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.

References for this chapter

[Ben-Ari06] M. Ben-Ari
Principles of Concurrent and Distributed Programming

[Barnes2006] Barnes, John
Programming in Ada 2005

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

```plaintext
if x <= y -> m := x
elseif x > y -> m := y
fi
```

Selection is non-deterministic 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

Non-determinism by design

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

\[
\text{if } x \leq y \rightarrow m := x \\
\text{else } x \geq y \rightarrow m := y
\]

- Guards refer 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

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

- 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 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

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

- 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

Selective Synchronization

Message-based selective synchronization in Ada

```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;
```

- 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 select-ed entry (if any) is executed and the select statement completes.
Selective Synchronization

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.

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-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-terminate)

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 all currently 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.
- ... 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.
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

```
conditional_entry_call ::= select
entry_call_statement
[sequence_of_statements]
else
sequence_of_statements
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 immediately
The delay 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.

Timed entry-calls

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

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

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 outgoing call after some specified time-out.

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

... the possibility to withdraw an outgoing call and one delay alternative.
Correctness of non-deterministic programs

Partial correctness:
\[ (P(I) \land \text{terminates(Program(I,O))}) \Rightarrow Q(I,O) \]

Total correctness:
\[ P(I) \Rightarrow (\text{terminates(Program(I,O))} \land Q(I,O)) \]

Safety properties:
\[ (P(I) \land \text{Processes}(I,S)) \Rightarrow \square Q(I,S) \]
where \( \square Q \) means that \( Q \) does always hold

Liveness properties:
\[ (P(I) \land \text{Processes}(I,S)) \Rightarrow \diamond Q(I,S) \]
where \( \diamond Q \) means that \( Q \) does eventually hold (and will then stay true)
and \( S \) is the current state of the concurrent system

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.

Correctness of non-deterministic programs

\[ \Rightarrow \] Correctness predicates need to hold true 
irrespective of the actual sequence of interaction points.

or

\[ \Rightarrow \] 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.

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)
Correctness of non-deterministic programs

Concrete:

- Every time you formulate a non-deterministic 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-determinism 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