CCSP -A Formal System for Distributed Program Debugging

UMR C0mputer Science Technical Report Number 94-30

HANAN LUTFIYYA, [1] BRUCE Mc. MILLIN,
BETH ARROWSMITH, and CRISTINA SERBAN [2]

Abstract

One major problem with programming in a parallel/distributed environment is the difficulty in debugging the programs owing to the complex interactions of their component processes. Complete knowledge of the program's state is not generally attainable in a distributed system. This paper presents a distributed system for debugging distributed programs that allows for the execution and evaluation of embedded assertions expressed in Hoare's CSP [6]. We show examples of the use of this system, prove its correctness, and describe how the system can be used for the more general case of ensuring an application's correctness at run-time.

Keywords

Parallel Programming, Distributed Programming, Formal Methods, Run-Time Satisfaction, Executable Assertions, Debugging

1 Introduction

In this paper, we will discuss our method of detecting global predicates in a distributed program. Earlier methods either assume a limitation on the global predicates or are computationally intense or are centralized. This section is divided into three parts. The first part describes our model of distributed systems, the second part describes other methods with respect to the model and the third section describes our approach.

1.1 Model of Distributed Environment

A distributed program is one that runs (executes) on multiple processors connected by a communication network and consists of a set of n processes denoted by $\{\rho_1, \rho_2, ..., \rho_n\}$. The global state at any point in time is a member of the cartesian product of the processes' local states. The local state of a process is defined by the value of all its

[1]Supported in part by the National Sciences and Engineering Research Council of Canada (NSERC)

[2]Supported in part by the National Science Foundation under Grant Number MSS-9216479 and, in part, from the Air Force Office of Scientific Research under contract number F49620-92-J-0546 and F49620-93-1-0409, and, in part by a grant from the University of Missouri Research Board.

variables including its program counter. We assume that communication states are captured in the local states of the processes. Thus, the state is distributed across the network and no process has access to the global state at any instant.

Consistent with Hoare's CSP [6], there are three types of events that result in a new state of a process: send (!), receive (?), or internal action.

Definition 1.1 *Event e precedes event f in an execution, i.e., $e \rightarrow f$, if and only if any one of the following conditions hold [7]:*

1. *e and f are events of the same process, and e occurs before f,*
2. *e is a send event, and f is the corresponding receive event, or*
3. *there exists an event g, such that $e \rightarrow g$ and $g \rightarrow f$.*

Two events e, f are said to be *causally related* if either $e \rightarrow f$ or $f \rightarrow e$ holds. Otherwise, they are said to be *concurrent* events.

We are interested in a single execution of a distributed program. Each process ρ_i in an execution of a distributed program generates a finite sequence of local states. A happened-before relation can be defined between states [5] similar to that of Lamport's happened-before relation between events. This means that an execution defines a partial order of states. There are many total orders that are consistent with this partial order. A consistent total order corresponds to a view of the execution that could be obtained by some observer. For each consistent total order we can define an *execution sequence* s of global states $s_0, s_1, ...s_l$ associated with a consistent total order. If for any local state t in s_i and any local state u in s_j, $t \rightarrow u$, then s_i comes before s_j in the sequence. By contrast, concurrent events may occur in any order. Throughout this paper, we will interchangeably use the words total order and global sequence of states.

We will denote by S the set of global sequences each associated with a total order that could have been observed with respect to an execution. $s \in S$ represents a particular total order or execution sequence and s_i a state within that total order.

In the next subsection, related work with respect to this model is discussed.

1.2 Related Work

This section describes related work with respect to the model described in the previous subsection. Let P be a *global predicate*. A global predicate is an assertion that depends on the local states of multiple processes. In other words, a global predicate is an assertion on the global state. P is a *strong predicate* if it becomes true in all total orders (with respect to a particular partial order) i.e. if for each $s \in S$ there exists a s_i such that P is true in s_i. It is a *weak predicate* otherwise i.e. if there exists $s \in S$ such that P is true in s_i (as opposed to all $s \in S$).

P is a *stable predicate* if it is an invariant i.e. remains true once it becomes true. In other words, if we have $s \in S$ such that there exists a state s_i in which P is true

then for all states s_j where $j > i$, s_j is true. Examples of stable predicates include deadlock and termination. An *unstable predicate* may alternate between true and false.

To maintain appropriate behavior of the distributed system, it necessary to be able to detect global predicates that are a specification of the appropriate behavior. As a result, several approaches have been developed.

We begin our brief survey of approaches by examining algorithms based on taking global snapshots. The first is presented by Chandy and Lamport [2] for detecting global state predicates by taking a consistent global snapshot of the system and checking if the snapshot satisfies the global state predicate. This approach has been extended by Bouge [1] and Spezialetti and Kearns [11] to use repeated snapshots. The repeated snapshots are used to detect sequences of predicates in turn i.e. it is sufficient to detect each predicate of the sequence in the order they are presented. The work of Spezialetti and Kerns [12] detects sequences of predicates without repeated snapshots. The major problem with these approaches is that they do not work for unstable predicates, since an unstable predicate may be true only between two snapshots, but not in snapshots taken.

The detection of unstable predicates is more difficult than stable predicates, since their occurrences are transient. However, interesting algorithms have been developed that detect the first occurrence of a predicate becoming true. We will begin with the most general of these algorithms i.e. an algorithm that does detection of general global predicates during the execution of a distributed system. Cooper and Marzulla [3] do detection using a monitoring process (monitor) within the system. The monitoring process monitors the other processes to determine when some state predicate becomes true. Each monitored process will send a message to the monitor when its local state changes in a way significant with respect to Φ. The monitor builds a lattice of consistent global states that correspond to an observed execution. In their work, given an execution sequence and a predicate, Φ, they are able to evaluate whether a global state exists where the predicate Φ is true (weak unstable predicates). They also present an algorithm for determining that, for ALL executions consistent with the observed behavior, there was some global state that satisfies Φ (strong stable predicates). Since there is no way to bound the amount of time between the time a condition becoming true and the time the monitor detects the condition, the state space can become large making the detection of the predicates time consuming. In addition, the detection of general predicates is done in a central process, that increases communication costs.

To make detection of unstable global predicates more efficient, other researchers have defined a restricted form of global predicates [3]. We will now discuss these restricted forms and then the relevant work. A *local predicate* is a condition that is based entirely on the state of a single process. Simple predicates can be combined using the disjunctive operator to create a *disjunctive predicate* or they can be combined using the conjunctive operator to create a *conjunctive predicate*. Predicates can be combined to describe a sequence of events called a *linked* predicate. For example, we

may have a local predicate in a process ρ_i that says that x $<$ 5 and a local predicate in ρ_j that says that y $<$ 6 is true. x $<$ 5 $\wedge$ y $<$ 6 is a conjunction of local predicates and x $<$ 5 $\vee$ y $<$ 6 is a disjunction of local predicates. An assertion such as x + y $<$ 5 is not a local predicate or a conjunction or disjunction of local predicates. Note that the disjunctive, conjunctive and linked predicates a subclass of global predicates. Disjunctive predicates are easy to detect by incorporating a local predicate detection mechanism at each process.

The work in [4, 5] detects a subclass of global predicates by linearizing an observed partial order at a centralized checker process for disjunctive, conjunctive, and linked predicates (both weak and strong). Each process monitors for the truth of its local portion of the global predicate. On detecting that its local predicate is true, a process sends a timestamped notification message to the checker process. The checker process determines the global predicate to be true if, based on the received timestamps, the states are concurrent to each other.

The work of [13] is an extension of the approach presented in [4]. They do not assume only predicates that are either a disjunction, conjunction or a sequence of local predicates. They present algorithms that handle predicates of only the following form: $(x_0 + x_1 <$ C), where x_0 and x_1 are variables in processes ρ_0 and ρ_1, respectively, and C is a constant. They present a decentralized and centralized version of their algorithms. Their approach makes extensive use of intervals (defined to be sequences at a single process between message activity). For each interval in ρ_0, the minimum value of x_0 is determined. For each interval in ρ_1, the minimum value of x_1 is computed. Since intervals are defined to be sequences at a single process in between message activity then from the low and high ends of the intervals, it is possible to determine which intervals are concurrent to each other. For those intervals that are concurrent to each other, the predicate $(x_0 + x_1 <$ C) is evaluated. This approach cannot be done efficiently for a general N.

In summary, we can categorize the various approaches as follows: (i) Algorithms developed for detecting stable global predicates (ii) An algorithm for detecting more general forms of global predicates (both stable and unstable), but inefficient and centralized (iii) Algorithms that are more decentralized and efficient, but can only detect limited forms of global predicates.

1.3 Our Work

The goal behind our work was to develop a system that has the following properties. First, we want to be able to evaluate general unstable predicates. Second, we want this evaluation to be done in a decentralized manner. We believe that this reduces the communication costs. Third, the programmer should be able embed assertions to express predicates into the component processes of the distributed application at some specific point for evaluation during run-time. This allows the programmer to specify "where" the assertion should be true in contrast to the methods of the previous section that base their algorithms on detecting the first occurrence of a predicate becoming

true. We believe that by giving the programmer this ability, that the programmer has more control and is better able to narrow down the location of the bug. Although, we do not specifically show this in this paper, future work will show how our system can be rolled back to a consistent global state.

A programmer can debug their programs by embedding these assertions at specific points that they believe should be true if the program is correct. The programmer tests the assertions by running the program. If an assertion, P, evaluates to false, it means there exists $s \in S$ such there exists an s_i where P was false. It is assumed that the assertion P is suppose to be true in all execution sequences, hence, if we find that P evaluates to false then there is a bug that has manifested itself at the point in the program where P is being evaluated. In other words, a programmer making an assertion P implicitly requires P to be a strong predicate. Thus, if P evaluates to false in an execution sequence (thus, P is weak) then the assumption is incorrect and this indicates a bug in the program.

In section 2, we will describe the assertional language and programming system (CCSP) for stating assertions in distributed systems in more detail. We show an example of how our system works and the theory behind CCSP. We then describe related work and give a conclusion.

2 Assertional Language

In this section, the syntax of the language being used to state and evaluate assertions is described. We first describe how we state global predicates (or assertions) for our system. This involves a discussion of global auxiliary variables and why this approach works.

2.1 CCSP

CSP, having a rich background in formal methods theory, particularly in the use of mathematical and logical assertions [8, 10], has proven to be a powerful tool for reasoning about program structure [15]. A considerable portion of CSP concerns communication. We have added CSP-like communication statements to C resulting in CCSP. A parser creates a C process from source code that has the CSP-like communication statements and allows for this code to be run on any BSD UNIX network. This enables us to take advantage of the power of CSP.

In CSP, a distributed program consists of n processes that cooperate with each other using interprocess communication. No shared memory or variables are permitted. For example, if a traffic light process needed a car process to know that the light was green, a message is sent from the traffic light process to the car process as follows.

Example 2.1 *Process traffic_light::*
[*car* ! *light_type(green)*]

In the above instruction, "car" is the name of the car process, "green" is the message, and "!" indicates a send operation. To receive this message, the car process uses the following receive statement:

Example 2.2 *Process car::*
[*color light;*
traffic_light ? *light_type(light)*]

In the receive instruction, "traffic_light" is the sending process, "light"is a variable in which the color of the light is received, and "?" indicates a receiving operation.

In contrast to CSP, since CCSP is a real language, communication is not this simple. At the beginning of a CCSP program, a process name is explicitly paired with an address, allowing a process to be referenced by its process name. The message, however, is a bit more complex. For a message to be sent or received, the size of the message must be known in addition to the type and contents of the message. So the traffic light becomes:

Example 2.3 *Process traffic_light::*

```
[
mach1@car!light_type,green,light_size
]
```

Process car::

```
[ color light;
mach2@traffic_light?light_type,light,light_size
]
```

In the new receive instruction, "`light_type`" is the type of message, "`light_size`" is the size of the message, "mach1" and "mach2" are the two machine names in the Internet domain, and "`car`" and "`traffic_light`" are processes. In this example, the car process waits for the "`traffic_light`" process to send a message, and the "`traffic_light`" process waits for the car process to receive its message.

2.2 Assertions in CCSP

We use the `assert` statement to be able to state that an assertion must be true at the point in the program that the `assert` statement is located. During run-time, it will be checked to see if the assertion in the `assert` statement can be evaluated to true. For example, suppose the following code fragment is expressed in CCSP.

Example 2.4
```
[ integer occupancy=0;
*[
    [] if (occupancy<4) ->
       assert(occupancy<4);

/* Let someone in */

       occupancy := occupancy + 1;
       assert(occupancy <= 4);
]]
```

If, for some reason, in the first assertion, occupancy was greater than 4, an error condition is raised signaling the process and line number of the failed assertion. Since, in this example, the assertion follows an "if" test that guarantees its truth, the only way the assertion could fail is a machine or environment failure.

2.3 Global Auxiliary Variables

Predicates are not limited to assertions about a single process; in general, predicates (*global predicates*) can make assertions about multiple processes. The difficulty in checking the truthfulness of a global predicate arises for two reasons. First, the total number of total orders is exponential to the number of processes in the distributed program. Second, the global state is distributed among the processes of the system during an execution sequence.

To avoid the problem of combinatorial explosion and yet still allow for evaluation of general predicates, we allow the programmer to express predicates that make use of shared reference to *global auxiliary variables*. Auxiliary variables are special variables that serve as placeholders for arbitrary values; they appear only in predicates and not in program statements. They are global only in the sense of the assertional system, but do not correspond to actual program variables. Auxiliary variables do not change the behavior of the program but merely collect information about the state of program execution. The shared reference to auxiliary variables allows for the definition of predicates that are assertions over multiple process states as well as assertions over different execution traces.

These variables can be set within a process using the `equal` statement and retrieved by the `value` statement. For instance, if, in the above example, there were two processes, it might be desirable to make assertions about a global `occupancy` variable in some other process where the variable is not set. In Example 2.5, ρ_0 sets `occupancy` and ρ_1 reads a global copy of it with the `value` statement. Notice, that due to arbitrary interleavings of processes, ρ_1 may get different values for `occupancy_copy` during different runs. This concept is explored, further, in the following section.

Example 2.5

ρ_0 ρ_1

```
[ integer occupancy=0;                  [global occupancy_copy;
  global occupancy_copy;
  equal(occupancy_copy,occupancy);
*[                                       assert(value(occupancy_copy)<=4)
    [] if (occupancy<4) ->
       assert(occupancy<4);              ]

/* Let someone in */
       equal(occupancy_copy,occupancy)
       occupancy := occupancy + 1;
       assert(occupancy <= 4);
]]
```

In addition, the global auxiliary variable could be set during communication, via receiving a communication into an `equal` function.

Interference In Example 2.5, due to arbitrary interleavings of processes, ρ_1 may get different values for `occupancy_copy` during different runs. However, the assertion never fails since ρ_0's actions do not interfere with the truth of the assertion. More formally, let α and β be events. Suppose that, as in Hoare triples [8], the assertion P is should be true before α and the assertion Q should be true after α. In other words, if α begins in a state satisfying P and α terminates then the state will satisfy Q. However, this is not valid if while this process is in a state satisfying P and P contains shared reference to global auxiliary variables, another process executes an event β concurrently and alters the global auxiliary variables i.e. a process might *interfere* with an assertion in another process making it false. Conceptually, interference is the same as invalidating a strong predicate.

Therefore, the only assertions using auxiliary variables that are logically implied by the distributed program are those for which it can be shown that events that execute concurrently to where the assertions are assumed to be valid do not interfere i.e. if P is an assertion in process ρ then P is invariant over any arbitrary execution sequence. Let β be an event (in this case, execution of a statement) concurrent to P; the validity of P must be preserved after β has been executed. In other words, P is true after S has been executed in all execution sequences. This is the property of *noninterference* and is represented as $< P \wedge Pre(\beta) > \beta < P >$ where $Pre(\beta)$ denotes the assertion that must be true of the global state before β is executed.

Operational Evaluation CCSP is based on CSP, a completely distributed programming language whose design does not permit memory to be shared. While we can construct and reason with assertions containing global auxiliary variables, we cannot directly evaluate these assertions at run time since the auxiliary variables are not available outside of the process in which they are set. This forms the crux of our problem; *how do we provide a run-time environment that can evaluate predicates involving shared reference to global auxiliary variables.*

To evaluate predicates, at run time, that use auxiliary variables, each process maintains a local copy of each declared `global` variable. CCSP exchanges and updates these *global auxiliary variables* (GAVs) at each communication such that after each communication, processes have consistent, pairwise, states of each other's global auxiliary variables. The reason that this works is that the assertions are assumed to be strong predicates - events of other processes don't interfere with their truth.

Assume that α and β are two concurrent events. The effects of these two events, by linearizing the associated partial order, is that the resulting state is the same whether α precedes β or vice-versa (see Theorem 1 of [5]). Thus, concurrent events do not induce causal dependencies on the program state. What about global auxiliary variables? Arguing informally, if each process makes local changes to its copy of a `global` variable and if none of these events interferes with any global predicate, then these events are concurrent. Then, it is perfectly acceptable to introduce a particular causality (at message exchange) since the events are concurrent. An example of this is depicted in figure 1. Event β is concurrent with event α. Since α's global predicate is valid if executed before or after β, α and β do not interfere and we are free to delay the update of `b` until some causality is introduced. After communication, the change to β is propagated to process 1. Now, since there is causality between β, we can assert the stronger predicate `b=v2`.

Later, we also show the following: (1) If an assertion is true of a distributed program then it will evaluate to true in CCSP (2) If an assertion is true in a distributed program then CCSP will not evaluate it to false. By showing this, we have shown that are predicates are in fact general global predicates.

In the next two sections, we describe how we use this system for debugging by using the dining philosopher's problem as an example and formally describing how the auxiliary maintenance is done.

3 Using CCSP

The use of assertions is a method of stating mathematically the expected or desired run-time behavior of a program. Checking the validity of these assertions is embedded in CCSP as much as communication. In this section, we present an example of using CCSP for debugging a program that implements the dining philosopher's problem. We chose the dining philosopher's problem as the example in this paper for the following reasons. First, it is a well-known and understood problem in distributed

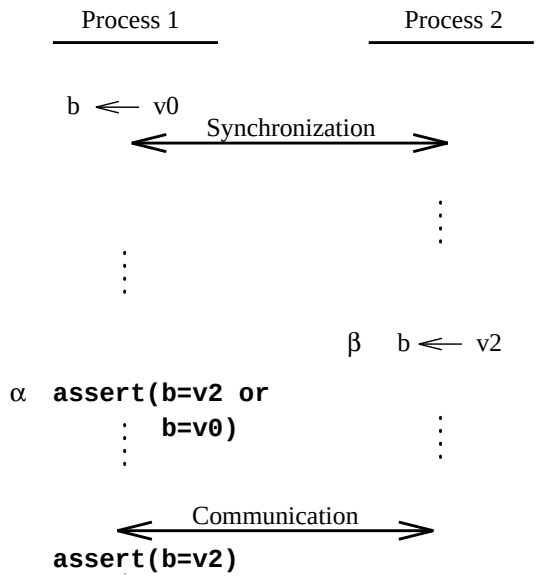

Figure 1: Concurrent Events forced to a particular causality

systems. Second, it is a distributed, multiple process program. Third, the processes that the program consists of are heterogeneous.

In this section, we present an example of assertions that can be made about the dining philosopher's problem, we give an example using the dining philosophers to show how global auxiliary variables are updated as part of communication, and then how debugging can be done by using CCSP's `assert()` statement.

3.1 Examples of Assertions

Suppose that five philosophers share a two room apartment. One room is a dining room and one room is a thinking room. The philosophers are poor. Their dining room has a table with five chairs and five plates, but they only possess five forks: one to the left and one to the right of every plate (Forks are shared). To eat the spaghetti each philosopher needs two forks. If a philosopher cannot get two forks, she will starve. Each philosopher picks up the fork to her left first and then the fork to her right. Should she be unable to get the fork immediately, she can get the fork as soon as another philosopher releases it. As long as one philosopher can eat, deadlock will not occur. The eating philosopher will eventually put down both forks allowing others to eat. However, if all the philosophers are at the table and none can eat, then deadlock has occurred, and the philosophers will starve. A possible solution and the solution implemented is to only permit four philosophers in the dining room at one time. Thus, this solution always allows at least one philosopher to eat, preventing deadlock. The philosopher and fork processes are straightforward and are depicted in Example 3.1 for Philosopher 1 and Fork 1. The room process is in Example 3.5.

Example 3.1

Philosopher 1

```
global in0, in1, in2, in3, in4;
*[
Thinking;
assert(I and value(in1)==false);
room!enter,&come_in,4;
assert(I and value(in1)==true);
printf("Phili is in the room\n");
fork1!pickup,nullbuf,0;
assert(I and value(in1)==true);
fork2!pickup,nullbuf,0;
assert(I and value(in1)==true);
Eating;
fork1!putdown,nullbuf,0;
assert(I and value(in1)==true);
fork2!putdown,nullbuf,0;
room!eggsit,&go_out,4;
assert(I and value(in1)==false);
] ]
```

Fork 1

```
global in0, in1, in2, in3, in4;
*[
*[ /* Philosopher to the right */
  []if (Phil1?pickup,nullbuf) ->
      printf("Phil1 picks up Forki");
      assert(I and value(in1)==true);
      Phili?putdown,nullbuf;
      printf("Phil1 puts down Fork1");
      assert(I and value(in1)==true);

  /* Philosopher to the left */
  []if (Phil2?pickup,nullbuf) ->
      printf("Phil2 picks up Fork1");
      assert(I and value(in2)==true);
      Phil2?putdown,nullbuf;
      printf("Phil2 puts down Fork1");
      assert(I and value(in2)==true);
 ] ]
```

To prevent deadlock, certain conditions (that are represented using assertions) must be met. In particular, an invariant (i.e. a condition that is always or invariantly true) is needed to ensure that all of the philosophers are not in the dining room simultaneously.

Example 3.2 `#define I  (not(value(in0) and value(in1) and value(in2) and value(in3) and value(in4)) and value(occupancy_copy)<=4)`

This invariant contains six global auxiliary variables: "`in0`" to "`in4`" and "`occupancy_copy`". The "`ini`"'s store the same numbered philosophers location: the dining room (denoted by the value 1) or the thinking room (denoted by the value 0). The global auxiliary variable "`occupancy_copy`" contains the number of philosophers in the dining room. All dining philosopher processes as well as the fork and room processes depend on this invariant, I, being true.

Additionally, a philosopher must verify after picking up or putting down a fork that she is in the dining room and that I, the invariant, is true. For Phil0, this assertion is:

Example 3.3 `assert(I and value(in0) == TRUE)`

We use the `assert` statement to be able to state that an assertion must be true at the point in the program that the `assert` statement is located. During run-time, it will be checked to see if the assertion in the `assert` statement can be evaluated to true. In example 3.1 since, we want `I` to be invariantly true, then one use of the `assert` statement may be to check to see that `I` is true at every point in execution (although in practical use, the programmer may only choose some points to test the truth of `I` at run-time).

3.2 Global Auxiliary Variable Maintenance

In this subsection, the method of maintaining global auxiliary values is described and an example is presented using the dining philosopher's problem. In the philosopher example, the global auxiliary variables "`in0`" to "`in4`" and "`occupancy_copy`" were used. However, CCSP is based on CSP, a completely distributed programming language whose design does not even permit memory to be shared. To allow such variables, each process must possess a local copy of the global auxiliary variables. These variables can be set within a process using the equal statement. For instance, in the room process (Example 3.5), "`occupancy_copy`" is set to zero by "`equal(occupancy_copy, occupancy);`". To retrieve this value, just use "`value(occupancy_copy);`". In addition, the global auxiliary variable could be set during communication, either explicitly via receiving a communication into an equal function, or implicitly.

In an implicit global auxiliary variable exchange, process A sends to process B (and vice versa) every global variable changed since the last communication with process B.

Example 3.4

```
                Process Room            Process Phil4
Before communication:
                Time: 23               Time:  20
in0             value: 1  time: 21     value:  0   time: 16
in1             value: 1  time: 17     value:  1   time: 17
in2             value: 1  time: 19     value:  1   time: 19
in3             value: 1  time: 23     value:  1   time: 18
in4             value: 0  time: 20     value:  0   time: 20
occupancy_copy  value: 3  time: 23     value:  4   time: 19

Communication:
(Phil4?enter,equal(in4,x))              room!enter,&come_in,4;
After communication:
                Time: 24               Time:  24
in0             value: 1  time: 21     value:  1   time: 21
in1             value: 1  time: 17     value:  1   time: 17
in2             value: 1  time: 19     value:  1   time: 19
```

```
in3             value: 1  time: 23    value:  1   time: 23
in4             value: 1  time: 24    value:  1   time: 24
occupancy_copy  value: 3  time: 23    value:  3   time: 23
```

Each communication in CSP is augmented in CCSP with functions [10] for global auxiliary variable maintenance. These functions are used by each process to ensure that all copies of the global auxiliary variables sent are the most recent information and that only those variables changed since the last communication with the other process will be sent. These functions prepare copies of a processes' global auxiliary variables for communication and unions these variables into a process' local state. Thus, above, prior to a communication, Room and Phil4 prepare their copies of the global variables for transmission using these functions. Here, we will give a high-level description and in the next section a concise definition of these functions is presented.

To determine which global auxiliary variables are most recent, it is necessary to timestamp them. However, there is no global clock for truly distributed processes. Agreement on time is difficult, but since only relative, not absolute, times are needed, Lamport clocks are sufficient [7]. Each process begins with a local Lamport clock set to 0. At each communication, the Lamport clock is changed to the maximal value from the two processes and then incremented by one [10]. This is demonstrated by the trace of Room and Phil4. The clock in Phil4 jumps from 20 to 24, the original time of the Room process plus 1, to make 24. The Room clock, however, merely increments by one, already having the maximal time in that communication. Additionally, the update functions must prepare to send all copies of global auxiliary variables it has heard so far from other processes, along with their timestamps.

Upon receiving Room's copy of the global auxiliary variables, Phil4 stores them and then transmits its own copies of the variables to Room. After the global variables are sent, Phil4 updates its current back copies of the global variables based on Room's information using a predefined function [10].

This function ensures that only the most recent copies of the global variables, whether from Phil4 or Room, are kept. For example, in Phil4, global variable in0, that has a value of 0, assigned at time 16, is changed to room's more current information to value 1, assigned at time 21. This spread of "new" information ensures that the logical assertions from a CSP specification are preserved in CCSP's run time environment. In other words, the property of noninterference is preserved.[3]

3.3 Debugging with CCSP

In this section, we describe how CCSP was used to catch an incorrectly written Room program using the `assert()`.

Traditional debuggers are nearly incapable of debugging a multiple process program. However, by using CCSP's `assert()` statement, debugging a distributed mul-

[3] "`occupancy_copy`" is 3 rather than 4 because the room process has not incremented "`occupancy_copy`" yet. The increment occurs after the communication.

tiple process program, such as the Dining Philosophers, is relatively simple. A successful Dining Philosopher's solution is an endless round of philosophers thinking, picking up the forks, and eating.

The philosopher and a fork are straightforward processes to write. The philosopher, as can be seen in example 4.1, has no conditionally-reached states. One command merely follows another, sequentially. The only assertion necessary that only four philosophers are in the room is the invariant assertion discussed in examples 4.3 and 4.4.

A fork process is nearly as straightforward. To have a correctly functioning fork, two conditional statements are required, one to see if the philosopher on the right wants it (using a conditional, nonblocking receive statement) and one to see if the philosopher on the left wants it. Also, to ensure that only philosophers in the room can pick up or put down a fork, an additional assertion requiring the philosopher requesting the fork to be in the dining room is needed. Finally, each fork process has the same invariant as the philosopher process.

The most difficult process type, as is seen in Example 3.5 to write correctly is the room process.

Example 3.5 (Room)

```
*[ assert (I);
*[ []if ((Phili?enter,equal(ini,x)) AND occupancy < 4) ->
        assert(I and value(ini)==true and value(occupancy_copy) < 4);
        occupancy := occupancy + 1;
        equal(occupancy_copy,occupancy);
        assert(I and value(ini)==true and value(occupancy_copy) <= 4);
   [] if (Phili?eggsit,equal(ini,x)) ->
        assert(I and value(ini)==false  and value(occupancy_copy) <= 4);
        occupancy := occupancy - 1;
        equal(occupancy_copy,occupancy);
        assert(I and value(ini) == false  and value(occupancy_copy) < 4);
/*The other philosophers are symmetrical.*/
] ]
```

The room monitors the philosopher processes. The room process needs to ensure three things:

1. the invariant, presented in the philosopher and the fork processes, holds,
2. the relevant philosopher is in the room, and
3. the number of philosophers is less than or equal to four.

To maintain these assertions, the room process has two conditional statements for each philosopher: one to exit and one to enter. To exit correctly, the philosopher sends a message to the room that she intends to leave. The global variable representing

this philosopher (in_num, where num is the philosopher number), is changed to reflect this fact. The more difficult problem occurs when the philosopher enters, for at this point in time, two criteria must be met

1. the philosopher is ready to enter and
2. fewer than four philosophers are present.

For philosopher 0, this code is:

Example 3.6 *if ((Phil0?enter,equal(in0, x)) AND (occupancy < 5)) − >*

However, this code fails spectacularly if a philosopher enters the room when there are four other processes in it. Assume that we have the following sequence of events:

Example 3.7

```
Output of faulty program:
Phil0 is thinking
           Phil1 is thinking
                       Phil2 is thinking
                                   Phil3 is thinking
                                               Phil4 is thinking
Phil0 is in the room
           Phil1 is in the room
                       Phil2 is in the room
                                   Phil3 is in the room
Assertion failed: file "room.c", line 47
Assertion failed: file "Phil4.c", line 53
```

Tracing the failure in the C programs, the first assertion that failed, the assertion on line 47, of room.c which is: "assert(I);". I is defined as:

```
I = (!( value(g[0],in0)&&value(g[0],in1)&&value(g[0],in2)
        &&value(g[0],in3)&&value(g[0],in4))
        &&value(g[0],occupancy_copy) <= 4).
```

The example in 3.4 is a reflection of the current values of the global auxiliary variables when `Phil4` enters the room. Looking at example 3.4, we see that the each `ini` is equal to one and hence their sum, which is in the auxiliary variable `occupancy_cost`, is five. This violates the invariant I that says that `occupancy_cost` should not exceed four.

The next failed assertion is on line 53 in the Phil4.c program and is as follows:

```
assert(I&&value(g[0],in4) == true);
```

where I is as previously defined. This assertion merely states that the fourth philosopher is in the dining room and that less than five philosophers are present in the dining room. A careful examination of the invariant indicates that all five philosophers were in the room; the room's occupancy was more than four. The program's output indicates that four philosophers were in the room when philosopher 4 tried to enter the room. Somehow, philosopher 4 entered the room when she should not have. To enter the room, the program requires philosopher 4 to query for permission to enter the room. Then the algorithm checks if the number of occupants is less than four. The above philosopher code only requires permission from the room to enter. Ergo, the above guard statement is out of order and thus that the order should be changed to:

Example 3.8 *if ((occupancy < 4) AND (Phil4?enter, equal(in0, x))) ->*

When this code is tried, it works perfectly. An eternal round of philosophers thinking and eating occurs.

4 Underlying Theory

Now that we have described how debugging in CCSP works in practice, we will now discuss its underlying theory. In the previous section, we described how each process must possess a local copy of each global auxiliary variable and that at communication time, processes exchange time-stamped copies of the global auxiliary variables. In this way, the state of the system, as represented by the global auxiliary variables, is diffused throughout the processes in the system. This concept is similar to the gossip messages of [14] except that, here, only a restricted history of size 1 is disseminated, whereas, in [14] an entire event history is treated.

This section, mathematically describes how the global auxiliary variables are updated in CCSP and that the method of updating preserves the property of noninterference.

4.1 Definitions of Functions

In this section, the method of updating the global auxiliary variables is presented by mathematically describing the functions that do the updates. Before we define the functions, we describe the representation of global auxiliary variables.

The local copies of global auxiliary variables is represented as a set of pairs, where each pair is a global auxiliary variable and its associated time-stamp.

Definition 4.1 *A set of* **time-stamped elements** *is a set* $X = \{ (x, t_x) \}$ *of pairs consisting of an element* x *and its time-stamp* t_x*, such that* $\forall (x, t_x) \in X, \forall (y, t_y) \in X, x \neq y$.

Definition 4.2 *The* **fresh-union** *of two sets A and B of time-stamped elements is:*

$$\begin{aligned} A \cup_f B = \quad & \{(a,t_a)|(a,t_a) \in A; \forall (x,t_x) \in B : x \neq a\} \ \cup \\ & \{(b,t_b)|(b,t_b) \in B; \forall (x,t_x) \in A : x \neq b\} \ \cup \\ & \{(c,t_c)|\exists (a,t_a) \in A, \exists (b,t_b) \in B : c = a = b, t_c = max(t_a,t_b)\} \end{aligned}$$

Let g_{ij} be the set that consists of entries of the form (a, v_a, t) denoting auxiliary variable a has value v_a at time t. Let g_{ij} contain the changes that were made to the global auxiliary variables in p_i since the last communication with p_j Let $\mathcal{G} = \{g_{ij}|i,j = 1,n\}$ be the set of all g_{ij} sets as previously defined for processes ρ_i.

Let $\mathcal{V}^{(i)}$ be the variable space for the process ρ_i, composed of local program variables, local auxiliary variables, and global auxiliary variables:

$$\mathcal{V}^{(i)} = V_{L,P}^{(i)} \ \cup V_{L,A}^{(i)} \ \cup V_{G,A}^{(i)}$$

Let $\mathcal{V}_{G,A}$ be the global auxiliary variable space for all processes:

$$\mathcal{V}_{G,A} = \{V_{G,A}^{(i)}|i = 1,n\}$$

Functions for the Computation Phase We now describe the effect on global auxiliary variables as the result of a global auxiliary variable's value changed as the result of an assignment statement.

Consider an assignment statement α of the form $a := v_a^{'}$ executed at time t' by the process ρ_i.

Definition 4.3 *The effect of α on processes' $\mathcal{G}$ is given by the function $\lambda 1 : \mathcal{D}_{\mathcal{G}} \longrightarrow \mathcal{D}_{\mathcal{G}}$:*

$$\lambda 1(g_{jj}) = g_{jj}^{new}, \ where$$

$$g_{jj}^{new} = \begin{cases} g_{jj} & if \ i \neq j \\ g_{jj} \cup_f \{(a, v_a^{'}, t')\} & if \ i = j \end{cases}$$

Definition 4.4 *The effect of α on the global auxiliary variable space is given by the function $\lambda 2 : \mathcal{D}_{\mathcal{V}_{\mathcal{G},\mathcal{A}}} \longrightarrow \mathcal{D}_{\mathcal{V}_{\mathcal{G},\mathcal{A}}}$:*

$$\lambda 2(V_{G,A}^{(j)}) = V_{G,A}^{(j)(new)}, \ where$$

$$V_{G,A}^{(j)(new)} = \begin{cases} V_{G,A}^{(j)} & if \ i \neq j \\ V_{G,A}^{(j)} \cup_f \{(a, v_a^{'}, t')\} & if \ i = j \end{cases}$$

ρ_s:
ψ /* Propagate Local Changes to Auxiliary Variables Within ρ_s */
Send a Message containing g_{sr}^{new} from ρ_s to ρ_t

ρ_t:
Receive Message containing g_{sr}^{new} at ρ_t from ρ_s
ϕ_1 /* Receive Auxiliary Variable Copies */
ϕ_2 /* Apply Auxiliary Variables to Local Variable Copies */

Figure 2: Functions for Augmented Communication

Functions for the Communication Phase We now describe the effect on global auxiliary variables as the result of a global auxiliary variable's value changed as the result of communication. Communication is organized into two phases, *primary communication* and *augmented communication*. Primary communication consists of (!) or (?) in the CCSP program between a sender and a receiver process. Augmented communication exchanges values of the global auxiliary variables between the same sender and receiver as specified in the primary communication. An augmented communication invokes the functions depicted in Figure 2 sending from process ρ_s to ρ_t.

Definition 4.5 *The propagation of local changes within the sending process' $\mathcal{G}$ before an augmented communication is given by the function $\psi : \mathcal{D}_\mathcal{G} \times \mathcal{D}_\mathcal{G} \longrightarrow \mathcal{D}_\mathcal{G}$:*

$$\psi(g_{ii}, g_{jk}) = g_{jk}^{new}, \textit{ where}$$

$$g_{jk}^{new} = \begin{cases} g_{jk} & \textit{if } i \neq j \\ g_{ii} \cup_f g_{jk} & \textit{if } i = j \textit{ and } i \neq k \\ \emptyset & \textit{if } i = j = k \end{cases}$$

Definition 4.6 *The propagation of received changes within the receiving process' $\mathcal{G}$ after an augmented communication is given by the function $\phi 1 : \mathcal{D}_\mathcal{G} \times \mathcal{D}_\mathcal{G} \longrightarrow \mathcal{D}_\mathcal{G}$*

$$\phi 1(g_{ij}, g_{kl}) = g_{kl}^{new}, \textit{ where}$$

$$g_{kl}^{new} = \begin{cases} g_{kl} & \textit{if } j \neq k \\ g_{ij} \cup_f g_{kl} & \textit{if } j = k \textit{ and } i \neq l \\ \emptyset & \textit{if } j = k \textit{ and } i = l \end{cases}$$

Definition 4.7 *The effect of received changes on the global auxiliary variable space of the receiving process after an augmented communication is given by the function* $\phi 2 : \mathcal{D}_{\mathcal{V}_{\mathcal{G},\mathcal{A}}} \times \mathcal{D}_{\mathcal{G}} \longrightarrow \mathcal{D}_{\mathcal{V}_{\mathcal{G},\mathcal{A}}}$ *:*

$$\phi 2(V_{G,A}^{(i)}, g_{jk}) = V_{G,A}^{(i)(new)}, \text{ where}$$

$$V_{G,A}^{(i)(new)} = \begin{cases} V_{G,A}^{(i)} & \text{if } i \neq j \\ V_{G,A}^{(i)} \cup_f g_{jk} & \text{if } i = j \end{cases}$$

4.2 Preserving Noninterference

In this section, we shall show that if an assertion is logically implied by the program then the noninterference of assertions is preserved in the distributed operational environment of CCSP.

First we prove the following theorem:

Theorem 4.1 *Assume that the global auxiliary variables are updated using the functions defined in Section 4.1. After a communication occurs, both participating processes have the same values for the global auxiliary variables and common values are the most recently assigned to global auxiliary variables. We will refer to this property as the* update *property.*

The proof is by induction on the number of communications.

Base case: One communication.

Let ρ_1 and ρ_2 be a pair of communicating processes, and let (a, v_a, t_0) be a GAV a, with value v_a at time t_0 in both processes.

Figure 3 depicts the exchange.

At time t_1 in process ρ_1, a is assigned a value v_1. Then:

$\lambda 1(g_{11}) = g_{11} \cup_f \{(a, v_1, t_1)\}$, and

$\lambda 2(V_{G,A}^{(1)}) = V_{G,A}^{(1)} \cup_f \{(a, v_1, t_1)\}$.

At time t_2 in process ρ_2, a is assigned a value v_2. Then:

$\lambda 1(g_{22}) = g_{22} \cup_f \{(a, v_2, t_2)\}$, and

$\lambda 2(V_{G,A}^{(2)}) = V_{G,A}^{(2)} \cup_f \{(a, v_2, t_2)\}$.

Without loss of generality suppose that no other changes are made to a during this phase (or take t_1 to be the time of the latest change to a).

The processes communicate at time t_{12}.

Prior to the actual communication the change is propagated in process ρ_1 to local histories, including g_{12}:

$\psi(g_{11}, g_{12}) = g_{12}^{new}$, thus $(a, v_1, t_1) \in g_{12}^{new}$.

Similarly in process ρ_2 the change is propagated to local histories, including g_{21}:

$\psi(g_{22}, g_{21}) = g_{21}^{new}$, thus $(a, v_2, t_2) \in g_{21}^{new}$.

After the communication the received changes are applied to the GAVs:

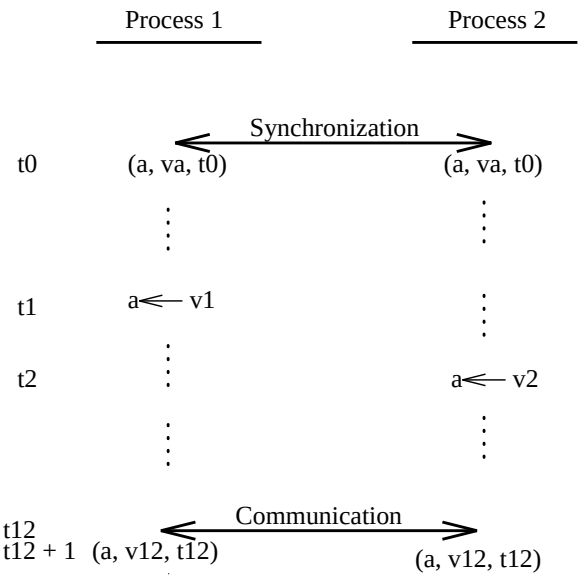

Figure 3: Base case: 1 communication.

In process ρ_1: $\phi 2(V^{(1)}_{G,A}, g_{21}) = V^{(1)new}_{G,A}$

To obtain the new set, the value retained for a is given - according to the fresh-union definition - by:

$$\{(a, v_1, t_1)\} \cup_f \{(a, v_2, t_2)\} = \begin{cases} \{(a, v_1, t_1)\} & \text{if } t_1 > t_2 \\ \{(a, v_2, t_2)\} & \text{if } t_1 \leq t_2 \end{cases}$$

Denote this tuple (a, v_{12}, t_{12}).

Similarly in process ρ_2:

$\phi 2(V^{(2)}_{G,A}, g_{12}) = V^{(2)new}_{G,A}$, where the new value for a is given by:

$$\{(a, v_2, t_2)\} \cup_f \{(a, v_1, t_1)\} = \begin{cases} \{(a, v_1, t_1)\} & \text{if } t_1 > t_2 \\ \{(a, v_2, t_2)\} & \text{if } t_1 \leq t_2 \end{cases}$$

This is the tuple denoted (a, v_{12}, t_{12}), the same one obtained in process ρ_1.

Thus at time $t_{12} + 1$ a is represented by (a, v_{12}, t_{12}) in both P_1 and P_2.

This proofs the case for number of communications equal to 1.

The second base case is for two communications and it is necessary because it involves the function $\phi 1$, that was not present in the previous case.

This case is presented in Figure 4.

In the same setting as before, assume the process ρ_2 communicates with the process ρ_3 at time t_{23}, between t_2 and t_{12}, exchanging value changes for a. The process ρ_3 had previously set the value of a to v_3, at time t_3, where $t_3 < t_2$.

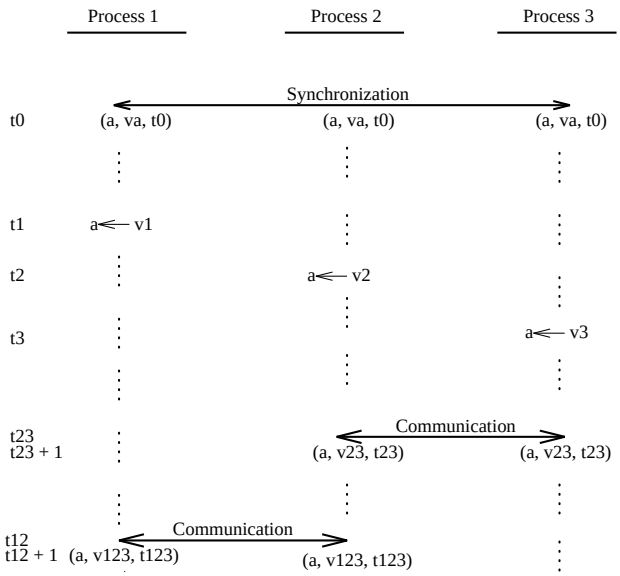

Figure 4: Second base case: 2 communications.

As in the previous case, at time $t_{23}+1$ both ρ_2 and ρ_3 have the common value (a, v_{23}, t_{23}) for a, where $t_{23} \geq t_2$ and $t_{23} \geq t_3$.

Also at time $t_{23}+1$ in process ρ_2 the changes received for a are propagated to all local histories, including g_{21} that contains the changes to be sent later to ρ_1:

$\phi 1(g_{32}, g_{21}) = g_{21}^{new}$, thus $(a, v_{23}, t_{23}) \in g_{21}^{new}$.

Assume (a, v_{23}, t_{23}) is ρ_2's value for a at time t_{12}. Then after the communication between ρ_1 and ρ_2 a is represented by (a, v_{123}, t_{123}), where $t_{123} = max(t_1, t_{23})$ and v_{123} its corresponding value.

In conclusion, regardless of the comparison between t_1 and t_3, at time t_{12} the process ρ_1 has the most recent value for a.

This completes the proof for number of communications equal to two.

Induction step:

Inductive Hypothesis: Assume that the update property is preserved for n communications.

We now that this property holds for $n+1$ communications.

Case 1: The extra communication is within the same set of processes $\mathcal{P} = \{\rho_1, \rho_2, ..., \rho_m\}$ as the other n communications.

- If the $(n+1)st$ communication occurs before the other n communications, it has been shown as a base case that one communication preserves the update property, so the $(n+1)st$ communication preserves the update property. By the inductive hypothesis the group of n communications also preserves the update property. Therefore $n+1$ communications preserve the update property.

- Similarly, if the $(n+1)st$ communication occurs after the other n communications.

- If k communications take place first, with $0 < k < n$, then by IH they preserve the update property. Then the $(n+1)st$ communication occurs, and it preserves the update property also, by base case. Finally the $n-k$ remaining communications take place and by IH they preserve the update property too. Therefore $n+1$ communications preserve the update property.

Case 2: The extra communication is between a process ρ_{m+1} from outside $\mathcal{P}$ and one of the processes in $\mathcal{P}$.

Regard the set $\mathcal{P}$ as a single process ρ_M inside which, by IH, the update property is preserved. Then the communication between ρ_M and ρ_{m+1} preserves the update property too, by base case.

Case 3: The extra communication is between two processes from outside $\mathcal{P}$.

Since the two sets do not communicate with each other, there is no interference. Noninterference is preserved inside the set $\mathcal{P}$, by IH, and within the other two processes, by base case.

This completes the proof for $n+1$ communications, and also for the communication phase and the whole theorem. □

We will first show that the noninterference of assertions is preserved in the operational environment.

Theorem 4.2 *Assume that α is an event in process ρ_i and that β is an event ρ_j. Assume that the global predicate P is logically implied by the distributed program and should be true before α executes and that β is any event concurrent to α that does not interfere with P. The operational environment as described in 4.1 preserves the validity of P in all execution sequences.*

Proof: Let $Pre(\beta)$ denote the assertion that must be true of the global state before β is executed.

There are two cases to consider.

Case 1: β does not involve global auxiliary variables. Therefore, since β is local to process ρ_j, it does not affect the truth or falsity of assertion P.

Case 2: β makes an assignment to a global auxiliary variable that is used in P. In theorem 4.1, we showed that at communication time the two concurrent processes containing α and β will have the same values of auxiliary variables when they communicate i.e. eventually the change is communicated to the process ρ_i. If this invalidates P then this is equivalent to $< P \wedge Pre(\beta) > \beta < \neg P >$. However, since β does not interfere with P ($< P \wedge Pre(\beta) > \beta < P >$), this forms a contradiction. □

4.3 Truth Preservation

We will now show the following: (1) If an assertion is logically implied by a distributed program then it will evaluate to true using the `assert` statement (2) If an assertion is always evaluated to true using the `assert` statement, then the assertion is logically implied by a distributed program.

Theorem 4.3 *If an assertion P is logically implied by a distributed program then* `assert(P)` *evaluates to true.*

Proof: Assume that P is logically implied by a distributed program, but `assert(P)` evaluates to false in some execution sequence. Then there exists some concurrent event, β, that modifies global auxiliary variables that are used by P. In Theorem 4.2 it was shown that in the operational environment concurrent events do not interfere with the validity of P. Hence, we have a contradiction if `assert(P)` evaluates to false. □

Theorem 4.4 *If* `assert(P)` *always results in true (P is a stable property), then P is logically implied by the distributed program.*

Proof: Assume that `assert(P)` always results in true, but P is not logically derived from the distributed program. This implies that there is a concurrent event α concurrent to P such that β invalidates P. In other words, β is an event that makes some assignment $x := y$ that, at some future time, is communicated to the process in which P is embedded into. Theorem 4.2 shows that the method of updating global auxiliary variables described in section 4.1 has the same effect as if the processes truly shared access to these variables. Thus, if `assert(P)` always evaluates to true, then this is equivalent to saying that in all execution sequences, P is true. This contradicts that P is not logically implied by a distributed program. Hence, we conclude that P is logically implied by the distributed program. □

5 Other Uses

In addition, to debugging, there exist other applications for CCSP. One application is fault-tolerant distributed computing especially in application-oriented fault tolerance [9]. Assertions can be embedded into the program code that can be used to detect errors. We need to ensure that the state information of a communicating process is compared to the expected result in the receiving process, so that a fault in the sending process can be detected by the receiving process. Another application is in the monitoring of distributed systems. From the perspective of managing the performance of a distributed application, it is necessary that information about the behavior of the underlying systems and networks is essential. Assertions can be used to state what the expected behavior should be.

6 Conclusions

We have presented the CCSP system for debugging distributed programs using programs annotated with assertions and the concept of global auxiliary variables. The state of these global auxiliary variables is maintained by diffusing the state through the system as communication occurs in the debugged program. Using CCSP for

debugging programs can detect general unstable predicates at specific points in the program's execution.

References

[1] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in csp. *Theoretical Computer Science*, 49:63–75, 1987.

[2] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed computing. *ACM Transactions on Computing Systems*, pages 163–173, February 1985.

[3] R. Cooper and K. Marzulla. Consistent detection of global predicates. *Proceedings ACM/ONR Workshop on Parallel Distributed Debugging*, pages 163–173, 1991.

[4] V. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. *Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, 1992.

[5] V. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transaction on Parallel and Distributed Systems*, 5(3):299–307, MARCH 1994.

[6] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[7] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[8] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[9] H. Lutfiyya. Fault tolerance in concurrent systems through formal methods. *Ph.D Thesis*, Computer Science Department, University of Missouri-Rolla, 1992.

[10] H. Lutfiyya, M. Schollmeyer, and B. McMillin. Formal generation of executable assertions for application-oriented fault tolerance. *UMR Department of Computer Science Technical Report Number CSC 92-15*, 1992.

[11] M. Spezialetti and P. Kearns. Efficient distributed snapshots. *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[12] M. Spezialetti and P. Kearns. A general approach to recognizing event occurrences in distributed computations. *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 300–307, 1988.

[13] A. Tomlinson and G. Garg. Detecting relational global predicates in distributed systems. *Proceedings ACM/ONR Workshop on Parallel Distributed Debugging*, pages 21–31, 1993.

[14] G. Wuu and A. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proc. of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 233–242, 1984.

[15] Zhiwei Xu. Structured principles for developing parallel computing programs. *IEEE International Conference on Systems, Man, and Cybernetics*, pages 655–660, 1991.