Analyzing Synchronous Distributed Algorithms

Jeff Magee

Department of Computing, Imperial College London,
South Kensington campus, London SW7 2AZ, UK
j.magee@imperial.ac.uk

Abstract. Synchronous distributed systems are those in which there is assumed to be a known upper bound on each processing step, a known upper bound on message transmission and processes have perfectly synchronized physical clocks. When these assumptions hold for a system, execution can be arranged to proceed in synchronous rounds. This paper is a tutorial on modeling and analyzing these systems using finite state machines to model processes and communication links, linear temporal logic to express required properties and model-checking to demonstrate that models satisfy the required properties. The paper focuses on models of Atomic Commitment protocols and illustrates the use of the LTSA tool in analyzing these models.

1 Introduction

A model is a simplified representation of the real world. This simplified representation lets us focus on some aspects of a problem while deferring the consideration of others. The synchronous model of distributed systems provides an idealized version of distributed computation that is a good basis for studying Atomic Commitment algorithms. Atomic Commitment is by far the distributed agreement protocol most commonly used in practice. In the following, we model both two-phase and three-phase commit protocols as a way of gaining a detailed understanding of the operation and properties of these protocols. The paper also discusses the aspects that need to be addressed when translating models into implementations. The objectives of this paper are twofold; firstly, to provide a detailed exposition of Atomic Commitment protocols and secondly, and perhaps more importantly, to explain a modeling approach and its associated tools that can be applied to synchronous distributed algorithms in general.

The approach adopted for modeling is essentially that outlined in [1] except the book looks at models for concurrent programs whereas here we are concerned with distributed algorithms. Processes and communication links are modeled using a form of state machine known as a Labeled Transition System (LTS). For other than small problems, it is cumbersome to directly specify LTSs, so as in [1], we use the Finite State Process (FSP) notation to specify processes and use the LTSA tool [2] to generate the corresponding LTSs. In the following, we will only briefly and informally describe FSP, the reader is referred to [1] for a comprehensive treatment.

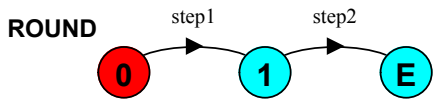
To specify properties, we use a form of linear temporal logic called fluent linear temporal logic (FLTL). The focus in this paper is on using FLTL to express the properties required of atomic commitment protocols. The reader is referred to [3] for more information. In essence, we model systems using actions and events and FLTL lets us specify properties in terms of abstract states of these systems. The LTSA model-checking tool is used to mechanically verify that systems satisfy the properties required of them.

The treatment of synchronous distributed algorithms follows that outlined in Nancy Lynch's book [4]. We adopt a model checking approach to analyzing algorithms while [4] uses a proof theoretic approach. We view the approaches as complementary rather than competitive. Our approach has the advantage of mechanically providing example executions as counter-examples to property violations and witnesses to property satisfaction, as an aid to understanding. The proof theoretic approach has the advantage of showing that all system configurations satisfy the required properties rather than only a specific configuration as in model-checking.

In the following, section 2 develops a precise model for synchronous networks, section 3 develops a precise specification using FLTL for the Atomic Commitment (AC) problem and sections 4 & 5 model the two-phase and three-phase AC protocols. Finally, section 6 discusses the approach and its extension to other distributed system models.

2 Synchronous Network Model

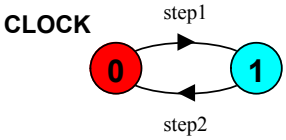
A network consists of a set of point-to-point connections or links that permit processes to exchange messages. A network is synchronous if there is a known upper bound for the time it takes a process to execute a local processing step and a known upper bound on message transmission delay. We also assume that processes have access to perfectly synchronized physical clocks. In practice, when the first two properties hold, approximately synchronized (with a known bounded drift $\varepsilon > 0$, from each other or from real time) clocks can be implemented. However, for modeling purposes, it is simpler to assume perfectly synchronized clocks. The consequence of these properties are firstly that we can use timeouts to detect link and process failures and secondly that we can organize computation into rounds. A round consists of sending messages to a set of processes, receiving all messages that have been sent and then changing state. More precisely, following Lynch [4], a round consists of two steps; in step1, processes send messages and in step 2, processes receive messages and change state. We model a round as consisting of two actions `step1` and `step2` that are shared by all processes that constitute a model of a synchronous system. A single round is modeled in FSP as shown in Figure 1 together with the LTS that it generates. The FSP "`->`" denotes action-prefix and means that a process engages in the action preceding the arrow and then becomes the process following the arrow – which in the case of `ROUND` is another action-prefix.




```
ROUND = (step1 -> step2 -> END).
```

Fig. 1. FSP for `ROUND` and associated LTS

A synchronous distributed system continuously executes rounds as modeled by the process `CLOCK` shown in Figure 2 together with its LTS. The process is constructed from `ROUND` using sequential composition ";" and recursion. In addition to the actions `step1` and `step2`, `CLOCK` has the action `never` added to its alphabet. This action is never executed by `CLOCK` or any other process that shares the action. The need for the action will be seen in the next section concerned with specifying properties.




```
CLOCK = ROUND;CLOCK+{never}.
```

Fig. 2. FSP for `CLOCK` and associated LTS.

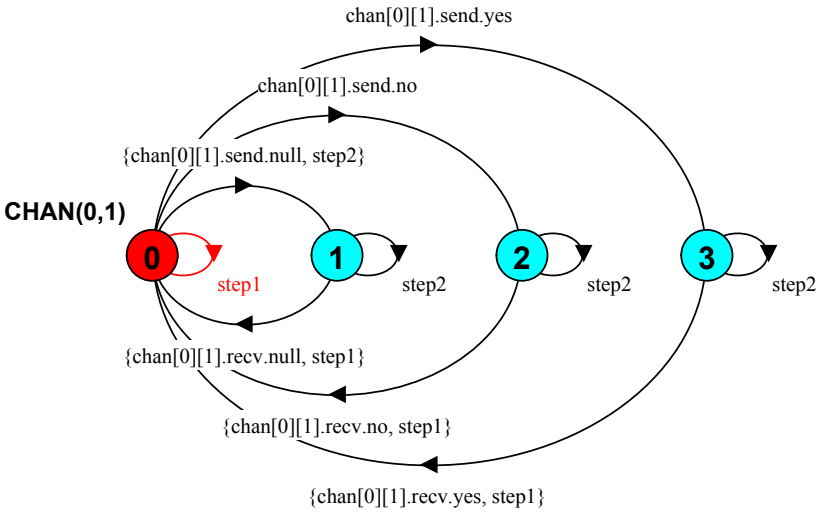
The process `CHAN`, shown in Figure 3, models a communication link between the process numbered `From` and the process numbered `To`. A message `m`, which takes a value from the set `Msg`, is sent to the channel by the action `chan[From][To].send[m:Msg]` and received from the channel by the action `chan[From][To].recv[m]`. From the LTS of Figure 3, it can be seen that `Msg` has been defined to be the set of actions `{null, yes, no}`. The channel can buffer at most one message. This is sufficient for a synchronous model since messages are always received in the same round that they are sent. To model the effect of a timeout, if a sending process crashes, `CHAN` delivers a `null` message in step 2 of a round if no message has been input to the channel in step 1. Similarly, if a message is not received in step 2, it is deleted from the channel when the next step 1 commences. The definition of `CHAN` uses the FSP choice operator "|" which specifies an alternative set of actions that a process may in engage in.

Using `CHAN` , we can now define a synchronous network to fully interconnect `N` processes numbered `0..N-1` by the following parallel composition:

```
||NETWORK = (forall[i:0..N-1][j:0..N-1]
               if (i!=j) then CHAN(i,j)
             ||CLOCK
             ).
```

For `N = 3`, this composition is exactly equivalent to the parallel composition:

```
||NETWORK = ( CHAN(0,1) || CHAN(0,2) || CHAN(1,0)
           || CHAN(1,2) || CHAN(2,0) || CHAN(2,1) || CLOCK).
```




```
CHAN(From=0, To =1)
   =(chan[From][To].send[m:Msg] -> CHAN[m]
    | step1 -> CHAN
    | step2 -> CHAN['null]
    ),
CHAN[m:Msg]
   =(chan[From][To].recv[m] -> CHAN
    | step1 -> CHAN
    | step2 -> CHAN[m]
    ).
```

Fig. 3. FSP for `CHAN(0,1)` and associated LTS

Process Failure

For reasons that we will discuss later, we initially do not permit channels to fail, however we permit processes to fail by *stopping* somewhere in the middle of their execution. This is sometimes known as *crash failure*. Process failure is signaled for a process `Id` by the action `fail[Id]`. This action may occur before or after step 1 or step 2 and in addition may occur in the middle of performing step 1 – meaning that a process may succeed in putting only a subset of the messages it is supposed to produce into message channels. Figure 4 lists a process `SEND_ALL` that sends a message from a process numbered `Id` to all processes numbered `Id+1..N-1`, thus for `Id=1` and `N=4`, the process sends a message to the processes numbered 2 and 3. We use this utility process in sections 4 & 5 in modeling two- and three-phase commit protocols. Note that after a failure, a process may only engage in the actions `step1` & `step2`. It may not send or receive messages – that is, it has *crashed*. In the following, we will refer to a process that has not failed as *correct* and a process that has failed as *crashed*.

```
SEND_ALL(From=0,M='null)
   = if ((N-1)>From) then TX[N-1-From] else (step2->END),
TX[n:0..N-1-From]
   = (when (n>0)  chan[From][From+1..N-1].send[M] -> TX[n-1]
     |when (n>0)  fail[From] -> ENDED
     |when (n==0) step2 -> END
     ),
ENDED
   = ({step1,step2}->ENDED).
```

Fig. 4. `SEND_ALL` process

3 Atomic Commitment

Atomic Commitment (AC) is important in the implementation of distributed database systems where it is important that a transaction takes effect in all participating sites or none of them. We formulate the problem as follows.

We assume we have a set of N processes numbered 0 to N-1 that participate in the execution of a distributed transaction. After processing its own part of the transaction (some actions), each process has its own initial "opinion" about whether the transaction should be committed (its results become permanent in the distributed database state) or aborted (results discarded). This opinion (or "vote") of the process is represented by the values *yes* and *no* respectively. A process will generally favor committing the transaction if all its local computations regarding that transaction have been completed successfully; otherwise will it favor abort. The processes communicate amongst each other to decide on the final outcome, commit or abort. If possible, the outcome should be commit. Again, we use the values *yes* and *no* to represent commit and abort respectively.

Correctness Conditions
Any algorithm that provides a solution to the AC problem must satisfy the following correctness conditions or properties. In other words, these are the specifications of any such algorithm.

- **Agreement**: No two processes (whether correct or crashed) should decide on different values.
- **Validity**:
  1. If *any* process votes *no*, then *no* is the only possible decision value.
  2. If *all* processes vote *yes*, and there are no failures, then *yes* is the only possible decision value.
- **Termination**: *(there are two types)*
  - **Strong Termination**: All correct processes eventually decide.
  - **Weak Termination**: If there are no failures, then all processes eventually decide.

Note that the agreement and validity conditions are *safety* properties and that the termination property is a *liveness* property. Informally, a safety property asserts that something "bad" does not happen during system execution and a liveness property asserts that eventually something "good" happens. The two types of termination give rise to two different AC problems termed the Strong AC problem (strong termination) and the Weak AC problem (weak termination). An atomic commitment algorithm is said to be non-blocking if it permits transaction termination to proceed at correct participants despite failures of some participants or failures of links. Such an algorithm must obviously satisfy the strong termination property above. Surprisingly, it has been proved that there is no non-blocking algorithm that solves AC in the presence of persistent link failures [5]. Consequently, as mentioned in the previous section, we have chosen not to model link failures initially. Although Weak AC can be solved for both process and link failure, Strong AC can be solved only for process failures. We examine the effect of link failures on a Strong AC algorithm in section 5.

Fluent Linear Temporal Logic (FLTL)

We use FLTL as defined in [3] to encode the correctness properties for AC so that they can be directly checked against models of AC protocols. Fluents (time-varying properties of the world) are true at particular time-points if they have been initiated by an action occurrence at some earlier time-point, and not terminated by another action occurrence in the meantime. Similarly, a fluent is false at a particular time-point if it has not been previously terminated and not initiated in the meantime [6]. We define a fluent by a pair of sets, a set of initiating actions and a set of terminating actions. To allow us to specify fluents relevant to the AC problem, we identify the following actions to record votes and decisions made by models:

```
range ID  = 0..N-1
vote[i:ID].yes          //  process i votes yes
vote[i:ID].no           //  process i votes no
decide[i:ID].yes        //  process i decides yes
decide[i:ID].no         //  process i decides no
fail[i:ID]              //  process i fails
```

Given these actions, we can declare the follow fluents that record the abstract state of models. The occurrence of any action in the first set of actions defining affluent makes the fluent true and the occurrence of any action in the second set makes the fluent false. In all the declarations, the second set consists of the single action `never`, which as discussed in the previous section can never occur. Consequently, when the fluents declared below become true, they remain true. This modeling "trick" means that we can model check single instances of protocol executions rather than continuous repetitive executions.

```
fluent VOTE[i:ID][v:{yes,no}] = <vote[i][v],never>
fluent DECIDED[i:ID]          = <decide[i].{yes,no},never>
fluent COMMIT [i:ID]          = <decide[i].yes, never>
fluent ABORT  [i:ID]          = <decide[i].no, never>
```

The declaration `ABORT[i:ID]` above is simply declaring a set of fluents, `ABORT[0], ABORT[1], ABORT[2]` and `ABORT[3].`

The concrete syntax for FLTL formulas used in the LTSA follows as closely as possible the LTL syntax used in SPIN [7]. The following operators are defined:

Unary operators (*unop*):		Binary operators(*binop*):	
`[]`	always (**G**)	`U`	strong until
`<>`	eventually (**F**)	`&&`	logical AND
`X`	next time	`||`	logical OR
`!`	logical negation	`->`	implication
		`<->`	equivalence

An FLTL formula Φ := `True` | `False` | *prop* | (Φ) | *unop* Φ | Φ *binop* Φ, where *prop* is a fluent, action or set of actions. In the following, we use *exists* and *forall replicators* where:

exists[*i*:1..*N*] Φ[i] ≡ Φ[1] || …|| Φ[*N*]

and

forall[*i*:1..*N*] Φ[i] ≡ Φ[1] && …&& Φ[*N*].

In addition, a fluent proposition of the form *FL*[1..*N*] ≡ *FL*[1] || …|| *FL*[*N*]. We are now in a position to encode the correctness conditions for the AC problems as follows:

- **Agreement**: No two processes (whether correct or crashed) should decide on different values.

```
assert AGREEMENT =  []!(COMMIT[ID] && ABORT[ID])
```

  This formula is read as: it is always not (never) the case that one of the processes 0..N-1 can have committed and also one of these processes can have aborted i.e. have decided on different values.

- **Validity**:
  1. If *any* process votes *no*, then *no* is the only possible decision value.

```
assert VALID_1 = [](VOTE[ID]['no] -> !COMMIT[ID])
```

     The reading of this formula is that it is always the case that if one of the processes 0..N..1 has voted no then it can not be the case that one of these processes has committed i.e. decided other than no.

  2. If *all* processes vote *yes*, and there are no failures, then *yes* is the only possible decision value.

```
assert VALID_2 = [](forall[i:ID]
                      (VOTE[i]['yes] && !CRASHED[i])
                       -> !ABORT[ID])
```

     This reads that it is always the case that if all processes vote yes and are not crashed (i.e. correct) then it cannot be the case that one of the processes is

aborted (i.e. decides other than yes). We use not aborted rather than directly using the committed fluent since commitment is not true initially and may never be true due to failure.

- **Termination**:
  - **Strong Termination**: All correct processes eventually decide.

```
assert
STRONGTERM = <>(forall[i:ID](!CRASHED[i]-> DECIDED[i]))
```

This reads: it is eventually the case the if a process has not crashed then it will decide.

  - **Weak Termination**: If there are no failures, then all processes eventually decide.

```
assert
WEAKTERM = ([]forall[i:ID] !CRASHED[i]
                -> <>forall[i:ID] DECIDED[i])
```

This reads: if its always the case that all processes do not crash then eventually all processes reach the decided state.

In the next section we model the two-phase commit protocol and examine which of the above properties it satisfies.

4 Two-phase commit

The most used practical algorithm for atomic commitment is the two-phase commit. It consists of two rounds and assumes a distinguished process usually termed the coordinator, which we number 0. The algorithm description below is adapted from [4] following [8].

Round 1: All processes except for the coordinator send their vote$\in$\{*yes*,*no*\} to the coordinator, and any process whose vote is *no* decides *no*. The coordinator collects all these votes together with its own initial vote. If all the votes are *yes* then the coordinator decides *yes*. If there is a *no* vote or a missing vote – because a vote was not received from the sending process – then the coordinator decides *no*.

Round 2: The coordinator broadcasts its decision to all the other processes. Any process, other than the coordinator, that receives a message at round 2 and has not already decided at round 1 decides on the value it receives in that message.

The communication pattern for a failure free run of two-phase commit is depicted in Figure 5. Note that in practical implementations of the two-phase commit, a extra round is added at the beginning in which the coordinator requests votes from the other

participating processes. In the interests of simplicity, we chose not to model this additional round since its does not alter the properties of the algorithm.

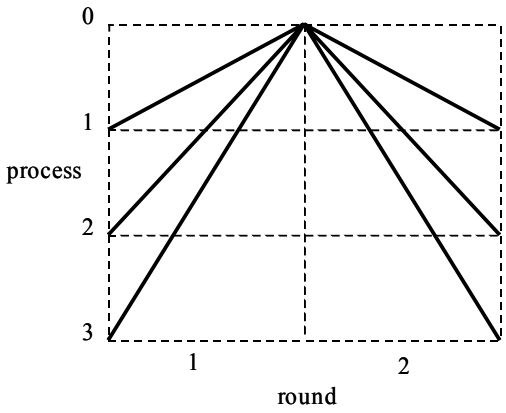


Fig. 5. Communication pattern for Two-phase Commit

We model the behavior of the two-phase commit protocol using two processes, a `COORDINATOR` numbered 0 and `PARTICIPANT` processes numbered 1..N-1. Figure 6 lists the FSP definition of the `PARTICIPANT` process together with some declarations and the auxiliary process `DECIDE`.

```
const    N        = 4
set      Msg      = {yes,no, null}

DECIDE(Id=0, D='null)
 = if (D=='yes || D=='no) then (decide[Id][D]->END) else END.

PARTICIPANT(Id=1) = ROUND1,
ROUND1
  = (vote[Id][v:{yes,no}]->step1->SEND[v]),
SEND[v:Msg]
  = (chan[Id][0].send[v] -> step2 ->
       if (v=='no) then DECIDE(Id,v);ENDED else ROUND;ROUND2
    |fail[Id] -> ENDED),
ROUND2
  = (chan[0][Id].recv[m:Msg] -> DECIDE(Id,m);ENDED
    |fail[Id] -> ENDED
    ),
ENDED
  = ({step1,step2}->ENDED)
    +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg]}.
```

Fig. 6. FSP definition of `PARTICIPANT` process.

Note that the PARTICIPANT process decides immediately in round 1 if it votes no otherwise the decision is based on receiving a message from the coordinator, via chan[0][Id] in round 2. Figure 7, below lists the COORDINATOR process.

```
COORDINATOR(Id=0)
  = (vote[Id][v:{yes,no}]->ROUND;ROUND1[v]),
ROUND1[v:{yes,no}]
  = (chan[Id+1..N-1][Id].recv[m:Msg]
      -> if (v =='no) then ROUND1['no]
         else if (m == 'no || m == 'null) then ROUND1['no]
         else if (m == 'yes && v == 'yes) then ROUND1['yes]
     |step1 -> ROUND2[v]
     ),
ROUND2[v:{yes,no}]
   = DECIDE(Id,v);SEND_ALL(Id,v);ENDED,
ENDED
   = ({step1,step2}->ENDED)
      +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg]}.
```

Fig. 7. FSP definition of COORDINATOR process.

In round 1, step 2, the coordinator receives messages from all participant processes. The end of this receiving phase is signaled by round 2, step 1. If at this point all *yes* messages have been received and the coordinator decision was *yes* then it decides *yes* otherwise, the decision is *no*. The actions signaling step 1 and step 2 are lower priority than receive actions so all messages that are sitting in channels are received before round 2. The actions are marked as low priority as shown below in the composition that combines the coordinator and participant processes with the network to form the system to be model-checked:

```
||SYS = (  COORDINATOR(0)
        || forall[i:1..N-1] PARTICIPANT(i)
        || NETWORK
        || FCONSTRAINT(2)
        )>>{step1,step2}.
```

The process FCONSTRAINT is used to constrain the number of failures that are allowed to occur in the model. With a parameter of 2 as above, a maximum of 2 of the N processes may fail. The process is defined below:

```
FCONSTRAINT(F=0) = FAIL[0],
FAIL[f:0..F]     = (when (f<F) fail[0..N-1] -> FAIL[f+1]
                   )+{fail[0..N-1]}.
```

Model-checking

We are now in a position to check that the model defined by SYS satisfies the properties for AC algorithms specified in section 3. As is well known, two-phase commit satisfied all the properties with the exception of Strong Termination. Two-phase commit solves only the Weak AC problem. When model-checking

STRONGTERM with processes constrained to vote yes and a single failure permitted, the LTSA tool produces the following counter-example:

```
Violation of LTL property: @STRONGTERM
Trace to terminal set of states:
     vote.0.yes
     vote.1.yes
     vote.2.yes
     vote.3.yes
     step1
     chan.1.0.send.yes
     chan.2.0.send.yes
     chan.3.0.send.yes
     step2
     chan.1.0.recv.yes
     chan.2.0.recv.yes
     chan.3.0.recv.yes
     step1
     decide.0.yes              DECIDED.0
     fail.0                    CRASHED.0 && DECIDED.0
     step2                     CRASHED.0 && DECIDED.0
     chan.0.1.recv.null        CRASHED.0 && DECIDED.0
     chan.0.2.recv.null        CRASHED.0 && DECIDED.0
     chan.0.3.recv.null        CRASHED.0 && DECIDED.0
     step1                     CRASHED.0 && DECIDED.0
Cycle in terminal set:
     step2                     CRASHED.0 && DECIDED.0
     step1                     CRASHED.0 && DECIDED.0
LTL Property Check in: 270ms
```

The counter-example trace gives the action on the left and lists all the fluents that hold after that action has occurred on the right. The trace shows the typical situation in which the two-phase commit algorithm blocks – that is, the coordinator crashes before sending the decision in round 2 and as a result, processes that voted *yes* in round 1 cannot decide – in this case processes 1,2 & 3. Even if the participants were to elect a new coordinator and try to terminate, they cannot, since the original coordinator could have voted *no* initially and it would then have decided *no* rather than *yes* as above. Participants cannot distinguish these two situations and as a result are blocked until the coordinator recovers.

Witness executions

In addition to generating counter-examples to show why a property has been violated, we can use the model checker to generate a witness execution that satisfies a property. To do this, we simply model-check the negation of a property and the counter-example is now a witness to the property. The trace below is a witness execution to the agreement property. It is a violation of the property:

```
assert WITNESS_AGREEMENT = !AGREEMENT
```

The trace shows the scenario in which all the participants vote *no* and the coordinator makes a decision before failing so all processes decide abort.

```
Violation of LTL property: @WITNESS_AGREEMENT
Trace to terminal set of states:
     vote.0.yes
     vote.1.no
     vote.2.no
     vote.3.no
     step1
     chan.1.0.send.no
     chan.2.0.send.no
     chan.3.0.send.no
     step2
     chan.1.0.recv.no
     chan.2.0.recv.no
     chan.3.0.recv.no
     decide.1.no                                          ABORT.1
     decide.2.no                                ABORT.1 && ABORT.2
     decide.3.no                    ABORT.1 && ABORT.2 && ABORT.3
     step1                          ABORT.1 && ABORT.2 && ABORT.3
     decide.0.no    ABORT.0 && ABORT.1 && ABORT.2 && ABORT.3
     fail.0         ABORT.0 && ABORT.1 && ABORT.2 && ABORT.3
```

The LTSA tool always produces the shortest trace possible as a counter-example, which is why it chose the abort scenario above. However, we can produce an example execution that commits using the LTSA Animator as shown in Figure 8.

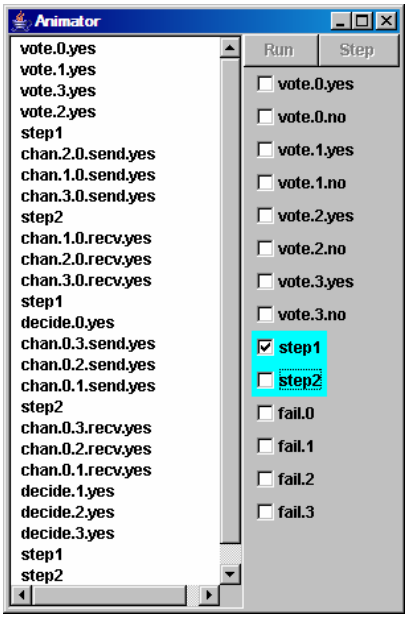


Fig. 8. Trace showing commit for Two-phase AC algorithm

5 Three-phase commit

The three-phase commit is an extension of the two-phase commit that guarantees strong termination in the presence of process failures. The difference is that the coordinator does not decide *yes* unless every process that has not failed is *ready* to decide *yes*. This requires an extra round. The first three rounds of the three-phase commit are as follows:

Round 1: All processes except for the coordinator send their vote$\in$ {*yes*,*no*} to the coordinator, and any process whose vote is *no* decides *no*. The coordinator collects all these votes together with its own initial vote. If all the votes are *yes* then the coordinator becomes *ready*. If there is a *no* vote or a missing vote – because a vote was not received from the sending process – then the coordinator decides *no*.

Round 2: The coordinator broadcasts either *ready* or *no* to all the other processes. Any process that receives *no* decides *no*. Any process that receives *ready* becomes *ready*. The coordinator decides *yes* if it has not previously decided *no*.

Round 3: If the coordinator decided yes, it broadcasts *yes* to all the other processes. Any process that receives *yes* decides *yes*.

The communication pattern for a failure free execution of three-phase commit is shown in Figure 9.

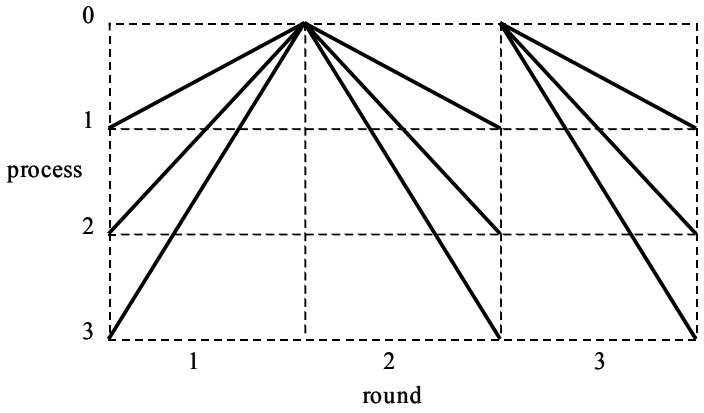


Fig. 9. Communication pattern for successful three-phase commit.

As it stands, the protocol will still block if the coordinator fails. However, the blocking situation in the two-phase protocol is avoided by the additional *ready* state – the coordinator cannot have committed if it fails during round 2. A termination protocol that starts at round 4 is outlined below:

Round 4: All (not yet failed) participant processes numbered 2..N-1 send their current decision status: *yes*, *no*, *ready* or *null* (uncertain) to process 1. Process 1 now acts as the coordinator. If process 1 has previously decided *no*, or receives *no* from some other process, it decides *no*. If process 1 has previously decided *yes*, or receives *yes* form some other process, it decides *yes*. If process 1 was *ready* or received *ready* from some other process, it becomes *ready* otherwise if it is uncertain and receives uncertain from all other processes, it decides *no*.

Round 5: This proceeds as round 2 except that process 1 is the coordinator. The only difference is that process 1 may send a *yes* decision in this round if it previously decided *yes*.

Round 6: This proceeds exactly as round 3.

After round 6, the protocol continues with three similar rounds coordinated by each of the processes 2..N-1. Each group of three rounds is termed an epoch with process *i* coordinating epoch *i*. Process N-1 decides in the second round of epoch N-1 and since it does not communicate with any other process, the last round of this epoch is superfluous. Figure 10 depicts the communication pattern for three-phase commit with termination in which process 0 fails before sending ready to all processes in round 2 and process 1 fails similarly in round 5.

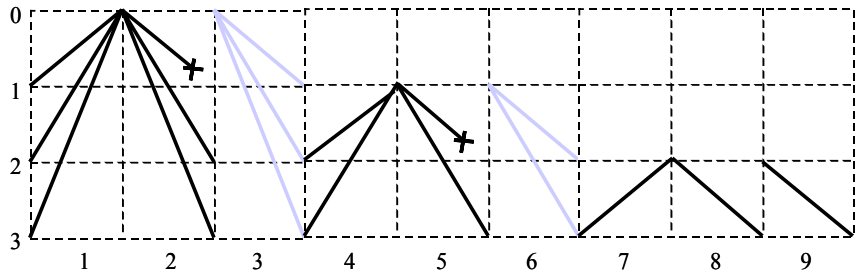


Fig. 10. Communication pattern for three-phase commit with failures.

In modeling the processes for three-phase commit, the coordinator for epoch 0, process 0 is the simplest since it only participates in the first three rounds. The FSP model is listed in Figure 12. The model of coordinator rounds 2 & 3 has been placed in the utility process `ROUND2_3`, since it is also used by `PARTCIPANT` processes when they act as coordinators – Figure 11. The three-phase commit model reuses the network models of section 2. The message set for three-phase commit is as defined below:

```
set Msg = {yes, no, ready, null}
```

```
// partcipant process, acts as coordinator for epoch Id
PARTICIPANT(Id=1) = ROUND1,
ROUND1
  = (vote[Id][v:{yes,no}]->step1->SEND[v]),
SEND[v:Msg]                                  // epoch 0
  = (chan[Id][0].send[v] -> step2
      -> if (v=='no)
         then DECIDE(Id,v);ROUND;ROUND;TERMINATE[v][0]
         else ROUND;PROUND2['null][0]
    |fail[Id] -> ENDED
    ),
SEND[v:Msg][epoch:1..N-1]
  = (chan[Id][epoch].send[v] -> step2
      -> if (v=='no || v=='yes)
         then DECIDE(Id,v);ROUND;ROUND;TERMINATE[v][epoch]
         else ROUND;PROUND2[v][epoch]
    |fail[Id]->ENDED
    ),
PROUND2[d:Msg][epoch:0..N-1]
  = (chan[epoch][Id].recv[m:Msg]
      -> if (m=='no || m=='yes)
         then DECIDE(Id,m);ROUND;TERMINATE[m][epoch]
         else if (m=='null)
         then ROUND;PROUND3[d][epoch]
         else ROUND;PROUND3[m][epoch]
    ),
PROUND3[d:Msg][epoch:0..N-1]
  = (chan[epoch][Id].recv[m:Msg]
      -> if (m!='null)
         then DECIDE(Id,m);TERMINATE[m][epoch]
         else DECIDE(Id,m);TERMINATE[d][epoch]
     ),
TERMINATE[d:Msg][epoch:0..N-1]
  =  if (epoch==N-1) then ENDED
     else if (Id==epoch+1)then ROUND;COORD[d]
     else  (step1->SEND[d][epoch+1]),
COORD[d:Msg]
  = (when (Id+1 < N) chan[Id+1..N-1][Id].recv[m:Msg]
      -> if (d =='yes || d=='no) then COORD[d]
         else if (m=='no || m =='yes) then DECIDE(Id,m);COORD[m]
         else if (m=='ready )then COORD[m]
         else COORD[d]
    |step1
      -> if (d=='null)
         then DECIDE(Id,'no);ROUND2_3(Id,'no);ENDED
         else ROUND2_3(Id,d);ENDED
    ),
ENDED
  = ({step1,step2}->ENDED)
    +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg],fail[Id]}.
```

Fig. 11. FSP definition of three-phase commit `PARTCIPANT` process.

```
// utility process for coordinator rounds epoch*3+2 and +3
ROUND2_3(Id=0,M='null)
  = SEND_ALL(Id,M);ROUND2,
ROUND2
  = if (M=='ready) then DECIDE(Id,'yes);ROUND3 else END,
ROUND3
  = (step1 -> SEND_ALL(Id,'yes);END).

// coordinator process for epoch 0
COORDINATOR(Id=0)
  = (vote[Id][v:{yes,no}] -> ROUND;ROUND1[v]),
ROUND1[v:{yes,no}]
  = (when (Id+1 < N) chan[Id+1..N-1][Id].recv[m:Msg]
      -> if (v =='no) then ROUND1['no]
         else if (m == 'no || m == 'null) then ROUND1['no]
         else if (m == 'yes && v == 'yes) then ROUND1['yes]
     |step1
      -> if (v=='yes)
         then ROUND2_3(Id,'ready);ENDED
         else DECIDE(Id,'no);ROUND2_3(Id,'no);ENDED
     ),
ENDED
  = ({step1,step2}->ENDED)
    +{chan[ID][Id].recv[Msg],chan[Id][ID].send[Msg],fail[Id]}.
```

Fig. 12. FSP definition of three-phase commit `COORDINATOR` process.

The complete model of the three-phase commit algorithm is formed exactly as for the two-phase model by composing the coordinator and participant processes with the network:

```
||SYS = (  COORDINATOR(0)
        || forall[i:1..N-1] PARTICIPANT(i)
        || NETWORK
        || FCONSTRAINT(3)
        )>>{step1,step2}.
```

The three-phase commit algorithm satisfies all the conditions we specified in section 3, including strong termination. The trace of Figure 13 is a witness to this property. It depicts the same situation that violated the strong termination property in the two-phase commit model. The coordinator (process 0) fails at the beginning of round 1 and thus participants do not receive the *ready* message. However, because no process receives a ready message, it is safe for process 1 to take over as coordinator and decide no since process 0 cannot have committed.

The algorithm as we have modeled it always takes 3*N-1 rounds even if there are no failures. In a practical protocol, it is easy to insert an extra round to detect that all processes have decided and then discontinue the termination protocol.

```
Trace to property violation in WITNESS_STRONGTERM:
      vote.0.yes
      vote.1.yes
      vote.2.yes
      vote.3.yes
      step1                                              // round 1
      chan.1.0.send.yes
      chan.2.0.send.yes
      chan.3.0.send.yes
      step2
      chan.1.0.recv.yes
      chan.2.0.recv.yes
      chan.3.0.recv.yes
      step1                                              // round 2
      fail.0              CRASHED.0
      step2               CRASHED.0
      chan.0.1.recv.null  CRASHED.0
      chan.0.2.recv.null  CRASHED.0
      chan.0.3.recv.null  CRASHED.0
      step1               CRASHED.0                      // round 3
      step2               CRASHED.0
      chan.0.1.recv.null  CRASHED.0
      chan.0.2.recv.null  CRASHED.0
      chan.0.3.recv.null  CRASHED.0
      step1               CRASHED.0                      // round 4
      chan.2.1.send.null  CRASHED.0
      chan.3.1.send.null  CRASHED.0
      step2               CRASHED.0
      chan.2.1.recv.null  CRASHED.0
      chan.3.1.recv.null  CRASHED.0
      step1               CRASHED.0                      // round 5
      decide.1.no         CRASHED.0 && DECIDED.1
      chan.1.2.send.no    CRASHED.0 && DECIDED.1
      chan.1.3.send.no    CRASHED.0 && DECIDED.1
      step2               CRASHED.0 && DECIDED.1
      chan.1.2.recv.no    CRASHED.0 && DECIDED.1
      decide.2.no         CRASHED.0 && DECIDED.1 && DECIDED.2
      chan.1.3.recv.no    CRASHED.0 && DECIDED.1 && DECIDED.2
      decide.3.no         CRASHED.0 && DECIDED.1 && DECIDED.2
                                                     && DECIDED.3
  Analysed in: 60ms
```

Fig. 13. Witness execution for property `STRONGTERM`

Link Failure

So far, because of the impossibility result mention in section 3, we have ignored link failure and focused on process failure. However, it is instructive to examine the behavior of the three-phase commit algorithm in the presence of link failures. To do this, we adapt the channel model shown in figure 3 as shown in figure 14.

```
CHAN(From=0, To =0)
   =(chan[From][To].send[m:Msg] -> CHAN[m]
    | step1 -> CHAN
    | step2 -> CHAN['null]
    ),
CHAN[m:Msg]
   =(chan[From][To].recv[m] -> CHAN
    | step1 -> CHAN
    | step2 -> CHAN[m]
    | step2 -> linkfail[From][To] -> CHAN['null]
    ).
```

Fig. 14. FSP model of `CHAN` which can fail.

The channel now has a non-deterministic choice at step 2. It can either fail changing the value of the message stored in the channel to `null` or leave the message intact.

Figure 15 depicts a counter example that violates the `AGREEMENT` property if we permit link failures. In this counter-example, all processes send *yes* and successfully inform the coordinator. However, all subsequent messages from the coordinator (process 0) are lost due to link failure. After the coordinator sends *ready* messages it decides *yes*. However, in round 4, *no* process has received *ready* or *yes* and so they send *null* to the new coordinator process 1. Consequently, in round 5, process 1 decides *no* thus violating the `AGREEMENT` property.

In practice, link protocols overcome transient failures using retransmission and networks can overcome permanent link failure using rerouting. However, the situation depicted in the counter-example can still occur if a network partition occurs – in this case isolating process 0 from processes 1,2 & 3. This is the reason that in practice two-phase commit is much more widely used than three-phase commit. The two-phase algorithm may block as a result of link or process failure, however, it does not violate the agreement or validity safety properties whereas if network partition can occur, three-phase commit violates these safety properties. We can of course check that the model of the two-phase commit algorithm does not violate `AGREEMENT` using the channel model of figure14.

```
Trace to property violation in AGREEMENT:
    vote.0.yes
    vote.1.yes
    vote.2.yes
    vote.3.yes
    step1
    chan.1.0.send.yes
    chan.2.0.send.yes
    chan.3.0.send.yes
    step2
    chan.1.0.recv.yes
    chan.2.0.recv.yes
    chan.3.0.recv.yes
    step1
    chan.0.1.send.ready
    chan.0.2.send.ready
    chan.0.3.send.ready
    step2
    decide.0.yes              COMMIT.0
    linkfail.0.1              COMMIT.0
    chan.0.1.recv.null        COMMIT.0
    linkfail.0.2              COMMIT.0
    chan.0.2.recv.null        COMMIT.0
    linkfail.0.3              COMMIT.0
    chan.0.3.recv.null        COMMIT.0
    step1                     COMMIT.0
    chan.0.1.send.yes         COMMIT.0
    chan.0.2.send.yes         COMMIT.0
    chan.0.3.send.yes         COMMIT.0
    step2                     COMMIT.0
    linkfail.0.1              COMMIT.0
    chan.0.1.recv.null        COMMIT.0
    linkfail.0.2              COMMIT.0
    chan.0.2.recv.null        COMMIT.0
    linkfail.0.3              COMMIT.0
    chan.0.3.recv.null        COMMIT.0
    step1                     COMMIT.0
    chan.2.1.send.null        COMMIT.0
    chan.3.1.send.null        COMMIT.0
    step2                     COMMIT.0
    chan.2.1.recv.null        COMMIT.0
    chan.3.1.recv.null        COMMIT.0
    step1                     COMMIT.0
    decide.1.no               COMMIT.0 && ABORT.1
Analysed in: 711ms
```

Fig. 15. Counter-example violating `AGREEMENT` property

6 Discussion & Conclusion

The paper has illustrated an approach to modeling and mechanically verifying synchronous distributed algorithms. The models have the advantage over less formal descriptions that they permit tool supported interactive exploration of the operation of an algorithm. In addition, model checking provides counter-example traces to explain property violation and witness traces that provide sample executions that satisfy properties. Models are considerably faster to produce than implementations or even simulations and in addition are more amenable to exploration as we have described in the foregoing.

In modeling synchronous algorithms, we have avoided explicit modeling of time and timeouts by arranging that computation proceed in rounds (following [4]) and modeling the effects of timeouts by null messages. We could model in more detail by explicitly including time. For example, an approach to discrete time models is described in [1]. Chkiaev, van der Stok and Hooman [9] have taken the approach of explicitly including time in mechanically verifying, using automated theorem proving, the non-blocking Atomic Commitment Protocol from [10]. This has the advantage of models that are closer to implementations but the disadvantage of models that have larger state spaces.

Moving away from the strictly synchronous approach leads to the need for more complex channel models that can buffer multiple rather than single messages. Again this leads to larger state spaces that can sometimes but not always be dealt with by the use of Partial Order reduction techniques (see [11]) for further details. Mechanical verification of complex asynchronous and partially synchronous algorithms is an active research area.

In modeling Atomic Commitment protocols, we have assumed that crashed processes remain crashed forever. However, clearly processes recover and there are recovery protocols that permit processes to discover the outcome of a transaction by a combination of stable storage logs and querying other participants (see [8]). These recovery protocols can be modeled using the techniques we have outlined in the paper.

We have chosen to model Atomic Commitment protocols since they have considerable practical application and a thorough understanding of their operation and limitations is desirable for distributed application designers. The properties of the protocols are of course well known, so the primary contribution of the models is pedagogic. However, the approach used is generally applicable and is being applied to new Web Services applications [12].

Finally, the reader will gain considerably more understanding of the models we have described in this paper by exploring them interactively using the LTSA tool. Both the tool and algorithm models are available from the author's website http://www.doc.ic.ac.uk/~jnm.

References

[1] J. Magee and J. Kramer, *Concurrency - State Models & Java Programs*. Chichester: John Wiley & Sons, 1999.

[2] J. Magee, "Behavioral Analysis of Software Architectures Using LTSA," presented at ICSE' 99. Proceedings of the 1999 International Conference on Software Engineering, Los Angeles, 1999.

[3] D. Giannakopoulou and J. Magee, "Fluent Model Checking for Event-Based Systems," presented at 4th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), Helsinki, 2003.

[4] N. A. Lynch, *Distributed Algorithms*: Morgan Kaufmann Publishers, Inc., 1996.

[5] J. N. Gray, "Notes on Database Operating Systems," in *Operating Systems: An Advanced Course*, vol. 60, *LNCS*: Springer-Verlag, 1978.

[6] R. Miller and M. Shanahan, "The Event Calculus in Classical Logic - Alternative Axiomatisations," *Linkoping Electronic Articles in Computer and Information Science*, vol. 4, pp. 1-27, 1999.

[7] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, 97.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*: Addison-Wesley Publishing Comp., 1987.

[9] D. Chkliaev, P. v. d. Stok, and J. Hooman, "Mechanical Verification of a Non-Blocking Atomic Commitment Protocol," presented at ICDCS Workshop on Distributed System Validation and Verification (DSVV'2000), 2000.

[10] O. Babaoglu and S. Toueg, "Non-blocking Atomic Commitment," in *Distributed Systems*, S. Mullender, Ed.: Addison-Wesley Publishing Comp, 1993, pp. 147-168.

[11] D. Peled, "Combining Partial Order Reductions with On-the-Fly Model Checking," presented at 6th International Conference on Computer Aided Verification (CAV'94), Stanford, California, 94.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based Verification of Web Service Compositions," presented at 18th IEEE International Conference on Automated Software Engineering (ASE 2003),, Montreal, 2003.