Abschnitt I

Asynchronous Shared Memory Model

**Inhalt:** specialization of I/O automata · processes and shared var’s · indistinguishable states · variable types · examples for variable types · behavior and composition for variable types

**Literatur:** The material is taken from [Lyn96, Chapter 9].
Intro

• ASMS ("asynchronous shared memory system") =
  – finite number of processes\(^1\)
  – communicating (internally) via shared variables

• port: interaction with the environment

• Cf. Figure 9.1, p 238

• modelling by I/O-automata:
  – one! big automaton per system.\(^2\)
  – rest is “convention/interpretation” and restriction on what the automaton is allowed to do

\(^1\)\(\neq\) automata
\(^2\)alternative are possible, but not more complex, if we want to stay in the chosen framework.
Asynchronous Shared Memory Model

Model: specifics for SM

• many things similar to before, but now one thinks of processes inside the automaton, providing some internal structure

• processes indexed 1, …, n,

• each states $states_i$, resp. start-state $start_i$,

• one shared var $x \Rightarrow$ value as state $values_x$, initially $initial_x$

• actions
  – each one associated with one of the processes
  – some of the internal actions may be (additionally) associated with a shared var
  – external actions (i/o) of process $i =$ communication “at port $i$”

• transitions $trans(A)$
– some **locality** restrictions, to reflect intended system structure

1. **processinternal** action: \((s, \pi, s')\), \(s, s' \in states_i\), non-trivial effect only for \(i\), rest unchanged

2. **process-variable** action:

3. effect:

\[(s, v), \pi, (s', v')\],

where \(s, s' \in states_i\), rest of the state-vector unchanged

* **enabledness** proviso: enabledness of a transition of \(i\) must depend only on the state of \(i\), not on the value of \(x\).

- **tasks:**

  – partitioning should be **consistent** with the process structure

  \(\Rightarrow\) each task (\(=\) eq. class) should include **locally controlled** actions of one process, only

  – often: 1 task per process (i.e., process is **sequential**
• $n$ processes, accessing one common shared var $x$

• the “first” process decides on the value

Signature:
input:
init($v_i$)
output:
decide($v_i$)
internal:
access$_i$

States of $i$
status $\in \{\text{idle, access, decide, done}\} = \text{idle}$
input : $V + \text{unknown} = \text{unknown}$
output : $V + \text{unknown} = \text{unknown}$

Transitions of $i$:
init($v_i$)
effect:
input := $v$
if status = idle
then status := access

access_i:
  precondition:
    status = access
  effect:
    if x = unknown then x := input
    output := x;
    status := decide

decide(v)_i
  precondition:
    status = decide
    output = v
  effect:
    status := done

• properties (informally)
  – liveness/progress/termination: decisions don’t take forever
  – agreement: decisions are consistent
  – validity: no trivial decisions are taken

• properties as trace properties, correctness claim: trace property \( P \) with \( \text{sig}(P) = \text{extsig}(A) \)
– if (Lynch: exactly) one \( \text{init}_i \) event appears in \( \beta \), then exactly one \( \text{decide}_i \) appears in \( \beta \) (for all \( i \))
– if no \( \text{init}_i \) appears in \( \beta \), then no \( \text{decide}_i \) event appears in \( \beta \) (for all \( i \))

**agreement** : if a \( \text{decide}(v)_i \) and a \( \text{decide}(w) \)-event appears in \( \beta \), then \( v = w \)

**validity** : if a \( \text{decide}_i(v) \)-event appears in \( \beta \), then some \( \text{init}_j(v) \) appears in

• Then

\[
[A]^{\text{fair}} \subseteq [P]^{\text{trace}}
\]
modeling the system environment explicitly as one or more I/O-automata

allows to specify assumptions about the environment by “programming” them

See Figure 9.2

Example 9.2.1:

– environment for the previous 1-variable process system
– one user automaton $U_i$ per system process $P_i$.
– each user process: request $\rightarrow$ wait $\rightarrow$ done (+ commonly-unreachable error-state, if unexpected decision comes)

$U_i$ automaton

Signature:

\[\text{alternative: logical description.}\]
Input: \( \text{decide}(v)_i, \ v \in V \)
Output: \( \text{init}(v)_i, \ v \in V \)
Internal: \( \text{dummy}_i \)

States:
- status: \{request, wait, done\} = request;
- decide: \( V + \text{unknown} = \text{unknown} \);
- error: \( \text{Bool} = \text{false} \);

Transitions:
- \( \text{init}(v)_i \):
  - precondition: status = request \lor error = true
  - effect: if error = false then status := wait
- \( \text{dummy}_i \):
  - precondition: error = true
  - effect: none;
- \( \text{decide}(v)_i \):
  - effect:
    - if error = false
    - then if status = wait
      - then decision := v
      - status := done
    - else error := true

Tasks: all locally controlled actions are in one class.
• cf. Figure 9.2, p 243

• properties: for every fair execution
  – there is exactly one $init_i$ and one $decide_i$-event
  – agreement, validity

• formally: trace property $Q$, over $\text{sig}(Q) = init, decide$

  termination $\beta$ contains exactly one $init_i$ event followed by exactly one $decide_i$ event.
  agreement if $decide(v)_i$ and $decide(w)_j$ both in $\beta$, then $v = w$
  validity if a $decide(v)_i$ occurs, then some $init(v)_j$ occurs in $\beta$

$$[A \times \prod U_i]^{fair} \subseteq [Q]^{trace}$$
Indistinguishable states

- useful for impossibility results later

- notion of “observability”: things “look” equal from a given perspective, e.g., from the perspective of one process/automaton (using projection)

- remember also indistinguishable executions for synchronous systems ($\sim_i$) and for I/O-automata, used (e.g.) for results in synchronous distributed consensus

- here: “observer” $i$ “sees”: his process’ + his user’s state + shared var’s (=all) it accesses

Definition 1. [Indistinguishable] Given states $s$ and $s'$ of system $A \times \prod U_i$. Then $s$ and $s'$ are indistinguishable to process $i$ ($s \sim_i s'$), if

1. state of process $i$, 


2. *the state of* $U_i$, *and*

3. *values of all shared var’s*

*are the same in* $s$ *and* $s'$.
Shared variable types (intro)

• so far: no restrictions on what is doable to a shared variable\(^4\)

• results depend on restrictions, for instance
  
  – write
  – read and give back
  – test
  – atomic combinations thereof

⇒ classification, shared var type

• note: not meant as restriction on the value domain,

• more: abstract data type/interface type intuition\(^5\)

\(^4\) except assumption of determinism.
\(^5\) another intuition could be objects with get- and set-methods (or other) + various “synchronization” disciplines.
Definition 2. [Variable type] A \textit{variable type} consists of

\begin{itemize}
  \item a set $V$ of \textit{values}
  \item an \textit{initial values} $v_0 \in V$
  \item set of \textit{invocations} and set of \textit{responses}
  \item a function:
  \[
  f : \text{invocations} \times V \rightarrow \text{responses} \times V
  \]
\end{itemize}
Asynchronous Shared Memory Model

Shared variable type in an automaton

- variable type $\neq$ I/O automaton
- “atomic” interaction: invocation and response at the same time = one! event
- shared variable $x$ of given type in a SMS $A$:
  - $values_x = V$
  - $initial_x = v_0$
  - transitions of $A$ wrt. $x$ must match the restrictions imposed by the var type
    * actions involving $x$ must be associated with one invocation $a$ of the var type.
    * describable$^6$ in a local guarded command style: given $p$ predicate on $state_i$ and $g \subseteq states_i \times responses \times states_i$

Transitions involving $i$ and $a$

Precondition: $p(state_i)$
Effect: $(b,x) := f(a,x)$  // effect as given by var type
       $state_i :=$  // $b = response$ => ‘‘sync.’’ with re
       any $s$ such that $(state_i,b,s) \in g$

$^6$In the examples: not necessarily explicitly.
Read/write variable

- most **common** variable type
- 2 **separate** interactions for reading and writing $\Rightarrow$ weak sync power
- **read/write** variable or (read/write) **register**
- arbitrary value domain, and one initial value

**interaction:**
- **invocations:** $\text{read}$, $\text{write}(v)$
- **responses:** $\text{ack}$

\[
\begin{align*}
\text{f(read, v)} & = (v, v) \\
\text{f(write(w), v)} & = (\text{ack}, w)
\end{align*}
\]
Asynchronous Shared Memory Model

- note: example on slide 5: not describable as register

States of \( i \)
- \( \text{status} \in \{\text{idle, access, decide, done}\} = \text{idle} \)
- \( \text{input} : V + \text{unknown} = \text{unknown} \)
- \( \text{output} : V + \text{unknown} = \text{unknown} \)

Transitions of \( i \):

- \( \text{init}(v)_i \)
  - effect:
    - \( \text{input} := v \)
    - if \( \text{status} = \text{idle} \) then \( \text{status} := \text{read} \)

- \( \text{read}_i \):
  - precondition: \( \text{status} = \text{read} \)
  - effect:
    - if \( x = \text{unknown} \) then \( \text{output} := \text{input} \)
    - \( \text{status} := \text{write} \)
    - else \( \text{output} := x \)
    - \( \text{status} := \text{decide} \)

- \( \text{write}(v)_i \):
  - precondition: \( \text{status} = \text{write} \)
    - \( v = \text{input} \)
  - effect:
    - \( x := v \)
    - \( \text{status} := \text{decide} \)

- \( \text{decide}(v)_i \)
Asynchronous Shared Memory Model

precondition: status = decide
output = v

effect: status := done
Read/write more explicitly

• given code not literally in the required form; conceptually it is (unlike representation on slide 5)

• for instance: \( \text{write}_i(v) \):
  
  – guard is \( \text{status} = \text{read} \)
  
  – effect \( g \subseteq \text{states}_i \times (V + \text{unknown}) \times V \), given by:

    \[
    \begin{align*}
    \text{if} & \quad b = \text{unknown} \\
    \text{then} & \quad \text{output} := \text{input} \\
            & \quad \text{status} := \text{write} \\
    \text{else} & \quad \text{output} := b \\
            & \quad \text{status} := \text{decide}
    \end{align*}
    \]

• for \( \text{write}(v)_i \)-action
  
  – guard-predicate \( p: \text{status} = \text{write} \)
  
  – effect \( g \) is the set of triples \( (s, b, s') \in \text{states}_i \times (V + \text{unknown}) \times V \), given by:
status := decide

- note: agreement does no longer hold!
Asynchronous Shared Memory Model

Read-modify-write

- another important, more sophisticated shared var type
- more powerful
- one instantaneous operation on $x$
  1. read $x$
  2. compute (depending on $x$): change own state and calculate value for $x$
  3. write $x$

- complex to implement on a multiprocessor architecture, not only atomic access ("mutex"), also fairness is required\(^7\)

- problem: how to model rmw-variable as variable type?

\(^7\) arbitration
• higher-order definition:
  – invocation: state-change function $h : V \rightarrow V$
  – response: value of variable
  – effect-function $f : (V \rightarrow V) \times V \rightarrow (V \times V)$

$$f(h, v) = (v, h(v))$$

Example 1. Cf. Example on slide 5:

$$h_v(x) = \begin{cases} v & \text{if } x = \text{unknown} \\ x & \text{otherwise} \end{cases}$$

• further variable types: special instances of read-modify-write
  – compare-and-swap
  – swap
  – test-and-set
  – fetch-and-add
Other variable types

- \textit{compare\_and\_swap}(u, v)

\[
f(\text{compare\_and\_swap}(u, v), w) = \begin{cases} 
(w, v) & \text{if } u = w \\
(w, w) & \text{otherwise}
\end{cases}
\]

- \textit{swap}(u)

\[
f(\text{swap}(u), v) = (v, u)
\]

- \textit{test\_and\_set}()

\[
f(\text{test\_and\_set}, v) = (v, 1)^8
\]

- \textit{fetch\_and\_add}(u)

\[
f(\text{fetch\_and\_add}(u), v) = (v, v + w)
\]

\[^8\text{assuming } 1 \in V.\]
 executions: as for I/O-automata: sequence of states and interface actions

finite or infinite

\[ v_0a_1b_1v_1a_2b_2 \ldots v_r \]
\[ v_0a_1b_1v_1a_2b_2 \ldots \]

- as specified by the automaton
  * \( v_0 \): initial value of the var type
  * \( (v_k, a_{k+1}, b_{k+1}v_{k+1} \) satisfy the functions of the var type:

\[ (b_{k+1}, v_{k+1}) = f(a_{k+1}, v_k) \]

traces: “interface” behavior: ignore the states, consider only the operations
Composition

- straightforward definition (interleaving)
- a countable collection \( \{ T_i \}_{i \in I} \): compatible, if the sets of invocations are disjoint, same for the responses

Definition 3. [Composition of variable types] Given \( \{ T_i \}_{i \in I} \) compatible. Then the composition \( T = \prod_{i \in I} T_i \) is defined by (as expected):

- \( V = \) cartesian product, initial value \( v_0 \) accordingly
- sets of invocations (resp. responses) is the —disjoint— union of the invocations (resp. responses) of the \( T_i \).
- effect-function: pointwise (but interleaving): assume, \( a \) is an invocation of \( i \), then \( f(a, w) \) is given by: apply \( f \) to the \( i \)th component of \( w \) \( \Rightarrow \) yields \( (b, v) \), then set \( i \)th component of \( w \) to \( v \)

\(^9\)No harm=synchronization done —except human confusion— if one’s var’s invocation matches another var’s response.
<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Then</td>
</tr>
</tbody>
</table>

Asynchronous Shared Memory Model
Complexity measures

- **time** complexity measure
  - special case of the definition of I/O automata
  - per task $C'$: upper bound $l \Rightarrow$ upper bound for time between successive chances by task $C'$ to perform a task
  - time until some event in $\pi = \supremum$ of times assignable to $\pi$ respecting the upper bounds; likewise time between events
  - not measured: “contention” time

- other potential (static) measures: number of shared vars, size of their value sets
Failures, randomization

- failures
  - remember: failures in synchronous network model
    * process failures: stopping, Byzantine
    * link failures: message loss
    * channels with “failures” in the asynchronous network model: losing, duplicating, reordering, (finite) duplication
  - just use the definition for I/O-automata:
    * probabilistic: transitions of the form $(s, \pi, P)$
    * non-deterministic: transitions of the form $(s, \pi, S)$. 
Literatur