Formal Methods – An Appetizer

Chapter 8: Concurrency

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

An Abstraction Mechanism

Concurrency

A system often consists of a number of processes that execute in parallel and interact with one another by exchanging information.

The processes may interact via shared variables.

The processes may communicate over channels.

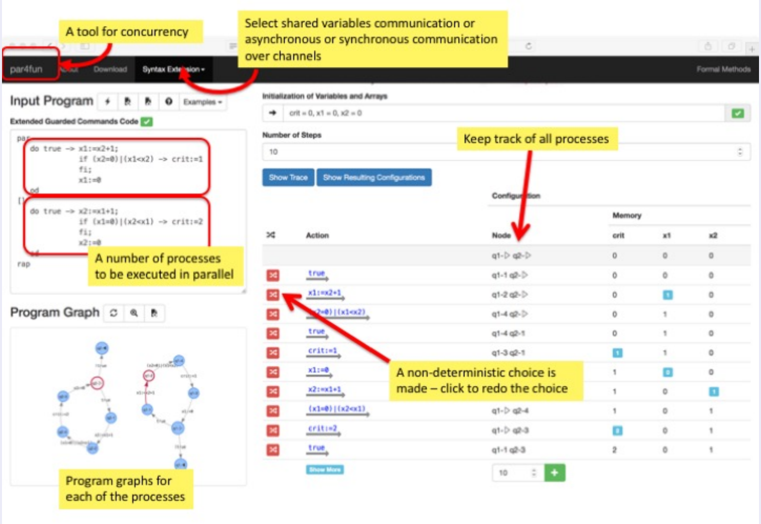
– asynchronous communication
– synchronous communication
– broadcast communication

Challenge

How does this change our notion of semantics?

The memory model is changed to record the configurations of all the processes (and channels).

FormalMethods.dk/par4fun



8.1 Shared Variables

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

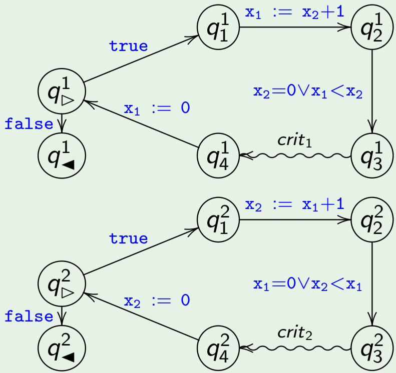
FormalMethods.dk

Example: Mutual Exclusion Algorithm

Bakery Algorithm (FM p 110)

```
par do true →
            x1 := x2 + 1 ;
            if x2 = 0 ∨ x1 < x2 →
                  critical section1
            fi ;
            x1 := 0
      od
[]    do true →
            x2 := x1 + 1 ;
            if x1 = 0 ∨ x2 < x1 →
                  critical section2
            fi ;
            x2 := 0
      od
rap
```

Two processes want to access a shared resource; however only one is allowed to do so at a time.



Semantics For Shared Variables

Program Graphs

We construct n disjoint program graphs, one for each process of `par` C_1 [] $\cdots$ [] C_n `rap`.

Configurations

We keep track of a node q_i for each process and the shared memory σ:

$$\langle q_1 \cdots q_n; \sigma \rangle$$

Execution Step (FM p 110)

Whenever $(q^i_\circ, \alpha, q^i_\bullet)$ is an edge in one of the program graphs then we have an execution step

$$\langle q^1 \cdots q^i_\circ \cdots q^n; \sigma \rangle \stackrel{\alpha}{\Longrightarrow} \langle q^1 \cdots q^i_\bullet \cdots q^n; \sigma' \rangle \quad \text{if } \mathcal{S}[\![\alpha]\!]\sigma = \sigma'$$

FormalMethods.dk/par4fun (FM p 110, 111)

Use the tool to construct an execution sequence for the Bakery Algorithm. Experiment with redoing some of the non-deterministic choices. Can you find an execution sequence where the values of the two variables $\mathtt{x}_1$ and $\mathtt{x}_2$ become arbitrarily large?

Hands On: Peterson's Algorithm

Peterson's Algorithm (FM p 112)

```
par do true →
        b₁ := 1; x := 2;
        if x = 1 ∨ b₂ = 0 →
             crit := 1
        fi;
        b₁ := 0
    od
 []  do true →
        b₂ := 1; x := 1;
        if x = 2 ∨ b₁ = 0 →
             crit := 2
        fi;
        b₂ := 0
    od
rap
```

FormalMethods.dk/par4fun (FM p 112)

Use the tool to construct an execution sequence for Peterson's algorithm.

Convince yourself that the two processes cannot be in a configuration where they both can update the variable crit.

Swop the first two assignments in each of the processes to have $x := 2; b_1 := 1$ and $x := 1; b_2 := 1$, respectively.

Use the tool to construct an execution sequence leading to a configuration where both processes can update the variable crit in their next step.

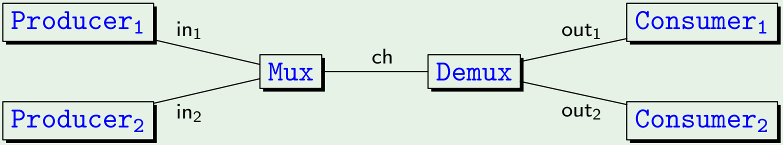
8.2 Asynchronous Communication

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

`FormalMethods.dk`

Channel Communication

Multiplexer Scenario



Output And Input

- $c!a$: outputs the value of a on the channel c.
- $c?x$: inputs a value on the channel c and assigns it to x.

Multiplexer Mux (FM p 112)

$$
\begin{array}{ll}
\texttt{loop} & \texttt{in}_1\texttt{?x} \rightarrow \\
& \quad \texttt{ch!}(2 * \texttt{x}) \\
[\,] & \texttt{in}_2\texttt{?x} \rightarrow \\
& \quad \texttt{ch!}(2 * \texttt{x} + 1) \\
\texttt{pool} &
\end{array}
$$

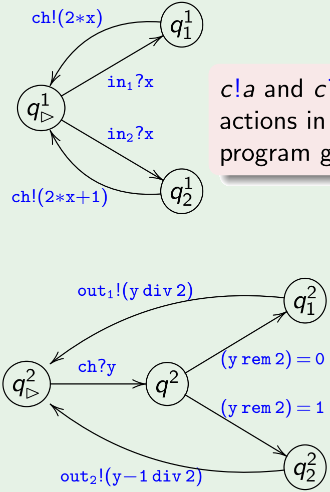
Output and input may be used as commands and as guards within a non-terminating version of the `do` $\cdots$ `od` command called `loop` $\cdots$ `pool`.

Program Graphs For Channel Communication

Guarded Command For Mux And Demux (FM p 112)

```
par
   loop in₁?x →
          ch!(2 * x)
   []   in₂?x →
          ch!(2 * x + 1)
   pool
[]
   loop ch?y →
         if (y rem 2) = 0 →
            out₁!(y div 2)
         [] (y rem 2) = 1 →
            out₂!(y − 1 div 2)
         fi
   pool
rap
```

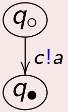
Program Graphs (FM p 113)



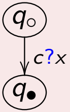
$c!a$ and $c?x$ are actions in the program graphs

Construction Of Program Graphs

$c!a$

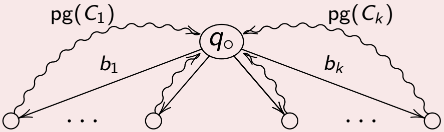


$c?x$

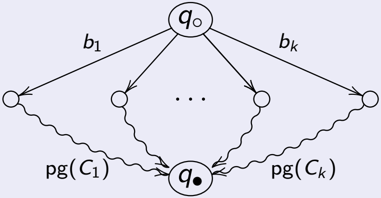


Output and input may only occur in guards in `if` ⋯ `fi` and `loop` ⋯ `pool` commands.

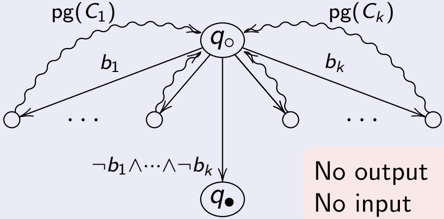
$\texttt{loop}\ b_1 \to C_1\ [] \cdots []\ b_k \to C_k\ \texttt{pool}$



$\texttt{if}\ b_1 \to C_1\ [] \cdots []\ b_k \to C_k\ \texttt{fi}$



$\texttt{do}\ b_1 \to C_1\ [] \cdots []\ b_k \to C_k\ \texttt{od}$



No output
No input

Asynchronous Semantics

Channels are Buffers

- $c!a$ places the value of a in the buffer for c
- $c?x$ obtains a value from the buffer of c (and assigns it to x)

The Semantic Domain (FM p 115)

- For variables:
  $\sigma \in \mathbf{Mem} = \mathbf{Var} \to \mathbf{Int}$
- For channels:
  $\kappa \in \mathbf{Buf} = \mathbf{Chan} \to \mathbf{Int}^*$

$\mathcal{S}_{\mathrm{B}}[\![\cdot]\!] : \mathbf{Act} \to (\mathbf{Mem} \times \mathbf{Buf} \hookrightarrow \mathbf{Mem} \times \mathbf{Buf})$

$$\mathcal{S}_{\mathrm{B}}[\![c!a]\!](\sigma, \kappa) = \begin{cases} (\sigma, \kappa[c \mapsto \vec{z} :: z']) & \text{if } z' = \mathcal{A}[\![a]\!]\sigma \text{ and } \kappa(c) = \vec{z} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathrm{B}}[\![c?x]\!](\sigma, \kappa) = \begin{cases} (\sigma[x \mapsto z'], \kappa[c \mapsto \vec{z}]) & \text{if } \kappa(c) = z' :: \vec{z} \text{ and } x \in \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathrm{B}}[\![x := a]\!](\sigma, \kappa) = \begin{cases} (\sigma[x \mapsto z], \kappa) & \text{if } z = \mathcal{A}[\![a]\!]\sigma \text{ and } x \in \mathrm{dom}(\sigma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Semantics For Asynchronous Communication

Program Graphs

We construct n disjoint program graphs, one for each process of $\texttt{par}\ C_1 \,[]\, \cdots \,[]\, C_n\ \texttt{rap}$.

Configurations (FM p 116)

We keep track of a node q_i for each process as well as the shared memory σ and the channel buffers κ:

$$\langle q_1 \cdots q_n; \sigma, \kappa\rangle$$

Execution Step

Whenever $(q^i_\circ, \alpha, q^i_\bullet)$ is an edge in one of the program graphs then we have an execution step

$$\langle q^1 \cdots q^i_\circ \cdots q^n; \sigma, \kappa\rangle \stackrel{\alpha}{\Longrightarrow} \langle q^1 \cdots q^i_\bullet \cdots q^n; \sigma', \kappa'\rangle$$

if $\mathcal{S}_{\mathrm{B}}[\![\alpha]\!](\sigma, \kappa) = (\sigma', \kappa')$

Alternative Semantics (FM p 117)

This semantics assumes that the buffers can have arbitrary size. An alternative is to put a positive bound k on the size of the buffers.

Hands On: Asynchronous Multiplexer Scenario

FormalMethods.dk/par4fun (FM p 116)

Use the tool to construct an execution sequence for the multiplexer scenario. Inially you may assume that the buffer for $\mathtt{in}_1$ is $[1, 2, 3]$ and that the one for $\mathtt{in}_2$ is $[2, 4, 6]$ while the other channels have empty buffers.

Producers

$$
\begin{array}{l}
\mathtt{loop}\ \mathtt{in}_1!\mathtt{u}_1 \rightarrow \\
\qquad\qquad \mathtt{u}_1 := \mathtt{u}_1 + 2 \\
\mathtt{pool}
\end{array}
$$

$$
\begin{array}{l}
\mathtt{loop}\ \mathtt{in}_2!\mathtt{u}_2 \rightarrow \\
\qquad\qquad \mathtt{u}_2 := \mathtt{u}_2 + 2 \\
\mathtt{pool}
\end{array}
$$

Extend the system with processes for the producers. Construct execution sequences from a memory where $\mathtt{u}_1$ and $\mathtt{u}_2$ are initialised to 1 and 2, respectively, and where all buffers are empty.

Experiment with different choices of buffer size; what differences do you observe?

Extend the system with processes for the consumers and repeat the above experiments.

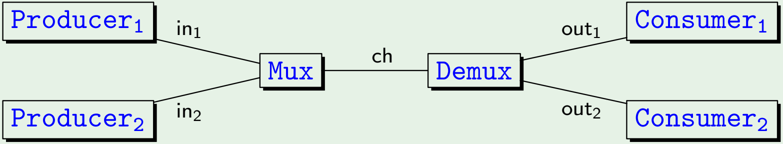
8.3 Synchronous Communication

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Synchronous Communication

Multiplexer Scenario



Another Communication Mechanism

If one process is ready to do an output and another process is ready to do an input over the same channel, then they can perform a joint action exchanging the value.

Changes To The Semantics

- We do not need the buffers
- We need a mechanism for two processes to perform a joint action

The syntax and the construction of program graphs are as before.
The configurations have the form $\langle q_1 \cdots q_n; \sigma \rangle$.

Hands On: Synchronous Multiplexer Scenario

Mux And Demux (FM p 117)

```
loop in₁?x →
        ch!(2 ∗ x)
[]   in₂?x →
        ch!(2 ∗ x + 1)
pool

loop ch?y →
      if (y rem 2) = 0 →
           out₁!(y div 2)
      [] (y rem 2) = 1 →
           out₂!(y − 1 div 2)
      fi
pool
```

Producers And Consumers ($i = 1, 2$)

```
loop inᵢ!uᵢ → uᵢ := uᵢ+1
pool

loop outᵢ?zᵢ → skip
pool
```

FormalMethods.dk/par4fun (FM p 118)

Use the tool with the synchronous semantics to construct execution sequences for the system.

Discuss the difference between the synchronous semantics and the asynchronous semantics where all buffers have length 1.

Synchronous Semantics

Individual Execution Step (FM p 118)

Whenever $(q^i_\circ, \alpha, q^i_\bullet)$ is an edge in one of the program graphs and α is of the form `skip`, $x := a$ or b then we have an execution step

$$\langle q^1 \cdots q^i_\circ \cdots q^n; \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle q^1 \cdots q^i_\bullet \cdots q^n; \sigma'\rangle \qquad \text{if } \mathcal{S}_{\mathrm{B}}[\![\alpha]\!](\sigma) = \sigma'$$

Synchronous Execution Step (FM p 118)

Whenever $(q^i_\circ, c!a, q^i_\bullet)$ and $(q^j_\circ, c?x, q^j_\bullet)$ are edges distinct program graphs (so $i \neq j$) then we have an execution step

$$\langle q^1 \cdots q^i_\circ \cdots q^j_\circ \cdots q^n; \sigma\rangle \overset{c!a?x}{\Longrightarrow} \langle q^1 \cdots q^i_\bullet \cdots q^j_\bullet \cdots q^n; \sigma'\rangle$$

if $\sigma' = \sigma[x \mapsto z]$ where $z = \mathcal{A}[\![a]\!]\sigma$

If one process is ready to do an output $c!a$ and another process is ready to do an input $c?x$ over the same channel c, then they can perform a joint action $c!a?x$ exchanging the value.

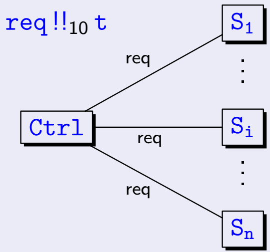
8.4 Broadcast and Gather

Flemming Nielson, Hanne Riis Nielson:
Formal Methods – An Appetizer.
ISBN 9783030051556, Springer 2019.

FormalMethods.dk

Broadcast And Gather

Generalisation to allow more than one sender and more than one receiver in communications.



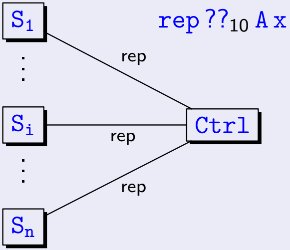
Broadcast $c\,!!_k\,a$ (FM p 119)

$c\,!!_k\,a$ outputs to all processes ready to do an input on c (using $c?$); at least k processes must be to do so as otherwise the action is stuck.

Gather $c\,??_k\,A\,x$ (FM p 119)

$c\,??_k\,A\,x$ inputs from all processes able to do an output on the channel c (using $c!$); at least k processes must be able to do so as otherwise the action is stuck.

The values received are placed in the array A; x equals the number of values received.



Example: A Controller And 16 Sensors (FM p 120)

A Scenario Exploiting Broadcast And Gather

A controller uses 16 sensors to measure temperature, pressure and humidity.

$$\texttt{par SENSOR}_0 \; [] \cdots [] \; \texttt{SENSOR}_{15} \; [] \; \texttt{CONTROL rap}$$

CONTROL

$$
\begin{array}{l}
\texttt{req}\,!!_{10}\,\texttt{temp}\,;\\
\texttt{rep}\,??_{10}\,\texttt{A x}\,;\\
\texttt{i} := 0\,;\\
\texttt{s} := 0\,;\\
\texttt{do i} < \texttt{x} \rightarrow\\
\quad \texttt{s} := \texttt{s} + \texttt{A[i]}\,;\\
\quad \texttt{i} := \texttt{i} + 1\\
\texttt{od}\,;\\
\texttt{avg} := \texttt{s}/\texttt{x}
\end{array}
$$

The controller requests at least 10 sensors to report their meassurements and computes the average of the (at least 10) responses.

$\texttt{SENSOR}_i$

$$
\begin{array}{l}
\texttt{loop}\\
\quad \texttt{req?t}_i \rightarrow \texttt{rep!Reg}_i\texttt{[t}_i\texttt{]}\\
\texttt{[]}\\
\quad \texttt{true} \rightarrow \cdots \texttt{update Reg}_i \cdots\\
\texttt{pool}
\end{array}
$$