Our goal is to show that both a bounded buffer (port) and a more complex module (net-port, described below) meet the specification of the general-port.

User u and paging server s use two general-ports for communication: *us* and *su*. The user sends pages to be backed up and requests for backed-up pages to *us*. The server returns backed-

up pages through *su*. The *user* has send access to *us* and receive access to *su;* similarly, the server has send access to *su* and receive access to *us*. No other process may access these general-ports.

Type
item: either a page request or
page contents

Variables
in: auxiliary history of item,
initially null
out: auxiliary history of item,
initially null
empty: auxiliary boolean,
initially true

Invariant

$$out \underline{\preceq} in$$

Operations

put (i: item)
pre: *true*
post:

$$in = in' \ @ \ i$$

live:

$$(at\ put \sim\!> after\ put) \wedge$$

$$(at\ put \sim\!> \neg empty)$$

get (var i: item)
pre: *true*
post:

$$out = out' \ @ \ i$$

live:

$$(at\ get \wedge \Diamond \neg empty) \sim\!> after\ get$$

It is obvious that the bounded buffer (our port) meets these specifications with assumption (A) and the assumption that only one process can call *put*. In the rest of this section we specify a net-port and show that it also meets the specification of a general-port.

Net-Port Description

A *net-port* is a complex module that consists of five sub-modules: three ports (bounded buffers), a network server, and the network. The network server *ns* is an intermediary process that provides alias ports for the sending processes; it has a mapping δ from local port names to remote port names. The network *nw* takes messages from the network server and delivers them to their proper destination ports. Note that both *ns* and *nw* are compound modules; we will not consider their underlying implementations.

The sending process will send messages to the input port *ip*. The network server *ns* takes messages from *ip,* adds the destination address $\delta(ip)$, and sends the composite message to the network port *np*. The network *nw* takes the messages from *np,* strips off the address, and delivers the message to the output port $\delta(ip)$, where the receiving process will receive the message.

Network Server Specification

The network server is shared by many users (or paging servers) at a given node (we will call them all users, for convenience). For each user *u, ns* has an input port ip_u. Associated with each input port is a destination address $\delta(ip_u)$. *Ns* sends messages consisting of pairs [*addr,data*] to *np*.

Invariant

$$\forall u \ (proj(np.in,addr = \delta(ip_u)).data \preceq ip_u.out)$$

We define *proj* as the projection function on histories. The formula $proj(a,b = c)$ means the history obtained from *a* by deleting all messages *m* for which $m.b \neq c$. We extend the record sub-field notation to histories, so that *h.f* for history *h* represents the history with its i^{th} element equal to the field *f* of the i^{th} element of *h*. Therefore, the invariant states that the messages in *np* come from the various *ip's*.

Let *I* be the assertion $\Box\Diamond\neg np.full$.

Commitment

$$(I \supset \forall u \ (\Box\Diamond \neg ip_u.full)) \wedge$$

$$(I \supset \forall d \ \forall u \ (([d] \in ip_u.in) \sim\!> [\delta(ip_u),d] \in np.in)))$$

Assuming that the network repeatedly takes messages from *np, ns* guarantees to repeatedly take messages from all of its input ports and to pass on messages from its input ports to *np*.

Network Specification

The specification of the network module *nw* is similar to that of the network server *ns*. For each node *n*, *nw* has an input port *n.np*. The messages in these ports are pairs: [*addr*,*data*], where *addr* is one of the destination ports. (We let *d* range over destination ports and *n* range over nodes.)

Invariant

$$\forall d\ \exists n\ (d.in \preceq proj(n.np.out, addr = d).data)$$

Let J be the assertion $\forall d\ (\Box\Diamond\neg d.full)$.

Commitment

$$(J \supset \forall n\ (\Box\Diamond\neg n.np.full)) \wedge$$

$$(J \supset (\forall x \forall n\ (x \in n.np.in) \sim>$$

$$(x.data \in x.addr.in))) \wedge$$

$$(J \supset (\forall x \forall n\ (x \in n.np.in \wedge x \notin n.np.out) \sim>$$

$$(\neg x.addr.empty)))$$

The final clause of the commitment implies that if there is a message for a destination port in a network input port, that destination port will eventually be non-empty.

Net-Port Verification

It should be obvious that the net-port does accurately implement the general-port. Consider the compound net-port module *ip - ns - np - nw - δ(ip)*. The *put* procedure for the module is implemented by *ip.put;* the *get* procedure by *δ(ip).get*. The auxiliary variable *empty* is the same as *δ(ip).empty*. Assumption (A) on the destination ports implies (J), which by the *nw* commitment implies (I), which by the *ns* commitment guarantees that the modules that we created do not violate (A) on the input ports. The commitments of *ns* and *nw* guarantee that once *ip.put* is called then $\Diamond\neg\delta(ip).empty$. The general-port invariant is implied by the conjunction of the invariants of the five sub-modules.

6. Example: Distributed Registration Server

Our final example is derived from the registration server of the Grapevine mail system [3]. The Grapevine registration server manages a distributed database of information on *individuals* and *groups*. Groups are named lists; their members may be individuals or other groups. Grapevine uses groups primarily for message distribution, but it can also maintain group membership lists for other applications, such as access lists for a file system. In this discussion, the only functions of the registration server to be considered are those concerned with group membership lists.

Grapevine is a distributed system. A user can interact with any site in the system to add or delete a *name* in a *group*, or ask whether a name is in the group. The database is replicated, and updates are automatically propagated between sites. Timestamps are used to keep the various copies of the database consistent, in a way that will be discussed below. However, it is possible for the databases to be inconsistent for a short time, and this is considered acceptable for the applications in question. For example, if an individual is added to a group, and a message is immediately mailed to the members of the group, the newly added individual may or may not receive the message. A few minutes are normally sufficient for the update to propagate throughout the system.

Grapevine is designed to perform reliably in the face of failures of some host machines and communication channels. For most of our discussion, we will ignore the issues involved in reliable operation, and assume that the underlying hardware operates correctly. We briefly consider the impact of failures at the end of this section.

The major components of the system at each site are a *clock*, a *forwarder* process, which propagates updates to other sites, a *queue* of updates to be processed by the forwarder, and a *group* monitor for each group being maintained. The monitor for group *g* at site *s* is denoted *s.g*. It contains a timestamped record of the most recent add or delete operation performed on each name; initially this record is null and the group has no members. The group monitor provides update procedures *s.g.Add(n: name)* and *s.g.Delete(n: name)*, and a query procedure *s.g.IsMember(n: name, var present: boolean)*. The update procedures *Add* and *Delete* stamp the new update with a time obtained from the local clock. They then merge the timestamped update into the record of updates, discarding any but the most recent update for *n*. Note that the determination of which update is most recent is based on timestamp ordering; this may not correspond to real-time ordering because the clocks at different sites are not perfectly synchronized. In particular, an update may already have been re-

ceived from another site with a timestamp later than the one being processed. After the merge, the new update is added to the forwarder queue for propagation to other sites. The query procedure *s.g.IsMember(n, present)* returns *true* if the most recent update for *n* recorded in *s.g* is an add, and *false* if the most recent update was a delete or if there is no record of any update.

The specifications of the registration server are based on monotonic predicates. The clock module illustrates this use of monotonic predicates in a simple context, so we will describe it first. Next we will give the system specifications, and then the specifications for the other system components. The section ends with a verification of the system specifications.

Clock Specifications

The clock itself is a distributed system, with one component at each site. The components have their own local clocks, which are synchronized so that they never differ by more than a fixed tolerance. The clock operation *s.GetTime* returns a timestamp consisting of the site id and the local time. Timestamps are totally ordered: ties are broken by an arbitrary order among sites.

Types
    timestamp: record
                s:site;
                t:time
            end

Constants
    tolerance: time

Variables — for each site s
    s.clock: timestamp

Invariant

$$\forall s1,s2\ (s1.clock - s2.clock) < tolerance$$

Monotonic

$$(s.clock = t) \supset \Box(s.clock \geq t)$$

Commitment

$$(s.clock = t) \rightsquigarrow (s.clock > t)$$

Operations — for each site s

s.GetTime (var ts: timestamp)
  pre:

$$t < s.clock$$

  post:

$$t < ts < s.clock$$

  live:

$$at\ s.GetTime \rightsquigarrow after\ s.GetTime$$

The monotonic specification, which is really a special kind of invariant, asserts that the local clocks do not run backwards; the commitment guarantees that they actually move forward. The invariant asserts that the maximum difference between local clocks is less than the required tolerance.

The service specifications for *s.GetTime* show how monotonic predicates can be used to specify clock-like behavior. The pre- and post-conditions imply that for any *t*, if $t < s.clock$ when *s.GetTime* is called, on its return it delivers a timestamp whose value is between *t* and the current value of *s.clock*. Here *t* is a free variable that is universally quantified over the pre- and post-conditions, in accordance with Hoare's axioms for procedure calls [13].

Note that the service specifications use the variable *s.clock*, which is not local to the calling module. However, the assertions are still *robust* for the calling module because they are monotonic. The value of *s.clock* can never decrease, so no actions of any module in the system can make the pre- or post-conditions false.

Now these specifications are useful only if they allow us to prove properties of timestamps that are needed in other modules of the system. For example, timestamps from a single site increase monotonically, so it should be possible to prove that after executing

s.GetTime(t1); s.GetTime(t2)

we have $t1 < t2$. The proof outline below indicates how such a proof could be constructed. It should be easy to see that the assertions satisfy the service specifications of s.GetTime.

$\{true\}$

s.GetTime(t1);

$\{t1 < s.clock\}$

s.GetTime(t2);

$\{t1 < t2 < s.clock\}$

In a similar way, we can show that after executing

s1.GetTime(t1); s2.GetTime(t2)

we have $t1 \neq t2$, regardless of whether *s1* and *s2* are different or the same. We can also prove that this program segment leaves $t1 < t2 + tolerance$:

$\{true\}$

s1.GetTime(t1);

$\{t1 < s1.clock\}$

$\{(t1 - tolerance) < s2.clock\}$

s2.GetTime(t2);

$\{(t1 - tolerance) < t2 < s2.clock\}$

$\{t1 < (t2 + tolerance)\}$.

Here $(t1 - tolerance) < s2.clock$ is implied by $t1 < s1.clock$ and the clock invariant.

System Specifications

We have already described the system structure and operations. In the implementation of the system, a record of the most recent update for each name/group pair is maintained at each site. For the purpose of formal specifications, we use auxiliary variables that record all updates, not just the most recent ones.

Types
AddorDel: (add, delete)
StampedName: record
n: name;
t: timestamp;
ad: AddorDel
end

Variables — for all sites s, groups g
s.clock: timestamp
s.g.Updates: auxiliary set of StampedName, initially null
s.g.privUpdates: private auxiliary set of StampedName, initially null

Monotonic

$[n,t,ad] \in s.g.Updates \supset$

$\Box\ [n,t,ad] \in s.g.Updates$

$[n,t,ad] \in s.g.privUpdates \supset$

$\Box\ [n,t,ad] \in s.g.privUpdates$

The first variable is the local clock, which we have discussed already. The other variables are sets that record the updates applied to group g at site s. These variables resemble history variables, and their values grow monotonically. They are sets rather than sequences because the timestamp on each update record gives all the ordering information needed. *Updates* is a global history that records all updates performed at this site; *privUpdates* is a private variable that records the updates performed by each module that calls the system. The invariant asserts that the updates known to the system are exactly those that have been given by some user and recorded in a private variable.

Invariant

$$\bigcup_s s.g.Updates = \bigcup\bigcup_s{}_m\, s.g.privUpdates[m].$$

Note that the invariant does not imply that

$$s.g.Updates = \bigcup_m s.g.privUpdates[m].$$

In fact, any item that was added to *s.g.Updates* because of propagation from another site will not be in *s.g.privUpdates*[*m*], for any *m*. However, it must be in *ss.g.privUpdates*[*m*] for some site *ss*, and this is all the invariant asserts.

Because updates are automatically propagated between sites, the system commitment can assert that once an item has been recorded at any site, it will eventually be recorded at all sites.

Commitment

$\exists s\ ([n,ts,ad] \in s.g.Updates) \rightsquigarrow$

$\forall s\ ([n,ts,ad] \in s.g.Updates)$

Finally the system provides three operations for accessing membership lists. A user can add a name to a group, remove a name from a group, or inquire whether a name is already in a group.

Operations
Add (g: group, n: name)
pre:

$\forall s\ (t_0 < s.clock)$

post:

$\exists s,t,ts\ (ts = [s,t] \wedge t_0 < ts < s.clock \wedge$

$(s.g.privUpdates[*] =$

$s.g.privUpdates'[*] \cup \{[n,ts,add]\}))$

live:

$$at\ Add \sim> after\ Add$$

Delete (g: group, n: name)
pre:

$$\forall s\ (t_0 < s.clock)$$

post:

$$\exists s,t,ts\ (ts = [s,t] \wedge t_0 < ts < s.clock \wedge$$

$$(s.g.privUpdates[*] =$$

$$s.g.privUpdates'[*] \cup \{[n,ts,del]\}))$$

live:

$$at\ Delete \sim> after\ Delete$$

IsMember (g: group, n: name,
var present: boolean)
pre:

$$\forall s\ ([n,ts,ad] \in s.g.Updates)$$

post:

$$((ad = add) \supset (present \vee$$

$$\exists s_1,ts_1\ (ts_1 > ts \wedge$$

$$[n,ts_1,del] \in s_1.g.Updates))) \wedge$$

$$((ad = del) \supset (\neg present \vee$$

$$\exists s_1,ts_1\ (ts_1 > ts \wedge$$

$$[n,ts_1,add] \in s_1.g.Updates)))$$

live:

$$at\ IsMember \sim> after\ IsMember$$

The specifications for *Add* and *Delete* are quite similar, so we will discuss only *Add*. Its pre- and post-conditions imply that the effect of an *Add* is to insert a timestamped entry for name n in a private update set for g at some site s. The invariant implies that $[n,ts,add]$ is also inserted in the global update set at s. The timestamp on entry is not returned to the user. However, it is possible to deduce something about its relationship with the timestamps of previous updates using the techniques discussed under clock specifications.

The specifications for *IsMember* indicate that once an update has reached all sites, its effect will be reflected by *IsMember* until an opposite update is processed at some site.

These specifications are intentionally indefinite about the behavior of the system under certain circumstances. For example, the specification of *IsMember* does not state what answer will be returned if different sites have different values for the update sets. This accurately reflects the nondeterminism in the system. If one adds a name to a group and then asks whether it is present, the answer may be either yes or no depending on where the query is processed, whether the update has been propagated to that site, and whether another user has deleted the name at about the same time or later. However, the system's behavior can be predicted after a certain stabilization period, and this can be proved from the specifications. For example, if an item is added, the post condition of *Add* implies that an add is in a private update set at some site. The invariant implies that it must also be in the global update set at that site. The commitment implies that it will eventually be in the global update sets at all sites. From that point on, an *IsMember* query is guaranteed to return present until a delete is processed at some site.

We now consider the specifications of the remaining components of the system.

Group Specifications

The monitor for group g at site s is named $s.g$. Its data structures are similar to those already described for the system. The chief addition is another private variable *propUpdates*, which records items propagated from the forwarders at other sites. The specifications also refer to the private variable $s.q.in[s.g]$, which is defined in the queue monitor. It is a sequence of tuples of the form $[g,n,ts,add]$ or $[g,n,ts,del]$, which indicates that name n is to be added or deleted to group g with timestamp ts.

Variables
s.g.Updates: auxiliary set of StampedName,
initially null
s.g.privUpdates: private auxiliary set of
StampedName,
initially null
s.g.propUpdates: private auxiliary set of
StampedName,
initially null

All of the variables grow monotonically in the way described by the system monotonic specification.

The invariant asserts that the elements of *s.g.Updates* are those from calls to *Add* or *Delete* at this site plus those propagated from other sites. It

also states that items placed in the queue by the monitor correspond to adds or deletes processed at this site. The commitment implies that every item added or deleted locally is eventually passed on to the queue for propagation.

Invariant

$(s.g.Updates =$

$(\bigcup_m s.g.privUpdates[m]) \cup (\bigcup_m s.g.propUpdates[m]))$

$\wedge$ $(([gg,n,ts,ad] \in s.q.in[s.g]) \supset$

$(gg = g \wedge [n,ts,ad] \in s.g.Updates))$

Commitment

$[n,t,ad] \in s.g.Updates \rightsquigarrow [g,n,t,ad] \in s.q.in$

The operations provided by a group monitor are *Add, Delete,* and *IsMember,* which are quite similar to the operations provided by the system, plus *Propagate,* which is called by forwarders at other sites.

Operations

s.g.Add (n: name)
pre:

$t_0 < s.clock$

post:

$\exists t,ts(ts = [s,t] \wedge t_0 < ts < s.clock \wedge$

$(s.g.privUpdates[*] =$

$s.g.privUpdates'[*] \cup \{[n,ts,add]\})$

live:

$(at\ s.g.Add \wedge \Box\Diamond \neg q.full) \rightsquigarrow after\ s.g.Add$

s.g.Delete
pre:

$t_0 < s.clock$

post:

$\exists t,ts\ (ts = [s,t] \wedge t_0 < ts < s.clock \wedge$

$s.g.privUpdates[*] =$

$s.g.privUpdates'[*] \cup \{[n,ts,del]\}$

live:

$(at\ s.g.Delete \wedge \Box\Diamond \neg q.full) \rightsquigarrow after\ s.g.Delete$

s.g.IsMember (n: name, var present: boolean)
pre:

$[n,ts,ad] \in s.g.Updates$

post:

$((ad = add) \supset (present \vee$

$\exists ts_1\ (ts_1 > ts \wedge$

$[n,ts_1,del] \in s.g.Updates))) \wedge$

$((ad = del) \supset (\neg present \vee$

$\exists ts_1\ (ts_1 > ts \wedge$

$[n,ts_1,add] \in s.g.Updates)))$

live:

$at\ s.g.IsMember \rightsquigarrow after\ s.g.IsMember$

s.g.Propagate(n: name, ts: timestamp,
ad: AddorDel)
pre: *true*
post:

$s.g.propUpdates[*] =$

$s.g.propUpdates'[*] \cup \{[n,ts,ad]\}$

live:

$at\ s.g.Propagate \rightsquigarrow after\ s.g.Propagate$

Comparing the *Add, Delete,* and *IsMember* operations provided by group *s.g* and those provided by the system, we see that the chief difference is that site *s,* which is a quantified variable in the system specifications, is bound in the group specifications. The only other difference is in the live-conditions of *Add* and *Delete.* In the system, these operations were guaranteed to terminate, while the group operations can be blocked trying to add items to a full queue. In fact, the system operations are implemented by the group operations, but at the system level we can prove that such blocking will not go on forever, because the *forwarder* will ensure that the queue does not remain full.

The *propagate* procedure is not visible at the system level, because the mechanics of propagation do not need to be visible to the user. Its post-condition states that the propagated item is added to the private *propUpdates* set; the invariant implies that it is added to *s.g.Updates* as well.

Queue Specifications

We have already discussed the specifications of a queue, and will not repeat them here.

Forwarder Specifications

S.forwarder is a process that reads items from *s.q* and propagates the updates to other sites. It does this by calling the procedure *ss.g.propagate* at each site *ss*. The forwarder defines no visible variables and provides no operations, so its specification consists of an invariant and a commitment. The invariant states that all items propagated by *s.forwarder* came from the queue, while the commitment guarantees that the queue will not remain full and that any item in the queue will eventually be propagated.

Invariant

$$\forall ss\ ([n,t,ad] \in ss.g.propUpdates[s.forwarder] \supset [g,n,t,ad] \in s.q.out)$$

Commitment

$$\Box\Diamond \neg s.q.full \wedge ([g,n,t,ad] \in s.q.in \sim> \forall ss \neq s\ ([n,t,ad] \in ss.g.propUpdates[s.forwarder]))$$

System Verification

We now show that the system specifications can be derived from the specification of the system components. We will not verify the component specifications; this would be done either by proving them directly from the code or by decomposing them into sub-modules.

Invariant

The system invariant is

$$\bigcup_s s.g.Updates = \bigcup_s \bigcup_m s.g.privUpdates[m].$$

To prove this, first note that the group invariant implies that for each *s.g*,

$$s.g.Updates = (\bigcup_m s.g.privUpdates[m]) \cup (\bigcup_m s.g.propUpdates[m]).$$

Now, *s.g.propUpdates*[*m*] is *null* for all modules *m* except the *forwarders*, because they are the only modules that call *s.g.propagate*. So

$$s.g.Updates = (\bigcup_m s.g.privUpdates[m]) \cup (\bigcup_{ss} s.g.propUpdates[ss.forwarder]). \quad (*)$$

But, any element in *s.g.propUpdates*[*ss.forwarder*] must also be in *ss.q.out* (forwarder invariant), and thus in *ss.q.privIn*[*ss.g*] (queue invariant), and thus in *ss.g.privUpdates*[*m*] for some module *m* (group invariant). Thus

$$s.g.propUpdates[ss.forwarder] \subseteq \bigcup_m ss.g.privUpdates[m]. \quad (**)$$

Taking the union over *s* of both sides of equation (*) above, and simplifying the right-hand side with (**) gives the required result.

Monotonicity

The system monotonic specification follows immediately from the group monotonic assertions.

Commitment

We need to show that if [*n,ts,ad*] is in *s.g.Updates* for any *s*, then eventually it will be in *ss.g.Updates* for all *ss*. We first observe that the commitment for group *s.g* implies

$$[n,t,ad] \in s.g.Updates \sim> [g,n,t,ad] \in s.q.in.$$

The commitment for *s.forwarder* is

$$[g,n,t,ad] \in s.q.in \sim> \forall ss \neq s\ ([n,t,ad] \in ss.g.propUpdates[s.forwarder]).$$

Combining these commitments with the group invariant, which implies that

$$ss.g.propUpdates[s.forwarder] \subseteq ss.g.Updates,$$

gives the required result.

Operations

The system operations *Add, Delete,* and *IsMember* are implemented by non-deterministically choosing a site and calling the corresponding operation of the group monitor at that site. It is easy to see that this implementation satisfies the pre- and post-conditions of the system operations, because the site operations have exactly the same pre- and post-conditions except for the quantification over site. The live-condition for *IsMember* is also obviously satisfied, but the live-conditions for *Add* and

Delete require further comment. The live-condition for *s.g.Add* is

$$(at\ s.g.Add \wedge \Box\Diamond\neg s.q.full) \sim> after\ s.g.Add.$$

The commitment for *s.forwarder* implies $\Box\Diamond\neg s.q.full$, so calls to *s.g.Add* always terminate. But then the system *Add* always terminates as well. The proof for *Delete* is the same.

Fault Tolerance

Up to now, we have ignored the possibility of site crashes and failures in the communication medium. The Grapevine registration server can provide reliable service in the face of a number of such failures. This is chiefly accomplished by having each site periodically transmit its update sets to all other sites. When a site receives such a transmission, it can merge the received update set with its own, based on the timestamp order. Thus if site s_1 is down when an update is processed at site s_2, the forwarder at s_2 can simply omit propagation to s_1. When s_1 recovers, it will learn of the update through the regular exchange of update sets.

If we incorporate this mechanism in our specifications, there is no change at all in the safety properties of the various modules. All of the invariants, monotonic predicates, and pre- and post-conditions are valid for the new system. Liveness properties change: for example, the forwarder commitment cannot guarantee that an update will propagate to all sites. The system commitment can remain much the same: if an update is know at any site, it will eventually be know at all sites, *provided that failures remain within certain bounds.* Verifying this commitment in a system with failures would require a careful definition of these bounds.

7. Conclusions and Comparison to Related Work

The specification and verification technique presented here greatly simplifies the process of reasoning about concurrent programs. Because the proofs from code involve only sequential reasoning, their construction is relatively straight forward. The specifications of each module can be verified without consideration of other modules (except for the service specifications of called procedures). Composition of system specifications is then based on logical reasoning, without any reference to code.

Perhaps the work most closely related to ours is the Gypsy language [8]. Gypsy provides modular specifications for a language with processes communicating through message buffers. It includes a built-in form of private history variables. It does not treat liveness at all, and instead of invariants uses *blocked assertions,* which hold when a process is blocked at message transmission. The *blocked assertion* of a compound module holds when all of its components are blocked. In most systems this would never occur, and the assertion is primarily used to show that deadlock is impossible.

Jones [17] describes a quite different modular method for safety proofs. He allows processes to share variables directly, and uses *guarantees* to capture what each process promises not to do to the shared variables.

There has been much work on proof techniques using temporal logic. Our work differs from the following because our technique depends on the modularity of the concurrent programs we prove and the independence of those modules; theirs does not. Schwartz, Melliar-Smith, Keller, Ramamritham, and Wolper have developed new, higher-level temporal operators [28, 30, 33]. Clarke and Emerson [6] have a method of constructing concurrent programs in which the synchronization skeleton of a program is automatically synthesized from branching-time temporal logic specifications. Ben-Ari and Pnueli have developed proofs of some complex synchronization algorithms [1]; together with Manna, they have explored aspects of branching-time temporal logic [2]. Lamport has developed a new interpretation for the conventional Hoare partial-correctness assertions in the context of concurrent programs [19]. He also uses temporal logic to specify liveness properties.

8. References

1] Mordechai Ben-Ari and Amir Pnueli. Temporal logic proofs of concurrent programs. Submitted for publication, 1980. (Department of Mathematical Sciences, Tel Aviv University.)

2] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. *Eighth Annual ACM Symposium on Principles of Programming Languages* (Williamsburg), pages 164–176, January 1981.

3] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine. In preparation, Computer Science Laboratory, Xerox Palo Alto Research Center, April 1981.

4] Per Brinch Hansen. *Operating System Principles.* Prentice-Hall, 1973.

5] Per Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM,* **21** (11):934–941, November 1978.

6] E. M. Clarke and E. A. Emerson. The design and synthesis of synchronization skeletons using temporal logic. *Proceedings of the Workshop on Programming Logics* (Yorktown Heights), May 1981. To be published by Springer-Verlag.

7] Department of Defense. Preliminary ADA reference manual. *Sigplan Notices,* **14** (6), June 1979.

8] Donald I. Good and Richard M. Cohen. Principles of proving concurrent programs in Gypsy. *Sixth Annual ACM Symposium on Principles of Programming Languages* (San Antonio), pages 42–52, January 1979.

9] John V. Guttag, Ellis Horwitz, and David Musser. Abstract data types and software validation. *Communications of the ACM,* **21** (12):1048–1064, January, 1979.

10] Brent T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* PhD Thesis, Stanford University, 1980. Available as technical report 195, Computer Systems Laboratory, Stanford University, August 1980.

11] Brent Hailpern and Susan Owicki. Modular verification of computer communication protocols. Submitted for publication, 1981. Available as research report RC8726, IBM T. J. Watson Research Center, March 1981.

12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM,* **12** (10):576+, October 1969.

13] C. A. R. Hoare. Procedures and parameters: An aximomatic approach. *Symposium on the Semantics of Algorithmic Languages,* pages 102–116. Springer-Verlag, 1971.

14] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM,* **17** (10):549–557, October 1974.

15] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM,* **21** (8):666–677, August 1978.

16] John H. Howard. Proving monitors. *Communications of the ACM,* **19** (5):273–279, May 1976.

17] C. B. Jones. Tentative steps towards a development method for interfering programs. In preparation, Programming Research Group, Oxford University, October 1980.

18] Leslie Lamport. "Sometimes" is sometimes "not never": On the temporal logic of programs. *Seventh Annual ACM Symposium on Principles of Programming Languages* (Las Vegas), pages 174–185, January 1980.

19] Leslie Lamport. The 'Hoare Logic' of concurrent programs. *Acta Informatica,* **14** :21–37, 1980.

20] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM,* **20** (8):564–576, August 1977.

21] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual (version 5.0). Technical report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

22] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM,* **19** (5):279–285, May 1976.

23] Susan Owicki. Specifications and proofs for abstract data types in concurrent programs. In F. L. Bauer and M. Broy, editors, *Program Construction,* pages 174–197. Springer-Verlag, 1979.

24] Susan Owicki. Specification and verification of a network mail system. In F. L. Bauer and M. Broy, editors, *Program Construction,* pages 198–234. Springer-Verlag, 1979.

25] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. Submitted for publication, 1980. (Computer Systems Laboratory, Stanford University.)

26] Amir Pnueli. The temporal logic of programs. *The 18th Annual Symposium on Foundations of Computer Science* (Providence), pages 46–57, October 1977.

27] Amir Pnueli. The temporal semantics of concurrent programs. *Semantics of Concurrent*

Computation, pages 1–20. Springer-Verlag, 1979.

28] Krithivasan Ramamritham and Robert M. Keller. On synchronization and its specification. Submitted for publication, 1981. (Department of Computer Science, University of Utah.)

29] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In preparation, Department of Computer Science, Carnegie-Mellon University, April 1981.

30] Richard L. Schwartz and P. M. Melliar-Smith. Temporal logic specifications of distributed systems. *Proceedings of Second International Conference on Distributed Systems* (Paris), April 1981.

31] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Communications of the ACM,* **20** (8):553–564, August 1977.

32] N. Wirth. Modula: A language for modular multiprogramming. *Software Practice and Experience,* **7** (1):3–35, January-February 1977.

33] Pierre Wolper. Temporal logic can be more expressive. In preparation, Computer Science Department, Stanford University, 1981.