# Partial Orders for Parallel Debugging

*C. J. Fidge*
*Department of Computer Science*
*Australian National University*
*Canberra, ACT, Australia*

## ABSTRACT

**Parallel programs differ from sequential programs primarily in that the temporal relationships between events are only partially defined. However, for a given distributed computation, debugging utilities typically linearize the observed set of events into a total ordering, thus losing information and allowing potentially capturable temporal errors to escape detection. We explore use of the partially ordered relation "happened before" to augment both centralized and distributed parallel debuggers to ensure that such errors are always detected and that the results produced by the debugger are unaffected by the non-determinism inherent in the partial ordering. This greatly reduces the number of tests required during debugging. Assertions are based on time intervals, rather than treating events as dimensionless points.**

## 1. INTRODUCTION

This paper presents techniques for reliably detecting temporal errors in a distributed program, where a "temporal error" is defined to be a violation of the intended partial ordering of events. This is proposed as a supplement to existing debugging utilities as a way of increasing their consistency in the face of non-determinism, and thus reduce the number of tests required for a programmer to be satisfied that a program is correct.

As a compromise between efficiency and usefulness the approach is developed methodically from first principles by starting with the simplest possible temporal relationship, "happened before", for atomic, dimensionless *events*, and extended to finite time *intervals*. The presentation is motivated by two styles of assertion, one event and one state-based.

Two different implementations are considered. Firstly, a centralized system is discussed, based on existing debugging systems. A more attractive distributed approach is then presented which attempts to minimize the impact of the debugging code on the program under test.

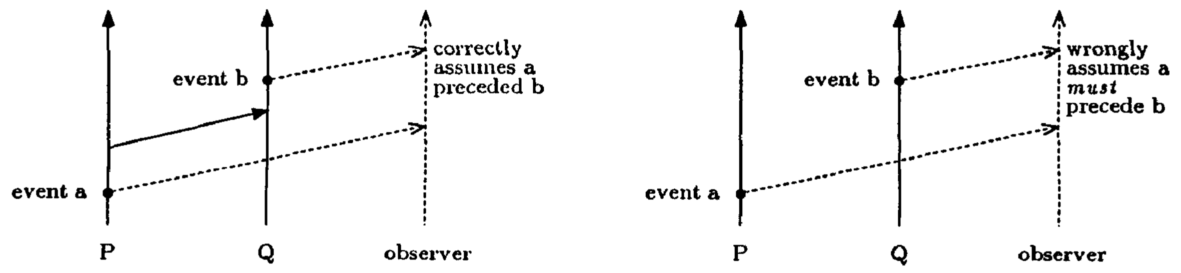
It is assumed that message-passing is the only form of inter-process communication. Both synchronous and asynchronous communication are discussed, although this work was originally motivated by consideration of synchronous systems. Note that we are only concerned with relative event orderings; no consideration is made of the issues associated with real-time or "performance" debugging.

## 2. MOTIVATION

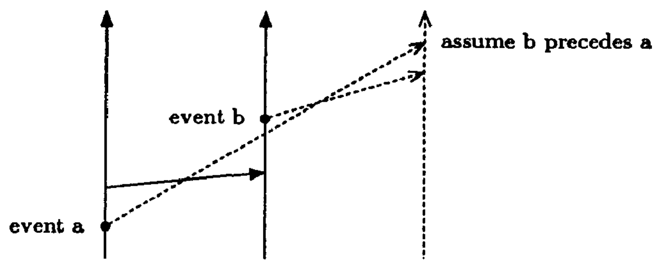
Faced with a black-box parallel program and a set of assertions specifying the intended behaviour of the program, a natural approach to debugging is to insert an *observer* (or *monitor*) process which receives reports from the program under test and checks that the assertions hold [2]. The observer may or may not delay the program while checking the assertions. This philosophy also underlies those systems which display significant events on a graphics terminal (where the onus is on the user to ensure that the program is seen to behave correctly) or those in which a linear *trace* is generated for post-mortem analysis [7].

This work was motivated by the observation that any such system implicitly totally orders the events of interest. This has a number of associated problems.

Firstly it is impossible for an observer process to distinguish between the case where the temporal ordering between two events is, and is not, enforced by a causal relationship:

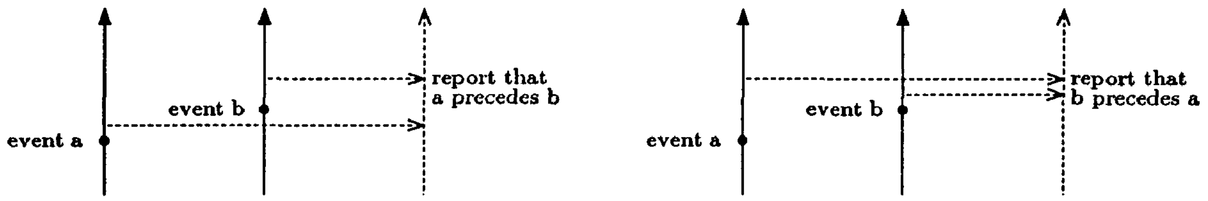

event b
event a
correctly assumes a preceded b
P
Q
observer
event b
event a
wrongly assumes a *must* precede b
P
Q
observer

Indeed it is even possible for an observer in an asynchronous system to perceive an incorrect total ordering due to "overtaking", even if FIFO queueing is guaranteed for each inter-process channel:



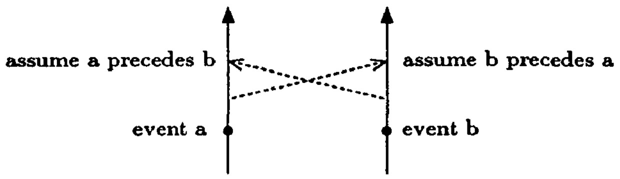
(Admittedly this can not occur in the synchronous case, or if an asynchronous observer blocks the observed processes by the use of "ack" signals.)

Additionally, the debugger itself may be non-deterministically affected by the (global time) interleaving of events. For exactly the same computation, with the same inputs, following the same control paths, and generating the same results, the debugger may produce different results due to differences in interleaving, and delays in reporting events. For example, assuming synchronous communication:



This forces the user to execute the <u>same</u> test again and again to ensure that the debugger has not missed a temporal error simply due to a fortuitous interleaving of events. Experience in this area has shown that it is necessary to run a particular test several times with different process priorities in the hopes of forcing an error to manifest itself [7]. It is important to note that this situation exists even with the use of reproducibility tools that do not totally order event traces (section 3.1).

Also, the perceived ordering differs for each process in the system. This problem is inherent in any attempt to add *auxiliary* communications to a parallel system to distribute information about past events. As a degenerate example, two processes simply informing each other of events will inevitably infer different temporal orderings in the absence of any additional information:



Finally, the observer itself may alter the behaviour of the system being studied [9]. This so-called *probe effect* [6] is particularly noticeable when the observer blocks processes.

The stand taken here is that the interleaving of concurrent events should not affect the results produced by a parallel debugging system. Any pronouncement on the correctness, or otherwise, of a particular test must be valid for any possible event ordering defined by that computation. Additionally, the perceived event ordering must be consistent for all observers.

We aim to achieve this by always working with the entire partial ordering of events defined by the computation, rather than a single, arbitrarily selected total ordering.

## 3. BACKGROUND

This section reviews related work on which the current presentation is based and defines some of the assumptions made.

### 3.1 Reproducibility

The value of reproducible tests for parallel programs as a means of controlling non-determinism during debugging is now widely recognized [16] and we accept it as a fundamental capability. This effectively allows efficiency issues to be disregarded by assuming that the debugging tools are added to prerecorded tests; in particular an observer can block monitored processes without altering the system behaviour [12].

Reproducibility is assumed to exist in its least intrusive form in which each process separately records the non-deterministic choices made and uses them to guide later replay [4]. This ensures that the replayed computations follow the same control paths and that the partial ordering of events is preserved, but it is important to note that the global time interleaving is not necessarily duplicated each time the test is replayed. In practice this effect is noticeable if several processes write to the terminal screen; the interleaving of messages may be different each time the computation is replayed.

It is further assumed that any observer processes or other auxiliary debugging code is not traced. Thus the debugger may see different event orderings just as the user at the terminal does. A debugger may therefore still produce non-deterministic results despite reproducibility!

## 3.2 Semantic Model

We adopt Reisig's semantic model in which a parallel program is defined as a partially ordered set of local states and actions [15]. There is no notion of global state. Each process is defined as a potentially infinite sequence of actions and their resultant states. An *action* is a single program statement (assignment, input, output) or a boolean guard. A synchronous message-pass is treated as a single, shared assignment action. A *state* consists of the current value of the program counter and the local variables for this process. We use "event" to denote a particular dynamic instance of a statically defined "action".

Thus for the following non-deterministic program (superscripts represent PC values),

$$\begin{array}{l} {}^{1}\mathrm{x} := 0;\ {}^{2}{*}[\ \mathrm{x} \leq 10 \rightarrow {}^{3}\mathrm{x} := \mathrm{x} + 2 \\ \qquad\quad \square\, \mathrm{c?x} \rightarrow {}^{4}\mathrm{skip}] \\ \| \\ {}^{5}\mathrm{y} := 5;\ {}^{6}{*}[\ \mathrm{c!y} \rightarrow {}^{7}\mathrm{y} := \mathrm{y} - 1] \end{array}$$

assuming synchronous communication, one of the many possible computations defined is:

$$\begin{tikzcd}
\text{(PC=1)} \arrow[r] & x := 0 \arrow[r] & \text{(PC=2, x=0)} \arrow[dr] & & \text{(PC=4, x=5)} \arrow[r] & skip \arrow[r] & \text{(PC=2, x=5)} \arrow[r] & x \leq 10 \arrow[r] & {} \\
& & & x := y \arrow[ur] \arrow[dr] & & & & & \\
\text{(PC=5)} \arrow[r] & y := 5 \arrow[r] & \text{(PC=6, y=5)} \arrow[ur] & & \text{(PC=7, y=5)} \arrow[r] & \cdots & & &
\end{tikzcd}$$

where state nodes are bracketed and action nodes are in italics. With asynchronous communication, the partial ordering becomes asymmetric. The semantics of the entire program consists of the union of all possible computations.

## 3.3 Time Intervals

Given this definition we distinguish two types of time interval that naturally arise.

Firstly an *event interval* [13] maps directly onto a lexical code segment within a particular process. The simplest case is a single statement, but in general it is used to refer to a block of code (this is very natural in a language such as occam[†] with its clearly delineated block structure). It is assumed that there is some mechanism for labelling code segments with some unique identifier. For example a mutual exclusion problem could be represented by labelling each critical region of code and asserting that control is never in the two regions at the same time. Referring to the computation graphs presented above, an event interval is a contiguous segment defined by the first and last action nodes.

A *state interval* is the period during which some boolean predicate on a process state holds. In this case the endpoints of the interval are the first and last local state node in which the predicate is true. Note that a debugger attempting to test such a predicate must not attempt to use undefined variables as this is yet another way of possibly introducing non-determinism into the debugger output; there must be a special value for "undefined".

Note that in both cases an interval is local to a particular process. The intended meaning of defining a state interval such as $x \leq y$, where $x$ and $y$ are in different processes, is unclear and poses many implementation difficulties. Our definition is restricted so that it is always possible to tell locally

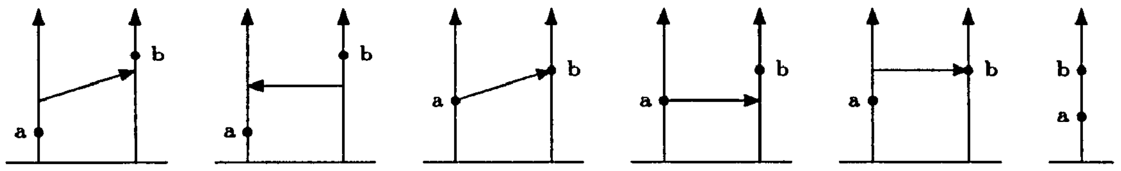
[†] occam is a trade mark of the INMOS Group of Companies

whether a process is "in" an interval or not. It is only via the interval relations themselves that inter-process comparisons can be made. Also note that an interval definition does not necessarily define a unique segment of the graph—this can be achieved by indexing.

One of the advantages of using intervals is that assertions are invariant under finite state repetition [11]. Thus executing null statements has no effect on the assertion checker—this is not the case in systems that count the number of discrete states in which a predicate holds.

## 3.4 "Happened Before"

"Happened before" is a transitive, irreflexive relation that directly describes causality in a parallel system [10]. Denoted $a \longrightarrow b$, it is true iff event $a$ must occur before event $b$, in any possible interleaving defined by the computation in which the two events occurred. The relation holds in all of the following cases (horizontal arrows represent synchronous communication):



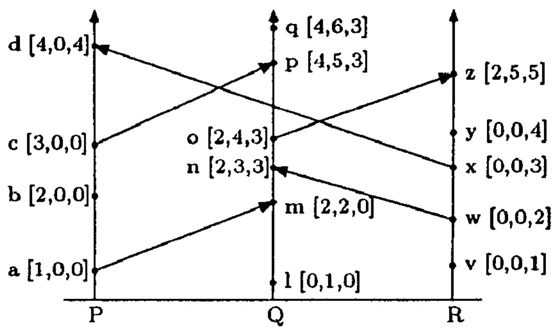
We adopt $\longrightarrow$ as the basis for interval assertions since it is the simplest possible temporal relation, and has been proven to be practically implementable (see below). Also, since it is a "past operator" it can be evaluated dynamically unlike "future operators" that require each computation to have successfully terminated before they can be tested [7].

### 3.4.1 Partially ordered clocks

It is possible to implement partially ordered logical clocks for message-passing systems that allow events to be timestamped for later comparison, preserving the full set of relationships defined by $\longrightarrow$ [5].

To achieve this each process maintains a clock consisting of a vector of integers, one element for each process in the network. The entire vector represents the current time and can be recorded to timestamp any event of interest (e.g. written to a trace file or sent to a monitor process). Each process ensures that its own vector element is incremented at least once between each atomic event (or at least before each event that will be timestamped).

Since communication events define the inter-process temporal orderings in a parallel computation special action must be taken at i/o events. In the asynchronous case the sender of a signal piggybacks the current time on the outgoing message (and adds 1 to its own slot on the outgoing vector to allow for transit time). The recipient compares the vector received with its local time and sets each element in the local vector to be the larger of the two. Later, when two vectors are compared, the slots for the process owning the first event are checked and $\longrightarrow$ holds only if the second timestamp has a larger value. Thus in the following example,



we can determine that $a \longrightarrow z$ (since $1 < 2$), $v \longrightarrow o$ (since $1 < 3$) and $a \longrightarrow c$ (since $1 < 3$). However, $\neg(b \longrightarrow o)$ since $2 \not< 2$, and $\neg(z \longrightarrow a)$ since $5 \not< 0$. The algorithm is similar in the synchronous case, except that clock vectors are exchanged during communication (in line with the symmetry of the rendezvous).

In effect the algorithm provides a way of labelling the action nodes in the computation graph. It can also be used to label state nodes, the only extension required being to "tick" the clocks immediately after the exchange in the synchronous case (so that the two local states immediately following communication do not receive the same timestamp).

The algorithm overcomes the problems of different observers seeing different event interleavings and the inability of observer processes to detect causal relationships. By timestamping significant events with partially ordered clock values, all debugging processes see an accurate, and identical, view of the computation graph (assuming that any debugging processes are not included in the clock algorithm).

### 3.4.2 Distributed synchronization

Since $a \rightarrow b$ relies on the ability of the first process to causally affect the second, we can say that the relation holds if it is possible to know, when $b$ is executed, that $a$ has already occurred. Thus by piggybacking a flag indicating the completion of $a$ onto all outgoing communications it is possible to check the assertion locally when $b$ is executed without the need to add any extra communications (indeed there must be no other ways of distributing this information apart from the original communications). This approach has been exploited as a way of synchronizing access to shared resources in distributed networks [14], but it is equally suitable for error detection.

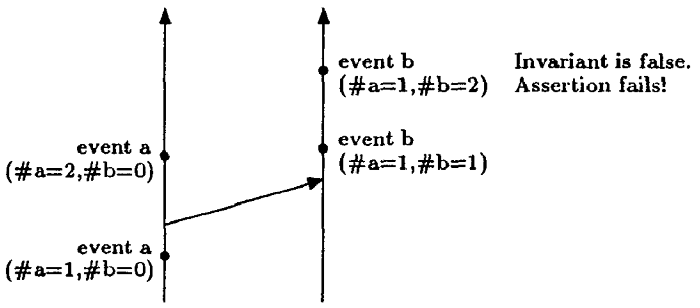
It can be generalized to assertions of the form

$$\forall i \, . \, a_i \rightarrow b_i$$

by having each process maintain auxiliary variables $\#a$ and $\#b$ representing the number of times events $a$ and $b$ have occurred respectively [3]. These *event counters* are carried on outgoing messages. The assertion may then be expressed as the global invariant

$$\#a \geq \#b$$

which is checked whenever either auxiliary variable changes. An error is signalled if the global invariant is ever false:



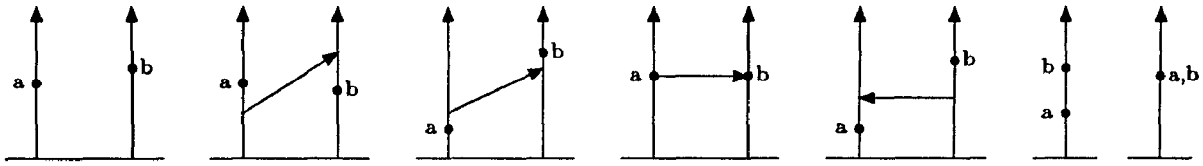
### 3.5 Interval Relations

Time intervals can be characterized by relations between their endpoints [1]. Using this as a starting point this section describes some interval relationships of interest that will motivate the following presentation.

Firstly we define some shorthand notations based on $\rightarrow$. Following [10], let $a \not\rightarrow b$ denote $\neg(a \rightarrow b)$. This includes the cases where $a = b$, $b \rightarrow a$, and where $a$ and $b$ are incomparable in the partial ordering in which case $a$ may occur before $b$.

If there is no relationship between two events they may occur in either order, or be thought of as "potentially concurrent". This is denoted $a \leftrightarrow b$ and is equivalent to $a \not\rightarrow b \wedge b \not\rightarrow a$.

Finally, let $a \rightsquigarrow b$ denote $a \rightarrow b \vee a \leftrightarrow b$. This could be interpreted as "a could occur before, or equals, b". Cases where this holds include:



From these definitions we now define some interval relations. For an interval $e$, let $Se$ and $Ee$ represent the start and endpoints respectively. The interval relation *precedes* is defined as

$$e \; precedes \; f \quad \equiv \quad Ee \rightarrow Sf$$

In other words, "the whole of $e$ must precede the whole of $f$, i.e. $e$ must complete before $f$ can begin" [13] in all possible global time interleavings.

A second interval relation which has an obvious application to mutual exclusion assertions is *may overlap*, defined as

$$e \text{ and } f \text{ may overlap} \quad \equiv \quad Se \rightsquigarrow Ef \wedge Sf \rightsquigarrow Ee$$

Informally, this assertion holds if there is any possible interleaving that could result in these two intervals overlapping in global time.

We also briefly refer to *includes*:

$$e \text{ includes } f \quad \equiv \quad Se \longrightarrow Sf \wedge Ef \longrightarrow Ee$$

## 4. CENTRALIZED ERROR DETECTION

This section outlines the algorithm required for a central observer process to reliably detect temporal errors. Given reproducibility it is assumed that the observer can block processes. This makes it impossible to perceive an incorrect interleaving and we can concentrate on detecting those orderings that do exist.

In summary the approach is:

i) the program under test supports partially ordered clocks,
ii) each time a process enters or leaves an interval, notification is sent to the observer, including a timestamp,
iii) the observer checks that the user's assertion still holds after each notification, using the timestamps to check temporal orderings that are not detectable from the interleaving of notifications.

The following sections define this in more detail for the two types of interval.

### 4.1 Event Intervals

As a motivational example consider the assertion

$$\forall i \;.\; p_i \text{ precedes } q_i \wedge q_i \text{ precedes } p_{i+1}$$

where $p$ and $q$ are event intervals. This assertion states that $p$ and $q$ must proceed in a strict interleaving starting with $p$. In practice $p$ can be thought of as a producer that creates some sort of structured object (e.g. a block of integers) and $q$ a consumer that uses the entire object (e.g. adding a checksum to create a data packet). The two intervals must never operate on the shared object at the same time (e.g. there is a buffer process holding the block which the producer and consumer can manipulate one integer at a time).

Care must be taken to ensure that the error detection code interacts correctly with the clock algorithm. To emphasize this we will make the clock code explicit; let "*tick*" represent incrementing the local vector element, "*exchange*" the vector exchange and update (synchronous communication), and "*now*" the current value of the logical clock vector. Since the clock algorithm nominally timestamps event nodes we must ensure that the timestamp sent to the observer is that which "belongs" to the action statements delineating the interval.

This is trivial in the case of sequential code as shown by the following pseudo-occam code fragment which we assume has been labelled "e":

```
e: SEQ                 SEQ
     x := y              tick
     z := x + 1    ⇒     observer ! Se; now
                         SEQ
                           x := y
                           tick
                           z := x + 1
                         observer ! Ee; now
                         tick
```

Note that if "e" was a single statement, both the start and endpoints would have the same timestamp.

Complications are introduced if the first or last event in the interval is a communications event, particularly if it is guarded. Since both "halves" of a synchronous communication event must have the same timestamp, the clock exchange must be done before notification is sent to the observer. Taking some liberties with the occam syntax, the following transformation represents the code required to label a guarded command "f":

```
ALT                          tick
  f: x=0 & c ? y             ALT
       d ! z          =>       x=0 & exchange
                                   observer ! Sf; now
                                   c ? y
                                   tick
                                   exchange
                                   d ! z
                                   observer ! Ef; now
                                   tick
```

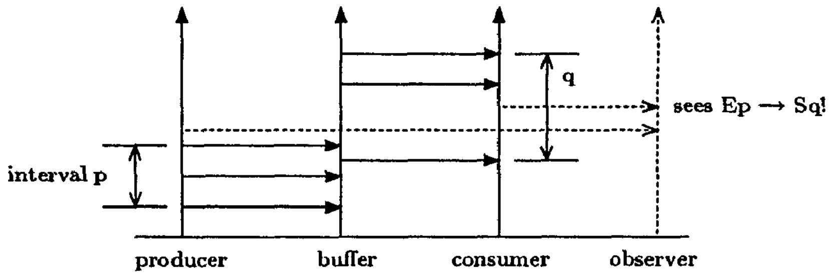
This reverses the clock exchange sequence from that described in [5], however as long as the first channel appearing in the exchange is that appearing in the original communication event, such a transformation cannot introduce deadlock, or change the semantics of the original program, since it does not alter the "committment" to communicate (this also holds for the asynchronous case as long as FIFO signal queueing is guaranteed).

Thus, looking at the exchange in more detail, the transformation for an output mirrors that for an input:

```
c ! x   =>   c ! localvector          d ? y   =>   d ? othervector
             c' ? othervector                      d' ! localvector
             merge vectors                         merge vectors
             ...                                   ...
             c ! x                                 d ? y
```

where *c'* and *d'* are the "reverse channels".

These transformations ensure that the start of an interval is reported to the observer immediately before it is entered. It is inadequate to notify the observer after the first event in the interval since delays in reporting let the observer see invalid orderings despite blocking (unless it is assumed that reporting to the observer is an atomic part of each observed event [2], an assumption that makes it impossible to express our algorithms at the source level):



Although the timestamps could distinguish the discrepency, it makes the code for the observer much more complicated.

We now briefly turn to the code required for the observer to detect an assertion violation in this example. Firstly note that the observer itself can maintain event counters for the four "meta-events" of interest. Given the restrictions described above the observer must see a valid ordering of these endpoint events. The only danger is that it cannot see all event orderings, i.e. the inability to detect causal relationships. The observer must be ready to accept notification of any event at any time (otherwise it may introduce deadlock). Upon receiving this it checks that the event is "congruent with the partial order" [2], i.e. $< Sp, Ep, Sq, Eq, Sp, \ldots >$, and that the appropriate "happened before" relationship enforces this. If not there is a potential error. Otherwise the implementation is very similar that of Baiardi *et al.*

In the general case, it is even possible to check such assertions if notification is made asynchronously with unpredictable delays and signal overtaking. The observer must maintain an array of timestamps for each event, and each time notification has arrived for both operands of a $\rightarrow$ relation it can be directly checked. The disadvantage, of course, apart from complexity, is that no obvious bounds can be placed on the storage requirements.

## 4.2 State Intervals

For state intervals the algorithm is slightly simpler since there are no shared nodes in the computation graph.

For each interval *s* a boolean variable *in_s* (initially false) is maintained which is true if the predicate defining the interval is currently true. Whenever any of the variables appearing in the interval definition could be changed (by assignment or input), the predicate is checked. If either of the interval endpoints has been reached, notification is sent to the observer. Thus for a state interval "*s*" defined as "$x > y$" the code transformation is

```
SEQ                       tick
  k := 1                  SEQ
  x := k + 2      ⇒         k := 1
  l := 5                    tick
                            x := x + 2
                            tick
                            check
                            l := 5
                            tick
```

where *check* tests to see whether the predicate has been changed by the assignment:

```
IF
  (NOT in_a) AND x > y
    SEQ
      in_a := true
      observer ! Ss; now
  in_a AND NOT (x > y)
    SEQ
      in_a := false
      observer ! Es; now
      ...
```

In the synchronous case the code following an input is equally simple

```
c ? x        ⇒    SEQ
                    exchange
                    c ? x
                    tick
                    check
```

(In fact the *tick* before the *check* is not strictly necessary since an output cannot change the state of the sender so it would not be possible for two state interval endpoints to receive the same timestamp anyway.)

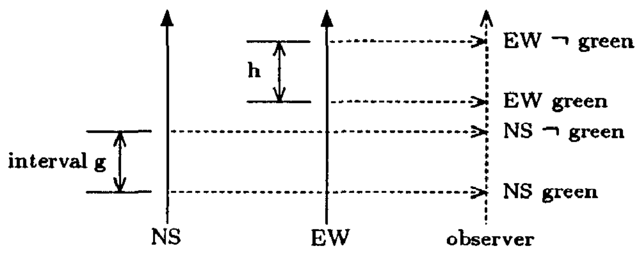
As an example, consider the following assertion

$$\nexists g, h \ . \ g \text{ and } h \text{ may overlap}$$

i.e. no instances of the state intervals *g* and *h* may ever overlap.

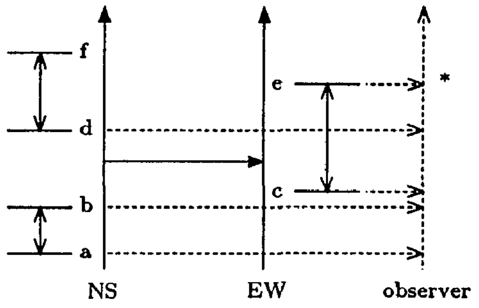
The classic example is a traffic light controller consisting of two processes, one representing the EW and the other the NS signals. The desired safety property, of course, is that both sets of signals must not show "green" at the same time. We assume that each process has a local variable representing the current colour displayed. Messages are sent to the external environment (which is not part of the system under test, see section 6.1) to effect the actual colour changes.

Again we note that an observer process cannot reliably detect this sort of error without a timestamping mechanism. The following obviously violates the intended system behaviour, but the observer never "sees" both intervals being active at the same time:



In this example the observer merely needs to maintain the four endpoint times and check that $\neg(Sg \rightsquigarrow Eh \wedge Sh \rightsquigarrow Eg)$ holds each time one is updated (after they have all been initialized). There is no need to count events in this case since it does not matter which instance of each interval is currently active.

Surprisingly the endpoints checked do not necessarily need to belong to the same interval instance as the following example shows:



At the point marked with an asterisk the observer can detect that $\neg(d \rightsquigarrow e \wedge c \rightsquigarrow b)$ which captures the error, even though event $d$ has overwritten event $a$!

The centralized error detection algorithm is also suitable (albeit somewhat redundant) for intra-process debugging (i.e. when both intervals are in the same process), and for post-mortem analysis of a totally ordered trace (the observer can be thought of as reading a pre-prepared trace, rather than receiving dynamic notifications).

## 5. DISTRIBUTED ERROR DETECTION

A centralized observer process, with it numerous links into the network under study, is expensive and may be difficult to implement in an inflexible h/w environment. This section examines the possibility of distributing information only via existing communication channels so that errors can be detected locally, within the existing processes.

In general the approach is:

i) each process maintains monotonically increasing auxiliary variables representing the number of times each endpoint has been passed,

ii) current values are distributed via existing communications (bi-directionally during synchronous communication),

iii) the global invariant defined by the assertion is checked whenever one of the aux vars changes.

However, as we shall see, our second motivational example is difficult enough to require a different approach.

### 5.1 Event Intervals

The event interval example is actually easier to check without an observer (unfortunately this is not generally true). This is because the assertion can be expressed as a conjunction of $\rightarrow$ relations and it is therefore possible to guarantee failure if the following expression is ever true

$$\#Sq > \#Ep \vee \#Sp > (\#Eq + 1)$$

As a disjunction this can be easily checked locally.

The debugging code inserted closely follows that of section 4.1. Immediately before the first event, and after the last event of the interval the appropriate aux var is incremented and and the invariant checked. The same considerations must be given to communication events marking the endpoints of an interval.

It is also possible to report success as well as failure. In general, for an assertion of the form $\forall i \, . \, a_i \rightarrow b_i$ success can be reported whenever $\#a$ and $\#b$ are greater than or equal to $i$, assuming that failure has not already been detected (we assume that ordinary execution is interrupted by the first failed instance). The user then sees a display of the form

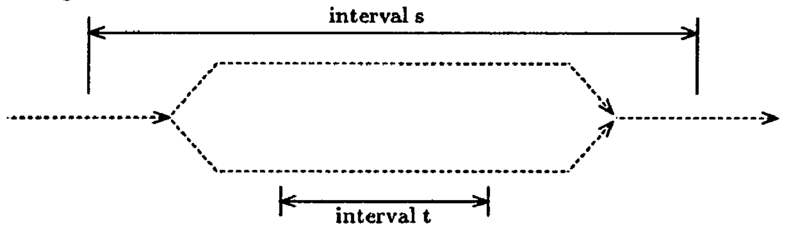
```
proven a1 → b1.
proven a2 → b2.
a3 → b3 failed!
```

This positive feedback can be very useful from the user's point of view since each report that the assertion holds for a particular iteration is a confirmation that the assertion must hold for this case (while conversely a failure report states that an error may occur).

## 5.2 State Intervals

On the other hand, the state interval example is particularly difficult to check in the distributed case because an error is possible if there was no communication. Since the expression to check failure is a conjunction of → relations, communication is necessary to detect failure.

To overcome this we firstly make a general assumption regarding termination: when a set of processes all successfully terminate, the auxiliary variables are merged and the invariant checked. This ensures that disjoint parts of the computation graph are checked at program termination, and it also maps naturally to languages with nested parallelism so that, for example, *s includes t* will be correctly detected in the following,



where the dotted lines represent control flow. This assumption means that we can guarantee that a temporal error will be detected if the computation is allowed to run to completion (analogous to a proof of partial correctness).

Instead of event counters each process maintains an auxiliary variable *last_time_s* for each interval *s* representing the last time that the predicate was true. For simplicity assume that this "time" is implemented using partially ordered clocks (the full generality of the clock algorithm is not necessary and since the logical clocks are themselves merely event counters, an extension to the endpoint counter variables could be used). Theoretically *last_time_s* is set to *now* whenever the predicate holds for the current local state, but efficiency can be improved by only re-setting it at the following points:

i) when the interval is entered,
ii) when the interval is left, and
iii) immediately before a communication event or process termination, if we are currently "in" the interval.

In this last case this must be done before the clock vectors are merged so that the two local states have different timestamps. It does not hurt to redundantly set the variable at any point, so it is not neccessary to ensure that communication will take place if it is not clear whether an i/o event will be selected at a non-deterministic choice. The current values of these auxiliary variables are distributed via existing communications as usual. Thus if $s$ is defined as $y = 5$,

```
SEQ                      SEQ
  y := x                   tick
  z := y + 2      ⇒        y := x
  c ? y                    set last time
                           check
                           tick
                           z := y + 2
                           set last time
                           exchange aux vars
                           check
```

```
tick
c ? y
set last time ...
```

where ***set last time*** sets $last\_time\_s$ to ***now*** if $y = 5$, ***exchange aux vars*** swaps and merges the auxiliary variables, and ***check*** tests the assertion.

Returning to the traffic light example, note that in terms of the semantic model, the assertion states that there must never be two local states $u$ and $v$ in which the predicates $g$ and $h$ hold respectively such that $u \leftrightarrow v$. Therefore it is merely necessary to check the invariant

$$\neg(last\_time\_g \leftrightarrow last\_time\_h)$$

whenever either auxiliary variable changes.

Overall, the distributed algorithm is simpler than the centralized case since the communications that define temporal relationships are the only means of distributing information. The global time interleaving therefore <u>cannot</u> have any effect on the results produced. In the centralized case, or whenever a debugger introduces additional communications, considerable effort must be expended on determining the original temporal relationships.

## 6. DISCUSSION AND FUTURE WORK

This section discusses unsolved problems and indicates directions for future work.

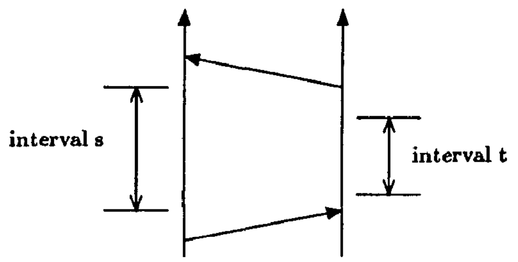
### 6.1 Assertion Language

For the purposes of this paper we have been deliberately evasive about exactly which language is used to express temporal assertions. In fact one of the aims of this work is to determine which language is most suitable. Ideally it should also be amenable to formal verification. A number of alternatives already exist, but each has drawbacks:

- Temporal logic is by far the most obvious choice, however since in its most common form it is based on operators that quantify over an <u>infinite</u> number of <u>future</u> states (*henceforth* and *eventually*), its applicability to dynamic evaluation during execution is unclear. One approach is to assume that the program <u>will</u> terminate and perform post-mortem analysis [7].
- For their debugger, Baiardi *et al* define a language based on observable communication events with operators for sequence, iteration, non-deterministic choice, etc. Our main objection to this language is that the assertions merely tend to parrot the original code. As shown by the examples in [2], the assertions map directly onto the original code, and in some cases are even bulkier.
- A totally new approach called "timesets" was proposed by Lamport in [11]. This paper influenced the work herein to some extent, but its reliance on a notion of global state does not match our semantic model.
- Hoare [8] expresses temporal relations as global invariants on trace lengths, however this is a very low level approach that maps directly onto the use of auxiliary event counter variables.

### 6.2 Global vs. "Observable" Time

We have assumed a notion of global time that allows us to say that the following two intervals overlap:



Arguably, however, this should not be considered true unless there is a process in the network that can observe this—with respect to <u>what</u> do they overlap? In the traffic light example there is no danger unless there is a process (the external environment) that is capable of "seeing" both lights green. This assumption greatly simplifies the distributed implementation, but we have avoided it for two reasons.

Firstly it seems counter-intuitive and prevents us from making assertions of the form that two intervals may overlap (in global time) which could be used to check that the desired level of concurrency is being achieved, i.e. the user wants to check that two intervals are unordered.

Secondly it means that the system under test must be "closed", with no communication to the environment. In the traffic light example a process must be added to simulate the environment since this is the only place where the mutual exclusion violation may be detected. This contradicts the aim of the distributed algorithm to avoid changing the communications graph.

### 6.3 Early Error Detection

Neither approach considered herein can detect an error at the earliest possible point. Communication within intervals, for example, often allows temporal relationships to be detected without waiting for the endpoints to be reached. The distributed algorithm in particular is limited by available communications—disjoint parts of the computation graph may even prevent assertions being checked until program termination. A hybrid approach in which each process maintains event counters with attached partially ordered timestamps representing the last time they were updated can alleviate this problem in some cases, but the additional complexity seems to outweigh the advantages.

## 7. CONCLUSION

For a parallel computation, the debugger itself produces non-deterministic results. We have shown that it is possible to avoid this by treating each test as a partially ordered set of events, rather than totally ordered. Some applications have been illustrated by example. Experimentally, our interest has been focussed on the distributed implementation and some limited state interval tests have been developed using our local CSP simulator. This work continues at the time of writing.

The author first became interested in this field while debugging a CSP program that generated graphical output (on a SUN workstation using Suncore primitives). A controller process initialized the graphics display, then a number of processes drew on the screen, and the controller closed the "viewsurface". During testing this program would non-deterministically crash towards the end of the computation (hence our initial interest in reproducibility). When finally tracked down the error was found to be due to insufficient synchronization between the controller and its subordinate processes, allowing the controller to attempt to close the display before all processes had finished writing to it. This hypothesis would have been immediately testable if we had had some easy way of asserting that the interval during which processes write *precedes* the shutdown procedure.

## REFERENCES

[1] J.F. Allen, "Maintaining Knowledge about Temporal Intervals", *Communications of the ACM*, Vol. 26, No. 11, November 1983.

[2] F. Baiardi, N. DeFrancesco and G. Vaglini, "Development of a Debugger for a Concurrent Language", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, April 1986.

[3] CORNAFION, *Distributed Computing Systems: Communication, Cooperation, Consistency*, Elsevier Science Publishers, 1985.

[4] C.J. Fidge, "Reproducible Tests in CSP", *Proceedings of the 10th Australian Computer Science Conference*, Deakin University, February 1987. Reprinted in *The Australian Computer Journal*, Vol. 19, No. 2, May 1987.

[5] C.J. Fidge, "Timestamps in Message-Passing Systems That Preserve the Partial Ordering", *Proceedings of the 11th Australian Computer Science Conference*, University of Queensland, February 1988.

[6] J. Gait, "A Probe Effect in Concurrent Programs", *Software—Practice and Experience*, Vol. 16, No. 3, March 1986.

[7] G.S. Goldszmidt, S. Katz and S. Yemeni, "Interactive Blackbox Debugging for Concurrent Languages", Israel Institute of Technology, Department of Computer Science, Technical Report 469, November 1987.

[8] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[9] J. Joyce, G. Lomow, K. Slind and B. Unger, "Monitoring Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 5, No. 2, May 1987.

[10] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[11] L. Lamport, "Timesets: A New Method for Temporal Reasoning About Programs", in *Logics of Programs*, D. Kozen (ed.), Springer-Verlag Lecture Notes in Computer Science No. 131, 1981.

[12] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, Vol. C-36, No. 4, April 1987.

[13] V. Pratt, "Modelling Concurrency with Partial Orders", *International Journal of Parallel Programming*, Vol. 15, No. 1, 1986.

[14] P. Quinton and J-P. Verjus, "Distributed Synchronization of Parallel Programs: Why and How?", in *Parallel Algorithms and Architectures*, M. Cosnard et al (eds.), North-Holland, 1986.

[15] W. Reisig, "Partial Order Semantics Versus Interleaving Semantics for CSP-like Languages and its Impact on Fairness", *11th Colloquium on Automata, Languages and Programming*, Belgium, LNCS 172, July 1984.

[16] S.M. Shatz and J-P. Wang, "Introduction to Distributed Software Engineering", *IEEE Computer*, Vol. 20, No. 10, October 1987.