CSc72010

The Asynchronous Model and Properties

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)
```

Schedule

```
schedule
  do
    while(true) do
      fire output P1.SEND(P1.pending,1,2);
      fire output P2.SEND(P2.pending,2,3);
      fire output P3.SEND(P3.pending,3,4);
      fire output P4.SEND(P4.pending,4,1);

      fire output C1.RECEIVE(C1.toSend,1,2);
      fire output C2.RECEIVE(C2.toSend,2,3);
      fire output C3.RECEIVE(C3.toSend,3,4);
      fire output C4.RECEIVE(C4.toSend,4,1)


    od
  od
```

Executions

[<7,T,unknown>, {}, <3,T,unknown>, {}, <2,T,unknown>,{},<5,T,unknown>,{}],
SEND(2,3,4),
[<7,T,unknown>,{},<3,T,unknown>,{},<2,F,unknown>,{2},<5,T,unknown>,{}],
SEND(3,2,3),
[<7,T,unknown>,{},<3,F,unknown>,{3},<2,F,unknown>,{2},<5,T,unknown>,{}],
RECEIVE(3,2,3),
[<7,T,unknown>,{},<3,F,unknown>,{},<3,T,unknown>,{2},<5,T,unknown>,{}],
SEND(7,1,2),
[<7,F,unknown>,{7},<3,F,unknown>,{},<3,T,unknown>,{2},<5,T,,unknown>,{}],
SEND(3,3,4),
[<7,F,unknown>,{7},<3,F,unknown>,{},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
RECEIVE(7,1,2),
[<7,F,unknown>,{},<7,T,unknown>,{},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
SEND(7,2,3),
[<7,F,unknown>,{},<7,F,unknown>,{7},<3,F,unknown>,{2,3},<5,T,unknown>,{}],
RECEIVE(7,2,3),
[<7,F,unknown>,{},<7,F,unknown>,{},<7,T,unknown>,{2,3},<5,T,unknown>,{}],
SEND(7,3,4),
[<7,F,unknown>,{},<7,F,unknown>,{},<7,F,unknown>,{2,3,7},<5,T,unknown>,{}]

Composition

Properties

We would like composition to behave appropriately, i.e., we can recover the original automata from the composition.

Definition. For an execution α and an automaton A_i, define the restriction of α to A_i as $\alpha|A_i$ = the same execution with all actions not in A_i (and the subsequent states) removed, and with states projected onto states of A_i.

Restriction onto Process(1,4,2,7):
For a trace β and an automaton A_i, define the restriction of β to A_i as $\beta|A_i$ = the same trace with all actions not in A_i removed.

Example: see red, above, for restriction to P(3,2,4,2). Actions of P(3,2,4,2) are:
RECEIVE(m, 2, 3)
SEND(m, 3, 4)
leader(2)

Theorem 8.1. Projection.
1. If $\alpha \varepsilon$ execs(A) then $\alpha|A_i \varepsilon$ execs(A_i) for every i
2. If $\beta \varepsilon$ traces(A) then $\beta|A_i \varepsilon$ traces(A_i) for every i

Exercise for May 5

Theorem 8.2. Pasting.
1. If $\alpha_i \varepsilon$ execs(A_i) for all i, β is a sequence of actions in ext(A) such that $\beta|A_i$ = trace(α_i) for all i, then there is an execution α of A such that β = trace(α) and $\alpha_i = \alpha|A_i$ for all i.
2. If β is a sequence of actions in ext(A) and $\beta|A_i$ e traces(A_i) for all i then $\beta \varepsilon$ traces(A).

Part 1 is exercise to hand in on May 5

Consider β=SEND(7,1,2), SEND(3,2,3), SEND(2,3,4), SEND(5,4,1), RECEIVE(7,1,2), RECEIVE(3,2,3), RECEIVE(2,3,4), RECEIVE(5,4,1), SEND(7,2,3), SEND(3,3,4), RECEIVE(7,2,3), RECEIVE(3,3,4), SEND(7,3,4), RECEIVE(7,3,4), SEND(7,4,1), RECEIVE(7,4,1)

Note that it is significant that β restricted to A_i must be legal for each A_i
Consider building up the execution α inductively.

Implementation

We say that A *implements* A' if every trace of A is a trace of A'. Very important.

Consider the reliable FIFO channel. We can show that it implements a reliable channel that doesn't guarantee in-order delivery, which in turn implements a channel that delivers every message at least once, but possibly out of order, which in turn implements a best-effort channel. (This should be intuitively obvious.)

So, if an algorithm works with an unreliable channel, for example, we don't have to prove it again for a reliable channel. This relies on the following substitutivity result:

Use above theorems to prove:
Theorem 3. Substitutivity
Suppose A and A' have the same external signature and traces(A) $\subseteq$ traces(A'). Similarly for B and B'. Then traces(AxB) $\subseteq$ traces(A'xB').

Proof: Let $\beta' \in$ traces(AxB). Then by Theorem 8.1 $\beta'|A \in$ traces(A) and $\beta'|B \in$ traces(B) and so $\beta'|A' \in$ traces(A') and $\beta'|B' \in$ traces(B'). Then (using part 2 of Theorem 8.3) implies that $\beta' \in$ traces(A'×B').

Fairness

We need to know that each task keeps getting turns to do steps (even if no steps are enabled).

Definition

Formally, an execution fragment α is defined to be fair if for all C $\in$ tasks(A), one of the following holds:

1) α is finite and no action of C is enabled in the final state of α.
2) α is infinite and contains infinitely many steps with actions in C.
3) α is infinite and contains infinitely many states in which C is not enabled.

fairexecs(A) is the set of fair executions
fairtraces(A) is the set of fair traces (i.e., traces of fair executions)

Examples

Universal reliable FIFO channel

`{}, send(i,j,a), {}|-a, receive(i,j,a), {}, send(i,j,b), {b}, receive(i,j,b), {}` is fair

`{}, send(i,j,a), {}|-a, receive(i,j,a), {}, send(i,j,b), {b}` is not fair, because receive is enabled

`{}, send(i,j,a), {}|-a, send(i,j,a), {}|-a|-a, send(i,j,a), {}|-a|-a|-a, ..., send(i,j,a), {}|-a|-a...|-a, ...`
is unfair because one of j's tasks (the set of receive actions) never gets to execute.

What are the fair sequences?

Finite: must end in empty queue

Infinite: Every message sent is received (note this is part of the definition of a reliable channel – without the fairness condition, we can't prove reliability, because the channel could just stop otherwise.)

Clock

Clock, p. 213 (actually, this is a translation to the language)

```
automaton Clock

signature
  input request
  internal tick
  output clock(t:Int)

states
  counter:Int := 0,
  flag:Bool := false

transitions
input request
eff
  flag := true

output clock(t:Int)
pre
  flag = true /\ counter = t
eff
  flag := false

internal tick
pre
  true
eff
  counter := counter+1

tasks
  { tick };
  { clock(t:Int) }
```

What are the fair sequences?

No finite sequence of ticks because tick is always enabled

Every request for time must be answered

Properties

Theorem 8.4: Let $\{A_i\}_{i \varepsilon I}$ be a compatible collection of automata and let $A = \Pi_{i \varepsilon I} A_i$.

1. If $\alpha \varepsilon$ fairexecs(A), then $\alpha | A_i$ ε fairexecs(A_i), for every i ε I.
2. If $\beta \varepsilon$ fairtraces(A), then $\beta | A_i$ ε fairtraces(A_i), for every i ε I.

Exercise – part 1 to hand in for May 5

The following are analogous to pasting for arbitrary executions:

Theorem 8.5: Let $\{A_i\}_{i\varepsilon I}$ be a compatible collection of automata and let $A = \Pi_{i\varepsilon I}A_i$. Suppose α_i is a fair execution of A_i for every i ε I, and suppose β is a sequence of actions in ext(A) such that $\beta |A_i = trace(\alpha_i)$ for every i ε I. Then there is a fair execution α of A such that β = trace(α) and $\alpha_i = \alpha|A_i$ for every i ε I.

Exercise.

Theorem 8.6: Let $\{A_i\}_{i\varepsilon I}$ be a compatible collection of automata and let $A = \Pi_{i\varepsilon I}A_i$. Suppose β is a sequence of actions in ext(A). If $\beta |A_i$ ε fairtraces(A_i) for every i ε I, then β ε fairtraces(A).

Exercise. By using 8.5

Theorem 8.7: Every finite execution (or finite trace) can be extended to a fair execution (or fair trace).

Exercise.

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

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) ⊆ traces(P) or maybe fairtraces(A) ⊆ 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.

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. λ is in P
2. if α is in P, all prefixes of α are in P
3. if all prefixes of α are in P, then α 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
A process eventually gets an IP address
A browser eventually gets a page

Note: you will have to make assumptions on the channels, e.g., no failures, or with replication, at most one failure, etc., to get these properties.

Surprising fact

Every trace property can be expressed as a combination of a liveness property and a safety property.

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) ∩ 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) ∪ {β | β is a finite sequence and no extension of β is in traces(P) }.

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

Clearly, traces(P) ⊆ traces(S) ∩ traces(L).

Let β∈ (traces(S) ∩ traces(L)) - traces(P). This implies β∈ traces(L) - traces(P). So β must be a finite sequence with no extension in traces(P). But also β∈ 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.
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.

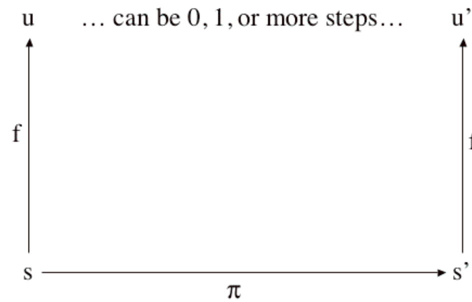
Show that for each execution of the lower-level algorithm, there exists a related execution of the higher-level algorithm.

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

1. If s ∈ start(A) then f(s) ∩ start(B) ≠ ∅

2. If s,u are reachable states of A and B respectively, with u∈ f(s), and if (s, π, s') is a transition of A, then there is an execution fragment α of B starting with u, and ending with u' ∈ f(s'), with trace(α) = trace(π).

Simulation Relation



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

Example: Implementing two channels with one.