A Simple Agent

A CCS agent is described both by a structural diagram and one or more algebraic equations. The diagram is for readability and is not part of the formalism.

The structural diagram shows the agent as a circle. The name of the agent is written inside the circle.

At the borders of the circle are the ports through which the agent interacts with other agents. Ports are represented as black dots. Each port has a name. Names with overbars are often interpreted as “output” ports while names without overbars are often interpreted as “input” ports.

The agent pictured below is named “C” and has one input port, “in”, and one output port, “out”. What does C do? What is the behavior of C?

\[ \text{C} \]

\[ \text{in} \quad \text{C} \quad \text{out} \]

An Agent with Sequential Behavior

The behavior of an agent is expressed by algebraic equations. Suppose that we want C to behave so that it accepts a value at its input port and transmits that value at its output port. The corresponding equations are:

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

These equations use the CCS “prefix” operator represented by a dot (\( . \)). The operator expresses sequential behavior.

The above equations are read as follows: agent C accepts an input value at its port in and then acts like the agent C’ with that input value. Agent C’ transmits the value at its out port and then acts like agent C.

Later, the values will be dropped and only the port names will be important.

An Agent 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 are inserted.

\[ \text{big} \quad \text{little} \]

\[ 2p \quad 1p \]

\[ \text{collect} \]

\[ \text{VM} = 2p.\text{big}.\text{collect}.\text{VM} + 1p.\text{little}.\text{collect}.\text{VM} \]

Composing Agents

Agents are autonomous, free to take their actions whenever their behavior allows such actions. Agents can 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).

Composed and communicating agents can synchronize their behaviors through their willingness or unwillingness to communicate. This reflects a rendezvous style of interaction used in CSP and Ada.

Agents:

\[ A = a.A' \]
\[ B = c.B' \]
\[ A' = c.A \]
\[ B' = b.B \]

A system of agents:

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

The vertical bar is the composition operator.

Encapsulating Agents

The agents A and B can interact through their ports c and \( \overline{c} \). However, any other agents may also use these names an 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:

\[ A = a.A' \quad B = c.B' \]
\[ A' = \overline{c}.A \quad B' = \overline{c}.B \]

An encapsulated system of agents:

\[ \text{System} = ( A \mid B ) \mid \{ c \} \]

where “\( \{ \ldots \} \)” is the restriction operator.

Thus, the port names c and \( \overline{c} \) are no longer visible to other agents.
Reusing an Agent Definition

It is useful to be able to create a system by composing several agents that have the same behavior. To allow them interact it is necessary to change the names of their ports.

If $A$ is an agent the re-labeling operator

$A \left[ \text{new-name} / \text{old-name} \right]$ 

is an agent where all of the ports named old-name are changed to new-name and all ports named old-name are changed to new-name.

Note that the re-labeling applies to both barred and unbarred forms of the port name.

Next Steps

- Using CCS to model interesting systems?
- Using CCS 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 CCS 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:

$$\text{Lock} = \text{lock.LOCK}$$
$$\text{Locked} = \text{unlock.LOCK}$$

A generic process enters and exits the critical region and follows the locking protocol is:

$$\text{Process} = \text{lock.enter.exit.unlock.Process}$$

A system of two processes is:

$$\text{Process1} = \text{Process} \left[ \text{enter1} / \text{enter}, \text{exit1} / \text{exit} \right]$$
$$\text{Process2} = \text{Process} \left[ \text{enter2} / \text{enter}, \text{exit2} / \text{exit} \right]$$

$$\text{System} = (\text{Process1} | \text{Process2} | \text{Lock}) \setminus \{\text{lock, unlock}\}$$

A "specification" for this system is:

$$\text{SafeSpec} = \text{enter1.exit1.SafeSpec} + \text{enter2.exit2.SafeSpec}$$

Modeling a Bounded Buffer

Suppose that the buffer has get and put operations and can hold up to three messages.

- $\text{Buffer}_0 = \text{put.Buffer}_1$
- $\text{Buffer}_1 = \text{put.Buffer}_2 + \text{get.Buffer}_0$
- $\text{Buffer}_2 = \text{put.Buffer}_3 + \text{get.Buffer}_1$
- $\text{Buffer}_3 = \text{get.Buffer}_2$

It is sometimes useful to think of the “agent” as being $\text{Buffer}_1$ while the other equations define “states” of this agent. So, $\text{Buffer}_2$ might be thought of as the “state” of the buffer when there are two messages present.

Notice that this captures the idea that a get operation is not possible when the buffer is empty (i.e., in state $\text{Buffer}_0$) and a put operation is not possible when the buffer is full (i.e., in state $\text{Buffer}_3$).

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

$$\text{CELL} = \bar{a.b}.\text{CELL}$$
$$\text{C0} = \text{CELL} \left[ c / b \right]$$
$$\text{C1} = \text{CELL} \left[ c / a \right]$$
$$\text{BUFF2} = (\text{C0} | \text{C1}) \setminus \{c\}$$
$$\text{C0} = \text{CELL} \left[ c / b \right]$$
$$\text{C1} = \text{CELL} \left[ c / a, d / b \right]$$
$$\text{C2} = \text{CELL} \left[ d / a \right]$$
$$\text{BUFF3} = (\text{C0} | \text{C1} | \text{C2}) \setminus \{c, d\}$$

$\text{BufferImpl} = \text{BUFF3} \left[ \text{put} / a, \text{get} / b \right]$
Recall:

\[
\begin{align*}
A &= a.A' \\
B &= c.B' \\
A' &= \tau.A \\
B' &= \delta.B
\end{align*}
\]

System = \(( A \mid B ) \setminus \{ c \} \)

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

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

But how do we know (formally) that this graph correctly depicts the behavior of the agents?

Answer: the following axioms.

### The Axioms

- \( \text{Act} \) \( a, E \vdash_{E} a \)
- \( \text{Sum} \) \( a, E + b.0 \vdash_{E} E \)
- \( \text{Act} \) \( a, F \vdash_{E} a, E \)
- \( \text{Com} \) \( (a, E + b.0) \vdash_{E} a, F \)
- \( \text{Res} \) \( (a, E + b.0) \vdash_{E\setminus a} (E|F) \vdash_{a} (E|F) \)

### Equivalence of Agents

\[
\begin{align*}
A &\equiv a, A_1 \\
B &\equiv b, A_1 + b, B_1 \\
A_1 &\equiv b, A_2 + c, A_2 \\
B_1 &\equiv c, B_2 \\
A_2 &\equiv a, A \\
B_2 &\equiv b, B \\
\end{align*}
\]
Equivalence of Agents

\[ a \xrightarrow{\varepsilon} A \quad c \xrightarrow{\varepsilon} B \]

\[ A \equiv a.A' \quad B \equiv c.B' \]

\[ A' \equiv \varepsilon.A \quad B' \equiv \varepsilon.B \]

Now consider the composite agent \( A:B \):

\[ a \xrightarrow{\varepsilon} A \quad c \xrightarrow{\varepsilon} B \]

Equivalence of Agents

\[ \text{start} \rightarrow (A(B))c \rightarrow (A(B))c \rightarrow \tau \]

From this we can see that \((A(B))c\) is behaviourally equal to \(C_6\), where we define the agents \(C_0, \ldots, C_5\) by:

\[ C_0 \equiv \varepsilon.C_1 + a.C_4 \]

\[ C_1 \equiv a.C_2 \]

\[ C_2 \equiv \varepsilon.C_5 \]

\[ C_5 \equiv \varepsilon.C_6 \]

Alternatively, we may say that:

\[(A(B))c = a \times C, \text{ where } C \equiv a \times C + \varepsilon \times C\]

The Edinburgh Concurrency Workbench

Welcome to the Concurrency Workbench (Version 6.1)

(1) Command: sort Buff3
(2) (a, b)
(3) Command: size Buff3
(4) Buff3 has 11 states.
(5) Command: min Buff3
(6) BuffMin has 4 states.
(7) Command: we 3 BuffMin
(8) a >= a >=
(9) a + b =>=
(10) Command: quit

Figure 1: Sample session 1

The Edinburgh Concurrency Workbench

Welcome to the Concurrency Workbench (Version 6.1)

(1) Command: sort Buff3
(2) (a, b)
(3) Command: size Buff3
(4) Buff3 has 11 states.
(5) Command: min Buff3
(6) BuffMin has 4 states.
(7) Command: we 3 BuffMin
(8) a >= a >=
(9) a + b =>=
(10) Command: quit

Figure 2: Sample session 2