CSP as a general Concurrency Model

Dr. M. Manjunathaiah (Manju)
Lecturer (Asst. Professor)
Computer Science and Informatics
University of Reading (Redding !)

PoPP11 – WCPATH, Feb. 12th 2011, San Antonio, USA
• July 2007

  - **Student Nominated award for Outstanding contribution to Teaching and Learning in the Faculty of Science, University of Reading**

  - **512 nominations**
    - 5 awards across the University each year!

• Teaching

  - **Part 1**: Discrete Mathematics and Algorithms
  - **Part 2**: Essential and Further Algorithms, Compilers, Operating Systems
  - **Part 3**: Concurrent Systems (CSP)
“Big Picture” – Research Interests

Polyhedral
Hierarchical Parallel framework for Scalability of PetaScale Applications
- Unify several models of Massively Parallel computing

CSP
Pervasive Parallel Computing on Chip
- Unify CSP, Polyhedral for Co-design

MapReduce
Map Reduce abstraction for seamless parallel computing on multi-core (Nvidia Tesla)
- Develop Transformation tools and Applications
  - Phylogeny

BSP
Architecture independent Data Distribution for PetaScale Applications
- Hypergraph partitioning

Establish a “languages and architecture” research programme to develop Models, Analysis and Synthesis tools for designing future high performance systems in a unified coherent framework
Brief Bio - Parallel Computing

- Central Research Laboratory, Bangalore, India
  - Member Research Staff
  - Designed and implemented a 32-node MIMD parallel computer based on INMOS Transputer and C004 switching N/W

- University of Southampton, U.K. Parallel Computing Group
  - Commonwealth Research Scholar
  - Thesis on “Compile time Analysis of Array Sections for Parallelization and Parallel Program Verification”

- Research Fellowships
  - University of Reading, PEDAL Group
    - Polyhedral Model; collaboration with IRISA France
  - Oxford University Computing Laboratory
    - Hardware Software Co-design in CSP model; collaboration with INRIA, France
  - University of Manchester
    - Compilation techniques for chip multiprocessor; visit to SunLabs
Communicating Sequential Process (CSP)

We strike a balance between theory and practice
Motivation - “unifying principle”

Concurrent systems, although appearing radically different, can be understood as having an underlying unifying principle of computational processes interacting to achieve an overall computational goal.
Motivation - Challenges of Concurrent Systems

Sequential programs are deterministic

**Definition:**
given the same input, the program will execute the same set of instructions, and will produce the same result.

Concurrent Programs are Non-deterministic!

**Order of execution of program fragments is not pre-defined**

- If P, Q and R are run on different processors (or as Java threads)
- If the initial values of x=1, y = 0 and z=2,
- Output can be either
  - x=2, y = 2, z = 1
  - x=2, y = 1, z= 0

race-conditions, deadlocks, livelocks, starvation,….phenomena not present in sequential counterpart
Motivation - Communicating Sequential Processes

CSP is introduced as an alternative concurrency model to threads and monitors built into Java

- **Principle** method of co-ordination is through communicating events
- **System design** can then be completely characterized in terms of its communicating behaviours.
- ‘**architecture neutral’**

Elegant model to
- analyze concurrent programs
- Synthesize (derive) from specification

Design challenge growing in **dependable systems**
- fly-by-wire, space control, medical systems, industrial automation
- correctness over performance
- Safety, liveness, absence of deadlocks etc.
Motivation - Communicating Sequential Processes (CSP)

CSP: Communicating Sequential Processes

- Invented by Tony Hoare in 1978
- Followed by many even to this day!

Impact:
- concurrent software design: OCCAM, JCSP, PyCSP, Jibu, …
- concurrent hardware design: HandelC
- ventures: INMOS, Celoxica
- Continuing developments in hybrid models (CSP and B) and new applications like security.
- Turing award…..!
Course structure – Autumn Term 10 weeks

Pre-requisites:
- Basic grasp of set theory
  - Part 1 Discrete Maths and Algorithms course
- Programming maturity
  - Part 2 Programming in Java course
- Some familiarity with ‘concurrency’ – using threads

Hence this is offered as a final year ‘optional’ module
Designing concurrent systems

- Capture a range of interesting concurrent system behaviours
  - using a basic set of operators
  - by abstracting away from implementation details
  - and concentrating on its observable (communicating) events

*Prefix* ($\rightarrow$):

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>$VM = coin \to choc \to VM$</td>
<td></td>
</tr>
</tbody>
</table>

*Choice* ($\square$, $\square$):

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\square$</td>
<td>$TVM = Edinburgh \to 50 \to ticket \to TVM$</td>
<td></td>
</tr>
<tr>
<td>$\square$</td>
<td>$Oxford \to 5 \to ticket \to TVM$</td>
<td></td>
</tr>
</tbody>
</table>

*Parallel* ($\parallel$):

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\parallel$</td>
<td>$TVM \parallel (CUST = Edinburgh \to 50 \to STOP)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$TVM \parallel (CUST = Edinburgh \to 10 \to STOP)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$TVM \parallel (CUST = Edinburgh \to 50 \to CUST)$</td>
<td></td>
</tr>
</tbody>
</table>
Coursework I – Airport Car parking application

- A ‘real-world’ scenario where concurrency is realistic!
  
  - **Three main components**
    - Online booking,
    - email,
    - car park system
  
  - **Design approach**
    - Evolve component design without worrying about locks!
    - Add communication links
    - ‘compose’ larger system from sub-systems (components)
Example application - Component

Implementing processes

Modeling processes.

Note: we use the term process when referring to the implementation of a thread.

Implementing Processes

```java
import jcsp.lang.*;

public class Departs implements CSProcess {
    private ChannelInputInt depart;

    public Departs(ChannelInputInt in) {
        depart = in;
    }

    public void run() {
        while(true) {
            value = depart.read();
            System.out.println("Spaces left "+ value);
        }
    }
}
```

DEPARTS = depart → DEPARTS
Example application - Composition

Implementing Processes

```java
import jcsp.lang.*

public class CarPark {
    public static void main(String[] args) {
        One2OneChannelInt arrive = new One2OneChannelInt();
        One2OneChannelInt depart = new One2OneChannelInt();

        Parallel CarPark = new Parallel{
            new CSProcess[] { new Arrivals(arrive),
                             new Control(arrive, depart),
                             new Departs(depart)}
        };

        CarPark.run();
    }
}
```

CAR-PARK = ARRIVALS || CONTROL || DEPARTS
A number of insights are gained from this programming task:

- **The design formalism ‘scales’**
  - The realisation that a minimal set of processes is indeed adequate to describe the entire behaviour of a complex system comes as a surprise.
  - Reveals the simplicity of underlying concepts and the power of the formalism.

- **Compositionality**
  - Parallel composition of systems is as simple as sequential composition of statements in procedural languages.
  - Familiar concepts of OO design are applicable thus easing the learning curve.

- **Simplicity**
  - absence of concerns for multi-processing side effects.
Analysing Concurrent systems

Second half of the course
- **delve into semantic models** – *traces, stable-failures, failure-divergences*
- **appreciate the rigorous foundations of CSP**
  - To characterize *deterministic* and *non-deterministic* systems
- **develop software design and verification principles**
  - *Refinement, safety, liveness, absence of deadlocks, …..*

Gain insights to answer broader questions of the following kind:
- Are two systems *equivalent*?
- Is one system ‘better’ than another?
Analysing Concurrent systems – ‘semantic models’

Key insights:

- **Traces model:**
  \[ \text{traces}(P) = \text{traces}(Q) \]
  - Two systems are equivalent
    \[ \text{traces}(Q) \subseteq \text{traces}(P) \]
  - Q is ‘safer’ than P hence ‘better’

- **Stable-failures model**
  - Traces cannot characterize non-deterministic systems
  - A non-deterministic system can refuse some events.
  - A **deadlock** occurs if the entire alphabet is refused

- **Failure-divergences model**
  - captures ‘livelock’ system behaviour

Three *semantic models* are studied to develop a ‘big picture’ of characterizing system properties:

- **by calculating traces**

Each model progressively gives more detailed observations.

- Provides a rigorous insight into *deterministic* and *non-deterministic systems*
Analysing Concurrent systems – ‘big picture’

- **Number of insights:**
  - The set of all possible designs forms a (complete) partial order.
    - bounded by terminal processes: ‘RUN’ and ‘STOP’
  - Safety requirement converges designs towards STOP.
  - Liveness requirement converges designs towards RUN.
  - In any concurrent system design we need a satisfactory level of liveness and safety
  - Refinement provides the framework for software design cycle
    - gradually move from specification to an implementation

A precise definition of deterministic systems:
- Predictable, and
- The most refined process under \( F D \)
Courwork 2 – ‘FDR tool for verification’

Develop insights into ‘semantic models’

- Use component models developed for Airport car park example to check for properties
  - safety, liveness, ....

- Perform ‘refinement’ checks using FDR

Refinement ‘big picture’:

- Refinement under various models are concerned with guarantees
  - More possible behaviours, less guarantees

- Refinement amounts to reducing non-determinism
CSP and multi-cores

Some viewpoints…since it is not covered in our course!

Quantifying performance is outside the scope of CSP formalism

- How do I compose a parallel system with a speed-up factor of $O(n)$ or $O(n^2)$?
- Use ‘pencil and paper’…..

In our course we take a ‘simplified’ view that any performance constraints can be met by utilising “abundant” computational bandwidth provided by multi-core technology!
CSP for multi-core programming

• ...but what we convey is that

  - **CSP** model is 'architecture neutral'

  - Programs can be implemented on single machine, multi-cores or distributed networks.

  - ‘network enabled’ versions of JCSP has been used in other undergraduate modules elsewhere.

  - For multi-cores
    • Focus should be on ‘data decomposition’ for optimizing memory hierarchy
    • Decomposition guided by some ‘performance metric’
    • CSP as an ‘implementation language’ for parallelism is sufficiently expressive
      - Meshes, trees, rings, .....can be specified
CSP for multi-core programming

Trends in Applications and Architectures are hybrid in nature:

- **Mobile phone does many things:**
  - web browsing, voice messaging, music, video....
- **Mobile phone architecture matches that**
  - as a SoC (system on chip) implementation
- **Desktop will be a ‘supercomputer’ by 2015 according to multi-core roadmap**

Hybrid programming model?

- CSP is good for ‘task parallelism’
- *Polyhedral model* is good for ‘data parallelism’
- we have looked at *bridging* the two models
  - *But this is at ‘research level’ !!!*
  - *At best can be taught at Master’s level – and not as a single course!*
Unified view in Polyhedral model: Software/hardware

Specification: Algorithm in high level form

\[ C_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j} \]

Compile into Parallel Programs

Space-Time Transformation

Synthesize (VLSI) Parallel Architectures

for \( t = 1, 2 \times N - 1 \)
for all \( p_1 = 1, N \)
for all \( p_2 = 1, N \)
\[ C[t, p_1, p_2] = \mathcal{F}(C, A, B) \]
Conclusions

- Our course aims to:
  - Convey that concurrent system design and implementation is challenging!
  - Convey that well developed formalism exists to approach it systematically and cope with the complexity.
  - Convey that the CS discipline has rich intellectual tradition of developing rigorous approaches.
  - Convey the benefits of the depths of insights about concurrent computational structures that are gained from studying the semantic models.
  - Convey the elegant interplay between theory and practice – to view multi-threaded ‘communicating’ Java programs in a new light!
  - Convey the ‘unifying principle’ of computational process interacting to an overall computational goal.

*Overall convey that concurrent software design is challenging and fascinating….and hence rewarding for the ‘aspiring computer scientist’ to rise to the challenge!*
... and finally

Thanks!

Acknowledgements:

• To all the contributors of CSP for making it an intriguing and rewarding experience to teach.

• To my students whose participation has enriched the experience!
Motivation - “unifying principle”

Concurrent systems, although appearing radically different, can be understood as having an underlying unifying principle of computational processes interacting to achieve an overall computational goal.
take home message!

...a remarkable example in a student’s write-up

Concurrency can be defined, in all situations computational or otherwise, as the “execution of two processes or operations simultaneously” (1) which may potentially be interacting with each other via shared resources. For example in a biological scenario, both the acts of breathing and swallowing use the mouth, but as we all know it is impossible to do both at the same time. If a system had not been implemented correctly it would be possible to do both concurrently and would result in choking, inhalation of food/drink and the possible, but fatal, deadlock situation of being able to do neither. In the computational world, concurrency can be seen a lot more easily in many situations from web servers, to multi-tasking operating systems and full-duplex communication systems.

Acknowledgement for this example:

~ Phill Flynn (Autumn 2008)