Using the Pi-Calculus for Formalizing Workflow Patterns

Frank Puhlmann

Hasso-Plattner-Institute at the University of Potsdam
http://bpt.hpi.uni-potsdam.de

(Joint work with Mathias Weske)

Outline

- Motivation
- The Pi-Calculus
- Pattern Representation
  - ECA Mapping
  - Basic Control Flow Patterns
  - Advanced Workflow Pattern:
    - Discriminator
- Conclusion

Motivation

- The pi-calculus, a process algebra, has been discussed as the formal foundation for workflow (The Third Wave, PiHype)
- However, no formal investigations on the capabilities of the pi-calculus regarding the workflow domain have been made so far
- Task: Show the capabilities of the pi-calculus to describe the behavioral perspective of workflow
- Solution: Investigate the representation of Workflow Patterns in the pi-calculus

The Pi-Calculus

- The pi-calculus consists of names and processes:
  - Names represent existing concepts like *links*, pointers, references, identifiers, etc.
    - Each name has a scope
  - Processes are defined as:

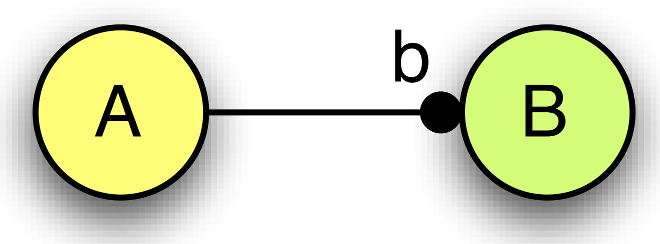
$$P ::= M \mid P|P' \mid \mathbf{v}zP \mid !P$$

  - The summations:

$$M ::= \mathbf{0} \mid \pi.P \mid M + M'$$

  - And the prefixes:

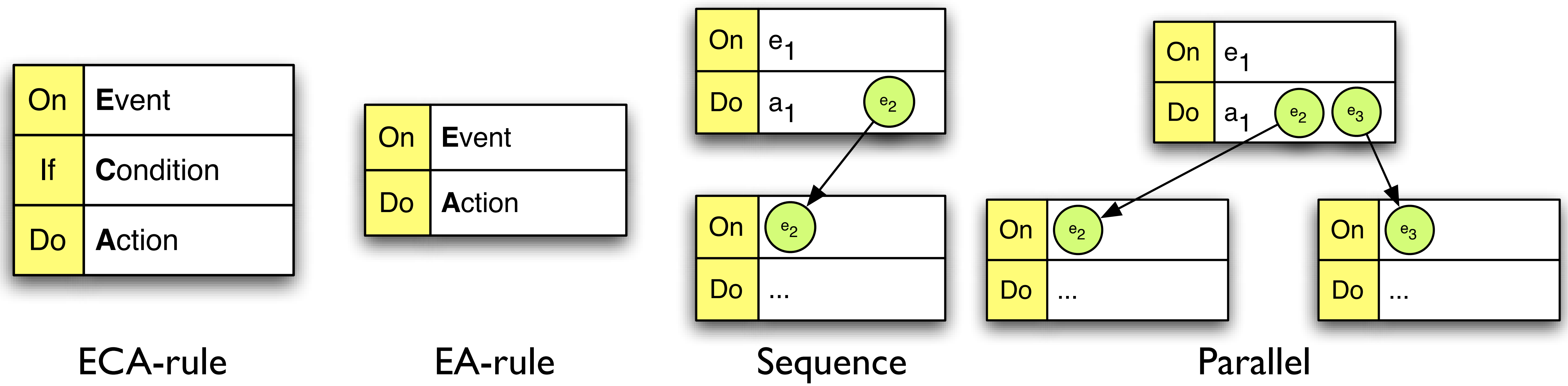
$$\pi ::= \overline{x}\langle y\rangle \mid x(z) \mid \tau \mid [x = y]\pi$$



$$A = \bar{b}\langle x\rangle.\mathbf{0} \quad B = b(x).\tau_B.\mathbf{0} \quad P = A|B$$

ECA Mapping

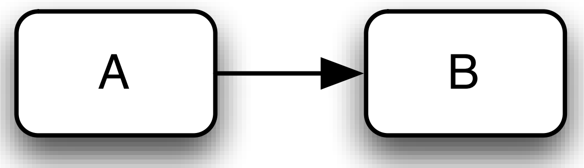
- Each workflow activity is mapped to a pi-calculus process with pre- and postconditions
- Based on the ECA approach:

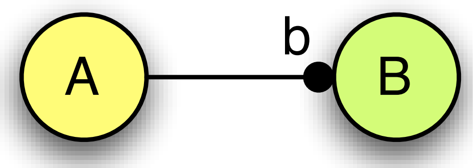


Precondition = Event and Condition	$x.[a = b]\tau.\overline{y}.\mathbf{0}$	Postcondition = Action

Basic Control Flow Patterns I

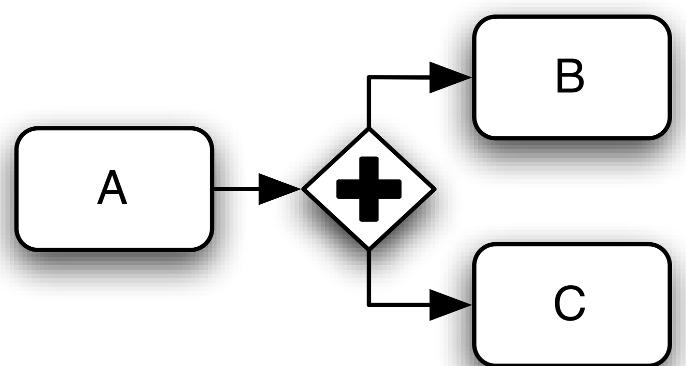
Sequence

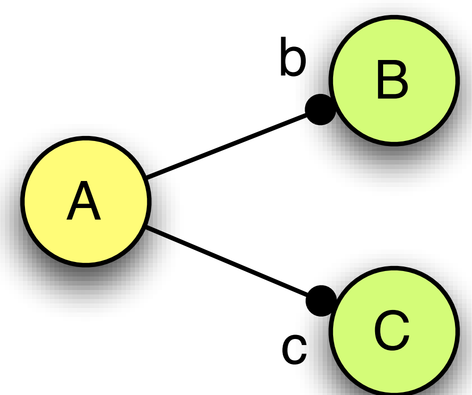




$$A = \tau_A.\overline{b}.\mathbf{0}$$
$$B = b.\tau_B.B'$$

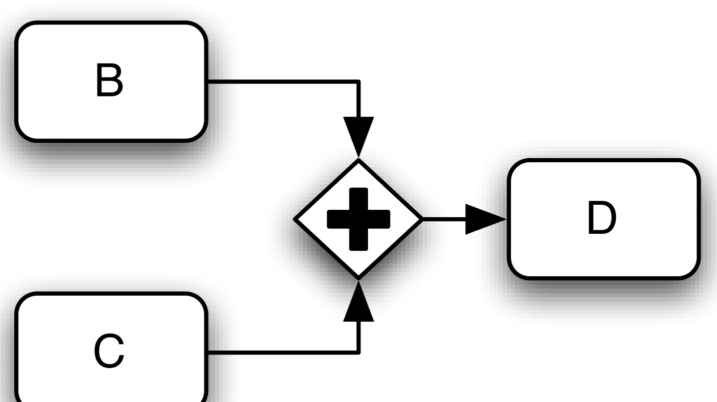
Parallel Split

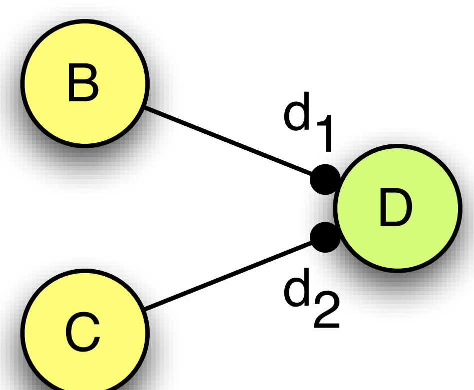




$$A = \tau_a.(\overline{b}.\mathbf{0}|\overline{c}.\mathbf{0})$$
$$B = b.\tau_B.B'$$
$$C = c.\tau_C.C'$$

Synchronization

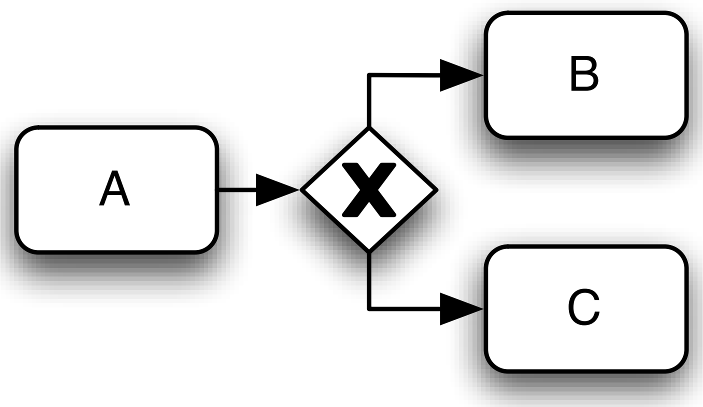


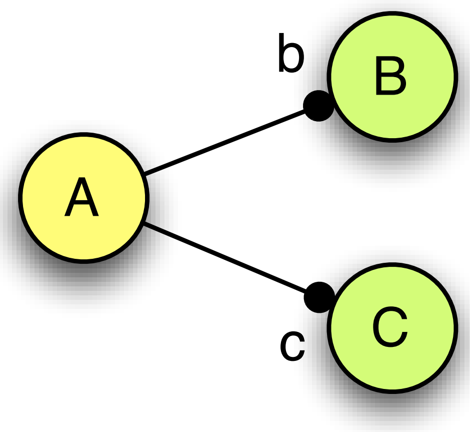


$$B = \tau_B.\overline{d_1}.\mathbf{0}$$
$$C = \tau_C.\overline{d_2}.\mathbf{0}$$
$$D = d_1.d_2.\tau_D.D'$$

Basic Control Flow Patterns 2

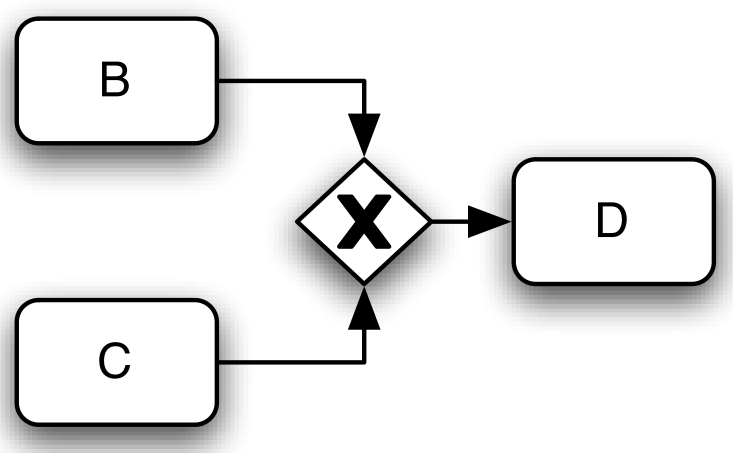
Exclusive Choice

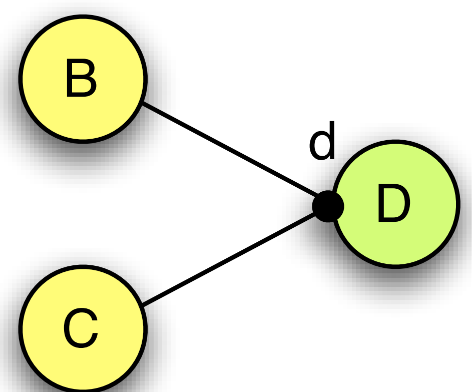




$$A = \tau_A.(\overline{b}.\mathbf{0} + \overline{c}.\mathbf{0})$$
$$B = b.\tau_B.B'$$
$$C = c.\tau_C.C'$$

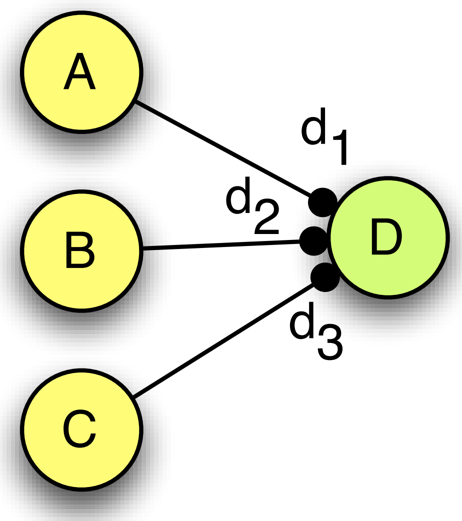
Simple Merge





$$B = \tau_B.\overline{d}.\mathbf{0}$$
$$C = \tau_C.\overline{d}.\mathbf{0}$$
$$D = d.\tau_D.D'$$

Discriminator



$$A = \tau_A.\overline{d1}.\mathbf{0} \quad B = \tau_B.\overline{d2}.\mathbf{0} \quad C = \tau_C.\overline{d3}.\mathbf{0}$$

$$D = (\mathbf{v}h, exec)(D_1|D_2)$$

$$D_1 = d_1.\overline{h}.\mathbf{0} \mid d_2.\overline{h}.\mathbf{0} \mid d_3.\overline{h}.\mathbf{0}$$

$$D_2 = h.\overline{exec}.h.h.D \mid exec.\tau_D.D'$$

Generic Discriminator (1-out-of-M-join):

$$D = (\mathbf{v}h, exec)((\prod_{i=1}^{m} d_i.\overline{h}.\mathbf{0}) \mid h.\overline{exec}.\{h\}_1^{m-1}.D \mid exec.\tau_D.D')$$

N-out-of-M-join:

$$D = (\mathbf{v}h, exec)((\prod_{i=1}^{m} d_i.\overline{h}.\mathbf{0}) \mid \{h\}_1^{n}.\overline{exec}.\{h\}_{n+1}^{m}.D \mid exec.\tau_D.D')$$

Conclusion

- Our work shows that all Workflow Patterns are directly representable in the pi-calculus
- Advanced features of the pi-calculus are required for the more complex patterns
- However, no new patterns have been found this time
- The results additionally provide a formal semantics for the Workflow Patterns, making them concise and unambiguous
- Future research can be based upon these results, e.g.
  - Formal foundation for (graphical) workflow notations
  - Tools for executing pi-calculus workflows

Further Information

- The interactive Pi-Workflow Website at:

  http://pi-workflow.org

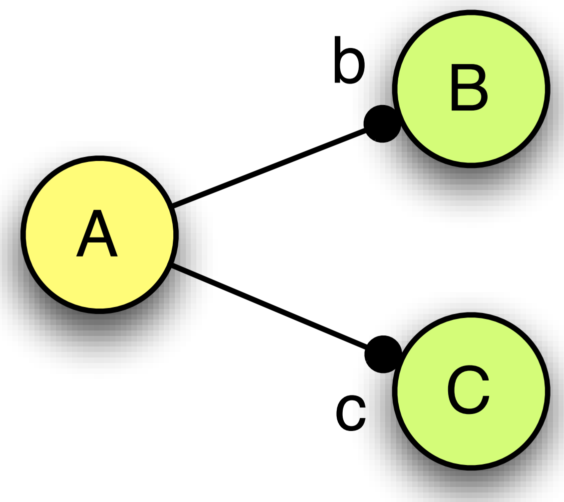
- Based on SnipSnap, a Wiki and Weblog software
- Everyone is invited to register and comment the content of the site
- The weblog will contain news about the research progress

Thank you!

Questions?

Multiple Choice

- How does the Multi-Choice Pattern work?



$$A = (\mathbf{v}exec)\tau_A.(A_1|A_2)$$

$$A_1 = \overline{exec}\langle b\rangle.\mathbf{0}+ \\ \overline{exec}\langle c\rangle.\mathbf{0}+ \\ \overline{exec}\langle b\rangle.\overline{exec}\langle c\rangle.\mathbf{0}$$

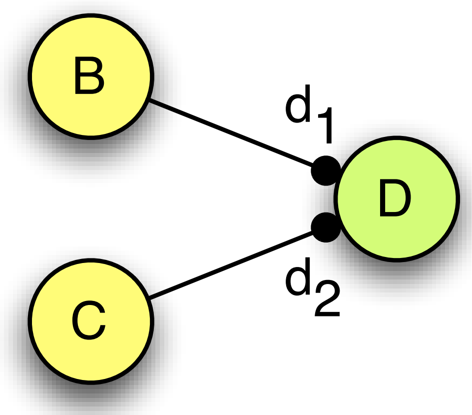
$$A_2 = !exec(x).\overline{x}.\mathbf{0}$$

$$B = b.\tau_B.B'$$

$$C = c.\tau_C.C'$$

OR-Joins

- How is the synchronizing merge handled?
- Pattern representation:

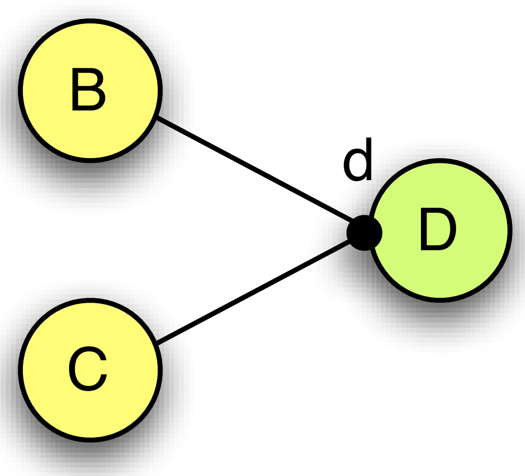


$$B = \tau_B.\overline{d_1}.\mathbf{0}$$
$$C = \tau_C.\overline{d_2}.\mathbf{0}$$
$$D = d_1.\tau_D.D' + d_2.\tau_D.D' + d_1.d_2.\tau_D.D'$$

- This pattern does not define how a runtime actually selects a summation; it just denotes all possibilities!
- A runtime could use different strategies, e.g.:
  - True/False token passing
  - Postphoned OR-join (YAWL like)

Multiple Merge

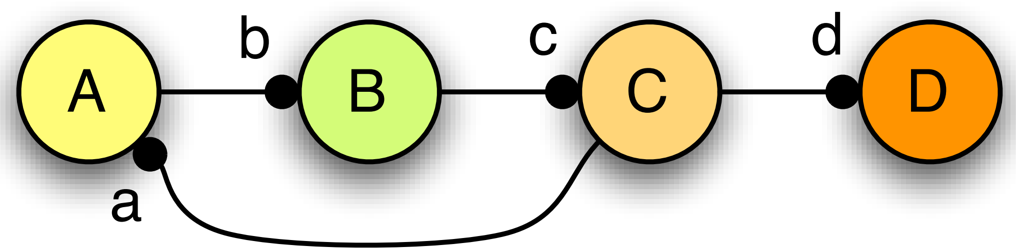
- How does the Multiple Merge Pattern work?



$$B = \tau_B.\overline{d}.\mathbf{0}$$
$$C = \tau_C.\overline{d}.\mathbf{0}$$
$$D = !d.\tau_D.D'$$

Arbitrary Cycles

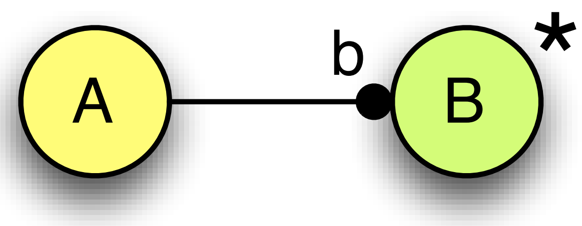
- How do arbitrary cycles work?



$$A = !a.\tau_A.\overline{b}.\mathbf{0}$$
$$B = !b.\tau_B.\overline{c}.\mathbf{0}$$
$$C = !c.\tau_C.(\overline{a}.\mathbf{0} + \overline{d}.\mathbf{0})$$
$$D = d.\tau_D.D'$$

MI without Synchronization

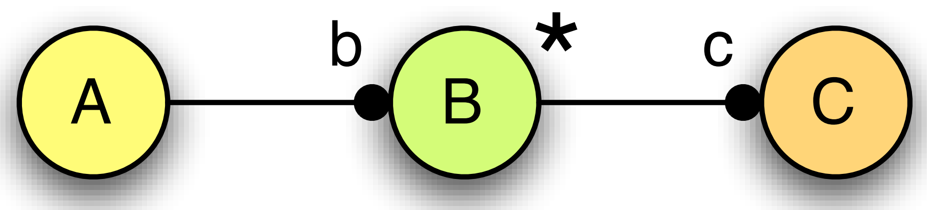
- Process A can spawn of any amount of multiple instances of a process B. No synchronization is required:



$$A = \tau_A.!\bar{b}.\mathbf{0}$$
$$B =!b.\tau_B.B'$$

MI with a priori Design Time Knowledge

- Process A spawns of a design time known number of instances of B that have to be synchronized afterwards:



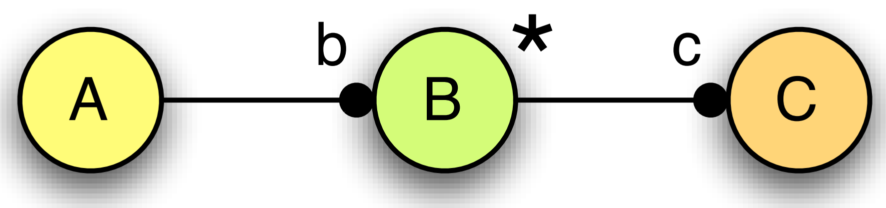
$$A = \tau_A.\overline{b}.\overline{b}.\overline{b}.\mathbf{0}$$
$$B = !b.\tau_B.\overline{c}.\mathbf{0}$$
$$C = c.c.c.\tau_C.C'$$

- For n design time copies the pattern is:

$$A \mid B \mid C \equiv \tau_A.\{\overline{b}\}_1^n.\mathbf{0} \mid !b.\tau_B.\overline{c}.\mathbf{0} \mid \{c\}_1^n.\tau_C.C'$$

MI with a priori Runtime Knowledge

- A process A can spawn of a runtime known number of instances of B that are started after all copies have been created. The copies of B have to be synchronized before another process C is activated:



$$A = (\mathbf{v}run)\tau_A.A_1(c) \mid run.!\overline{start}.\mathbf{0}$$

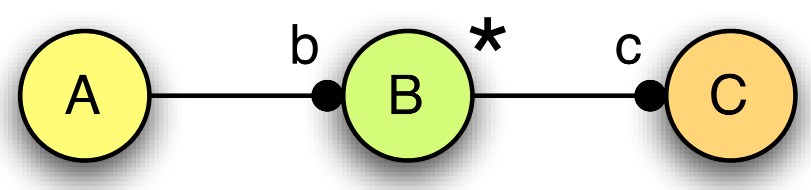
$$A_1(x) = (\mathbf{v}y)\bar{b}\langle y\rangle.y\langle x\rangle.A_1(y) + \overline{run}.\overline{x}.\mathbf{0}$$

$$B = !b(y).y(x).start.\tau_B.y.\overline{x}.\mathbf{0}$$

$$C = c.\tau_C.C'$$

MI with no priori knowledge

- Process A spawns multiple instances of another process B that have to be synchronized before the execution of C:



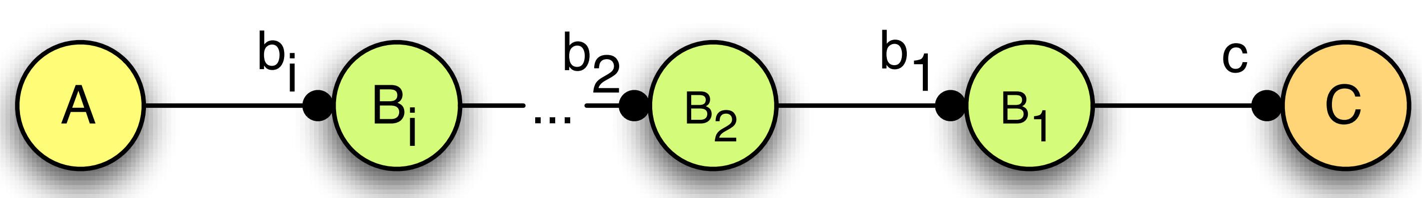
$$A = \tau_A.A_1(c)$$

$$A_1(x) = (\mathbf{v}y)\overline{b}\langle y\rangle.y\langle x\rangle.A_1(y) + \overline{x}.\mathbf{0}$$

$$B = !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0}$$

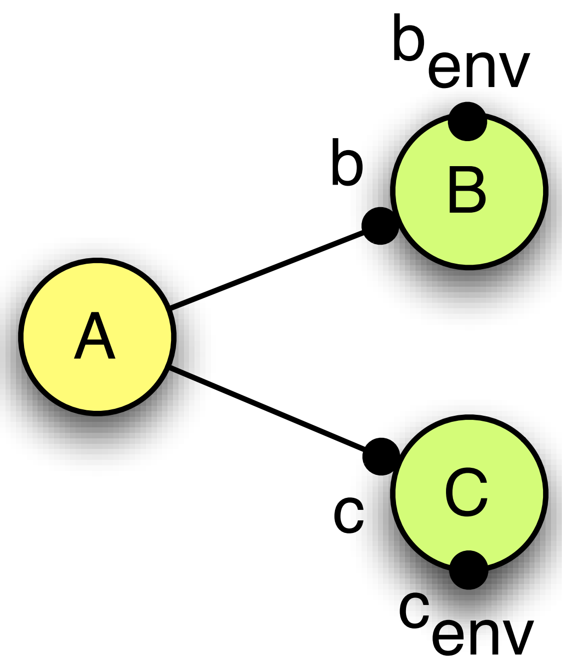
$$C = c.\tau_C.C'$$

The pattern works like a dynamic linked-list:



Deferred Choice

- How does the Deferred Choice Pattern work?



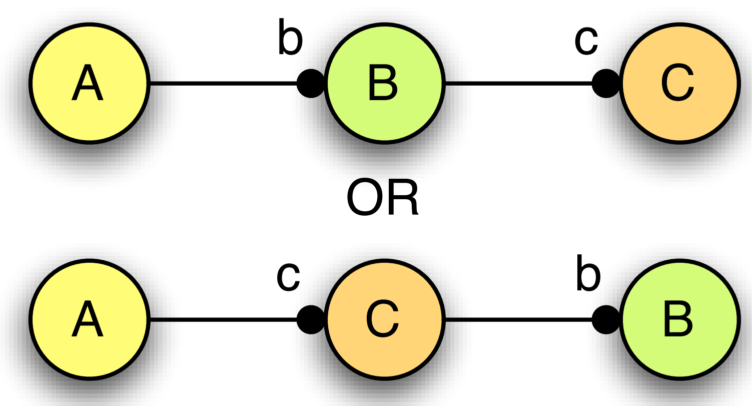
$$A = \tau_A.(\bar{b}.\mathbf{0} + \bar{c}.\mathbf{0})$$

$$B = b.(b_{env}.\overline{kill}.\tau_B.B' + kill.\mathbf{0})$$

$$C = c.(c_{env}.\overline{kill}.\tau_C.C' + kill.\mathbf{0})$$

Interleaved Parallel Routing

- How does the Interleaved Parallel Routing Pattern work?



$$A = \tau_A.\overline{x}.y.\overline{x}.y.A'$$
$$B = x.\tau_B.\overline{y}.\mathbf{0}$$
$$C = x.\tau_C.\overline{y}.\mathbf{0}$$

Milestone

- One possible representation of the Milestone Pattern:

$$\begin{aligned} A &= check(x).([x = \top]\tau_{A1}.A' + [x = \bot]\tau_{A2}.A'') \\ B &= M(\bot) \mid b.\overline{m}\left\langle \top \right\rangle .\tau_B.\overline{m}\left\langle \bot \right\rangle .B' \\ M(x) &= m(x).M(x) + \overline{check}\left\langle x \right\rangle .M(x) \end{aligned}$$

Cancel Patterns

- How does the Cancel Activity Pattern work?

$$A \mid \mathcal{E} \equiv a.\tau_A.A' + cancel.\mathbf{0} \mid !\tau_{\mathcal{E}}.\overline{cancel}.\mathbf{0}$$