Theoretical Foundations of Concurrency

A formal study of concurrency enables:
- understanding the essential nature of concurrency
- reasoning about the behavior of concurrent systems
- developing tools to aid in producing correct systems

The $\pi$-calculus of Robin Milner:
- an algebra (operators, expressions, reaction rules)
- an interpretation for communicating/mobile processes

The Structure of a Process

A process is a named entity possessing named ports through which it may communicate zero or more values with other processes.

In diagrams, a process is drawn as a circle. The name of the process is written inside the circle. The ports are represented by black dots drawn at the border of the circle.

Port names with overbars are often interpreted as “output” ports while names without overbars are often interpreted as “input” ports.

The process below is named “C” and has one input port, “in(x)”, and one output port, “$\overline{\text{out}}(x)$”. Both ports transmit a single value, x.
The behavior of a process is expressed by algebraic equations. Suppose that we want C to behave like a “cell” - it accepts a value at its input port and transmits that value at its output port. This is expressed as:

\[ C(\text{in}, \text{out}) = \text{in}(x).\text{out}<x>.0 \]

The dot ("."') is a prefix operation expressing sequential behavior.

The above equation is read as follows: process C has two ports, in and out; it accepts an input value at its port in, outputs that value at its port out, and then terminates (it acts like a process that takes no actions).

A Sequential Process

\[ \text{C}(\text{in}, \text{out}) = \text{in}(x).\text{out}<x>.0 \]

The behavior of a process is expressed by algebraic equations. Suppose that we want C to behave like a “cell” - it accepts a value at its input port and transmits that value at its output port. This is expressed as:

\[ C(\text{in}, \text{out}) = \text{in}(x).\text{out}<x>.0 \]

The dot ("."') is a prefix operation expressing sequential behavior.

The above equation is read as follows: process C has two ports, in and out; it accepts an input value at its port in, outputs that value at its port out, and then terminates (it acts like a process that takes no actions).

A Sequential Process

\[ \text{C}(\text{in}, \text{out}) = \text{in}(x).\text{out}<x>.0 \]

The behavior can also be written as

\[ \text{C}(\text{in}, \text{out}) = \text{in}(x).\text{C'}(\text{in, out, x}) \]
\[ \text{C'}(\text{in, out, x}) = \text{out}<x>.0 \]

In this case, process C accepts an input value at its port in and then acts like C' with that input value. Process C' transmits the value at its output port and then terminates.
A cell that continues to accept and send its current value can be expressed using a recursive style definition as:

\[ C(\text{in}, \text{out}) = \text{in}(x).\text{out}<x>.C(\text{in}, \text{out}) \]

The above equation is read as follows: process C has two ports, in and out; it accepts an input value at its port in, outputs that value at its port out, and then acts like process C.

---

The behavior

\[ C(\text{in}, \text{out}) = \text{in}(x).\overline{\text{out}<x>}.C(\text{in}, \text{out}) \]

can also be written as

\[ C'(\text{in}, \text{out}, x) = \text{in}(x).C'(\text{in}, \text{out}, x) \]

\[ C'(\text{in}, \text{out}, x) = \overline{\text{out}<x>}.C(\text{in}, \text{out}) \]
Using Ports for Synchronization

In addition to receiving a value, a port may also receive a synchronizing data-less signal. In this example, the const process returns a fixed value, but only when “asked”.

\[
\text{Const}(\text{query}, \text{reply}, c) = \text{query}.\text{reply}<c>.\text{Const}<\text{query}, \text{reply}, c>
\]

In this case, the query port transmits no data and can be written as:

\[
\text{Const}(\text{query}, \text{reply}, c) = \text{query}.\text{reply}<c>.\text{Const}<\text{query}, \text{reply}, c>
\]

Using Ports for Synchronization

An output port may transmit no data. For example, a cell that acknowledges the receipt of a new value may be written as:

\[
\text{C}(\text{in}, \text{ack}, \text{out}) = \text{in}(x).\text{ack}<x>.\text{out}<x>.\text{C}<\text{in}, \text{ack}, \text{out}>
\]

In this case, the ack port transmits no data and can be written as:

\[
\text{C}(\text{in}, \text{ack}, \text{out}) = \text{in}(x).\text{ack}.\text{out}<x>.\text{C}<\text{in}, \text{ack}, \text{out}>
\]
A Process with Alternative Behavior

A vending machine that dispenses chocolate candies allows either a 1p (p for pence) or a 2p coin to be inserted. After inserting a 1p coin, a button labelled “little” may be pressed and the machine will then dispense a small chocolate. After inserting a 2p coin, the “big” button may be pressed and the machine will then dispense a large chocolate. The candy must be collected before additional coins can be inserted.

\[
\text{big little}\quad 2p \quad 1p
\]

\[
\text{collect}
\]

\[
\text{VM(big, little, collect, 1p, 2p) = 2p.big.collect<largeChoc>.VM<big, little, collect, 1p, 2p> + 1p.little.collect<smallChoc>.VM<big, little, collect, 1p, 2p>}
\]

The plus (“+”) operator expresses alternative behavior.
Modeling a Bounded Buffer

Suppose that a buffer has get and put operations and can hold up to three data items. Ignoring the content of the data items, and focusing only on the operations, a buffer can be defined as:

\[
\begin{align*}
\text{Buffer}_0(\text{put, get}) &= \text{put.Buffer}_1(\text{put, get}) \\
\text{Buffer}_1(\text{put, get}) &= \text{put.Buffer}_2(\text{put, get}) + \text{get.Buffer}_0(\text{put, get}) \\
\text{Buffer}_2(\text{put, get}) &= \text{put.Buffer}_3(\text{put, get}) + \text{get.Buffer}_1(\text{put, get}) \\
\text{Buffer}_3(\text{put, get}) &= \text{get.Buffer}_2(\text{put, get})
\end{align*}
\]

Notice that this captures the idea that a get operation is not possible when the buffer is empty (i.e., in state Buffer_0) and a put operation is not possible when the buffer is full (i.e., in state Buffer_3).

Concurrent Processes

Processes can be composed, allowing them to communicate through ports with complementary names (i.e., one agent has an output port and the other has an input port with the same name).

Concurrent communicating agents can synchronize their behaviors through their willingness or unwillingness to communicate. This reflects a rendezvous style of interaction.

\[
\begin{align*}
\text{A}(a,c) &= a.A' <a,c> \\
\text{B}(c,b) &= c.B' <c,b>
\end{align*}
\]

\[
\begin{align*}
\text{A}'(a,c) &= c.A <a,c> \\
\text{B}'(c,b) &= b.B <c,b>
\end{align*}
\]

\[
\text{A system of processes: } \quad \text{System} = A \mid B
\]

The vertical bar is the composition operator.
Concurrent Processes

Processes:
\[ A(a,c) = a.A'_{a,c} \quad B(c,b) = c.B'_{c,b} \]
\[ A'(a,c) = c.A'_{a,c} \quad B'(c,b) = b.B'_{c,b} \]

A system:
\[ \text{System} = A \mid B \]

The behavior of this composite system of interacting agents is represented by the following transitions:
\[ ( A \mid B ) \xrightarrow{a} ( A' \mid B ) \xrightarrow{\tau} ( A \mid B' ) \xrightarrow{b} ( A \mid B ) \]

where the label “\( \tau \)” (pronounced “tau”) is used to denote the “handshake” communication between the two processes.

Encapsulating Interaction

Processes A and B can interact through their ports \( c \) and \( \bar{c} \). However, any other agents may also use these names to interact with either A or B. If such other interactions are not desired, then the names of the port may be encapsulated or restricted or given a scope that includes only agents A and B. This is done as follows:
\[ A(a,c) = a.A'_{a,c} \quad B(c,b) = c.B'_{c,b} \]
\[ A'(a,a) = c.A'_{a,c} \quad B'(c,b) = b.B'_{c,b} \]

An encapsulated system of agents:
\[ \text{System} = \text{new } c \ ( A \mid B ) \]

where “new” is the restriction operator.

Thus, the port names \( c \) and \( \bar{c} \) are no longer visible to other agents.
Encapsulating Interaction

To illustrate the effect of encapsulation, for this system:

\[
\begin{align*}
A(a,c) &= a.A' <a,c> & B(c,b) &= c.B' <c,b> \\
A'(a,c) &= \bar{c}.A <a,c> & B'(c,b) &= \bar{b}.B(c,b)
\end{align*}
\]

System = ( A | B )

A possible behavior is: \( \text{a.}\bar{c}.a.\bar{c}.b... \)

However the system: System = new c ( A | B )
has the behavior: \( \text{a.}\bar{c}.a.\bar{c}.a\bar{b}... \)

Reusing a Process Definition

\[
\begin{align*}
\text{CELL}(a,b) &= a.\bar{b} \cdot \text{CELL}<a,b> \\
C0 &= \text{CELL}(a, c) \\
C1 &= \text{CELL}(c, b) \\
\text{BUFF2} &= \text{new } c \ ( C0 \ | \ C1 ) \\
C0 &= \text{CELL} (a,c) \\
C1 &= \text{CELL} (c,d) \\
C2 &= \text{CELL} (d,b) \\
\text{BUFF3} &= \text{new } c,d \ ( C0 \ | \ C1 \ | \ C2 )
\end{align*}
\]
Next Steps

• Using $\pi$-calculus to model interesting systems?
• Using $\pi$-calculus to represent “specifications” and “implementations”
• Representing the “behavior” of a system in a way that it can be examined by a tool
• Proving that this representation is correct (equivalently, precisely defining the semantics of the $\pi$-calculus operators)
• Showing the equality between two agents for:
  • substitutability (can one implementation be replaced with another without changing the behavior of the overall system?)
  • satisfaction (does an implementation behavior according to its specification)

---

Modeling Mutual Exclusion

A lock to control access to a critical region is modeled by:

\[
\begin{align*}
\text{Lock} & \left(\text{lock, unlock}\right) = \text{lock.Locked}\left(\text{lock, unlock}\right) \\
\text{Locked} & \left(\text{lock, unlock}\right) = \text{unlock.Lock}\left(\text{lock, unlock}\right)
\end{align*}
\]

A generic process with a critical region follows the locking protocol is:

\[
\begin{align*}
\text{Process(enter, exit, lock, unlock)} & = \text{lock.enter.exit.unlock.Process(enter, exit, lock, unlock)}
\end{align*}
\]

A system of two processes is:

\[
\begin{align*}
\text{Process}_1 & = \text{Process}\left(\text{enter}_1, \text{exit}_1, \text{lock, unlock}\right) \\
\text{Process}_2 & = \text{Process}\left(\text{enter}_2, \text{exit}_2, \text{lock, unlock}\right) \\
\text{MutexSystem} & = \text{new lock, unlock (Process}_1 | \text{Process}_2 | \text{Lock})
\end{align*}
\]
Modeling Mutual Exclusion

A system of two processes is:

\[
\text{Process}_1 = \text{Process} \langle \text{enter}_1, \text{exit}_1, \text{lock}, \text{unlock} \rangle \\
\text{Process}_2 = \text{Process} \langle \text{enter}_2, \text{exit}_2, \text{lock}, \text{unlock} \rangle \\
\text{MutexSystem} = \text{new lock, unlock} (\text{Process}_1 \mid \text{Process}_2 \mid \text{Lock})
\]

A “specification” for this system is:

\[
\text{MutexSpec(enter}_1, \text{exit}_1, \text{enter}_2, \text{exit}_2) = \text{enter}_1.\text{exit}_1.\text{MutexSpec}(\text{enter}_1, \text{exit}_1, \text{enter}_2, \text{exit}_2) + \text{enter}_2.\text{exit}_2.\text{MutexSpec}(\text{enter}_1, \text{exit}_1, \text{enter}_2, \text{exit}_2)
\]

Mobility

Mobility in the \( \pi \)-calculus:

- refers to dynamic change in the communication topology among processes
- is accomplished by a process acquiring and losing ports through which it may communicate with other processes
- is realized by transmitting the name of a port as the value of some communication between two processes allowing the transmitted port to be known to the receiving process
Mobility

\[ A(x,y) = x.A(x,y) + \overline{y}(x).A'(y) \]
\[ B(y) = y(z).\overline{z}.B'(y,z) \]
\[ A'(y) = \ldots \]
\[ B'(y,z) = \ldots \]

\[ (A \mid B \mid C) = (x.A(x,y) + \overline{y}(x).A'(y) \mid y(z).\overline{z}.B'(y,z) \mid C) \]
\[ = (A'(y) \mid \overline{x}.B'(y,x) \mid C) \]
The transmitters can be modeled by:

\[
\text{Trans}(\text{talk}, \text{switch}, \text{gain}, \text{lose}) = \text{talk}.\text{Trans}<\text{talk}, \text{switch}, \text{gain}, \text{lose}>
+ \text{lose}(t, s).\text{switch}<t, s>.\text{Idtrans}<\text{gain}, \text{lose}>
\]

\[
\text{Idtrans}(\text{gain}, \text{lose}) = \text{gain}(t, s).\text{Trans}<t, s, \text{gain}, \text{lose}>
\]

The car’s behavior can be described as:

\[
\text{Car}(\text{talk}, \text{switch}) = \text{talk}.\text{Car}<\text{talk}, \text{switch}> + \text{switch}(t, s).\text{Car}<t, s>
\]
Mobility

A simple control system, that alternates between the two transmitters, is given by:

\[
\text{Control}_1 (\text{lose}_1, \text{lose}_2, \text{gain}_1, \text{gain}_2) = \text{lose}_1 <\text{talk}_2, \text{switch}_2> \text{gain}_2 <\text{talk}_2, \text{switch}_2> \text{Control}_2 (\text{lose}_1, \text{lose}_2, \text{gain}_1, \text{gain}_2) \\
\text{Control}_2 (\text{lose}_1, \text{lose}_2, \text{gain}_1, \text{gain}_2) = \text{lose}_2 <\text{talk}_1, \text{switch}_1> \text{gain}_1 <\text{talk}_1, \text{switch}_1> \text{Control}_1 (\text{lose}_1, \text{lose}_2, \text{gain}_1, \text{gain}_2)
\]

The mobile transmission system is:

\[
\text{System}_1 = \text{new talk}_1, \text{switch}_1, \text{gain}_1, \text{lose}_1, \text{talk}_2, \text{switch}_2, \text{gain}_2, \text{lose}_2 \\
\text{(Car<talk}_1, \text{switch}_1> \mid \text{Trans}_1 \mid \text{Idtrans}_1 \mid \text{Control}_1, \text{lose}_1, \text{lose}_2, \text{gain}_1, \text{gain}_2>)
\]

where \(\text{Trans}_1 = \text{Trans<talk}_1, \text{switch}_1, \text{gain}_1, \text{lose}_1>\) \\
\(\text{Idtrans}_1 = \text{Idtrans<gain}_1, \text{lose}_1>\)

Socket Protocol

(see handout for equations)
Modeling a Bounded Buffer

It was seen earlier how to model the operations of a bounded buffer as follows:

\[
\begin{align*}
\text{Buffer}_0\text{ (put, get)} &= \text{put}.\text{Buffer}_1 \langle\text{put, get}\rangle \\
\text{Buffer}_1\text{ (put, get)} &= \text{put}.\text{Buffer}_2 \langle\text{put, get}\rangle + \text{get}.\text{Buffer}_0 \langle\text{put, get}\rangle \\
\text{Buffer}_2\text{ (put, get)} &= \text{put}.\text{Buffer}_3 \langle\text{put, get}\rangle + \text{get}.\text{Buffer}_1 \langle\text{put, get}\rangle \\
\text{Buffer}_3\text{ (put, get)} &= \text{get}.\text{Buffer}_2 \langle\text{put, get}\rangle
\end{align*}
\]

This may be viewed as the “specification” of a correct bounded buffer.

Modeling a Bounded Buffer

The Buffer equations might be thought of as the “specification” of the bounded buffer because it only refers to states of the buffer and not to any internal components or machinery to create these states.

An “implementation” of the bounded buffer is readily available by re-labeling the BUFF3 agent developed earlier:

\[
\begin{align*}
\text{CELL} &= \text{a.b.CELL} \\
\text{C0} &= \text{CELL} \langle\text{put, c}\rangle \\
\text{C1} &= \text{CELL} \langle\text{c, d}\rangle \\
\text{C2} &= \text{CELL} \langle\text{d, get}\rangle \\
\text{BufferImpl} &= \text{new c, d ( C0 | C1 | C2 )}
\end{align*}
\]
Equality of Processes

We would like to know if two processes have the same behavior (interchangeable), or if an implementation has the behavior required by a given specification (conformance). For example:

\[
\begin{align*}
\text{is } & \text{ Buffer}_0 = \text{ BufferImpl } ? \\
\text{is } & \text{ MutexSystem = MutexSpec } ?
\end{align*}
\]

How do we tell if two behaviors are the same?

Structural Congruence

Two expressions are the same if one can be transformed to the other using these rules:

1. change of bound names: new a (a.P) = new c (c.P)
3. \( P | 0 = P, \ P | Q = Q | P, \ P | (Q | R) = (P | Q) | R \)
4. new x (P | Q) = P | new x Q if x is not a free name in P, new x 0 = 0, new x.y P = new y.x P

Example:

\[
\begin{align*}
P &= \text{new } z \ ( (\overline{x}<y> + z(w),\overline{w}<y> ) | x(u),\overline{u}<v> | \overline{x}<z> ) \\
   &= \text{new } z \ (x(u),\overline{u}<v> | \overline{x}<y> + z(w),\overline{w}<y> ) | \overline{x}<z> ) \text{ by (3)} \\
   &= x(u),\overline{u}<v> | \text{new } z \ ( (\overline{x}<y> + z(w),\overline{w}<y> ) | \overline{x}<z> ) \text{ by (4)}
\end{align*}
\]
An equation can be changed by the application of these rules that express the "reaction" of the system being described:

**TAU:** \[ \tau \cdot P + M \rightarrow P \]

**REACT:** \[ (x(y) \cdot P + M) \mid \overline{x} < z > \cdot Q + N) \rightarrow (z/y) \cdot P \mid Q \]

**PAR:** \[ \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \]

**RES:** \[ \frac{P \rightarrow P'}{\text{new } x \cdot P \rightarrow \text{new } x \cdot P'} \]

**STRUCT:** \[ \frac{P \rightarrow P'}{Q \rightarrow Q'} \quad \text{if } P = Q \text{ and } P' = Q' \]

Processes: \[ A(a,c) = a \cdot A' < a, c > \quad B(c,b) = c \cdot B' < c, b > \]

A system: \[ \text{System} = \text{new } c \cdot (A \mid B) \]

Show: new \( c \cdot (A' \mid B) \rightarrow \text{new } c(A \mid B') \)

by **REACT:** \[ \overline{c} \cdot A \mid c \cdot B' \rightarrow A \mid B' \]

by **RES:** \[ \text{new } c(\overline{c} \cdot A \mid c \cdot B') \rightarrow \text{new } c \cdot (A \mid B') \]

by definition: \[ \text{new } c(A' \mid B) \rightarrow \text{new } c(A \mid B') \]
Depicting an Agent’s Behavior

Recall:

- \( A = a.A' \)
- \( B = c.B' \)
- \( A' = \overline{c}.A \)
- \( B' = \overline{b}.B \)
- System = new \( c \ ( A \ | \ B ) \)

Draw a graph to show all possible sequences of actions. Here is the start:

More of the Behavior

\[
\begin{array}{c}
(A|B) \\
\downarrow a \\
(A|B) \\
\downarrow \tau \\
(A|B') \\
\end{array}
\]

\[
\begin{array}{c}
(A|B) \\
\downarrow a \\
(A|B) \\
\downarrow \tau \\
(A|B') \\
\end{array}
\]

\[
\begin{array}{c}
(A|B) \\
\downarrow b \\
(A|B) \\
\end{array}
\]

\[
\begin{array}{c}
(A|B) \\
\downarrow a \\
(A|B) \\
\end{array}
\]

\[
\begin{array}{c}
(A|B) \\
\downarrow b \\
(A|B) \\
\end{array}
\]
Depicting an Agent’s Behavior

\[ \begin{align*}
\bar{b} & \quad (A|\bar{B}) \\
& \quad \downarrow a \\
& \quad (A\bar{B}) \\
& \quad \downarrow \tau \\
& \quad (A|B') \\
& \quad \downarrow a \\
& \quad (A|B') \\
\end{align*} \]

Equivalence of Agents

\[ \begin{align*}
A & \equiv a.A_1 \\
A_1 & \equiv b.A_2 + c.A_3 \\
A_2 & \equiv 0 \\
A_3 & \equiv d.A \\
B & \equiv a.B_1 + a.B'_1 \\
B_1 & \equiv b.B_2 \\
B_2 & \equiv 0 \\
B_3 & \equiv d.B \\
B'_1 & \equiv c.B_3 \\
\end{align*} \]
Bisimulation

The behavior of two processes are equal when each can simulate exactly the behavior of the other.

I can do everything you can do!

P

Q

I can do everything you can do!

The Edinburgh Concurrency Workbench

The Edinburgh Concurrency Workbench
(Version 6.1, August, 1992)

(1) Command: bi Cell a.'b.Cell
(2) Command: bi C0 Cell[c/b]
(3) Command: bi C1 Cell[c/a,d/b]
(4) Command: bi C2 Cell[d/a]
(5) Command: bi Buff3 (C0 | C1 | C2){c,d}
(6) Command: bi Spec a.Spec'
(8) Command: bi Spec'' 'b.Spec' + a.'b.Spec''
(9) Command: eq
(9a) Agent: Buff3
(9b) Agent: Spec
(9c) true
(10) Command: quit

Figure 1: Sample session 1
Welcome to the Concurrency Workbench (Version 6.1)

* Assume we have the agents from Session 1

(1) Command: sort Buff3
(1_a) {a,'b}
(2) Command: size Buff3
(2_a) Buff3 has 11 states.
(3) Command: min Buff3
(3_a) Save result in identifier: Buff3Min
(3_b) Buff3Min has 4 states.
(4) Command: vs 3 Buff3Min
(4_a) === a a a ==> 
(4_b) === a 'b ==> 
(4_c) === 'b a ==> 
(5) Command: random 16 Buff3Min
(5_a) a,'b,a,'a,'b,'b,a,a,'b,'b,a,'b,a,a,'b
(6) Command: quit

Figure 2: Sample session 2