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.

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

\[
\begin{align*}
\text{if } x \leq y \rightarrow m := x \\
\text{if } x \geq y \rightarrow m := y \\
\end{align*}
\]

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.

References for this chapter

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

[Barnes2006]
Barnes, John
Programming in Ada 2005

[AdaRM2012]
ISO/IEC 8652:201x (E)
Non-determinism by design

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

```
if x <= y -> m := x
  Selection is non-deterministic for x=y
fi
```

- 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);
```

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

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).

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

Selective Synchronization

Message-based selective synchronization in Ada

```ada
select
  [guard] selective_accept_alternative
  [or] [guard] selective_accept_alternative
[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;
```

Non-determinism

Selective Synchronization

Message-based selective synchronization in Ada

Forms of selective waiting:

```ada
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

Non-determinism by interaction

Select function in POSIX

```ada
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
const struct timespec *timeout, 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 occurred.
- 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.

Selective Synchronization

Basic forms of selective synchronization

```ada
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.
Selective Synchronization

Basic forms of selective synchronization
(select-guarded-accept)

```plaintext
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 evaluates to 'true': the accept statement after those conditions is 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-else)

```plaintext
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)

```plaintext
select
  when <condition> => accept ...
or
  when <condition> => accept ...
or
  when <condition> => accept ...
  ...or
  when <condition> => terminate;
  ...else <statements>
  end select;
```

- 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.
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);
else
    Timed Request);
end select;
```

- If the call is not accepted immediately or the else alternative is chosen.
  This is e.g. useful to withdraw an entry call after some specified time-out.
- The delay alternative is chosen.
  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
    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.

... underlying concept: Dijkstra's guarded commands

Conditional entry-calls

```
conditional_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 immediately or the else alternative is chosen.
- The delay alternative is chosen.

... the possibility to withdraw an outgoing call.
... this might be restricted if calls have already been partly processed.

Non-determinism

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-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 or the else alternative is chosen.
- The delay alternative is chosen.

... the possibility to withdraw an outgoing call.
... this might be restricted if calls have already been partly processed.

Non-determinism

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 or the else alternative is chosen.
- The delay alternative is chosen.

... the possibility to withdraw an outgoing call.
... this might be restricted if calls have already been partly processed.

Non-determinism

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 or the else alternative is chosen.
- The delay alternative is chosen.

... the possibility to withdraw an outgoing call.
... this might be restricted if calls have already been partly processed.

Non-determinism

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 or the else alternative is chosen.
- The delay alternative is chosen.

... the possibility to withdraw an outgoing call.
... this might be restricted if calls have already been partly processed.
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.
- Networks & communication systems:
  - Traffic will arrive in an unpredictable way (non-deterministic).
  - Message passing systems react load depended.
- 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

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

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

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

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

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

Concrete:

```
select
  when <condition> => accept _
  or when <condition> => accept _
  or when <condition> => accept _
  ....
end select;
```

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