CCS goes .Net

IVNET’05
Instituto Superior de Engenharia do Porto (ISEP)
22 Junho 2005

Dep. Informática - Universidade do Minho
Braga, Portugal

Nuno F. Rodrigues - nfr@di.uminho.pt
Luis S. Barbosa – lsb@di.uminho.pt
Agenda

- CCS – An Overview
- Motivation
- Cw
- CCS to Cw
- Conclusions
Calculus of Communicating Systems (CCS) - An Overview

- A process Algebra
- Calculi to reason about concurrent systems
- Observational behaviour
- Observational equivalence, congruence, etc…
- Specific proof techniques
CCS – An Overview

- Operational semantics given by a labelled transition system specified by

\[
\begin{align*}
\alpha.E & \xrightarrow{\alpha} E \\
E & \xrightarrow{\alpha} E' \\
F & \xrightarrow{\alpha} F' \\
E + F & \xrightarrow{\alpha} E' \quad \text{or} \quad E + F \xrightarrow{\alpha} F'
\end{align*}
\]

\[
\begin{align*}
E & \xrightarrow{\alpha} E' \\
F & \xrightarrow{\alpha} F' \\
E \mid F & \xrightarrow{\tau} E' \mid F' \\
E & \xrightarrow{\alpha} E' \mid F \\
E \mid F & \xrightarrow{\alpha} E \mid F'
\end{align*}
\]

\[
\begin{align*}
\text{new } \{\beta\} & \xrightarrow{\alpha} \text{ new } \{\beta\} E' \\
\text{if } \alpha & \notin \{\beta, \overline{\beta}\}
\end{align*}
\]
A Simple Process

\[ M \equiv \text{coin.}(M + \text{give1coin}.M + \text{give3coins}.M) \]
Pedagogical Motivation

- Pedagogical reasons – MP4
  - CCS discipline
  - What do we gain by prototyping in CCS?
  - Is it possible to implement CCS systems?
  - How to implement the given systems in CCS?
- Add a concurrency template to Cw
- But our aim is not to formalize Cw
Motivation

- System requirements as a collection of process algebra expressions.
- How can such requirements be animated, or guide the entire system implementation?
An extension to C#

Data type support for XML and Table manipulation

A control flow extension for asynchronous wide-area concurrency
  - Async methods
  - Chords
Chords

public async method2();
public int method1(object obj) & method2() { ... }

Who gets the return value?

public async method2();
public async method3(int x);
public int method1(object obj)
  & method2() { ... }
  & method3(int x) { ... }
How does CCS goes .Net (again) ?

1. Represent CCS Systems in Cw
2. Implement CCS labelled transition semantics

\[
\begin{align*}
\alpha.E & \xrightarrow{\alpha} E \\
E + F & \xrightarrow{\alpha} E' \\
E + F & \xrightarrow{\alpha} F'
\end{align*}
\]
CCS to Cw

- Processes become classes in Cw

- Actions
  - Implemented by methods, or by chords to reflect cases of dependence:
    - Ports without complementary ports
    - Input ports with complementary output ports
    - Output ports with complementary input ports
Ports without complementary ports

- The simpler case of actions
- Not relevant from the architectural point of view

```java
public void c_p() {
    p code
}
```
Input ports with complementary output ports

- Implemented as chords of two methods, a synchronous and an asynchronous one.
- The asynchronous method is responsible for signalling that the complementary output port is available for simultaneous execution.

```java
public async obs_p();
public void p() & obs_p() {
    p code
}
```
Output ports with complementary input ports

- Wait for a request from their complementary input ports
- The request receiver cannot stop the current thread

```csharp
public async request_obs_p(object obj) { c_p(obj); }
public void c_p(object obj) {
    c_p code
    if(obj is CallerType1) {
        (CallerType1 obj).obs_p();
    }
    ...other possible requesters...
    if(obj is CallerTypeN) {
        (CallerTypeN obj).obs_p();
    }
}
```
Sequential Behaviour

- Methods implementing input ports are always called by their immediate preceding ports.
- Output ports must be guarded by a semaphore insuring the sequential evolution of the process

```csharp
public async allow_c_p();
public void c_p(object obj) & allow_c_p() {
    c_p code
    if(obj is CallerType1) {
        ((CallerType1) obj).obs_p();
    }
    ...other possible requesters...
    if(obj is CallerTypeN) {
        ((CallerTypeN) obj).obs_p();
    }
}
```
Reactions

- Obtained by the previous implementation of actions where

```
public async obs_p();
public void p() &
ob_p() { p code }
```

```
public async request_obs_p(object obj) { c_p(obj); }
public void c_p(object obj) & allow_c_p() {
  c_p code
  if(obj is CallerType1) {
    (CallerType1 obj).obs_p();
  }
  ...other possible requesters...
  if(obj is CallerTypeN) {
    (CallerTypeN obj).obs_p();
  }
}
```
Alternative Reactions

Alternative reactions are implemented by defining their initial ports as chords bounded to a semaphore (alternative())

```csharp
private async allow_choice();
public async request_obs_green(object obj) {c_green(obj);} public void c_green(object obj) & allow_choice() {
    if(obj is Rail) {
        ((Rail) obj).obs_green();
    }
    rail.request_obs_red(this);
}
red();
```
None Determinism

\[ P_{12} = \text{think}.P_{12} + \text{fork}_1.\text{fork}_2.\text{eat}.\text{fork}_1.\text{fork}_2.P_{12} \]

```csharp
public void c_think() {
    Console.WriteLine("Phil_12 is thinking...");
    Random ran = new Random();
    System.Threading.Thread.Sleep(ran.Next(5000));
    double d = ran.NextDouble();
    if (d >= 0.5) {
        c_think();
    } else {
        // Request an observational fork 1
        pfork1.request_obs_fork1(this);
        fork1();
    }
}
```
Every process prototype is equipped with an asynchronous `start()` method which wakes up the processes and starts up their execution.

```java
public async start() {
    c_think();
}
```
Putting it all together

CCS Specification

Web Service

Web Browser

Application

CCS2dotNet

Cw Code

CwC

CwC

CwC

User Defined Code
Limitations of the method

- Processes must be aware about other processes they are communicating with
- Actions with the same name must be renamed
- Generated code tends to be vast
- No coverage of parametric processes
switch (state) { 
    case st_thinking:
        bmp = humface;
        break;
    case st_hungry:
        bmp = switchstate ? humface2 : humface;
        break;
    case st_entering:
        bmp = switchstate ? humface2 : humface;
        pos -= distance;
        .
        .
        .

int st_thinking;
int st_hungry;
int st_entering;
int st_gotone;
int st_eating;
int st_leaving;
CCS Specification

\[ P_{12} \equiv \text{think}.P_{12} + \text{fork}_1.\text{fork}_2.\text{eat}.\text{fork}_1.\text{fork}_2.P_{12} \]
\[ P_{23} \equiv \text{think}.P_{23} + \text{fork}_2.\text{fork}_3.\text{eat}.\text{fork}_2.\text{fork}_3.P_{23} \]
\[ P_{34} \equiv \text{think}.P_{34} + \text{fork}_3.\text{fork}_4.\text{eat}.\text{fork}_3.\text{fork}_4.P_{34} \]
\[ P_{45} \equiv \text{think}.P_{45} + \text{fork}_4.\text{fork}_5.\text{eat}.\text{fork}_4.\text{fork}_5.P_{45} \]
\[ P_{51} \equiv \text{think}.P_{51} + \text{fork}_5.\text{fork}_1.\text{eat}.\text{fork}_5.\text{fork}_1.P_{51} \]
\[ F_1 \equiv \text{fork}_1.\text{fork}_1.F_1 \]
\[ F_2 \equiv \text{fork}_2.\text{fork}_2.F_2 \]
\[ F_3 \equiv \text{fork}_3.\text{fork}_3.F_3 \]
\[ F_4 \equiv \text{fork}_4.\text{fork}_4.F_4 \]
\[ F_5 \equiv \text{fork}_5.\text{fork}_5.F_5 \]

\[ S \equiv P_{12} \mid P_{23} \mid P_{34} \mid P_{45} \mid P_{51} \mid F_1 \mid F_2 \mid F_3 \mid F_4 \mid F_5 \]
public class Fork1 {
    public async start() { allow_c_fork1(); }

    public async request_obs_fork1(object obj) { c_fork1(obj); }

    public async allow_c_fork1();
    public void c_fork1(object obj) & allow_c_fork1() {
        if(obj is P51) {
            ((P51) obj).obs_fork1();
            //request an observational fork 1
            ((P51) obj).request_obs_fork1(this);
        }
        if(obj is P12) {
            ((P12) obj).obs_fork1();
            //request an observational fork 1
            ((P12) obj).request_obs_fork1(this);
        }
    }
    fork1();
}

public async obs_fork1();
public void fork1() & obs_fork1() { allow_c_fork1(); }
Dining Philosophers Prototype

\[ P_{12} \equiv \text{think}.P_{12} + \text{fork}_1.\text{fork}_2.\text{eat}.\text{fork}_1.\text{fork}_2.P_{12} \]
\[ P_{23} \equiv \text{think}.P_{23} + \text{fork}_2.\text{fork}_3.\text{eat}.\text{fork}_2.\text{fork}_3.P_{23} \]
\[ P_{34} \equiv \text{think}.P_{34} + \text{fork}_3.\text{fork}_4.\text{eat}.\text{fork}_3.\text{fork}_4.P_{34} \]
\[ P_{45} \equiv \text{think}.P_{45} + \text{fork}_4.\text{fork}_5.\text{eat}.\text{fork}_4.\text{fork}_5.P_{45} \]
\[ P_{51} \equiv \text{think}.P_{51} + \text{fork}_5.\text{fork}_1.\text{eat}.\text{fork}_5.\text{fork}_1.P_{51} \]
\[ F_1 \equiv \text{fork}_1.\text{fork}_1.F_1 \]
\[ F_2 \equiv \text{fork}_2.\text{fork}_2.F_2 \]
\[ F_3 \equiv \text{fork}_3.\text{fork}_3.F_3 \]
\[ F_4 \equiv \text{fork}_4.\text{fork}_4.F_4 \]
\[ F_5 \equiv \text{fork}_5.\text{fork}_5.F_5 \]

\[ S \equiv P_{12} \mid P_{23} \mid P_{34} \mid P_{45} \mid P_{51} \mid F_1 \mid F_2 \mid F_3 \mid F_4 \mid F_5 \]
Example – Distributed Counter

\[ C \triangleq up.(C \bowtie C) + dw.P \]
\[ P \triangleq \overline{d}.C + \overline{z}.Z \]
\[ Z \triangleq up.(C \bowtie Z) + zr.Z \]

\[ E \bowtie F = \mathbf{new} \ A (\{\overline{z'}/\overline{z}, \overline{d'}/\overline{d}\} E \ | \ \{u'/up, z'/zr, d'/dw\} F) \]
onde \ A = \{u', z', d'\}
Simple Rail Road Example

Road \equiv \text{car.up.ccross.dw.Road}
Rail \equiv \text{train.green.tcross.red.Rail}
Signal \equiv \text{green.red.Signal + up.dw.Signal}

C \equiv \text{new\{green, red, up, dw\}(Road|Rail|Signal)}
Two way Rail Road System
Conclusions and Future Work

- Good acceptance from students studying CCS process algebra.
- The generated code can be used in a number of different contexts
- Precise expression of requirements and calculation power to discuss correctness and refinement.
- Implement the same method using Groovy
- GUI derivation
- Reverse path observation of the method
- Development of an automatic Cw code generator
- Test with bigger/real examples
Thank you

Questions?

Nuno F. Rodrigues – nfr@di.uminho.pt
Luis S. Barbosa – lsb@di.uminho.pt
Transition System
How does CCS goes .Net?

1. Represent CCS Systems in C#
2. Implement CCS labelled transition semantics

\[ \alpha.E \xrightarrow{\alpha} E \]
\[ E \xrightarrow{\alpha} E' \]
\[ F \xrightarrow{\alpha} F' \]
\[ E + F \xrightarrow{\alpha} E' \]
\[ E + F \xrightarrow{\alpha} F' \]

\[ E \xrightarrow{\alpha} E' \]
\[ F \xrightarrow{\alpha} F' \]
\[ E | F \xrightarrow{\alpha} E' | F' \]
\[ E | F \xrightarrow{\alpha} E' | F \]
\[ E | F \xrightarrow{\alpha} E | F' \]
\[ E \xrightarrow{\alpha} E' \]
\[ \text{new}\ \{\beta\} \rightarrow \text{new}\ \{\beta\} \]

(if \( \alpha \notin \{\beta, \overline{\beta}\} \))
$M \equiv \text{coin.}\tau.\text{coffee}.M$

**coin**

- public void coin() { ... }

**τ**

- private void tau1() { ... }
- ... 
- private x tauN() { ... }

**coffee**

- public cof coffeeComp() { ... }
Processes

Processes become classes in C#

\[ M \equiv \text{coin.} \tau. \text{coffee.} M \]

```csharp
public class M {
    public void coin() { ... }
    private void tau1() { ... }
    ...
    private x tauN() { ... }
    public cof coffee() { ... }
}
```
Sequential Behaviour

- **Implemented based on 3 functions**
  - `initialPorts :: System -> Process -> [Port]`
  - `finalPorts :: System -> Process -> [Port]`
  - `predPorts :: System -> Port -> [Port]`

```java
public class M {
    private string state;
    public void coin() {
        if(state != null || state.Equals("coffee")) {
            state = "processing";
            "code from the coin computations"
            state = "coin";
        } else {
            throw new Exception("Process sequence violation.");
        }
    }
    ...
}
```
Alternative and Parallel Behaviour

- Alternative (P+Q) and Parallel (P | Q) behaviour is implemented by the previous Action implementation.

\[
\begin{align*}
E & \xrightarrow{\alpha} E' \\
E + F & \rightarrow E' \\
F & \xrightarrow{\alpha} F' \\
E + F & \xrightarrow{\alpha} F'
\end{align*}
\]
Restrictions

- Restriction
  - Implemented by assemblies execution spaces, with the `internal` member access controller

\[
\frac{E \xrightarrow{\alpha} E'}{\text{new } \{\beta\} \ E \xrightarrow{\alpha} \text{new } \{\beta\} \ E'} \quad (\text{if } \alpha \notin \{\beta, \overline{\beta}\})
\]
Synchronization is provided in CCS by complementary actions.

\[
\begin{align*}
E & \xrightarrow{\alpha} E' \\
F & \xrightarrow{\bar{\alpha}} F' \\
E \ | \ F & \xrightarrow{\tau} E' \ | \ F'
\end{align*}
\]

```csharp
public static void greenComp(bool b)
{
    if (state == null || state.Equals("red") || state.Equals("dw")) {
        if (!b) { Rail.green(true); }
        state = "processing";
        // (computational details to be supplied)
        state = "green";
    }
    else { throw new Exception("Process sequence violation."); }
}
```
Special Recursive Processes

Rec = a.Rec + b.(Rec | Rec)

- If port b is activated a new Rec process is born.
- In our approach, this kind of behaviour has to be addressed by the client layer of our C# processes.