# CSc72010

Proofs for asynchronous systems

## Examples

### *Leader*

```
type Status = enumeration of leader, unknown

automaton Process (pos : Int, prev : Int, next : Int,  rank: Int)
  signature
    input RECEIVE(m: Int , const prev, const pos : Int)
    output SEND (m: Int , const pos , const next : Int)
    output leader(const rank)

  states
%    pending : Mset [Int] := {rank},
    pending : Int := rank,
    status : Status := unknown,
    ready : Bool := true

  transitions
     input RECEIVE(m, j, i) where m > pending
       eff pending := m; ready := true

     input RECEIVE(m, j, i) where m < pending

     input RECEIVE(m, j, i) where m = rank
       eff status := leader

     output SEND (m, i, j)
       pre status ~= leader /\ m = pending /\ ready
       eff ready := false

     output leader(rank)
       pre status = leader
```

Representation of state: <4, T, unknown>, <7,F,leader>
That is, <*pending*, *ready*, *status*>

### *Channel*

```
automaton Channel(i:Int , j:Int)
  signature
    input SEND (m: Int , const i : Int, const j : Int)
    output RECEIVE(m : Int, const i : Int ,const j : Int)

  states
    queue : Seq[Int] := {}

  transitions
    input SEND (m, i, j)
      eff queue := queue |- m

    output RECEIVE(m,i,j)
      pre head(queue) = m
      eff queue := tail(queue)
```

Representaton of state: {}, {4}, {4,7}

That is, *queue*

### *Composition*

```
automaton LCR
  components

   P1: Process(1, 4, 2, 7);
   P2: Process(2, 1, 3, 3);
   P3: Process(3, 2, 4, 2);
   P4: Process(4, 3, 1, 5);
   C1: Channel(1, 2);
   C2: Channel(2, 3);
   C3: Channel(3, 4);
   C4: Channel(4, 1)
```

# Proof Techniques

## *Types of properties*

### Invariants

Properties of states that are true in all reachable states. Usually proof by induction. Step granularity is finer than rounds, so proofs are harder.

Here are invariants for leader:
p.pending >= p.rank for all processes p
Another:
$c_{ij}$.queue[k] >= $p_i$.rank for all i,j,k

Exercise – think about the inductive proof of these

How to state properties (section 14.1.2):

1) As an automaton – example, the Channel example that we are using can be used to specify the traces allowed by a reliable FIFO channel (so it is both a specification of such a channel and an example)
2) As axioms:
   There is a function *cause* mapping each receive to a send and satisfying
   a. For every receive event $\pi$, $\pi$ and cause($\pi$) contain the same message argument.
   b. cause is surjective (onto).
   c. cause is injective (one-to-one).
   d. cause preserves order, that is, there do not exist receive events $\pi_1$ and $\pi_2$ with $\pi_1$ preceding $\pi_2$ in $\beta$ and cause($\pi_2$) preceding cause($\pi_1$) in $\beta$.

Interpretation:

a. Every message received was first sent
b. Every send corresponds to a receive (reliable)
c. Different receives correspond to different sends (no duplicates).
d. This is the FIFO property.

## Trace properties

Any property of the external behavior sequences of automata. Formally, a trace property P is an external signature together with a set of sequences of actions in the signature, i.e., the allowable sequences:
<sig(P), traces(P)>

An automaton A satisfies a trace property P if it has the same external signature and traces(A) $\subseteq$ traces(P) or maybe fairtraces(A) $\subseteq$ traces(P).

All of the problems we will consider in asynchronous systems can be formulated as trace properties. Also, we'll usually be concerned with fairness, i.e., we'll be making statements about trace properties that hold for fair traces.

A trace property for leader:
In every trace, there is at most one leader(x) action:
Define the property by:

sig(P) = { SEND(m,i,j), RECEIVE(m,i,j), leader(r) | m $\in$ M, i,j $\in$ א, r $\in$ א }

traces(P) = { $\tau_1, \tau_2, \ldots$ | $\forall i(\tau_i \in \text{sig(P)}) \wedge \neg\exists i, j, r, s\ (\tau_i = \text{leader(r)} \wedge \tau_j = \text{leader(s)})$ }

## Safety properties

A safety property says that bad things don't happen.
In other words, traces(P) is non-empty, prefix-closed, limit-closed:

1. $\lambda$ is in P
2. if $\alpha$ is in P, all prefixes of $\alpha$ are in P
3. if all prefixes of $\alpha$ are in P, then $\alpha$ is in P

Examples:
At most one process declares itself as leader. (The traces are those including only one leader action.)
DHCP: No two processes get the same IP address
ARP: No two processes respond to the same ARP message
HTTP: No response to a GET is returned before the response to an earlier GET.
RIP (or any other routing protocol): No loops in routing tables. The path taken by any packet is the shortest path.
Mutual exclusion: No two simultaneous grants of resources

How to prove a safety property:

1. Relate it to a state invariant
2. Prove the state invariant

## Liveness properties

A liveness property says that good things do happen. traces(P) includes only those traces with the good things in them:
Every finite sequence over sig(P) has an extenson in traces(P).

Examples
A leader is eventually elected

traces(P) = { $\tau_1, \tau_2, \ldots \mid \forall i(\tau_i \in sig(P)) \wedge \exists i, r\ (\tau_i = leader(r))$ }

A process eventually gets an IP address
A browser eventually gets a page

Note: you will have to assum fairness to get these properties.
What we can say is that "Every *trace* of a correct leader election algorithm contains at most one leader action" and "Every *fair trace* of a correct leader election contains a leader action."

## Surprising fact

Every trace property can be expressed as a combination of a liveness property and a safety property. (e.g., a trace of a correct leader election algorithm contains exactly one leader action)

If <sig(P), traces(P)> is a trace property then
there exists a safety property <sig(P), traces(S)> and
there exists a liveness property <sig(P), traces(L)>
such that traces(P) = traces(S) $\cap$ traces(L).

So a good way to organize a specification is as a sequence of safety properties followed by a sequence of liveness properties.

Proof of theorem:
Let traces(S) = the prefix and limit closure of traces(P), making it a safety property, and obviously traces(S) contains traces(P)
Let traces(L) = traces(P) $\cup$ {$\beta \mid \beta$ is a finite sequence and no extension of $\beta$ is in traces(P) }.

We show that L is a liveness property:
Take any finite sequence $\beta$ of actions in sig(P). Either some extension of $\beta$ is in traces(P) or not. If it is, then it is "live"; if not, $\beta$ ( an extension of itself) is in traces(L). Hence L is a liveness property.

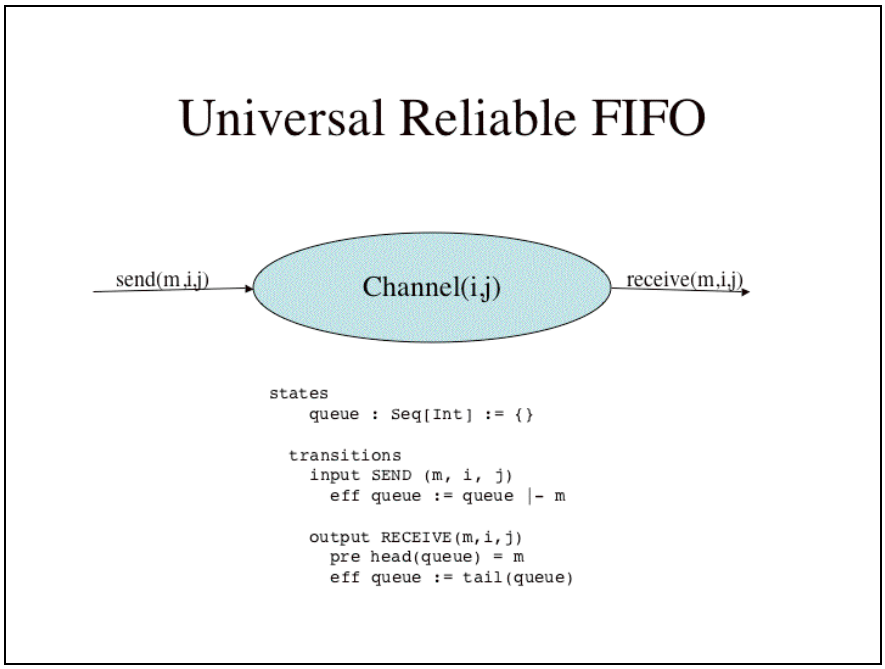
Clearly, traces(P) $\subseteq$ traces(S) $\cap$ traces(L).

Let $\beta \in$ (traces(S) $\cap$ traces(L)) - traces(P). This implies $\beta \in$ traces(L) - traces(P). So $\beta$ must be a finite sequence with no extension in traces(P). But also $\beta \in$ traces(S), and it is finite, so it must be a prefix of something in traces(P). Contradiction.

## *Hierarchical proofs*

This is an important strategy for complex algorithms. We formulate the algorithm in a series of levels. For example, we could have a high-level centralized algorithm (easy to

prove, almost a specification), then do a simple but inefficient decentralized version, then do an optimized version.
High level version:




## Simulation Relations

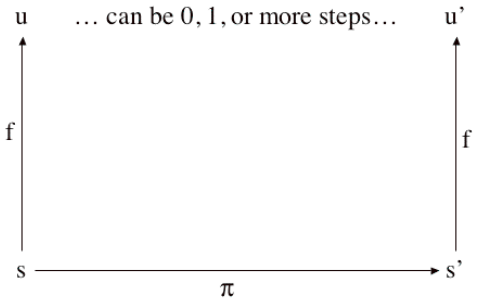
Lower levels are harder to understand, so we relate them to higher levels with a simulation invariant rather than trying to deal with them directly. This is similar to the simulation relation for synchronous algorithms:
Run them side by side.
Define an invariant relating the states.
The invariant is called a *simulation relation* and is usually shown via induction.

Show that for each execution of the lower-level algorithm, there exists a related execution of the higher-level algorithm.

**Definition**: Assume A and B have the same external signature. Let f be a binary relation on states(A)xstates(B).
Notation: $(s,u) \in f$ or $u \in f(s)$
Then f is a simulation relation from A to B provided that

1. If $s \in start(A)$ then $f(s) \cap start(B) \neq \emptyset$
2. If s,u are reachable states of A and B respectively, with $u \in f(s)$, and if $(s, \pi, s')$ is a transition of A, then there is an execution fragment $\alpha$ of B starting with u, and ending with $u' \in f(s')$, with $trace(\alpha) = trace(\pi)$.

# Simulation Relation




**Theorem**. If there's a simulation relation from A to B then traces(A) $\subseteq$ traces(B).
Proof: Take any execution of A, and iteratively construct the corresponding execution of B.

## Example proof

Example: Implementing the reliable FIFO channel with TCP (sort of)

## Altered spec

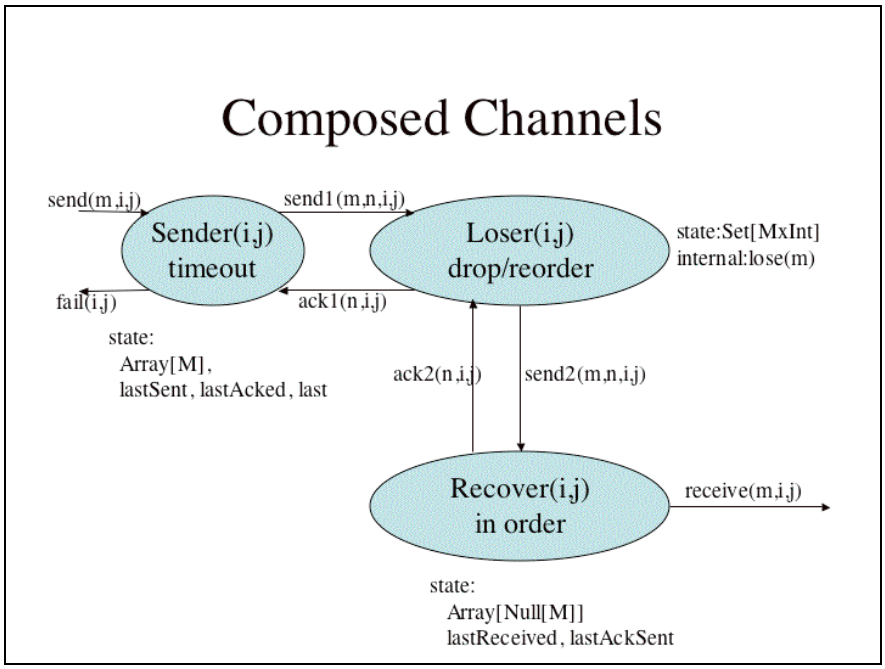
Add fail(i.j) transition to Reliable FIFO: all messages up to some point get sent in order. At some point there is a fail. No more messages can be sent after some point, but a subset of the messages in the queue can be received.

either there is a fail action or the cause is surjective.

if a send `cause(π)` is in `range(cause)` then so is every earlier send

## Implementation

This is an adaptation of TCP.




Discussion:
`Sender(i,j)` puts arriving messages in `Sender(i,j).Array[last+1]` and increments `last`. It sends messages in `Sender(i,j).Array[lastAcked+1...last]` repeatedly until it receives `ack1(n,i,j)`. Then it sets `lastAcked` to `max(lastAcked, n)`.

`Loser(i,j)` puts messages in `Loser(i,j).Set` when `send1(m,i,j)` happens. The precondition for `send2(m,i,j)` and for `delete(m,i,j)` is that `m ∈ Loser(i,j).Set`. The effect of `delete(m,i,j)` is to remove the message from the set, but `send2(m,i,j)` causes no change. This means that `Loser(i,j)` delivers each message to `Recover(i,j)` 0 or more times.

`Recover(i,j)` puts each message in its proper position in `Recover(i,j).Array`.
`Recover(i,j).ack2(n,i,j)` is enabled when there are no holes in `Recover(i,j).Array[1..n]` and `n > lastAckSent`. Its effect is to update `lastAckSent` to `n`.
`Recover(i,j).receive(m,i,j)` is enabled if `m = Recover(i,j).Array[lastReceived+1]` and `lastAckSent > lastReceived`.

## Simulation relation

We are comparing the composition of `Sender`, `Loser`, and `Recover` with all actions hidden except `send(m,i,j)`, `receive(m,i,j)`, and `fail(i,j)`. We need to show that the traces of this composition are contained in the traces of `Channel(i,j)`, modified by the fail action.

The important parts of the state are:

```
Sender(i,j).Array
Recover(i,j).Array
lastReceived
```

`lastAcked`

Facts:

`Recover(i,j).Array` $\subseteq$ `Sender(i,j).Array[1...lastSent]` (with holes)

`Recover(i,j).Array[1...lastReceived]` $\supseteq$ `Sender(i,j).Array[1...lastAcked]`

`Recover(i,j).Array[1...lastAckSent]` is the messages already delivered plus those that are ready to be delivered

The correspondence is that `Channel(i,j).Queue` corresponds to any state with all messages in `Channel(i,j).Queue` sent and no messages in `Channel(i,j).Queue` received (yet) – that is,

`Channel(i,j).Queue` corresponds to all states such that there exists `n` with
`Send(i,j).Array[n...last] = Channel(i,j).Queue` and
`n = Recover(i,j).lastReceived+1`

We must show that the simulation relation is preserved by all actions. Consider actions of the composition:

`send` adds a message to `Channel(i,j).Queue` and to `Sender(i,j).Array[last+1]` – this preserves the simulation relation

`send1, send2` don't change any of the relevant parts of the state

`receive` increases `Recover(i,j).lastReceived` and `Recover(i,j).lastAckSent`; also removes an element from `Channel(i,j).Queue` – this preserves the simulation relation

`ack2` and `ack1` have no effect on relevant parts of state

`fail` can happen any time but after `fail`, `Sender` quits doing anything - the only other thing that can happen is that messages get through the system and received. In `Channel`, no more messages can be sent but some subset of the messages already in the channel can be delivered.