Systems, Networks & Concurrency 2020

4

# Non-determinism

Uwe R. Zimmer - The Australian National University



## References for this chapter

**[Ben-Ari06]**
M. Ben-Ari
*Principles of Concurrent and Distributed Programming*
2006, second edition, Prentice-Hall, ISBN 0-13-711821-X

**[Barnes2006]**
Barnes, John
*Programming in Ada 2005*
Addison-Wesley, Pearson education, ISBN-13 978-0-321-34078-8, Harlow, England, 2006

**[AdaRM2012]**
*Ada Reference Manual - Language and Standard Libraries;*
ISO/IEC 8652:201x (E)

# Non-determinism

## *Definitions*

**Non-determinism** *by design*:

***A property of a computation which may have more than one result.***

**Non-determinism** *by interaction*:

***A property of the operation environment which may lead to different sequences of (concurrent) stimuli.***



## Non-determinism by design

Dijkstra's **guarded commands** (non-deterministic case statements):

```
if  x <= y -> m := x
❑   x >= y -> m := y
fi
```

Selection is non-deterministc for x=y

☞ The programmer needs to design the alternatives as 'parallel' options: all cases need to be covered and overlapping conditions need to lead to the same result

All true case statements in any language are potentially concurrent and non-deterministic.



## Non-determinism by design

Dijkstra's **guarded commands** (non-deterministic case statements):

```
if  x <= y -> m := x
❑   x >= y -> m := y
fi
```

Selection is non-deterministc for x=y

☞ The programmer needs to design the alternatives as 'parallel' options:
all cases need to be covered and overlapping conditions need to lead to the same result

All true case statements in any language are potentially concurrent and non-deterministic.

Numerical non-determinism in **concurrent statements** (Chapel):

```
writeln (* reduce [i in 1..10] exp (i));
writeln (+ reduce [i in 1..1000000] i ** 2.0);
```

Results may be non-deterministc depending on numeric type

☞ The programmer needs to understand the numerical implications of out-of-order expressions.

## Non-determinism by design

# Motivation for non-deterministic design

By explicitly leaving the sequence of evaluation or execution undetermined:

☞ The compiler / runtime environment can directly (i.e. without any analysis) translate the source code into a concurrent implementation.

☞ The implementation gains potentially significantly in performance

☞ The programmer does not need to handle any of the details of a concurrent implementation (access locks, messages, synchronizations, …)

A programming language which allows for those formulations is required!

☞ current language support: Ada, X10, Chapel, Fortress, Haskell, OCaml, …



## Non-determinism by interaction

# Selective waiting in Occam2

```
ALT
   Guard1
      Process1
   Guard2
      Process2
…
```

- Guards are referring to boolean expressions and/or channel input operations.
- The boolean expressions are local expressions, i.e. if none of them evaluates to true at the time of the evaluation of the `ALT`-statement, then the process is stopped.
- If all triggered channel input operations evaluate to false, the process is suspended until further activity on one of the named channels.
- Any Occam2 process can be employed in the `ALT`-statement
- The `ALT`-statement is non-deterministic (there is also a deterministic version: `PRI ALT`).

## *Non-determinism by interaction*

# *Selective waiting in Occam2*

```
ALT
  NumberInBuffer < Size & Append ? Buffer [Top]
    SEQ
      NumberInBuffer := NumberInBuffer + 1
      Top := (Top + 1) REM Size
  NumberInBuffer > 0 & Request ? ANY
    SEQ
      Take ! Buffer [Base]
      NumberInBuffer := NumberInBuffer - 1
      Base := (Base + 1) REM Size
```

- Synchronization on input-channels only (channels are directed in Occam2):

  ☞ to initiate the sending of data (`Take ! Buffer [Base]`),
  a request need to be made first which triggers the condition: (`Request ? ANY`)

CSP (Communicating Sequential Processes, Hoare 1978)
also supports non-deterministic selective waiting



## Non-determinism by interaction

# Select function in POSIX

```
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
                  const struct timespec *ntimeout, sigset_t *sigmask);
```

with:

- `n` being one more than the maximum of any file descriptor in any of the sets.
- after return the sets will have been reduced to the channels which have been triggered.
- the return value is used as success / failure indicator.

The POSIX select function implements parts of general selective waiting:

- `pselect` returns if one or multiple I/O channels have been triggered or an error occured.

¬ Branching into individual code sections is not provided.

¬ Guards are not provided.

After return it is required that the following code implements a *sequential* testing of *all* channels in the sets.

# Message-based selective synchronization in Ada

Forms of selective waiting:

```
select_statement ::= selective_accept        |
                     conditional_entry_call |
                     timed_entry_call       |
                     asynchronous_select
```

… underlying concept: Dijkstra's guarded commands

`selective_accept` implements …

- … wait for more than a single rendezvous at any one time
- … time-out if no rendezvous is forthcoming within a specified time
- … withdraw its offer to communicate if no rendezvous is available immediately
- … terminate if no clients can possibly call its entries



## Selective Synchronization

# Message-based selective synchronization in Ada

```
selective_accept ::= select
                          [guard] selective_accept_alternative
                   { or   [guard] selective_accept_alternative }
                   [ else sequence_of_statements ]
                     end select;

guard ::= when <condition> => selective_accept_alternative ::= accept_alternative |
                                                                delay_alternative  |
                                                                terminate_alternative

accept_alternative    ::= accept_statement [ sequence_of_statements ]
delay_alternative     ::= delay_statement  [ sequence_of_statements ]
terminate_alternative ::= terminate;

accept_statement ::= accept entry_direct_name [(entry_index)] parameter_profile [do
                        handled_sequence_of_statements
                     end [entry_identifier]];
delay_statement  ::= delay until delay_expression; | delay delay_expression;
```

# Basic forms of selective synchronization

(select-accept)

```
select
   accept …
or
   accept …
or
   accept …
…
end select;
```

- *If none of the entries have waiting calls*
  ☞ **the process is suspended** until a call arrives.
- *If exactly one of the entries has waiting calls*
  ☞ **this entry is selected**.
- *If multiple entries have waiting calls*
  ☞ **one of those is selected** (non-deterministically). The selection can be prioritized by means of the real-time-systems annex.

The code following the selected entry (if any) is executed and the `select` statement completes.

 

# Basic forms of selective synchronization

(select-guarded-accept)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
end select;
```

- *If all conditions are* ‘`true`’
  ☞ **identical to the previous form.**
- *If some condition evaluate to* ‘`true`’
  ☞ **the accept statement after those conditions are treated like in the previous form.**
- *If all conditions evaluate to* ‘`false`’
  ☞ `Program_Error` **is raised.**
  Hence it is important that the set of conditions covers all possible states.

This form is identical to Dijkstra’s guarded commands.

# Basic forms of selective synchronization

(select-guarded-accept-else)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
else
   <statements>
end select;
```

- *If all currently open entries have no waiting calls or all entries are closed*
  ☞ **The else alternative is chosen**, the associated statements executed and the select statement completes.
- Otherwise ☞ **one of the open entries with waiting calls is chosen as above.**

This form never suspends the task.

This enables a task to *withdraw* its offer to accept a set of calls if no tasks are currently waiting.

 



## Selective Synchronization

# Basic forms of selective synchronization

(select-guarded-accept-delay)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
or
   when <condition> => delay [until] …
      <statements>
or
   when <condition> => delay [until] …
      <statements>
…
end select;
```

- *If none of the open entries have waiting calls before the deadline specified by the earliest open* **delay** *alternative*
  ☞ **This earliest delay alternative is chosen** and the statements associated with it executed.
- *Otherwise* ☞ **one of the open entries with waiting calls is chosen as above.**

This enables a task to *withdraw* its offer to accept a set of calls if no other task is calling after some time.

## Selective Synchronization

# Basic forms of selective synchronization

(select-guarded-accept-terminate)

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
or
   when <condition> => terminate;
…
end select;
```

`terminate` cannot be mixed with `else` or `delay`

- *If none of the open entries have waiting calls and none of them can ever be called again*
  ☞ **The `terminate` alternative is chosen, i.e. the task is terminated.**

This situation occurs if:

☞ … all tasks which can possibly call on any of the open entries are terminated.

☞ or … all remaining tasks which can possibly call on any of the open entries are waiting on select-terminate statements themselves and none of their open entries can be called either. In this case all those waiting-for-termination tasks are terminated as well.

 

# *Message-based selective synchronization in Ada*

Forms of selective waiting:

```
select_statement ::= selective_accept       |
                     conditional_entry_call |
                     timed_entry_call       |
                     asynchronous_select
```

… underlying concept: Dijkstra's guarded commands

`conditional_entry_call` and `timed_entry_call` implements …

… the possibility to withdraw an outgoing call.

… this might be restricted if calls have already been partly processed.

 

## *Selective Synchronization*

# Conditional entry-calls

```
conditional_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   else
      sequence_of_statements
   end select;
```

Example:

```
select
   Light_Monitor.Wait_for_Light;
   Lux := True;
else
   Lux := False;
end;
```

- *If the call is not accepted immediately*
  ☞ **The `else` alternative is chosen.**

This is e.g. useful to probe the state of a server before committing to a potentially blocking call.

Even though it is tempting to use this statement in a “busy-waiting” semantic, there is usually no need to do so, as better alternatives are available.

There is only *one* entry-call and *one* `else` alternative.

 

## *Selective Synchronization*

# *Timed entry-calls*

```
timed_entry_call ::=
   select
      entry_call_statement
      [sequence_of_statements]
   or
      delay_alternative
   end select;
```

Example:

```
select
   Controller.Request (Some_Item);
   ------ process data
or
   delay 45.0; ------ seconds
   ------ try something else
end select;
```

- *If the call is not accepted before the deadline specified by the* `delay` *alternative*
  ☞ **The `delay` alternative is chosen.**

This is e.g. useful to withdraw an entry call after some specified time-out.

There is only *one* entry-call and *one* `delay` alternative.

 

## Selective Synchronization

# Message-based selective synchronization in Ada

Forms of selective waiting:

```
select_statement ::= selective_accept        |
                     conditional_entry_call |
                     timed_entry_call       |
                     asynchronous_select
```

… underlying concept: Dijkstra's guarded commands

`asynchronous_select` implements …

… the possibility to escape a running code block due to an event from outside this task.
(outside the scope of this course ☞ check: Real-Time Systems)



## Non-determinism

# *Sources of Non-determinism*

As concurrent entities are not in “lockstep” synchronization, they “overtake” each other and arrive at synchronization points in non-deterministic order, due to (just a few):

- Operating systems / runtime environments:
  - ☞ Schedulers are often non-deterministic.
  - ☞ System load will have an influence on concurrent execution.
  - ☞ Message passing systems react load depended.
- Networks & communication systems:
  - ☞ Traffic will arrive in an unpredictable way (non-deterministic).
  - ☞ Communication systems congestions are generally unpredictable.
- Computing hardware:
  - ☞ Timers drift and clocks have granularities.
  - ☞ Processors have out-of-order units.
- … basically: **Physical systems** (and **computer systems connected to the physical world**) are *intrinsically non-deterministic*.

## Non-determinism

# Correctness of non-deterministic programs

**Partial correctness**:

$$(P(I) \land terminates(Program(I,O))) \Rightarrow Q(I,O)$$

**Total correctness**:

$$P(I) \Rightarrow (terminates(Program(I,O)) \land Q(I,O))$$

**Safety properties**:

$$(P(I) \land Processes(I,S)) \Rightarrow \Box Q(I,S)$$

where $\Box Q$ means that $Q$ does *always* hold

**Liveness properties**:

$$(P(I) \land Processes(I,S)) \Rightarrow \Diamond Q(I,S)$$

where $\Diamond Q$ means that $Q$ does *eventually* hold (and will then stay true)
and $S$ is the current state of the concurrent system

## Non-determinism

# *Correctness of non-deterministic programs*

☞ Correctness predicates need to hold true *irrespective* of the actual sequence of interaction points.

or

☞ Correctness predicates need to hold true *for all possible* sequences of interaction points.

Therefore correctness predicates need to be based on **invariants**,
i.e. **invariant** predicates which are *independent* of the potential execution sequences,
*yet* support the overall correctness predicates.

## Non-determinism

# Correctness of non-deterministic programs

For example (in verbal form):
“Mutual exclusion accessing a specific resource holds true,
*for all possible* numbers, sequences or interleavings of requests to it”

An **invariant** would for instance be that the number of writing tasks inside a protected object is less or equal to one.

☞ Those **invariants** are the only practical way to guarantee (in a logical sense) correctness in concurrent / non-deterministic systems.

(as enumerating all possible cases and proving them individually is in general not feasible)

 

## Non-determinism

# Correctness of non-deterministic programs

```
select
   when <condition> => accept …
or
   when <condition> => accept …
or
   when <condition> => accept …
…
end select;
```

Concrete:

- ☞ Every time you formulate a non-determinstic statement like the one on the left you need to formulate an **invariant** which holds true whichever alternative will actually be chosen.

This is very similar to finding **loop invariants** in sequential programs

## Summary

# Non-Determinism

- **Non-determimism by design:**
  - Benefits & considerations
- **Non-determinism by interaction:**
  - Selective synchronization
  - Selective accepts
  - Selective calls
- **Correctness of non-deterministic programs:**
  - Sources of non-determinism
  - Predicates & invariants