Abstract

In many distributed computations it is necessary to ensure that the causal order in which events occur (a partial order) conforms to a total order that is derived from a correctness constraint on the computation. For example, the events of a distributed simulation must appear to be processed in simulation-time order. The Time Warp mechanism [Jefferson 84] uses the notion of global virtual time to schedule the events of a computation to be executed in an appropriate partial order. Under Time Warp, executions are subject to two sources of inefficiency, which we call transitivity delay and incremental rollback. We describe a new virtual-time synchronization protocol called Broadcast Time Warp which eliminates both of these undesirable phenomena, while retaining the synchronization semantics of Time Warp.

1 Introduction

In many distributed computations it is necessary to ensure that the order in which events occur conforms to a total order that is derived from a correctness constraint on the computation. Examples of such constraints are the serializability of database transactions, or the monotonicity of timestamps in a discrete-event simulation [9]. However, as Lamport noted in [7], the causal relationship of events (e.g. a → b because a is the sending of a message and b is the receipt) yields only a partial order, because of the absence of a global clock. He proposed the logical clocks mechanism as a method of extending the inherent partial order to a total order. Subsequently, Jefferson[5] observed that the converse approach is to specify a total order on the events, and then to provide a mechanism that causes the events of a computation to be executed in a partial order that conforms to the specified total order. He proposes a mechanism called Time Warp that allows the programmer to specify the virtual time at which an event (e.g. message receipt) must occur, and then synchronizes processes to ensure that the specified order is observed.

The original Time Warp protocol is subject to two sources of inefficiency, which we call transitivity delay and incremental rollback. In section 4 we describe a new virtual time synchronization protocol that overcomes these disadvantages.

2 The Time Warp Protocol

Time Warp synchronizes a set of processes that communicate using asynchronous message passing. Each process maintains a local “virtual clock” that records the local virtual time of the process. When a process sends a message, it labels the message with a virtual send time equal to the local virtual time when the message was sent, and a virtual receive time, the virtual time at which the message is to be received. The virtual receive time must be at least the virtual send time. Each process has an input queue and an output queue (Figure 1). When a message arrives at a process, it is placed in its input queue. When a process sends a message, a copy of the message is placed in the output queue of the process. A process consumes messages in its input queue by choosing a message whose virtual receive time is equal to the local virtual time of the process. It then acts upon the message (possibly causing the process to send messages of its own), and sets its local virtual clock to the next largest virtual receive time among the messages in its input queue, or ∞ if the queue is empty.

It is possible for a message to arrive whose virtual receive time is earlier (less) than the local virtual time of the process (figure 2). In this case the message has arrived “in the virtual past.” All of the work done by the process between its current virtual time and the virtual receive time of the new message is potentially...
incorrect, since it did not take the new message into account. To allow the process to undo this work, it checkpoints its state each time it increases its local virtual clock. Upon receiving a message in the virtual past, the process rolls back to its most recent checkpoint taken at or before the virtual receive time of the new message. The process completes its rollback by sending an antimessage for each of the messages in its output queue whose virtual send time is later than the current local virtual time. The antimessage has virtual send and receive times identical to the original, “positive” message. An antimessage has the effect of annihilating with its positive counterpart in the input queue of the receiving process, as well as with the copy of the positive message in the output queue of the sending process. Antimessages arriving in the virtual past are treated the same as positive messages: They cause the receiving process to roll back.

Finally, processes execute a separate protocol that periodically updates global virtual time (GVT). GVT is the minimum of the set of local virtual times and the virtual send times of all unreceived messages. This allows processes to discard all checkpoints and input messages older than GVT.

3 Disadvantages

In the execution of the Time Warp protocol, there may occur two phenomena that increase both its communication and processor overhead during rollback. The first of these is transitivity delay, in which a process is at the end of a long chain of antimessages that are sent when the process at the head of the chain initiates a rollback. Consider the sequence of events shown in figure 3. At local virtual time 5, process A sends the positive message +m to process B, with virtual receive time 5. Upon receiving +m, B sends +n to process C, and C subsequently sends +p to process D. After sending +m, process A receives a message whose virtual receive time is 4, and must therefore roll back and send an antimessage, −m. This results in B and C sending antimessages −n and −p, respectively. Thus, once A rolls back to time 4, there is a delay of three messages before D initiates its rollback, even though it is certain before the first antimessage is sent that D must roll back, since its receipt of +p is at the end of a chain of messages that begins with +m. Any computation performed by D after A begins its rollback is doomed to be undone. Therefore, D should
begin its rollback as soon as possible after $A$ rolls back, so that it may begin executing forward again sooner.

The second phenomenon, incremental rollback, is due to the receipt of multiple antimessages. Consider message $+r$, which $A$ sent directly to $D$ at the same virtual time it sent $+m$. When $A$ rolls back, it sends antimessage $-r$ in addition to $-m$. When $D$ receives $-p$, it rolls back to its checkpoint at virtual time 7. When it subsequently receives $-r$, it must roll back a second time to its checkpoint at virtual time 4. Thus a single rollback at $A$ has resulted in multiple rollbacks at $D$. A rollback by $A$ to a virtual time preceding 5 should cause only a single rollback (also to a time preceding 5) at each process that has received messages from $A$. Multiple "shallow" rollbacks consume additional processor time, and may cause more messages to be sent than a single, "deep" rollback. This is because a process may roll forward between shallow rollbacks, sending messages which will immediately be cancelled. A deep rollback does not give the process an opportunity to send these erroneous messages.

Both transitivity delay and incremental rollback could be avoided if it were possible for process $A$ to notify every process along the chain of messages beginning with $+m$ (and $+r$) that it must roll back to a local virtual time that precedes the virtual receive time of a message that is part of the chain. In the next section we describe an alternative to the Time Warp protocol that replaces the functionality of a chain of antimessages with a single kill message that is broadcast to all processes.

4 Broadcast Time Warp (BTW)

The BTW protocol is based upon the following observation: When a process (e.g. $A$) rolls back to a checkpoint at virtual time $t$ because it received a positive message in the virtual past (a spontaneous rollback), the need for another process (e.g. $D$) to roll back (an induced rollback) is completely determined by the existence of a dependence chain of messages between the first process and the second process, such that the virtual send time of the first message in the chain is greater than $t$. Therefore, to determine whether a process, $p$, needs to roll back, it is sufficient to record at $p$ the virtual send time of each message from each other process, $q$, that begins the most recent dependence chain from $q$ to $p$.

To make use of this information, the protocol must perform these functions:

- Include in each message the virtual send time(s) that begin the dependence chain(s) of which the message is a member.
- Provide a mechanism for process $q$ to inform process $p$ that $q$ has rolled back to time $t$.
- Remove messages that are part of the affected dependence chain(s) from the input queue of $p$.

To perform the second function, each process maintains a set of dependence attributes. A dependence attribute is of the form $(id, (vt, count))$, where $id$ is a process identifier, $vt$ is a virtual time, and $count$ indicates the number of times that process $id$ has previously executed events at local virtual time $> vt$, or, equivalently, the number of times $id$ has rolled back to time $vt$. Such an attribute stored at a process $p$, denotes that a dependence chain of messages from $id$ to $p$ is the count$^{th}$ such chain that began at virtual time $vt$. As described below, the count is necessary to distinguish among multiple rollbacks by a process that span a common virtual time interval.

A message is of the form $(alist, rt, data)$, where $alist$ is a list of dependence attributes, $rt$ is the virtual receive time of the message, and $data$ is the message's data portion. We refer to the components of message $m$ as $m.i, m.r$, and $m.d$, and the components of a dependence attribute, $a$, as $a.i, a.r$, and $a.c$. A kill
message contains a single dependence attribute, indicating the origin of a chain of messages that is to be killed. To simplify our protocol, we assume that any two messages from a single sender are received in the order they were sent (FIFOness w.r.t. senders).

4.1 Process Specification

Processes maintain an input queue of messages and a local virtual clock as in the original Time Warp protocol, but need not keep a queue of output messages. The dependence attributes of a process are captured by the messages of the input queue, and so need not be stored separately.

The behavior of a process is specified by the following four functions. queue(m) is executed in response to the arrival of message m at the process. kill(k) responds to the arrival of a kill message, k, and send(p, d) sends a message to process p containing data portion d. gvt(t) performs fossil collection (removal of checkpoints and messages that are no longer needed) upon receiving an update of global virtual time (assumed to be performed by a separate protocol). In following code, lvt refers to the local virtual time of the process, Q is the input queue, r is the virtual time beyond which a process must roll back upon receiving a kill message, and count(t) is the number of times the process has rolled back to time t. self is the unique identifier of the process.

queue(m) {
  if ∃ kill message k in input queue
    ∧(∃a ∈ m.l : a.i = k.i
      ∧a.v > k.v ∧ a.c = k.c) then
discard m;
  if m.r < lvt then { /* spontaneous rollback */
    restore checkpoint taken at t, t < m.r;
    broadcast(KILL, (self, (lvt, count(lvt))));
    foreach t', t ≤ t' ≤ lvt
      count(t') := count(t') + 1;
  }
  else
    place m in Q;
}

kill(k) {
  place k in Q;
  r := +∞;
  foreach m ∈ Q such that
    ∃a ∈ m.l : a.i = k.i ∧ a.v > k.v ∧ a.c = k.c
    if m.r < r then
      r := m.r;
      remove m from Q;
  }

send(p, d) {
  x := (self, (lvt, count(lvt))), lvt, d);
  foreach (i, (v, c)) ∈ m.l such that
    m ∈ Q ∧ v maximal for i
    ∧(i, (v, c)) not previously sent to p
    x.l := x.l ∪ {(i, (v, c))};
  send(p, x);
}

gvt(t) {
  discard checkpoints older than t;
  foreach m ∈ Q, m.r < t
    remove m from Q;
  foreach kill message k ∈ Q, k.v < t
    remove k from Q;
}

4.2 Execution

Unlike antimessages, which annihilate with only one positive message, kill messages remain in the input queue until removed by the GVT algorithm. When a process receives a message, it examines its input queue for any previously received kill message that kills a chain of messages to which the newly arrived message belongs. If a matching kill message is found, the new message is discarded. Otherwise, the virtual receive time of the new message is examined. If it is less than the current local virtual time (i.e. the message arrived in the virtual past), then a rollback is initiated to the checkpoint taken immediately prior to the virtual receive time of the message. After a rollback to time t, the object broadcasts a kill message containing its unique identifier, the rollback time, and a count that distinguishes the current kill message from previous kill messages for the same virtual time. If the new message has arrived in the virtual future, it is placed in the input queue.

When a kill message, (i, (v, c)) from object i is received, the receiving object removes from its input queue all those (previously received) messages that are part of a chain that begins at i after virtual time v. The kill message remains in the input queue to allow removal of any messages that are part of such a chain, but arrive after the kill message itself. However, mes-
sages that belong to subsequent chains originating at object \( i \) and at time \( v \) will not be killed by this kill message, since their count value will exceed \( c \). Subsequent to receiving a kill message, a process must roll back iff its local virtual time exceeds the virtual receive time of any of the killed messages, that is, if it has already processed a message that is now killed. The checkpoint to which it must roll back is the one taken immediately prior to the minimum of the virtual receive times of the killed messages. A rollback that occurs in response to the receipt of a kill message does not cause any additional kill messages to be sent, since the original kill message will induce any necessary rollbacks at other processes.

A message that is to be sent from process \( q \) to process \( p \) is constructed by including in the \( alist \) of the message a dependence attribute that contains the identifier of the sending process, the current local virtual time (i.e. the virtual send time\(^1\)), and the count value for that virtual time. In addition, any dependence attributes that have been received since the last time a message was sent to \( p \) are included in the \( alist \).

The condition "\( v \) maximal for \( i \)" ensures that the \( alist \) of the outgoing message extends to \( p \) only the most recent dependence chain from \( i \) to \( q \), for each process, \( i \), in the system. The condition further guarantees that the \( alist \) will contain at most one dependence attribute for each process, i.e., the length of the \( alist \) is at most linear in the number of processes. This is sufficient because \( p \) must be able to identify only the most recent dependence chain from \( i \) to \( p \) in order to be able to respond correctly to any subsequent kill messages sent by \( i \).

Finally, when a process receives a message informing it of a change in global virtual time, it discards any checkpoints that are older than GVT, as well as any (kill) messages whose virtual times precede GVT.

### 4.3 Example

Figure 4 shows an example execution. Messages \( r \), \( m \), \( n \), and \( p \) have been sent as in figure 3. For each of \( B \), \( C \), and \( D \) there is a dependence chain of messages that originates at process \( A \) at virtual time 5. At \( C \) and \( D \) there are also chains beginning at \( B:7 \), and at \( D \) there is a chain beginning at \( C:9 \). When \( A \) receives message \( z \), it must roll back to virtual time 4. This causes it to send a kill message, killing those chains beginning at \( A:5 \) whose count is 1. Since each of \( B \), \( C \), and \( D \) are on at least one of these chains, they

---

\(^1\) We have arbitrarily set the virtual receive time of each message equal to its virtual send time.

### 5 Comparison of BTW to TW

In comparing BTW to the original TW protocol, we assume a distributed environment in which all processes execute in parallel and the communication network is a broadcast-bus (e.g. Ethernet [8]), on which the unit cost (propagation delay and bandwidth consumed) for sending a broadcast message is identical to that for point-to-point messages.

#### 5.1 Transitivity Delay

BTW eliminates the transitivity delay of Time Warp by killing all messages in a dependence chain.
that is no longer valid with a single broadcast message. Processes needing to roll back are delayed for only constant (one message interval) time. By comparison, The delay suffered by TW processes is, in the worst case, linear in the number of processes: A rollback at one process may spawn a chain of antimas-
sages that reaches each other process. For n processes, the rollback of the last process in the chain is there-
fore delayed by n – 1 message intervals. In general
dependence chains may be arbitrarily long, and one
might therefore expect the transitivity delay of TW
to be unbounded in the worst case. However, the posi-
tion of any process in a dependency chain is at most
n – 1 messages from the head of the chain. Thus the
maximum transitivity delay is at most n – 1. Never-
theless, this linearity suggests that TW does not scale
well to large numbers of processes, a fact borne out by
simulation studies [3].

5.2 Incremental Rollback

Although it is still possible for BTW processes to roll back several times in a row by receiving several kill
messages, a single rollback at one process will never
cause multiple rollbacks at another process, because
only a single kill message is sent. When there exist
multiple dependence chains between the sender of a
call message and a receiver, the receiving process per-
foms a single rollback to the earliest checkpoint nec-
necessary. In TW, the worst-case number of induced roll-
backs by a single process in response to a single sponta-
aneous rollback is unbounded: Consider a process,
p, which has received messages +m0, +m1, ... , +mk,
each of which is part of one or more dependence
chains beginning at process q. When q rolls back,
p may receive the corresponding antimasages in re-
verse order2: −mk, −m_{k-1}, ..., −m1. In this case, p
will roll back k + 1 times, while under similar circum-
stances a BTW process would roll back exactly once.

5.3 Message Traffic & Storage

The number of messages sent per rollback is sub-
stantially reduced under BTW. Exactly one broadcast
message is sent per spontaneous rollback, and no ad-
nitional messages are generated by induced rollbacks.
In Time Warp, a single rollback may give rise to an
unbounded number of antimasages. This reduction in
the number of messages is achieved at the cost of in-
creased message size. The size of a BTW message
depends upon the size of its alist component. Since
each process identifier appears at most once in an al-
ist, the maximum length of an alist is linear in the
number of processes.

BTW also eliminates the need for keeping previ-
ously sent messages in an output queue. This reduc-
in storage requirements is offset by an increase in
the size of an average message in the input queue, and
the need to retain kill messages.

5.4 Performance Comparison

The performance of TW was compared to that of
BTW using a hand-coded simulation, executed on a
single-processor system. The simulation models the
parallel execution of a fixed number of TW/BTW
processes on a broadcast-bus network, using a FIFO
queue of messages to implement the broadcast chan-
el. Relative performance was measured under syn-
thetical workloads [3, 6] that are representative of typ-
ical TW applications in parallel discrete-event sim-
ulation. A synthetic workload is one that is gen-
erated using a mathematical model of a class of appli-
cations, rather than execution traces of an actual ap-
lication. By varying the parameters of the model,
synthetic workloads can be generated that capture a
wide range of possible behaviors of the modeled appli-
cation class. Our simulation uses the PHOLD model,
which has previously been used to study the perfor-
manve of Time Warp [3].

5.4.1 Parameters

The following parameters were used in the simulation:

- **Number of processes**: The number of TW or BTW
  processes logically executing concurrently. The
  effect of increasing the number of processes is to
  increase the number of messages that are con-
  sumed and generated concurrently.

- **Number of causality threads**: In the PHOLD
  model, messages correspond to events in a paral-
  lel discrete-event simulation in which the comple-
tion of an event causes exactly one new event to
  be scheduled for future execution. The consump-
tion of an input message by a TW/BTW pro-
cess causes the process to send exactly one out-
put message. Thus the total number of positive
messages in the system remains roughly constant
over time, and the number of causality threads is
determined by the number of initial messages
placed in the input queues of the TW/BTW pro-
cesses.
• **Temporal locality:** This parameter is a function that determines the virtual receive time of a message, given its virtual send time. In discrete-event simulation terms, the function determines how far in the future the event is scheduled to be processed.

• **Spatial Locality:** This function determines the set of possible destination processes for a message sent by a given process. In computations with high spatial locality, each process receives messages from only a small number of other processes.

### 5.4.2 Results

The parameters most significantly affecting the performance of the two protocols were found to be the number of processes and the number of causality threads. The data presented here show that the advantages of BTW become more pronounced as the computation is scaled to larger numbers of processes or threads. Although there are clear trends in the data, the performance of TW and BTW does not vary monotonically with the number of processes. This is due to the sensitivity of virtual-time synchronization to variation in message arrival times, which is in turn determined by the number of processes accessing the broadcast channel. The data shown here were obtained from simulation runs in which temporal locality varied uniformly over the interval [1…5], and the spatial locality for process \( p \) was uniformly distributed over processes \([p+1 \ldots p+2]\) (modulo the number of processes).

The relative performance of TW and BTW is most dramatically illustrated by the transitivity delay data. We measured the average number of antimessages that form a chain between a spontaneous rollback and an induced rollback for 4, 8 and 16 causality threads, over a range of 4 to 63 TW processes. Figure 5 shows that the delay is linear in the number of processes under TW, with the average length of a dependence chain being roughly half the number of processes. The observed delay also becomes more regular as the number of threads per process is increased. This suggests that transitivity delay is a significant obstacle to scaling TW computations to large numbers of processes: As the number of processes is increased each process spends a higher proportion of its time executing doomed computations—computations that will eventually be rolled back because of the arrival of an antimessage. Since transitivity delay is constant at \( i \) under BTW, its scalability is not affected.

Long chains of antimessages also contribute to increased message traffic. Consider that if the average transitivity delay under TW corresponds to a dependence chain \( d \) messages long, those \( d \) messages are replaced by a single kill message in an equivalent BTW computation.\(^3\) Given a TW computation in which the total number of antimessages is \( a \) and the number of induced rollbacks is \( i \), the total number of kill messages sent under BTW would be \( a - i + (i/d) \). The term \( a - i \) gives the number of antimessages which do not induce rollbacks, and so are "replaced" one-for-one by kill messages. The number of kill messages needed to induce \( i \) rollbacks is \( i/d \). Message traffic is further affected by the reduced tendency of BTW processes to send erroneous positive messages—messages that are doomed to be cancelled—since BTW processes spend less time in computations that produce such messages. The reduction in message traffic obtained under BTW is shown in figure 6 as the fraction of total message traffic that is eliminated under BTW, compared to TW under identical conditions. Here, the number of threads is the significant factor, with the number of messages being reduced by 40% with 8 threads.

### 6 Optimizations

There are two modifications that may be made to the basic BTW to further improve its efficiency. The first is the addition of the "lazy cancellation" mechanism described in [4], and the second is the integration of BTW with Associative Broadcast [1, 2] to reduce the number of kill messages received.

#### 6.1 Lazy Cancellation

In TW, lazy cancellation delays the sending of antimessages after a rollback until it is certain that upon reexecution, the process will send a different set of messages than it did before the rollback. Thus only those messages that were produced in error are "unsent," instead of simply cancelling messages according to their virtual send times.

In the same way, BTW may delay the sending of kill messages. This requires that processes maintain an output queue of messages in the same manner as TW processes. Now a kill message is not sent until it is certain that at least one of the messages in the output buffer heads a chain that must be killed.

\(^3\) An equivalent computation is one containing an identical pattern of positive messages.
Figure 5: Transitivity delay in Time Warp

Figure 6: Message traffic reduction in Broadcast Time Warp
6.2 Kill Message Reduction

We have thus far assumed that kill messages are sent using a simple broadcast primitive which transmits a kill message to all BTW processes. However, not all processes need actually receive every kill message. Only those processes that are part of the dependence chain being killed by a particular kill message must receive the message. The sender of the kill message, on the other hand, has no way of determining which processes are part of the chain. If the only communication primitives available are point-to-point messages or network-wide broadcast, then the latter must be used.

However, Associative Broadcast allows a set of processes to be specified as the target of a broadcast message, based on the local states of the candidate processes. After briefly describing the Associative Broadcast model, we show how it can be used to deliver kill messages only to those processes that are part of the killed dependence chain.

6.2.1 Associative Broadcast

Associative Broadcast uses message broadcasting as the fundamental mode of communication among processes. Communication is 1-N: A sender specifies a target set of processes that are to receive a message. Associated with each process is a profile, which consists of a set of attributes. An attribute is either a symbol (a simple attribute) or a (symbol, value) pair (a compound attribute). An attribute represents an abstraction of the state of the process, or an operation the process is capable of performing. For example, in a distributed hashing algorithm, the profile \{INSERT, FIND, (MIN, 0), (MAX, 10)\} might indicate that the process supports the INSERT and FIND operations on key values in the range 0–10.

A process is specified by a set of local data structures, and a set of operations. Receipt of a message causes the operation indicated in the message to be executed. The operation may modify local data, alter the profile of the process, and broadcast messages. A message is a tuple, ([selector], arg, ...), where [selector] is a propositional formula over attributes that specifies the target set of the message. For example, broadcast([INSERT ∧ MIN < 3], x) might send the value x to all processes whose MIN attribute is less than 3, and cause them to perform the INSERT operation using the value x. After a message is broadcast, the selector contained in the message is matched against the profile of each process. If the selector is true for the profile of a process, the message is received by the process. The broadcast primitive is not atomic: The profile of a process may change while the message is in transit. In this case, the process is not guaranteed to receive the message.

6.2.2 BTW using Associative Broadcast

A kill message can be directed to the appropriate subset of processes by specifying in its selector the identity of the sender, the virtual time to which the sender is rolling back, and the count value of the current rollback. Processes maintain their dependence attributes as part of their profiles. A kill message is delivered only to those processes whose profiles contain a dependence attribute that satisfies the selector.

The initial profile of each object is \{QUEUE, KILL, GVT\} specifying the functions that the process can perform. Only the queue() function of the previously given BTW code must be modified to use Associative Broadcast, as follows:

```plaintext
queue(m) {
    if ∃ kill message k in input queue
        ∧ ∃ a.i ∈ m.l : a.i = k.i
        ∧ a.v > k.v ∧ a.c ≤ k.c then
            discard m;
    if m.r < let then {
        restore checkpoint taken at t, t ≤ m.r
        (including profile);
        broadcast([KILL ∧ self.v > t
            ∧ self.c = count(t)], (self, (t, count(t))));
        foreach t', t ≤ t' ≤ let
            count(t') := count(t') + 1;
    } else {
        place m in Q;
        foreach a ∈ m.l {
            profile := profile ∪ {(a.i, (a.v, a.c))};
        }
    }
}
```

BTW processes behave as before, except that they place the elements of the alist of each input message in their profiles. They also specify in the selector of an outgoing kill message that the kill message is to be received by those processes whose profiles contain a dependence attribute that identifies the process as belonging to the a killed dependence chain. When the average number of processes affected by a kill message is much smaller than the total number of processes,
this significantly reduces the number of kill messages that each process must handle.

7 Conclusions

The Time Warp protocol, originally envisioned as a general-purpose synchronization mechanism, has been most successfully applied to parallel discrete-event simulation. As the parallelism in such simulation applications is increased, the overhead due to rollbacks, antimessages, and transitivity delay consumes a relatively larger proportion of the available computational resources. We have presented an alternative protocol—based on the same virtual-time synchronization semantics as Time Warp, and therefore identical to TW from the application perspective—which reduces the communication and rollback overhead by taking advantage of broadcast communication. By maintaining state information about the causal relationship among messages, large numbers of antimessages can be replaced by a single kill message. The resulting reduction in communication delay also leads to less wasted computation and fewer rollbacks.

References

[1] B. Bayerdorffer, Associative broadcast and the communication semantics of naming in concurrent systems, Ph.D. dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, Dec. 1993


