Distributed Algorithms

Nancy Lynch

July 15, 2024

Contents

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**:

Input :	Output:
$send(m)_{i,j}, m \in M$	$receive(m)_{i,j}, m \in M$

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

$send(m)_{i,j}$	$receive(m)_{i,j}$
Effect:	Precondition:
add m to $queue$	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 $j, 1 \leq j \leq n$:
$val(j) \neq null$
$v = f(val(1), ..., val(n))$
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 $s_0, \pi_1, s_1, \pi_2, ..., \pi_r, s_r$ or an infinite sequence $s_0, \pi_1, s_1, \pi_2, ...$, 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.

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

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 $s_i = s_i'$.
- $tasks(A) = \bigcup_{i \in I} tasks(A_i)$

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.

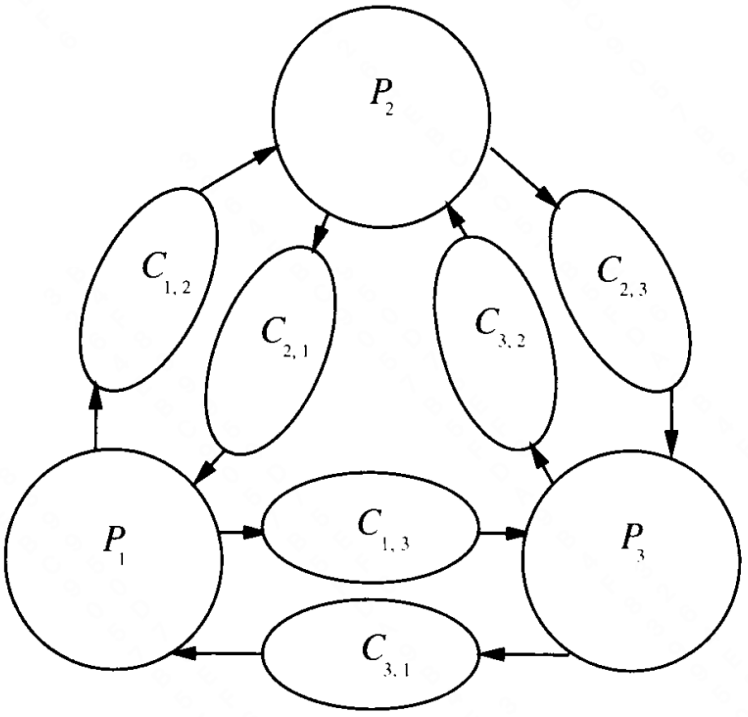


Figure 1: Composition of P_is and $C_{i,j}$s

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* $i \in I$.

Proof. 1. Execution of any automaton A_j where $j \neq i$ doesn't affect automaton A_i.

2. Immediately

□

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 $i \in I$.*

Proof. 1. If $\alpha \in fairexecs(A)$.

- If α is finite, then for each task C, C is not enabled in the final state of α, therefore each $C|A_i$ is not enabled in the final state of $\alpha|A_i$ too.
- If α is infinite, then blabla

Therefore $\alpha|A_i \in fairexecs(A_i)$

2. same

□

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. $(2 \Rightarrow (3)$: For any finite trace $\beta \in traces(A)$, there is $\beta' \in fairtraces(A)$ that starts in β. Thus $\beta \in traces(P)$ because of prefix-closedness.

$(3) \Rightarrow (1)$: For any infinite trace $\beta \in traces(A)$, we can have such a infinite sequence of traces $\beta_1, \beta_2, ...$ of A, where β_i is a prefix of β_{i+1} for any i, and $\beta = \bigcup_{i \in \omega} \beta_i$. Therefore $\beta \in traces(P)$ because of limit-closedness. □

Definition 1.10. A trace property P is a **trace liveness property**, or a **liveness property** for short, provided that every finite sequence over $acts(P)$ has some extension in $traces(P)$.

Example 1.9. Suppose $sig(P)$ consists of input $init(v)$, $v \in V$ and outputs $decide(v)$, $v \in V$. Suppose $traces(P)$ is the set of sequences β of $init$ and $decide$ actions in which, for every $init$ event in β, there is some $decide$ event occuring later in β. Then P is a liveness property.

Often one wants to prove that $fairtraces(A) \subseteq traces(P)$ for some automaton A and liveness property P. Methods based on **temporal logic** work well in practice for proving such claims. Another method for proving liveness claims, which we call the **progress function method**, is specially designed for proving that some particular goal is eventually reached.

Theorem 1.11. *If P is both a safety property and a liveness property, then P is the set of all (finite and infinite) sequence of actions in $acts(P)$.*

Proof. Suppose that P is both a safety and a liveness property and let β be an arbitrary sequence of elements of $acts(P)$. If β is finite, then since P is a liveness property, β has some extension β' in $traces(P)$. Then since P is a safety property, $\beta \in traces(P)$.

If β is infinite, then for each $i \geq 1$, define β_i to be the length i prefix of β. Then $\beta \in traces(P)$. □

Theorem 1.12. *If P is an arbitrary trace property with $traces(P) \neq \emptyset$, then there exist a safety property S and a liveness property L s.t.*

1. $sig(S) = sig(L) = sig(P)$.
2. $traces(P) = traces(S) \cap traces(L)$

Proof. Let $traces(S)$ be the prefix- and limit-closure of $traces(P)$. Let

$$\begin{aligned} traces(L) =& traces(P) \\ & \cup \{\beta : \beta \text{ is a finite sequence and no extension of } \beta \text{ is in } traces(P)\} \end{aligned}$$

Claim: L is a liveness property Now $traces(P) \subseteq traces(S) \cap traces(L)$. If there is $\beta \in traces(S) \cap traces(L) \setminus traces(P)$, then β is a finite sequence and no extension of β is in $traces(P)$. □

1.4.4 Compositional Reasoning

Theorem 1.13. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Let $\{P_i\}_{i \in I}$ be a (compatible) collection of trace properties and let $P = \prod_{i \in I} P_i$*

1. *If $extsig(A_i) = sig(P_i)$ and $traces(A_i) \subseteq traces(P_i)$ for every i, then $extsig(A) = sig(P)$ and $traces(A) \subseteq traces(P)$.*
2. *If $extsig(A_i) = sig(P_i)$ and $fairtraces(A_i) \subseteq traces(P_i)$ for every i, then $extsig(A) = sig(P)$ and $fairtraces(A) \subseteq traces(P)$.*

Proof. 1. If $\beta \in traces(A)$, then by Theorem 1.1, $\beta | A_i \in traces(A_i) \subseteq traces(P_i)$ for every $i \in I$. Then by Theorem 1.3, $\beta \in traces(P)$.

2.

□

Example 1.10 (Satisfying a product trace property)**.** Consider the composed system of Example 1.4. Each process automaton P_i satisfies a trace safety property that asserts that any $decide_i$ event has a preceding $init_i$ event. Also, each channel automaton $C_{i,j}$ satisfies a trace safety property that asserts taht the sequence of messages in $receive_{i,j}$ events is a prefix of athe sequence of messages in $send_{i,j}$ events.

Then it follows from Theorem 1.13 that the composed system satisfies the product trace safety property. This means that in an trace of the combined system, the following hold:

1. For every i, any $decide_i$ event has a preceding $init_i$ event

2. For every i and j, $i \neq j$, the sequence of messages in $receive_{i,j}$ events is a prefix of the sequence of messages in $send_{i,j}$ events.

Second, suppose that we want to show that a particular sequence of actions is a trace of a composed system $A = \prod_{i \in I} A_i$. Theorem 1.3 shows that it is enough to show taht the projection of the sequence on each of the system components is a trace of that component. Theorem 1.6 implies an analogous result for fair traces.

Third, consider the compositional proof of safety properties. Suppose we want to show that a composed system $A = \prod_{i \in I} A_i$ satisfies a safety property P. One strategy is to show that none of the components A_i is the first to violate P.

Let A be an I/O automaton and let P be a safety property with $acts(P) \cap int(A) = \emptyset$ and $in(P) \cap out(A) = \emptyset$. We say that A **preserves** P if for every finite sequence β of actions that does not include any internal actions of A, and every $\pi \in out(A)$, the following holds: If $\beta | acts(P) \in traces(P)$ and $\beta\pi | A \in traces(A)$, then $\beta\pi | acts(P) \in traces(P)$. This says that A is not the first to violate P, as long as A's environment only provides inputs to A in such a way that the cumulative behaviour satisfies P, then A will only perform outputs s.t. the cumulative behaviour satisfies P.

The key fact about preservation of safety properties is that if all the components in a composed system preserve a safety property, then so does the entire system. Moreover, if the composed system is closed, then it actually satisfies the safety property.

Theorem 1.14. *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Let P be a safety property with $acts(P) \cap int(A) = \emptyset$ and $in(P) \cap out(A) = \emptyset$.*

1. *If A_i preserves P for every $i \in I$, then A preserves P*

2. *If A is a closed automaton, A preserves P, and $acts(P) \subseteq ext(A)$, then $traces(A) | acts(P) \subseteq traces(P)$*

3. *If A is a closed automaton, A preserves P, and $acts(P) = ext(A)$, then $traces(A) \subseteq traces(P)$.*

1.4.5 Hierarchical Proofs

1.5 Complexity Measures

2 Modelling III: Asynchronous Shared Memory Model

2.1 Shared Memory Systems

We mode lthe entire system as one big I/O automaton A.

As in the synchronous network model, we assume that the processes in the system are indexed by $1, \ldots, n$. Suppose that each process i has an associated set of **states**, $states_i$, among which some are designated as **start states**, $start_i$. Also suppose that each shared variable x in the system has an associated set of **values**, $values_x$, among which some are designated as the **initial values**, $initial_x$. Then each state in $states(A)$ (the set of states of the system automaton A) consists of a state in $states_i$ for each process i, plus a value in $values_x$ for each shared variable x. Each state in $start(A)$ consists of a state in $start_i$ for each process i, plus a value in $initial_x$ for each shared variable x.

We assume that each action in $acts(A)$ is associated with one of the processes. In addition, some of the internal actions in $int(A)$ may be associated with a shared variable. The input actions and output actions associated with process i are used for interaction between process i and the outside world; we say they occur on **port** i. The internal actions of process i that do not have an associated shared variable are used for local computation, while the internal actions of i that are associated with shared variable x are used for performing operations on x.

The set $trans(A)$ of transitions has some locality restrictions, which model the process and shared variable structure of the system.

1. Consider an action π that is associated with process i but with no variable; as we noted above, π is used for local computation. Then only the state of i can be involved in any π step. That is, the set of π transitions can be generated from some set of triples of the form (s, π, s'), where $s, s' \in states_i$, by attaching any combination of states for the other processes and values for the shared variables to both s and s'

2. Consider an action π that is associated with both a process i and a variable x; π is used by i to perform an operation on x. The set of π transitions can be generated from some set of triples of the form $(s, v), \pi, (s', v')$, where $s, s' \in states_i$ and $v, v' \in values_x$, by attaching

any combination of states for the other processes and values for the other shared variables. There is a technicality: if π is associated with process i and variable x, then whether or not π is enabled should depend only on the state of process i

The task partitioon $tasks(A)$ must be consistent with the process structure: that is, each equivalence class (task) should include locally controlled actions of only one process.

Example 2.1 (Shared memory system)**.** Let V be a fixed value set. Consider a shared memory system A consisting of n processes, numbered $1, \ldots, n$, and a single shared variable x with values in $V \cup \{unknown\}$, initially $unknown$. The inputs are of the form $init(v)_i$, where $v \in V$ and i is a process index. The outputs are of the form $decide(v)_i$, The internal actions are of the form $access_i$. All the actions with subscript i are associated with process i, and in addition, the $access$ actions are associated with variable x.

After process i receives an $init(v)_i$ input, it accesses x. If it finds $x = unknown$, then it writes its value v into x and decides v. If it finds $x = w$, where $w \in V$, then it does not write anything into x, but decides w.

Formally:

States of i:

$status \in \{idle, access, decide, done\}$, initially $idle$
$input \in V \cup \{unknown\}$, initially $unknown$
$output \in V \cup \{unknown\}$, initially $unknown$

Transitions of i:

$init(v)_i$
  Effect:
    $input := v$
    if $status = idle$ then
      $status := access$

$decide(v)_i$
  Precondition:
    $status = decide$
    $output = v$
  Effect:
    $status := done$

$access_i$
  Precondition:
    $status = access$
  Effect:
    if $x = unknown$ then $x := input$
    $output := x$
    $status := decide$

There is one task per process, which contains all the $access$ and $decide$ actions for that process.

It is not hard to see that in every fair execution α of A, any process that receives an $init$ input eventually performs a $decide$ output. Moreover, every execution satisfies the "agreement property" that no two processes decide on different values, and the "validity property" that every decision value is the initial value of some process.

3 Mutual Exclusion

3.1 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.

3.2 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.

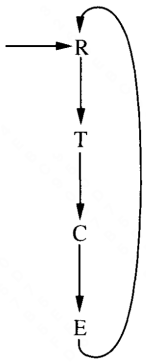


Figure 2: The cycle of regions of a single user

The inputs to process i are the try_i action, which models a request by user U_i for access to the resource, and the $exit_i$ action, which models an annoucement by user U_i that it is done with the resource. The outputs of process i are $crit_i$, which models the granting of the resource to U_i and rem_i, which tells U_i that it can continue with the reset of its work. The try, $crit$, $exit$, and rem actions are the only external actions of the shared memory system. The processes are responsible for performing the trying and exit protocols. Each process i acts as an "agent" on behalf of user U_i.

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:

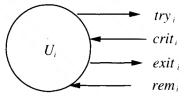


Figure 3: External interface of user U_i

The only thing that we assume about U_i is that it obeys the cyclic region protocol. 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.

In executions of U_i that do observe the cyclic order of actions, we say that U_i is

- in its **remainder region** initially and in between any rem_i event and the following try_i event
- in its **trying region** in between try_i event and the following $crit_i$ event
- in its **critical region** in between any $crit_i$ event and the following $exit_i$ event. During the time, U_i should be thought of as being free to use the resource
- in its **exit region** in between any $exit_i$ event and the following rem_i event

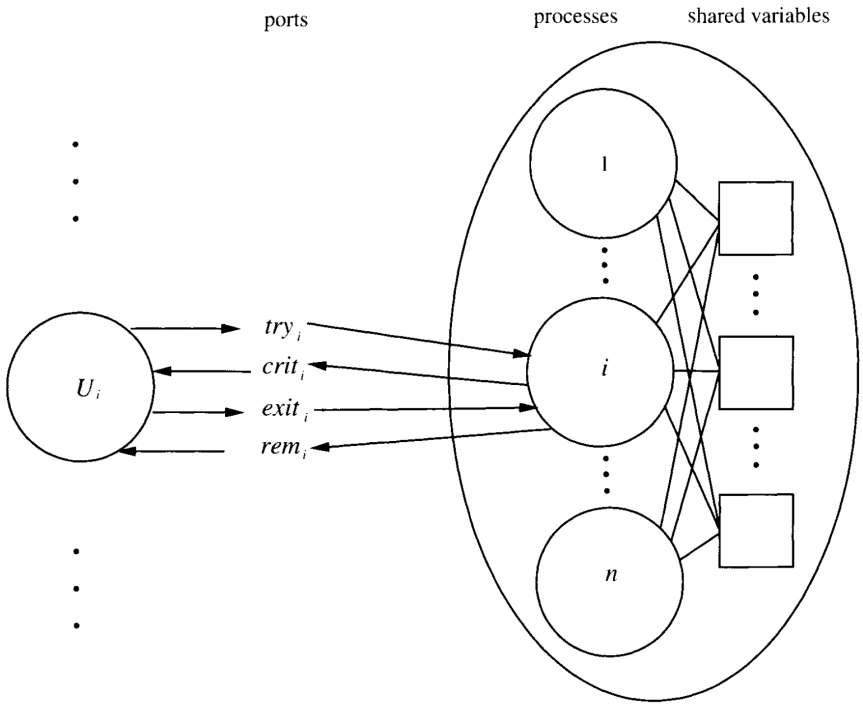


Figure 4: Interactions between components for the mutual exclusion problem

The combination of A and the users must satisfy the following conditions:

- **Well-formedness**: In any execution, and for any i, the subsequence describing the interactions between U_i and A is well-formed for i.
- **Mutual exclusion**: There is no reachable system state where more than one user is in the critical region C
- **Progress**: At any point in a *fair execution*
  1. (Progress for the trying region) If at least one user is in T and no user is in C, then at some later point some user enters C
  2. (Progress for the exit region) If at least one user is in E, then at some later point some user enters R.

We say that a shared memory system A **solves the mutual exclusion problem** provided that it solves it for every collection of users.

Lemma 3.1. *Let A be an algorithm that solves the mutual exclusion problem. Let $U_1, \ldots, U_n$ be any particular collection of users, and let B be the combination of A and the given collection of users. Let s be a reachable state of B.*

If process i is in its trying or exit region in state s, then some locally controlled action of process i is enabled in s.

Proof. WLOG, we may assume that each of the users always returns the resource.

Let α be a finite execution of B ending in s, and assume that process i is in either its trying or exit region in state s, and no locally controlled action of process i is enabled in s. Then we claim that no events involving i occur in any execution of B that extends α, after the prefix α. This follows from the fact that enabling of locally controlled actions is determined only by the local process state, plus the fact that well-formedness prevents inputs to process i while process i is in T or E.

Now let α' be a fair execution of B that extends α, in which no try events occur after the prefix α. Repeated use of the progress assumption, plus the fact that the users always return the resource, imply that process i must eventually perform either a $crit_i$ or a rem_i action. But this contradicts the fact that α' contains no further actions of i. □

3.3 Dijkstra's Mutual Exclusion Algorithm

3.3.1 The Algorithm

Shared variables:
  $turn \in \{1, \ldots, n\}$, *initally arbitrary, writable and readable by all processes*
for every i, $1 \leq i \leq n$*:*
  $flag(i) \in \{0, 1, 2\}$, *initially 0, writable by process* i *and readable by all processes*;
Process i**:**

Remainder region

try_i;
L $flag(i) := 1$;
while $turn \neq i$ **do**
  if $flag(turn) = 0$ **then**
    $turn := i$;
end
$flag(i) := 2$;
for $j \neq i$ **do**
  if $flag(j) = 2$ **then**
    go to L;
end
$crit_i$;

Critical region

$exit_i$;
$flag(i) := 0$;
rem_i;

Algorithm 1: DijkstraME algorithm

The $turn$ variable is a $multi\text{-}writer/multi\text{-}reader$ register. Each $flag(i)$ is a $single\text{-}writer/multi\text{-}reader$ register.

The state of each process should consist of the values of its local variables plus some other information that is not represented explicitly in the code, including

- temporary variables needed to remember values just read from shared variables
- a program counter

- temporary variables introduced by the flow of control of the program
- a region designation, R, T, C, or E

The unique start state of each process should consist of specified initial values for local variables, arbitrary values for temporary variables, and the program counter and the region designation indicating the remainder region.

There are some ambiguities in the code that need to be resolved in the automaton.

1. Although the code describes the changes to the local and shared variables, it does not say explicitly what happens to the implicit variables
2. The code also does not specify exactly which portions of the code comprise indivisible steps.

DijkstraME algorithm (rewritten)

Shared variables:
$turn \in \{1, \ldots, n\}$, initially arbitrary
for every i, $1 \leq i \leq n$:
    $flag(i) \in \{0, 1, 2\}$, initially 0
Actions of i:

Input:	Internal:
try_i	$set\text{-}flag\text{-}1_i$
$exit_i$	$test\text{-}turn_i$
Output:	$test\text{-}flag(j)_i, 1 \leq j \leq n, j \neq i$
$crit_i$	$set\text{-}turn_i$
rem_i	$set\text{-}flag\text{-}2_i$
	$check(j)_i, 1 \leq j \leq n, j \neq i$
	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag\text{-}1, test\text{-}turn, test\text{-}flag(j), set\text{-}turn, set\text{-}flag\text{-}2, check, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem.
    S, a set of process indices, initially $\emptyset$.
Transitions of i:

try_i
  Effect:
    $pc := set\text{-}flag\text{-}1$

$set\text{-}flag\text{-}1_1$
  Precondition:
    $pc = set\text{-}flag\text{-}1$
  Effect:
    $flag(i) := 1$
    $pc := test\text{-}turn$

$test\text{-}turn_i$
  Precondition:
    $pc = test\text{-}turn$
  Effect:
    if $turn = i$ then $pc := set\text{-}flag\text{-}2$
    else $pc := test\text{-}flag(turn)$

$test\text{-}flag(j)_i$
  Precondition:
    $pc = test\text{-}flag(j)$
  Effect:
    if $flag(j) = 0$ then $pc := set\text{-}turn$
    else $pc := test\text{-}turn$

$crit_i$
  Precondition:
    $pc = leave\text{-}try$
  Effect:
    $pc := crit$

$exit_i$
  Effect:
    $pc := reset$

$set\text{-}turn_i$
  Precondition:
    $pc = set\text{-}turn$
  Effect:
    $turn := i$
    $pc := set\text{-}flag\text{-}2$

$set\text{-}flag\text{-}2_i$
  Precondition:
    $pc = set\text{-}flag\text{-}2$
  Effect:
    $flag(i) := 2$
    $S := \{i\}$
    $pc := check$

$check(j)_i$
  Precondition:
    $pc = check$
    $j \notin S$
  Effect:
    if $flag(j) = 2$ then
      $S := \emptyset$
      $pc := set\text{-}flag\text{-}1$
    else
      $S := S \cup \{j\}$
      if $|S| = n$ then $pc := leave\text{-}try$

$reset_i$
  Precondition:
    $pc = reset$
  Effect:
    $flag(i) := 0$
    $S := \emptyset$
    $pc := leave\text{-}exit$

rem_i
  Precondition:
    $pc = leave\text{-}exit$
  Effect:
    $pc := rem$

3.3.2 A Correctness Argument

Lemma 3.2. *DijkstraME guarantees well-formedness for each user.*

Proof. By inspection of the code, it is easy to check that DijkstraME preserves well-formedness for each user. Since, by assumption, the users also

preserve well-formedness, Theorem 1.14 implies that the system produces only well-formed sequences. □

Lemma 3.3. *DijkstraME satisfies mutual exclusion*

Proof. By contradiction. Assume that U_i and U_j, $i \neq j$, are simultaneouly in region C in some reachable state. Consider the execution that leads to this state. By the code, both i and j perfrom $set\text{-}flag\text{-}2$ steps before entering their critical area. Consider the last such step for each process and assume, without loss of generality, that $set\text{-}flag\text{-}2_i$ comes first. Then $flag(i)$ is 2 from that point until i leaves C, contradicting the fact that j enters C.

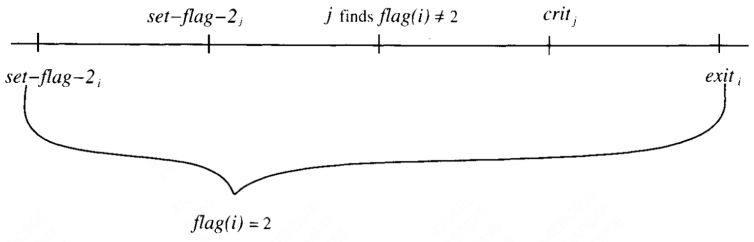


Figure 5: Order of events in the proof of Lemma 3.3

□

Lemma 3.4. *DijkstraME guarantees progress.*

Proof. Progress of exit region is easy.

We consider the progress condition for the trying region. Suppose for the sake of contradiction that α is a fair execution that reaches a point where there is at least one user in T and no user in C, and suppose that after this point, no user ever enters C.

Any process in E keeps taking steps, so after at most two steps, it must reach R. So after some point in α, every process must be in T or R. Second, since there are only finitely many processes in the system, after some point in α, no new processes enter T. Thus, after some point in α, every process is in T or R, and no process every again changes region. This implies that α has a suffix α_1 in which there is a fixed nonempty set of processes in T, continuing to take steps forever, and no region changes occur. Call these processes **contenders**.

Note that after at most a single step in α_1, each contender i ensures that $flag(i) \geq 1$ and it remains ≥ 1 for the rest of α_1. So we can assume, WLOG, that $flag(i) \geq 1$ for all contenders throughout α_1.

Claim: In α_1, turn eventually acquires a contender's index.

#+BEGIN$_{\text{proof}}$ Suppose not, that is, suppose the value of $turn$ remains equal to the index of a non-contender throughout α_1. Consider any contender i.

If pc_i reaches $test\text{-}turn$, which is happened since i fails to enter C, then $test\text{-}turn_i$ finds that $turn$ equal to some $j \neq i$. Then it performs a $test\text{-}turn_i$ and finds $flag(j) = 0$. Process i therefore performs $set\text{-}turn_i$, setting $turn$ to i. □

Once $turn$ is set to a contender's index, it is always thereafter equal to *some* contender's index. Then any later $test\text{-}turn$ and subsequent $test\text{-}flag$ yield $flag(turn) \geq 1$. Thus, $turn$ will not changed as a result of these tests. Therefore, eventually $turn$ stabilizes to a final indx. Let α_2 be a suffix of α_1 in which the value of $turn$ is stablized at some contender's index, say i.

Next, we claim that in α_2, any contender $j \neq i$ eventually ends up with its program counter looping forever between $test\text{-}turn$ and $test\text{-}flag$. So let α_3 be a suffix of α_2 where all contenders other than i loop forever between $test\text{-}turn$ and $test\text{-}flag$. Note that this means that all contenders other than i have their flag variables equal to 1 throughout α_3.

We conclude the argument by claiming that in α_3, process i (the one whose index is in turn) has nothing to stand in the way of its reaching C. #+END$_{\text{proof}}$

Theorem 3.5. *DijkstraME solves the mutual exclusion problem.*

3.3.3 An Assertion Proof of the Mutual Exclusion Condition

another proof of Lemma 3.3.

Assertion 3.6. *In any reachable system state,* $|\{i : pc_i = crit\}| \leq 1$

Assertion 3.7. *In any reachable system state, if* $pc_i \in \{leave\text{-}try, crit, reset\}$, *then* $|S_i| = n$

Assertion 3.8. *In any reachable system state, there do not exist* i *and* j, $i \neq j$, *s.t.* $i \in S_j$ *and* $j \in S_i$.

If both Assertions 3.7 and 3.8 are true, then Assertion 3.6 follows.

Assertion 3.7 is easy.

Assertion 3.9. *In any reachable system state, if* $S_i \neq \emptyset$, *then* $pc_i \in \{check, leave\text{-}try, crit, reset\}$.

Assertion 3.10. *In any reachable system state, if* $pc_i \in \{check, leave\text{-}try, crit, reset\}$, *then* $flag(i) = 2$

Putting these together, we see that:

Assertion 3.11. *In any reachable system state, if $S_i \neq \emptyset$, then $flag(i) = 2$.*

Now we can prove Assertion 3.8, again by induction on the length of an execution. For the inductive step, the only event that could cause a violation is one that adds an element j to S_i for some i and j, $i \neq j$, that is, a $check(j)_i$. Then it must be that $flag(j) \neq 2$ when this event occurs. But then $S_j = \emptyset$, so $i \notin S_j$. □

3.3.4 Running Time

We impose:

- an upper bound of l on the time between successive steps of each process (when these steps are enabled);
- all the precondition-effect code for one action is assumed to comprise a single step.
- an upper bound of c on the maximum time that any user spends in the critical region.

In terms of these assumed bounds, we can deduce upper bounds for the time required for interesting activity to occur.

Theorem 3.12. *In DijkstraME, suppose that at a particular time some user is in T and no user is in C. Then within $O(ln)$, some user enters C*

Proof. Suppose the lemma is false and consider an execution in which, at some point, process i is in T and no process is in C, and in which no process enters C for time at least kln, for some particular large constant k.

First, it is easy to see that the time elapsed from the starting point of the analysis until there is no process either in C or E is at most $O(l)$.

Second, we claim that the additional time until process i performs a $test\text{-}turn_i$ is at most $O(ln)$. This is because i can at worst spend this much time checking flags in the second stage before returning to $set\text{-}flag$-1.

Third, we claim that the additional time from when process i does $test\text{-}turn_i$ until the value of turn is a contender index is at most $O(l)$.

Fourth, after an additional time $O(l)$, a point is reached at which the value of $turn$ has stablized to the index of some particular contender, say j.

Fifth, we claim that by an additional time $O(ln)$, all contenders other than j will have their program counters in $test\text{-}turn, test\text{-}flag$.

Sixth and finally, within an additional time $O(ln)$, j must succeed in entering C

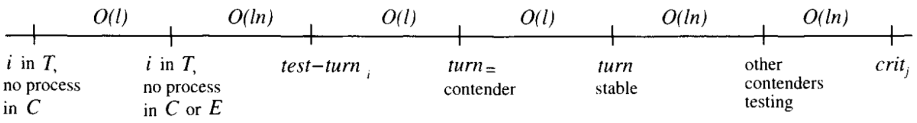


Figure 6: Order of events and time bounds in proof of Theorem 3.12

□

3.4 Stronger Conditions for Mutual Exclusion Algorithms

In order to distinguish these two types of fairness, we will call the fair execution of process steps and user automata steps **low-level fairness**, and the fair granting of the resource **high-level fairness**.

Another not-so-attractive property of Dijkstra's algorithm is that it uses a shared *multi-writer/multi-reader* register (turn). Such a variable is difficult and expensive to implement in many kinds of multiprocessor systems (as well as in nearly all message-passing systems). It would be better to design algorithms that use only *single-writer/multi-reader* registers, or even better, *single-writer/single-reader* registers.

Each of these properties is stated for a particular mutual exclusion algorithm A composed with a particular collection $U_1, \ldots, U_n$ of users.

Lockout-freedom: In any low-level-fair execution, the following hold:

1. (Lockout-freedom for the trying region) If all users always return the resource, then any user that reaches T eventually enters C.
2. (Lockout-freedom for the exit region) Any user that reaches E eventually enters R.

Time bound b: In any low-level-fair execution with associated times, the following hold:

1. (Time bound b for the trying region) If each user always returns the resource within time c of when it is granted, and the time between successive steps of each process in T or E is at most l, then any user that reaches T enters C within time b.
2. (Time bound b for the exit region) If the time between successive steps of each process in T or E is at most g, then any user that reaches E enters R within time b.

Number of bypasses a: Consider any interval of an execution starting when a process i has performed a locally controlled step in T, and throughout which it remains in T. During this interval, any other user j, $j \neq i$, can only enter C at most a times.

Theorem 3.13. *Let A be a mutual exclusion algorithm, let $U_1, \ldots, U_n$ be a collecion of users, and let B be the composition of A with $U_1, \ldots, U_n$. If B has any finite bypass round and is lockout-free for the exit region, then B is lockout-free.*

Proof. Consider a low-level-fair execution of B in which all users always return the resource, and suppose that at some point in the execution, i is in T. Assume for the sake of contradiction that i never enters C. Lemma 3.1 implies that eventually i must perform a locally controlled action in that trying region, if it has not already done so. Repeated use of the progress condition and of the assumption that users always return the resource together imply that infinitely many total region changes occur. But then some process other than i enters C an infinite number of times while i remains in T, which violates the bypass bound. □

Theorem 3.14. *Let A be a mutual exclusion algorithm, let $U_1, \ldots, U_n$ be a collection of users, and let B be the composition of A with $U_1, \ldots, U_n$. If B has any time bound b, then B is lockout-free.*

3.5 Lockout-Free Mutual Exclusion Algorithms

3.5.1 A Two-Process Algorithm.

If $i \in \{0, 1\}$, we write $\bar{i}$ to indicate $1 - i$.

Peterson2P algorithm

Shared variables:
$turn \in \{0, 1\}$, initially arbitrary, writable and readable by all processes
For every $i \in \{0, 1\}$: $flag(i) \in \{0, 1\}$, initially 0, writable by i and readable by $\bar{i}$
Process i:

Remainder region

try_i;
$flag(i) := 1$;
$turn := i$;
wait for $flag(\dot{i}) = 0$ or $turn \neq i$;
$crit_i$

Critical region

$exit_i$;
$flag(i) := 0$;
rem_i;

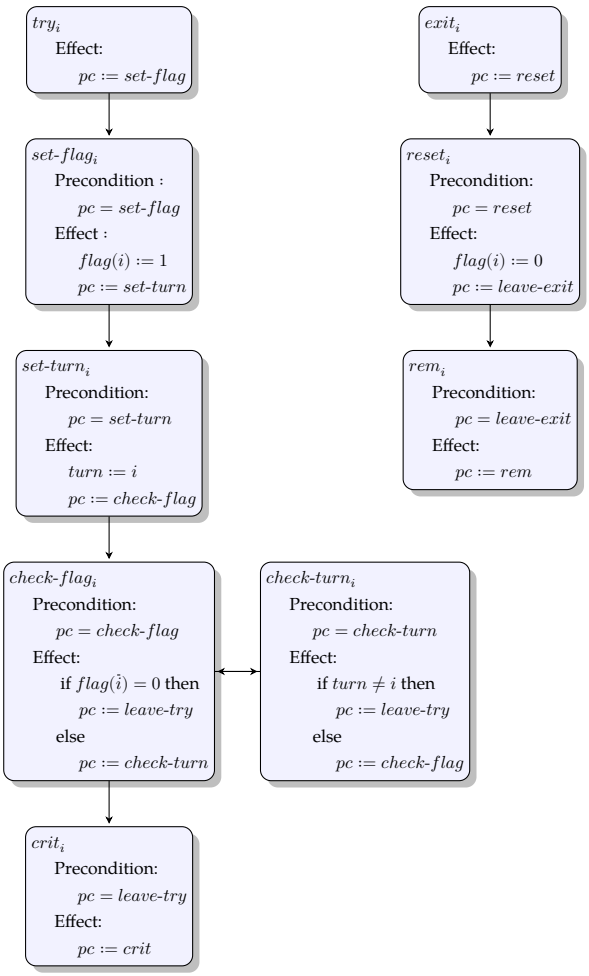
Algorithm 2:

Peterson2P algorithm (rewritten)

Shared variables:
$turn \in \{0, 1\}$, initially arbitrary
for every $i \in \{0, 1\}$:
    $flag(i) \in \{0, 1\}$, initially 0
Actions of i:

Input:	Internal:
try_i	$set\text{-}flag_i$
$exit_i$	$set\text{-}turn_i$
Output:	$check\text{-}flag_i$
$crit_i$	$check\text{-}turn_i$
rem_i	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem
Transitions of i:



Lemma 3.15. *Peterson2P satisfies mutual exclusion*

Proof.

Assertion 3.16. *In any reachable system state, if* $flag(i) = 0$*, then* $pc_i \in \{leave\text{-}exit, rem, set\text{-}flag\}$

Assertion 3.17. *In any reachable system state, if* $pc_i \in \{leave\text{-}try, crit, reset\}$*, and* $pc_{\bar{i}} \in \{check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset\}$*, then* $turn \neq i$*.*

That is, if i has won the competition, and if $\bar{i}$ is a competitor, then the $turn$ variable is set favorably for i.

Suppose both i and $\bar{i}$ are in C, then Assertion 3.17, applied twice for i and $\bar{i}$, implies that both $turn \neq i$ and $turn \neq \bar{i}$. □

Lemma 3.18. *Peterson2P guarantees progress.*

Proof. Suppose α is a low-level-fair execution that reaches a point where at least one of the processes, say i, is in T and neither process is in C, and suppose that after this point, neither process ever enters C.

1. If $\bar{i}$ is in T sometime after the given point in α, then both processes must get stuck in their $check$ loops, which is impossible
2. If $\bar{i}$ is never in T after the given point in α, we can show that $flag(\bar{i})$ eventually becomes and stays equal to 0.

□

Lemma 3.19. *Peterson2P is lockout-free*

Proof. Consider the lockout-freedom for trying region.

Suppose the contrary, that is, that at some point in execution α, process i is in T after having performed $set\text{-}flag_i$, and thereafter, while i remains in T, process $\bar{i}$ enters C three times.

Note that in each of the second and third times, it must be that $\bar{i}$ first sets $turn := \bar{i}$ and then sees $turn = i$; it cannot see $flag(i) = 0$. This means that there are at least two occurrences of $set\text{-}turn_i$ after the given point in α. But $set\text{-}turn_i$ is only performed once. □

Let l and c be upper bounds on process step time and critical section time, respectively.

Theorem 3.20. *In Peterson2P, the time from when a particular process i enters T until it enters C is most $c + O(l)$.*

3.5.2 An n-Process Algorithm

For n processes, we can use the idea of the *Peterson2P* algorithm iteratively, in a series of $n - 1$ competitions at levels $1, 2, \ldots, n - 1$. At each successive competition, the algorithm ensures that there is at least one **loser**. Thus, all n processes may compete in the level 1 competition, but at most $n - 1$ processes can win.

Shared variables:
for every $k \in \{1, \ldots, n-1\}$: $turn(k) \in \{1, \ldots, k\}$, initially arbitrary
for every i, $1 \leq i \leq n$: $flag(i) \in \{0, \ldots, n-1\}$, initially 0

Process i:

Remainder region

try_i;
for $k \in \{1, \ldots, n-1\}$ **do**
  $flag(i) := k$;
  $turn(k) := i$
  wait for $[\forall j \neq i : flag(j) < k]$ or $[turn(k) \neq i]$;
end

Critical region

$exit_i$;
$flag(i) := 0$;
rem_i;

Algorithm 3:

Ambiguities:

- one of the conditions in the waitfor statement involves the flag variables for all the other processes.
- we need to specify some conditions on the order in which process i checks the various $flag$ variables and the $turn(k)$ variable,

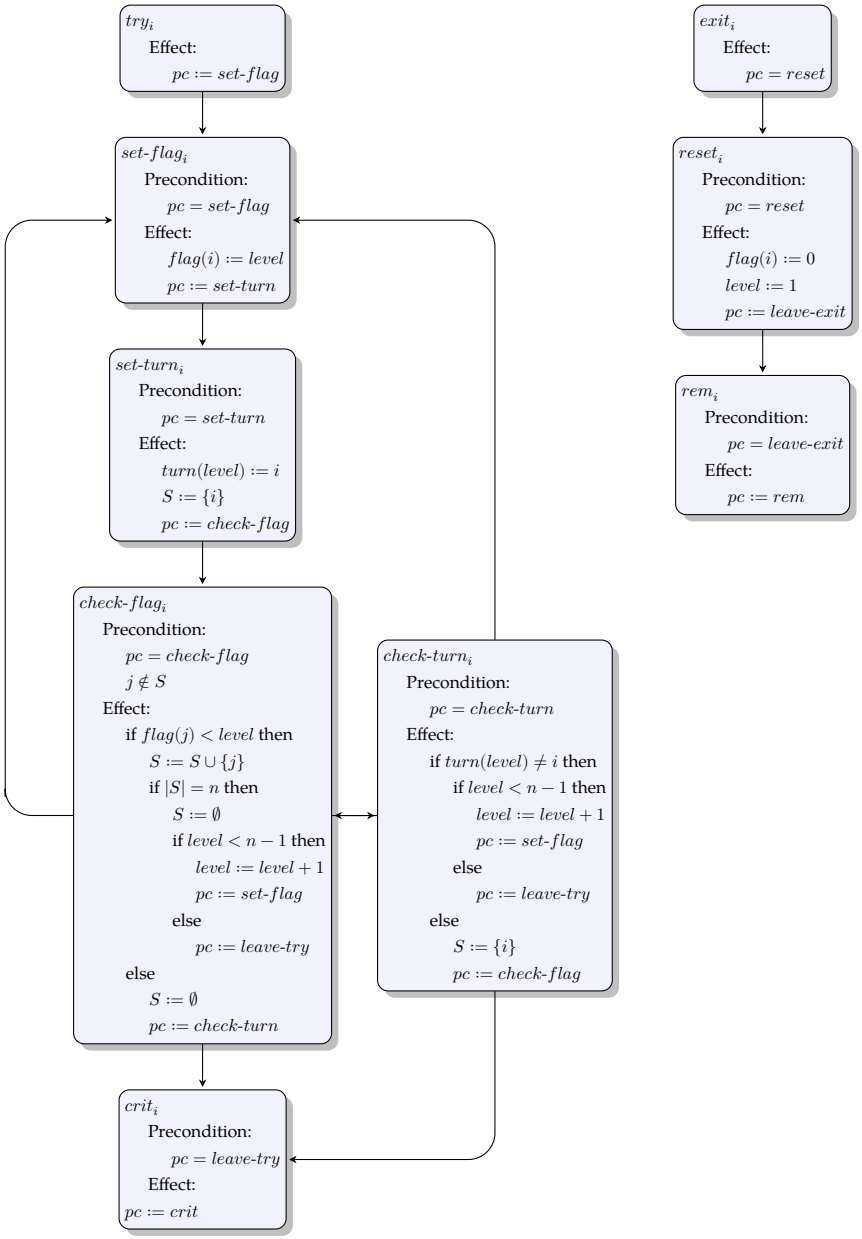
---

Shared variables:
for every $k \in \{1, \ldots, n-1\}$: $turn(k) \in \{1, \ldots, k\}$, initially arbitrary for every i, $1 \leq i \leq n$: $flag(i) \in \{0, \ldots, n-1\}$, initially 0
Actions of i:

Input:	Internal:
try_i	$set\text{-}flag_i$
$exit_i$	$set\text{-}turn_i$
Output:	$check\text{-}flag(j)_i$, $1 \leq j \leq n$, $j \neq i$
$crit_i$	$check\text{-}turn_i$
rem_i	$reset_i$

States of i:
$pc \in \{rem, set\text{-}flag, set\text{-}turn, check\text{-}flag, check\text{-}turn, leave\text{-}try, crit, reset, leave\text{-}exit\}$, initially rem
$level \in \{1, \ldots, n-1\}$, initially 1
S, a set of process indices, initially $\emptyset$

Transitions of i:



In any system state of *PetersonNP*, we say that a process i is a **winner** at level k provided that either $level_i > k$ or else $level_i = k$ and $pc_i \in$

$\{leave\text{-}try, crit, reset\}$. (The latter condition only arise for $k = n - 1$.) We also say that process i is a **competitor** at level k, provided that it is either a winner at level k or else $level_i = k$ and $pc_i \in \{check\text{-}flag, check\text{-}turn\}$.

Lemma 3.21. *PetersonNP satisfies mutual exclusion.*

Proof.

Assertion 3.22. *In any reachable system state of PetersonNP, the following are true:*

1. *If process i is a competitor at level k, if $pc_i = check\text{-}flag$, and if any process $j \neq i$ in S_i is a competitor at level k, then $turn(k) \neq i$*
2. *If process i is a winner at level k, and if any other process is a competitor at level k, then $turn(k) \neq i$.*

#+BEGIN$_{\text{proof}}$

1. If j is a winner,

□

#+END$_{\text{proof}}$

4 Q&A

1. 1.2.3. Need think.
2. 1.4.3