Distributed Algorithms

Nancy Lynch

June 30, 2024

Contents

1 Modelling II: Asynchronous System Model **1**

1.1 I/O Automata 1

1.2 Operations on Automata 4

1.2.1 Composition 4

1.2.2 Hiding 7

1.2.3 Fairness 7

1.3 Inputs and Outputs for Problems 9

1.4 Properties and Proof Methods 9

1.4.1 Invariant Assertions 9

1.4.2 Trace Properties 9

1.4.3 Safety and Liveness Properties 10

2 Mutual Exclusion **11**

2.1 Asynchronous Shared Memory Model 11

2.2 The Problem 11

3 Q&A **12**

1 Modelling II: Asynchronous System Model

1.1 I/O Automata

A **signature** S is a triple consisting of three disjoint sets of actions: the **input actions**, $in(S)$, the **output actions**, $out(S)$, and the **internal actions**, $int(S)$. We define the **external actions**, $ext(S)$, to be $in(S) \cup out(S)$; and **locally controlled actions**, $local(S)$ to be $out(S) \cup int(S)$; and $acts(S)$ to be all the actions of S. The **external signature**, $extsig(S)$, is defined to be the signature $(in(S), out(S), \emptyset)$.

An **I/O automaton** A, which we also call simply an **automaton**, consists of five components:

- $sig(A)$, a signature
- $states(A)$
- $start(A)$, a nonempty subset of $states(A)$ known as the **start states** or **initial states**
- $trans(A)$, a **state-transition relation** where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$.
- $tasks(A)$, a **task partition**, which is an equivalence relation on $local(sig(A))$ having at most countably many equivalence classes

We use $acts(A)$ as shorthand for $acts(sig(A))$, and similarly $in(A)$, and so on.

We call an element (s, π, s') of $trans(A)$ a **transition**, or **step**, of A. The transition (s, π, s') is called an **input transition**, **output transition**, and so on, based on whether the action π is an input action, output action, and so on.

If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is **enabled** in s. Since every input action is required to be enabled in every state, automata are said to be **input-enabled**. We say that state s is **quiescent** if the only actions that are enabled in s are input actions.

A task C is **enabled** in a state s means somes action in C is enabled in s.

Example 1.1 (Channel I/O automaton)**.** Consider a communication channel automaton $C_{i,j}$. Let M be a fixed message alphabet.

- **Signature**:

$$\begin{array}{ll} \text{Input}: & \text{Output:} \\ \quad send(m)_{i,j}, m \in M & \quad receive(m)_{i,j}, m \in M \end{array}$$

- **States**: $queue$, a FIFO queue of elements of M, initially empty

- **Transitions**:

```
$send(m)_{i,j}$
  Effect:
    add $m$ to $queue$

$receive(m)_{i,j}$
  Precondition:
    $m$ is first on $queue$
  Effect:
    remove first element of $queue$
```

- **Tasks**: $\{receive(m)_{i,j} : m \in M\}$

Example 1.2 (Process I/O automata)**.** Consider a process automaton P_i. V is a fixed value set, $null$ is a special value not in V, f is a fixed function, $f : V^n \to V$

- **Signature**:
  - Input:
    - $init(v)_i, v \in V$
    - $receive(v)_{j,i}$, $v \in V$, $1 \leq j \leq n$, $j \neq i$
  - Output:
    - $decide(v)_i$, $v \in V$
    - $send(v)_{i,j}$, $v \in V$, $1 \leq j \leq n$, $j \neq i$
- **States**: val, a vector indexed by $\{1, \ldots, n\}$ of elements in $V \cup \{null\}$, all initially $null$
- **Transitions**:

```
$init(v)_i, v \in V$
  Effect:
    $val(i) := v$

$receive(v)_{j,i}, v \in V$
  Effect:
    $val(j) := v$

$send(v)_{i,j}, v \in V$
  Precondition:
    $val(i) = v$
  Effect:
      none

$decide(v)_i, v \in V$
  Precondition:
    for all $j, 1 \leq j \leq n$ :
      $val(j) \neq null$
    $v = f(val(1), \ldots, val(n))$
  Effect:
    none
```

- **Tasks**: for every $j \neq i$: $\{send(v)_{i,j} : v \in V\}$, $\{decide(v)_i : v \in V\}$.

An **execution fragment** of A is either a finite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots$, of alternating states and actions of A s.t. $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. An execution fragment beginning with a start state is called an **execution**. We denote the set of executions of A by $execs(A)$. A state is **reachable** if it is the final state of a finite execution of A.

If α is a finite execution fragment of A and α' is any execution fragment of A that begins with the last state of α, then we write $\alpha \cdot \alpha'$ to represent the sequence obtained by concatenating α and α', eliminating the duplicate occurrence of the last state of α.

The **trace** of an execution α of A, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions. We say that β is a **trace** of A if β is the trace of an execution of A. We denote the set of traces of A by $traces(A)$.

Example 1.3 (Executions)**.** The following are three executions of the automaton $C_{i,j}$ described in Example 1.1 (assuming that the message alphabet M is equal to the set $\{1, 2\}$). Here we indicate the states by putting the sequences in *queue* in brackets; λ denotes the empty sequence.

$$[\lambda], send(1)_{i,j}, [1], receive(1)_{i,j}, [\lambda], send(2)_{i,j}, [2], receive(2)_{i,j}, [\lambda]$$
$$[\lambda], send(1)_{i,j}, [1], receive(1)_{i,j}, [\lambda], send(2)_{i,j}, [2]$$
$$[\lambda], send(1)_{i,j}, [1], send(1)_{i,j}, [11], send(1)_{i,j}, [111], \ldots$$

1.2 Operations on Automata

1.2.1 Composition

The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving π, so do all component automata that have π in their signatures.

We impose certain restrictions on the automata that may be composed.

1. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, we do not allow A to be composed with B unless the internal actions of A are disjoint from the actions of B.

   Otherwise, A's performance of an internal action could force B to take a step.

2. In order that the composition operation might satisfy nice properties, we establish a convention that at most one component automaton "controls" the performance of any given action; that is, we do not allow A and B to be composed unless the sets of output actions of A and B are disjoint.

3. We do not preclude the possibility of composing a countably infinite collection of automata, but we do require in this case that each action must be an action of only finitely many of the component automata.

A countable collection $\{S_i\}_{i \in I}$ of signatures to be **compatible** if for all $i, j \in I$, $i \neq j$, all of the following hold:

1. $int(S_i) \cap acts(S_j) = \emptyset$

2. $out(S_i) \cap out(S_j) = \emptyset$

3. No action is contained in infinitely many sets $acts(S_i)$

We say that a collection of automata is **compatible** if their signatures are compatible.

The **composition** $S = \prod_{i \in I} S_i$ of a countable compatible collection of signatures $\{S_i\}_{i \in I}$ is defined to be the signature with

- $out(S) = \bigcup_{i \in I} out(S_i)$

- $int(S) = \bigcup_{i \in I} int(S_i)$

- $in(S) = \bigcup_{i \in I} in(S_i) - \bigcup_{i \in I} out(S_i)$

Now the **composition** $A = \prod_{i \in I} A_i$ of a countable, compatible collection of I/O automata $\{A_i\}_{i \in I}$ can be defined. It is the automaton defined as:

- $sig(A) = \prod_{i \in I} sig(A_i)$

- $states(A) = \prod_{i \in I} states(A_i)$

- $start(A) = \prod_{i \in I} start(A_i)$

- $trans(A)$ is the set of triples (s, π, s') s.t., for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$.

Note that an action π that is an output of one component and an input of another is classified as an output action in the composition, not as an internal action. This is because we want to permit the possibility of further communication using π.

Example 1.4 (Composition of automata)**.** Consider a fixed index set $I = \{1, \ldots, n\}$ and let A be the composition of all the process automata P_i, $i \in I$ from Example 1.2. In order to compose them, we must assume that the message alphabet M for the channel automata contains the value set V for the process automata.

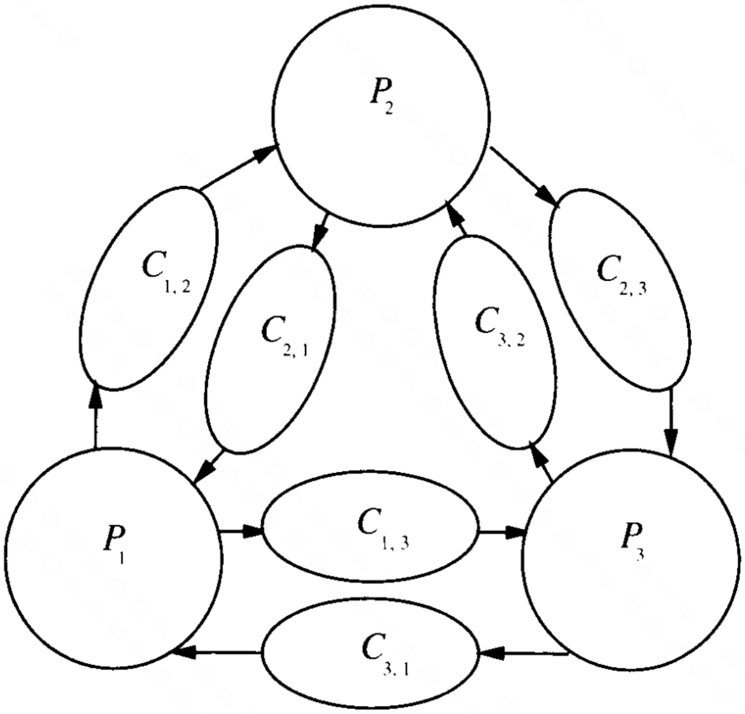

Figure 1: Composition of P_is and $C_{i,j}$s

1. An $init(v)_i$ input action, which deposits a value in P_i's $val(i)$ variable, $val(i)_i$.

2. A $send(v)_{i,j}$ output action, by which P_i's value $val(i)_i$ gets put into channel $C_{i,j}$.

3. A $receive(v)_{i,j}$ output action, by which the first message in $C_{i,j}$ is removed and simultaneously placed into P_j's variable $val(i)_j$.

4. A $decide(v)_i$ output action, by which P_i announcs its current computed value.

Given an execution $\alpha = s_0, \pi_1, s_1, ...$, of A, let $\alpha|A_i$ be the sequence obtained by deleting each pair pi_r, s_r for which π_r is not an action of A_i and replacing each remaining s_r by $(s_r)_i$.

Theorem 1.1. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$.*

1. *If $\alpha \in execs(A)$, then $\alpha|A_i \in execs(A_i)$ for every $i \in I$.*
2. *If $\beta \in traces(A)$, then $\beta|A_i \in traces(A_i)$ for every $i \in I$.*

Theorem 1.2. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ s.t. $\beta|A_i = traces(\alpha_i)$ for every $i \in I$. Then there is an execution α of A s.t. $\beta = trace(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$.*

Theorem 1.3. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose β is a sequence of actions in $ext(A)$. If $\beta|A_i \in traces(A_i)$ for every $i \in I$, then $\beta \in traces(A)$.*

1.2.2 Hiding

If S is a signature and $\Phi \subset out(S)$, then $hide_\phi(S)$ is defined to be the new signature S', where $in(S') = in(S)$, $out(S') = out(S) - \Phi$ and $int(S') = int(S) \cup \Phi$.

If A is an automaton and $\Phi \subseteq out(A)$, then $hide_\Phi(A)$ is the automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_\Phi(sig(A))$.

1.2.3 Fairness

An execution fragment α of an I/O automaton A is said to be **fair** if the following conditions hold for each class C of $tasks(A)$:

1. If α is finite, then C is not enabled in the final state of α
2. If α is infinite, then α contains either infinitely many events from C or infinitely many occurrences of states in which C is not enabled.

We use the term **event** to denote the occurrence of an action in a sequence.

- We can understand the definition of fairness as saying that infinitely often, each task C is given a turn. Whenever this happens, either an action of C gets performed or no action from C could possibly be performed since no such action is enabled.

- We can think of a finite fair execution as an execution at the end of which the automaton repeatedly gives turns to all the tasks in round-robin order, but never succeeds in performing any action since none are enabled in the final state.

We denote the set of fair executions of A by $fairexecs(A)$. We say that β is a **fair trace** of A if β is the trace of a fair execution of A, and we denote the set of fair traces of A by $fairtraces(A)$.

Example 1.5 (Fairness)**.** In Example 1.3, the first execution given is fair, because no $receive$ action is enabled in its final state. The second is not fair, because it is finite and a $receive$ action is enabled in the final state. The third is not fair, because it is infinite, contains no $receive$ events, and has $receive$ actions enabled at every point after the first step.

Theorem 1.4. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$.*

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

Theorem 1.5. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose α_i is a fair execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ s.t. $\beta | A_i = trace(\alpha_i)$ for every $i \in I$. Then there is a fair execution α of A s.t. $\beta = trace(\alpha)$ and $\alpha_i = \alpha | A_i$ for every $i \in I$.*

Theorem 1.6. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose β is a sequence of actions in $ext(A)$. If $\beta | A_i \in fairexecs(A_i)$ for every $i \in I$, then $\beta \in fairexecs(A)$.*

Example 1.6 (Fairness)**.** Consider the fair executions of the system of three processes and three channels in Example 1.4. In every fair execution, every message that is sent is eventually received.

In every fair execution containing least one $init_i$ event for each i, each process sends infinitely many messages to each other processes and each process performs infinitely many $decide$ steps

In every fair execution that does not contain at least one $init$ event for each process, no process ever performs a $decide$ step.

Theorem 1.7. *Let A be any I/O automaton.*

1. *If α is a finite execution of A, then there is a fair execution of A that starts with α.*

2. *If β is a finite trace of A, then there is a fair trace of A that starts with β.*

3. *If α is a finite execution of A and β is any sequence of input actions of A, then there is a fair execution $\alpha \cdot \alpha'$ of A s.t. the sequence of input actions in α' is exactly β*

4. *If β is a finite trace of A and β' is any sequence of input actions of A, then there is a fair execution $\alpha \cdot \alpha'$ of A s.t. $trace(\alpha) = \beta$ and s.t. the sequence of input actions in α' is exactly β'*

1.3 Inputs and Outputs for Problems

1.4 Properties and Proof Methods

1.4.1 Invariant Assertions

1.4.2 Trace Properties

A **trace property** P consists of the following:

- $sig(P)$, a signature containing no internal actions
- $traces(P)$, a set of (finite or infinite) sequences of actions in $acts(sig(P))$

That is, a trace property specifies both an external interface and a set (in other words, a property) of sequences observed at that interface. We write $acts(P)$ as shorthand for $acts(sig(P))$, and similarly $in(P)$, and so on.

The statement that an I/O automaton A satisfies a trace property P can be mean either of two different things:

1. $extsig(A) = sig(P)$ and $traces(A) \subseteq traces(P)$

2. $extsig(A) = sig(P)$ and $fairtraces(A) \subseteq traces(P)$

The fact that A is input-enabled ensures that $fairtraces(A)$ contains a response by A to each possible sequence of input actions. If $fairtraces(A) \subseteq traces(P)$, then all of the resulting sequences must be included in the property P.

Example 1.7 (Automata and trace properties)**.** Consider automata and trace properties with input set $\{0\}$ and output set $\{1, 2\}$. First suppose that $traces(P)$ is the set of sequences over $\{0, 1, 2\}$ that include at least 1. Then $fairtraces(A) \subseteq traces(P)$ means that in every fair execution, A must output at least one.

It is easy to design an I/O automaton for which this is the case - for example, it can include a task whose entire job is to output 1. The fairness condition is used to ensure that this task actually does get a change to output 1. On the other hand, there does not exist any automaton A for which $traces(A) \subseteq traces(P)$, because $traces(A)$ always includes the empty string λ, which does not contain a 1.

Now suppose that $traces(P)$ is the set of sequences over $\{0, 1, 2\}$ that include at least one 0. In this case, there is no I/O automaton A for which $fairtraces(A) \subseteq traces(P)$, because $fairtraces(A)$ must contain some sequence that includes no inputs.

A countable collection $\{P_i\}_{i \in I}$ of trace properties is **compatible** if their signatures are compatible. Then the **composition** $P = \prod_{i \in I} P_i$ is the trace property s.t.

- $sig(P) = \prod_{i \in I} sig(P_i)$.
- $traces(P)$ is the set of sequences β of external actions of P s..t $\beta | acts(P_i) \in traces(P_i)$ for all $i \in I$.

1.4.3 Safety and Liveness Properties

Definition 1.8. A trace property P is a **trace safety property**, or a **safety property** for short, provided that P satisfies the following conditions:

1. $traces(P)$ is nonempty
2. $traces(P)$ is **prefix-closed**, that is, if $\beta \in traces(P)$ and β' is a finite prefix of β, then $\beta' \in traces(P)$
3. $traces(P)$ is **limit-closed**, that is, if $\beta_1, \beta_2, ...$ is an infinite sequence of finite sequences in $traces(P)$, and for each i, β_i is a prefix of β_{i+1}, then $\beta = \bigcup_{i \in \omega} \beta_i \in traces(P)$.

Example 1.8 (Trace safety property)**.** Suppose $sig(P)$ consists of inputs $init(v)$, $v \in V$ and outputs $decide(v)$, $v \in V$. Suppose $traces(P)$ is the set of sequences of $init$ and $decide$ actions in which no $decide(v)$ occurs without a preceding $init(v)$ (for the same v). Then P is a safety property.

Proposition 1.9. *If P is a safe property, TFAE:*

1. $traces(A) \subseteq traces(P)$
2. $fairtraces(A) \subseteq traces(P)$
3. *finite traces of A are all in traces P.*

Proof. □

2 Mutual Exclusion

2.1 Asynchronous Shared Memory Model

The system is modelled as a collection of processes and shared variables, with interactions. Each process i is a kind of state machine, with a set statesi of states and a subset $start$ of $states_i$ indicating the start states, just as in the synchronous setting. However, now process i also has labelled $actions$, describing the activities in which it participates. These are classified as either $input$, $output$, or $internal$ actions. We further distinguish between two different kinds of internal actions: those that involve the shared memory and those that involve strictly local computation. If an action involves the shared memory, we assumethat it only involves one shared variable.

There is a transition relation $trans$ for the entire system, which is a set of (s, π, s') triples, where s and s' are **automaton states**, that is, combinations of states for all the processes and values for all the shared variables, and where π is the label of an input, output, or internal action. We call these combinations of process states and variable values "automaton states" because the entire system is modelled as a single automaton. The statement that $(s, \pi, s') \in trans$ says that from automaton state s it is possible to go to automaton state s' as a result of performing action π.

We assume that input actions can always happen, that is, that the system is input-enabled. Formally, this means that for every automaton state s and input action π, there exists s' such that $(s, \pi, s') \in trans$. In contrast, output and internal steps might be enabled only in a subset of the states. The intuition behind the input-enabling property is that the input actions are controlled by an arbitrary external user, while the internal and output actions are controlled by the system itself.

2.2 The Problem

The mutual exclusion problem involves the allocation of a single, indivisible, nonshareable resource among n **users**, $U_1, \ldots, U_n$.

A user with access to the resource is modelled as being in a **critical region**, which is simply a designated subset of its states. When a user is not involved in any way with the resource, it is said to be in the **remainder region**. In order to gain admittance to its critical region, a user executes a **trying protocol**, and after it is done with the resource, it executes an (often trivial) **exit protocol**. This procedure can be repeated, so that each user follows a cycle, moving from its *remainder region* (R) to its *trying region* (T), then to its *critical region* (C), then to its *exit region* (E), and then back again to its remainder region.

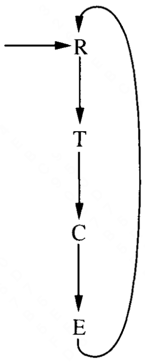

Figure 2: The cycle of regions of a single user

Each of the users U_i, $1 \leq i \leq n$, is modelled as a state machine (formally, an **I/O automaton**) that communicates with its agent process using the try_i, $crit_i$, $exit_i$ and rem_i actions:

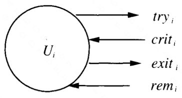

Figure 3: External interface of user U_i

We define a sequence of try_i, $crit_i$, $exit_i$ and rem_i actions to be **well-formed** for user i if it is a prefix of the cyclically ordered sequence $try_i, crit_i, exit_i, rem_i, try_i, ...$ Then we require that U_i **preserve** the **trace property** defined by the set of sequences that are well-ordered for user i.

3 Q&A

1. 1.2.3. Need think.