Non-Determinism

Non-determinism by design
Non-determinism by interaction
Non-Determinism

- 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: Guarded Commands

- Dijkstra’s Guarded Command Language (non-deterministic selection):
  
  \[
  \text{if } x \leq y \rightarrow m := x \\
  \mid x \geq y \rightarrow m := y \\
  \text{fi}
  \]

- Result is nondeterministic for \( x = y \)

- The programmer must design alternatives as ‘parallel’ options: all cases must be covered and overlapping conditions have same result

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

E. Dijkstra EWD472: Guarded commands, non-determinacy and formal derivation of programs
Non-Determinism by Design: Parallel Reduction

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

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

• Is reduction operation:
  – Commutative? \( x \otimes y = y \otimes x \)
  – Associative? \( (x \otimes y) \otimes z = x \otimes (y \otimes z) \)

• Programmers need to understand the numerical implications of out-of-order expressions
Non-Determinism by Design: Motivation

By explicitly leaving the sequence of evaluation or execution undetermined:

- Compiler / runtime environment can directly (i.e. without any analysis) translate source code into a concurrent implementation.
- Implementation may gain significantly in performance.
- Programmer does not need to handle details of concurrent implementation (access locks, messages, synchronizations, …)

A programming language which allows for those formulations is required!

Current language support: Ada, Chapel, X10, Fortran, Haskell, OCaml, …
Non-Determinism by Interaction (occam)

Selective waiting in occam2:

\[
\text{ALT} \\
\text{Guard1} \\
\text{Process1} \\
\text{Guard2} \\
\text{Process2}
\]

- Guards refer to boolean expressions and/or channel input operations
- Boolean expressions are local expressions; if none evaluates to true, the process is stopped
- If all triggered channel input operations evaluate to false, the process is suspended pending input on one of the named channels
- Any occam2 process can be employed in the ALT-statement
- Deterministic version: PRI ALT
Non-Determinism by Interaction (occam)

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

See also Communicating Sequential Processes (Hoare 1978)
Non-Determinism by Interaction (POSIX)

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

with:

- \( n \) = maximum of any file descriptor in any of the sets, plus one
- on return, fd_sets are reduced to the channels which were triggered

Implements some features of general selective waiting:

- returns if one or more I/O channels have been triggered or error occurred
- branching into individual code sections is not provided
- guards are not provided; after return, must test each channel in the read/write/exception sets
Message-Based Selective Synchronization in Ada

Forms of selective waiting:

\[
\text{select_statement} ::= \text{selective_accept} \mid \text{conditional_entry_call} \mid \text{timed_entry_call} \mid \text{asynchronous_select}
\]

underlying concept: Dijkstra’s guarded commands

selective_accept implements

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

Ada Reference Manual, Selective Accept
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)

```plaintext
select
  when <condition1> => accept ...
or
  when <condition2> => accept ...
or
  when <condition3> => accept ...
...
end select;
```

• If all conditions are ‘true’: identical to the previous form
• If some conditions evaluate to ‘true’: the guarded accept statements are treated as per select-accept
• If all conditions evaluate to ‘false’: Program_Error is raised.
  Hence it is important that the set of conditions covers all possible states
• Identical to Dijkstra’s guarded commands
Basic Forms of Selective Synchronization

(select-guarded-accept-else)

select
  when <condition1> => accept ... 
or
  when <condition2> => accept ... 
or
  when <condition3> => accept ... 
  ... 
else
  <statements>
end select;

• If all currently open entries have no waiting calls or all entries are closed: the else alternative is executed and the select statement completes
• Otherwise: one of the open entries with waiting calls is chosen as per select-guarded-accept.
• Does not suspend the task
• Enables a task to withdraw its offer to communicate if no task is currently waiting
Basic Forms of Selective Synchronization

(select-guarded-accept-delay)

```plaintext
select
  when <condition1> => accept ... 
or
  when <condition2> => accept ... 
  ... 
or
  when <condition> => delay [until] ... 
  <statements>
  or
  when <condition> => delay [until] ... 
  ... 
end select;
```

- If no open entries have waiting calls before deadline specified by earliest open delay alternative, the earliest delay alternative is executed
- Otherwise:
  one of the open entries with waiting calls is chosen as above
- Enables a task to withdraw its offer to communicate if no other task is calling after some time
Basic Forms of Selective Synchronization

(select-guarded-accept-terminate)

\[
\text{select} \\
\text{when } \langle \text{condition1} \rangle \Rightarrow \text{accept} \ldots \\
\text{or} \\
\text{when } \langle \text{condition2} \rangle \Rightarrow \text{accept} \ldots \\
\ldots \\
\text{or} \\
\text{when } \langle \text{condition} \rangle \Rightarrow \text{terminate}; \\
\text{end select;}
\]

- If none of the open entries have waiting calls and none of them can ever be called again, the task is terminated

This situation occurs if:
- … all tasks which can possibly call any open entries have terminated
- or … all remaining tasks which can possibly call on any of the open entries are waiting themselves

\text{terminate} cannot be mixed with \text{else} or \text{delay}
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` implement the possibility to withdraw an outgoing call (might be restricted if calls have already been partly processed)
Conditional Entry Calls

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

- If call is not accepted immediately, the else alternative is chosen
- Useful e.g. to probe the state of a server before committing to a potentially blocking call
- Although it is tempting to use this statement for “busy-waiting”, better alternatives are available
- Only one entry call and one else alternative
Timed Entry Calls

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

• If the call is not accepted before the deadline specified by the delay alternative, the delay alternative is chosen
• Useful to withdraw an entry call after some specified time-out
• Only one entry-call and one delay alternative
Message-Based Selective Synchronization in Ada

Forms of selective waiting:

\[
\text{select_statement ::= selective_accept |}
\]

\[
\text{conditional_entry_call |}
\]

\[
\text{timed_entry_call |}
\]

\[
\text{asynchronous_select}
\]

underlying concept: Dijkstra’s guarded commands

\text{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 - see Real-Time Systems)
Sources of Non-Determinism

As concurrent entities are not in “lockstep” synchronization, they arrive at synchronization points in non-deterministic order, due to e.g.:

• Operating systems / runtime environments:
  – Schedulers
  – Message passing systems

• Networks & communication systems:
  – Multiple routing paths
  – Communication systems congestion

• Computing hardware:
  – Timer drift and clocks granularity
  – Out-of-order execution

• … computer systems connected to the physical world are intrinsically non-deterministic.
Correctness of Non-Deterministic Programs

- Partial correctness:
  - \((P(I) \land \text{terminates (Program}(I,O))) \rightarrow Q(I,O)\)

- Total correctness:
  - \(P(I) \rightarrow (\text{terminates (Program}(I,O)) \land Q(I,O))\)

- \((P(I) \land \text{Processes}(I,S)) \rightarrow \square Q(I,S)\)

  where \(\square Q\) means that \(Q\) always holds and \(S\) is the current state of the concurrent system

- \((P(I) \land \text{Processes}(I,S)) \rightarrow \diamond Q(I,S)\)

  where \(\diamond Q\) means that \(Q\) eventually holds
Correctness of Non-Deterministic Programs

• Correctness predicates need to hold true \textit{irrespective} of the actual sequence of interaction points or 
  \textit{for all possible} sequences of interaction points.

• Therefore correctness predicates need to be based on \textit{invariants}, i.e. invariant predicates which are independent of the potential execution sequences, yet support the overall correctness predicates.
Correctness of Non-Deterministic Programs

- For example: “Mutual exclusion accessing a specific resource holds true, for all possible numbers, sequences or interleavings of requests to it”
- E.g. invariant: the number of writing tasks inside a protected object is less or equal to one.
- Such invariants are the only practical way to guarantee correctness, as enumerating all possible cases and proving them individually is in general not feasible.
Correctness of Non-Deterministic Programs

```
select
  when <condition1> => accept ...
or
  when <condition2> => accept ...
or
  when <condition3> => accept ...
...
end select;
```

Concretely:

- Whenever you use a non-deterministic statement like the one on the left you need to formulate an invariant which holds regardless of which alternative is actually chosen
- Similar to finding loop invariants in sequential programs