

Process Oriented Design for Java - Concurrency for All

Peter Welch
Computing Laboratory
University of Kent at Canterbury
(P.H.Welch@ukc.ac.uk)

PDPTA 2001, Las Vegas, Nevada (24th. June, 2001)

3-Jul-01

Copyright P.H.Welch

1

Nature is not organised as a single thread of control:

```
joe.eatBreakfast ();  
sue.washUp ();  
joe.driveToWork ();  
sue.phone (sally);  
US.government.sue (bill);  
sun.zap (office);
```

???

3-Jul-01

Copyright P.H.Welch

2

Nature is not bulk synchronous:

```
bill.support (this);  
bill.invade (serbia);  
bill.win (worldCup);  
bill.phone (monica);  
bill.blow (saxaphone);  
UNIVERSE.SYNC ();
```

???

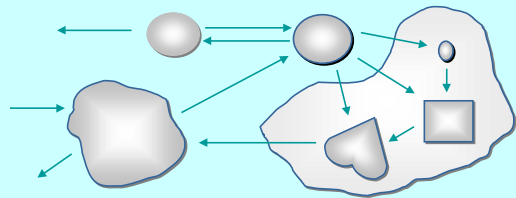
3-Jul-01

Copyright P.H.Welch

3

Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

... nuclear ... human ... astronomic ...



3-Jul-01

Copyright P.H.Welch

4

The Real(-Time) World and Concurrency

Computer systems - to be of use in this world - need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system ... it should be **simpler**.

Yet concurrency is thought to be an **advanced** topic, **harder** than serial computing (which therefore needs to be mastered first).

3-Jul-01

Copyright P.H.Welch

5

This tradition is WRONG!

... which has (radical) implications on how we should educate people for computer science ...

... and on how we apply what we have learnt ...



3-Jul-01

Copyright P.H.Welch

6

What we want from Parallelism

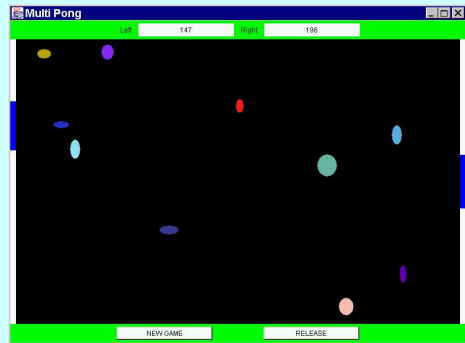
- A powerful tool for **simplifying** the description of systems.
- Performance that spins out from the above, but is **not** the primary focus.
- A model of concurrency that is mathematically clean, yields no engineering surprises and scales well with system complexity.

3-Jul-01

Copyright P.H.Welch

7

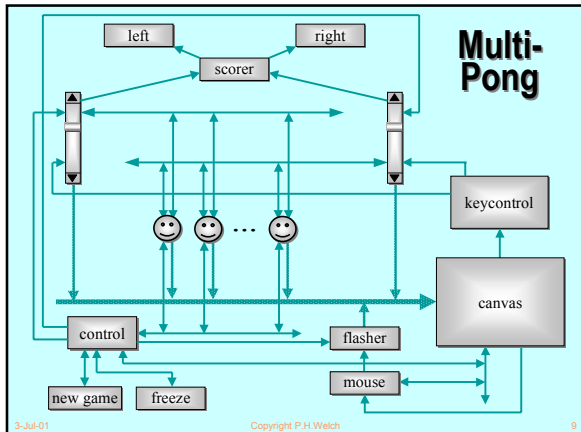
Multi-Pong



3-Jul-01

Copyright P.H.Welch

8



3-Jul-01

Copyright P.H.Welch

9

Good News!

The good news is that we can worry about each process on its own. A process interacts with its environment *through its channels*. It does not interact directly with other processes.

Some processes have *serial* implementations - these are just like traditional serial programs.

Some processes have *parallel* implementations - i.e. networks of sub-processes.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. This will scale!

3-Jul-01

Copyright P.H.Welch

10

What we want from Parallelism

- A powerful tool for **simplifying** the description of systems.
- Performance that spins out from the above, but is **not** the primary focus.
- A model of concurrency that is mathematically clean, yields no engineering surprises and scales well with system complexity.

3-Jul-01

Copyright P.H.Welch

11

Java Monitors - CONCERNS

- Easy to learn - but very difficult to apply ... safely ...
- Monitor methods are *tightly interdependent* - their semantics compose in *complex ways* ... the whole skill lies in setting up and staying in control of these complex interactions ...
- Threads have no structure ... there are no *threads within threads* ...
- Big problems when it comes to scaling up complexity ...

3-Jul-01

Copyright P.H.Welch

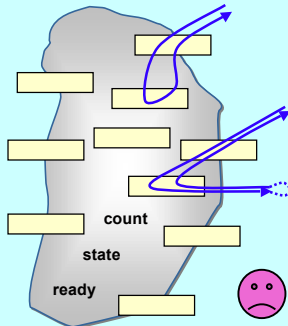
12

Objects Considered Harmful

Most objects are dead - they have no life of their own.

All methods have to be invoked by an external thread of control - they have to be *caller oriented* ...

... a somewhat curious property of so-called *object oriented* design.



3-Jul-01

Copyright P.H.Welch

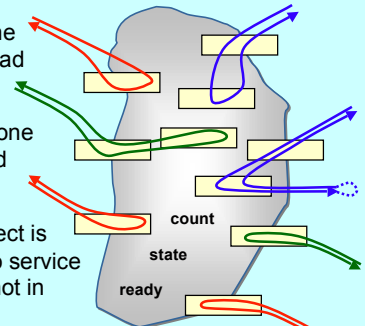
13

Objects Considered Harmful

The object is at the mercy of *any* thread that sees it.

Nothing can be done to prevent method invocation ...

... even if the object is not in a fit state to service it. The object is not in control of its life.



3-Jul-01

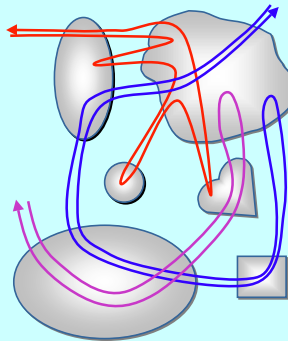
Copyright P.H.Welch

14

Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure.*



3-Jul-01

Copyright P.H.Welch

15

Java Monitors - CONCERNS

- Almost all multi-threaded codes making direct use of the Java monitor primitives that we have seen (*including our own*) contained race or deadlock hazards.
- Sun's Swing classes are not thread-safe ... why not?
- One of our codes contained a race hazard that did not trip for two years. This had been in daily use, its sources published on the web and its algorithms presented without demur to several Java literate audiences.

3-Jul-01

Copyright P.H.Welch

16

Java Monitors - CONCERNS

java.sun.com/products/jfc/tsc/articles/threads/threads1.html

- **"If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug."**
- **"Component developers do not have to have an in-depth understanding of threads programming: toolkits in which all components must fully support multithreaded access, can be difficult to extend, particularly for developers who are not expert at threads programming."**

3-Jul-01

Copyright P.H.Welch

17

Java Monitors - CONCERNS

java.sun.com/products/jfc/tsc/articles/threads/threads1.html

- **"It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications ... use of threads can be very deceptive ... in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism ... this is particularly true in Java ... we urge you to think twice about using threads in cases where they are not absolutely necessary ..."**

3-Jul-01

Copyright P.H.Welch

18

Java Monitors - CONCERNS

- No guarantee that any *synchronized* method will ever be executed ... (e.g. *stacking* JVMs)
- Even if we had above promise (e.g. *queueing* JVMs), standard design patterns for *wait / notify* fail to guarantee liveness ("Wot, no chickens?")

See:

<http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/3.html>

<http://www.nist.gov/itl/div896/emaildir/rt-j/msg00385.html>

<http://www.nist.gov/itl/div896/emaildir/rt-j/msg00363.html>

3-Jul-01

Copyright P.H.Welch

19

Java Monitors - CONCERNS

- Threads yield non-determinacy (and, therefore, scheduling sensitivity) straight away ...
- No help provided to guard against race hazards ...
- Overheads too high (> 30 times ???)
- Tyranny of Magic Names (e.g for *listener* callbacks)
- Learning curve is long ...
- Scalability (both in logic and performance) ???
- Theoretical foundations ???
 - ◆ (deadlock / livelock / starvation analysis ???)
 - ◆ (rules / tools ???)

3-Jul-01

Copyright P.H.Welch

20

Java Monitors - CONCERNS

- So, Java monitors are not something with which we want to think - certainly not on a daily basis.
- But concurrency should be a powerful tool for **simplifying** the description of systems ...
- So, it needs to be something I want to use - and am comfortable with - on a daily basis!

3-Jul-01

Copyright P.H.Welch

21

Communicating Sequential Processes (CSP)

A mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects.

CSP has a formal, and *compositional*, semantics that is in line with our informal intuition about the way things work.

Claim

3-Jul-01

Copyright P.H.Welch

22

Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about deadlock and livelock.
- Abstraction and refinement central to underlying theory.
- Robust **and commercially supported** software engineering tools exist for formal verification.

3-Jul-01

Copyright P.H.Welch

23

Why CSP?

- CSP libraries available for Java (**JCSP**, **CTJ**).
- Ultra-lightweight kernels* have been developed yielding **sub-microsecond** overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.
- Easy to learn and easy to apply ...

* not yet available for JVMs (or Core JVMs!)

3-Jul-01

Copyright P.H.Welch

24

Why CSP?

- After 5 hours teaching
 - ◆ exercises with 20-30 threads of control
 - ◆ regular and irregular interactions
 - ◆ appreciating and eliminating race hazards, deadlock, etc.
- CSP is (parallel) architecture neutral
 - ◆ message-passing
 - ◆ shared-memory



3-Jul-01

Copyright P.H.Welch

25

So, what is CSP?

CSP deals with **processes**, **networks** of processes and various forms of **synchronisation / communication** between processes.

A network of processes is also a process - so CSP naturally accommodates layered network structures (**networks of networks**).

We do not need to be mathematically sophisticated to work with CSP. That sophistication is pre-engineered into the model. We benefit from this simply by using it.

3-Jul-01

Copyright P.H.Welch

26

Processes

myProcess

- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data nor execute those algorithms! [They are not *objects*.]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

3-Jul-01

Copyright P.H.Welch

27

Processes



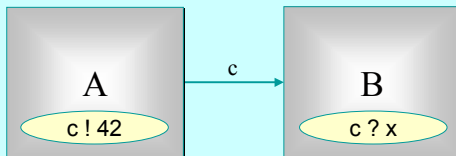
- The simplest form of interaction is *synchronised* message-passing along **channels**.
- The simplest forms of channel are **zero-buffered** and **point-to-point** (i.e. *wires*).
- But, we can have **buffered** channels (*blocking/overwriting*).
- And **any-1**, **1-any** and **any-any** channels.
- And structured multi-way synchronisation (e.g. **barriers**) ...
- And high-level (e.g. **CREW**) *shared-memory* locks ...

3-Jul-01

Copyright P.H.Welch

28

Synchronised Communication



A may write on *c* at any time, but has to wait for a *read*.

B may read from *c* at any time, but has to wait for a *write*.

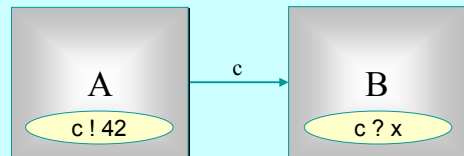
A (c) || B (c)

3-Jul-01

Copyright P.H.Welch

29

Synchronised Communication



Only when both *A* and *B* are ready can the communication proceed over the channel *c*.

A (c) || B (c)

3-Jul-01

Copyright P.H.Welch

30

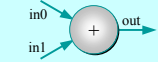
'Legoland' Catalog



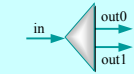
IdInt (in, out)



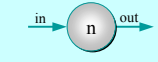
SuccInt (in, out)



PlusInt (in0, in1, out)



Delta2Int (in, out0, out1)



PrefixInt (n, in, out)



TailInt (in, out)

3-Jul-01

Copyright P.H.Welch

31

'Legoland' Catalog

- This is a catalog of fine-grained processes - think of them as pieces of hardware (e.g. chips). They process data (ints) flowing through them.
- They are presented not because we suggest working at such fine levels of granularity ...
- They are presented in order to build up fluency in working with parallel logic.

3-Jul-01

Copyright P.H.Welch

32

'Legoland' Catalog

- Parallel logic should become just as easy to manage as serial logic.
- This is not the traditionally held view ...
- But that tradition is **wrong**.
- CSP/occam people have always known this.



Let's look at some CSP pseudo-code for these processes ...

3-Jul-01

Copyright P.H.Welch

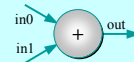
33



IdInt (in, out) = in?x --> out!x --> IdInt (in, out)



SuccInt (in, out) = in?x --> out!(x + 1) --> SuccInt (in, out)



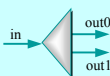
PlusInt (in0, in1, out) =
 (in0?x0 --> SKIP || in1?x1 --> SKIP);
 out!(x0 + x1) --> Plus (in0, in1, out)

Note the parallel input

3-Jul-01

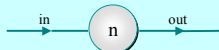
Copyright P.H.Welch

34



Note the parallel output

Delta2Int (in, out0, out1) =
 in?x --> (out0!x --> SKIP || out1!x --> SKIP);
 Delta2Int (in, out0, out1)



PrefixInt (n, in, out) = out!n --> Id (in, out)



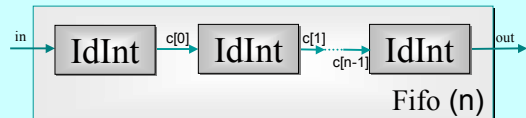
TailInt (in, out) = in?x --> Id (in, out)

3-Jul-01

Copyright P.H.Welch

35

A Blocking FIFO Buffer



Fifo (n, in, out) =
 IdInt (in, c[0]) ||
 (([| i = 0 FOR n-2 | IdInt (c[i], c[i+1])) ||
 IdInt (c[n-2], c[n-1])

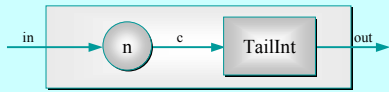
Note: this is such a common idiom that it is provided as a (channel) primitive in JCSP.

3-Jul-01

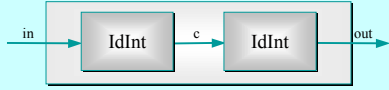
Copyright P.H.Welch

36

A Simple Equivalence



$(\text{PrefixInt } (n, \text{in}, c) \parallel \text{TailInt } (c, \text{out})) \setminus \{c\}$



$(\text{IdInt } (\text{in}, c) \parallel \text{IdInt } (c, \text{out})) \setminus \{c\}$

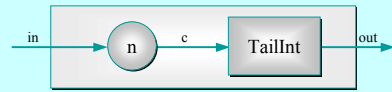
The outside world can see no difference between these two 2-place FIFOs ...

3-Jul-01

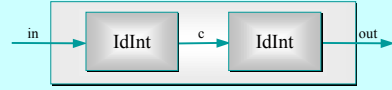
Copyright P.H.Welch

37

A Simple Equivalence



$(\text{PrefixInt } (n, \text{in}, c) \parallel \text{TailInt } (c, \text{out})) \setminus \{c\}$



$(\text{IdInt } (\text{in}, c) \parallel \text{IdInt } (c, \text{out})) \setminus \{c\}$

The proof that they are equivalent is a two-liner from the definitions of $!$, $?$, $-->$, \setminus and \parallel .

3-Jul-01

Copyright P.H.Welch

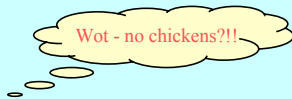
38

Good News!

The good news is that we can 'see' this semantic equivalence with just one glance.

[CLAIM] CSP semantics cleanly reflects our intuitive feel for interacting systems.

This quickly builds up confidence ...

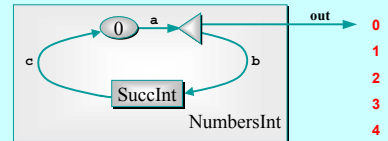


3-Jul-01

Copyright P.H.Welch

39

Some Simple Networks



$\text{NumbersInt } (\text{out}) = \text{PrefixInt } (0, c, a) \parallel \text{Delta2Int } (a, \text{out}, b) \parallel \text{SuccInt } (b, c)$

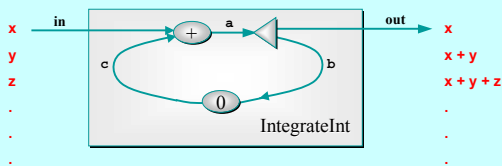
Note: this pushes numbers out so long as the receiver is willing to take it.

3-Jul-01

Copyright P.H.Welch

40

Some Simple Networks



$\text{IntegrateInt } (\text{out}) = \text{PlusInt } (\text{in}, c, a) \parallel \text{Delta2Int } (a, \text{out}, b) \parallel \text{PrefixInt } (0, b, c)$

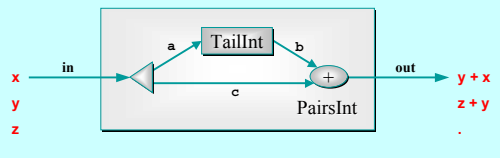
Note: this outputs one number for every input it gets.

3-Jul-01

Copyright P.H.Welch

41

Some Simple Networks



$\text{PairsInt } (\text{in}, \text{out}) = \text{Delta2Int } (\text{in}, a, c) \parallel \text{TailInt } (a, b) \parallel \text{PlusInt } (b, c, \text{out})$

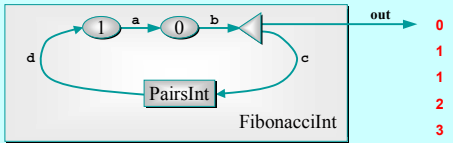
Note: this needs two inputs before producing one output. Thereafter, it produces one number for every input it gets.

3-Jul-01

Copyright P.H.Welch

42

Some Layered Networks



```

FibonacciInt (out) = PrefixInt (1, d, a) ||
                    PrefixInt (0, a, b) ||
                    Delta2Int (b, out, c) ||
                    PairsInt (b, c)
    
```

Note: the two numbers needed by PairsInt to get started are provided by the two PrefixInts. Thereafter, only one number circulates on the feedback loop. If only one PrefixInt had been in the circuit, deadlock would have happened (with each process waiting trying to input).

3-Jul-01

Copyright P.H.Welch

43

Some Layered Networks



```

SquaresInt (out) = NumbersInt (a) ||
                    IntegrateInt (a, b) ||
                    PairsInt (b, out)
    
```

Note: the traffic on individual channels:

```

<a> = [0, 1, 2, 3, 4, 5, 6, 7, 8, ...]
<b> = [0, 1, 3, 6, 10, 15, 21, 28, 36, ...]
<out> = [1, 4, 9, 16, 25, 36, 49, 64, 81, ...]
    
```

3-Jul-01

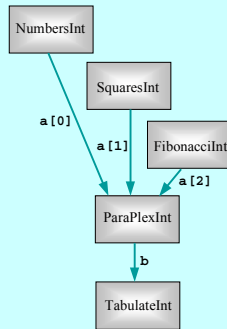
Copyright P.H.Welch

44

Quite a Lot of Processes

```

NumbersInt (a[0]) ||
SquaresInt (a[1]) ||
FibonacciInt (a[2]) ||
ParaPlexInt (a, b) ||
TabulateInt (b)
    
```



3-Jul-01

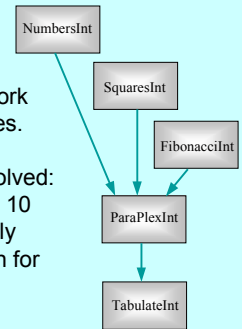
Copyright P.H.Welch

45

Quite a Lot of Processes

At this level, we have a network of 5 communicating processes.

In fact, 28 processes are involved: 18 non-terminating ones and 10 low-level transients repeatedly starting up and shutting down for parallel input and output.



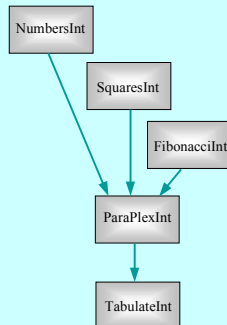
3-Jul-01

Copyright P.H.Welch

46

Quite a Lot of Processes

Fortunately, CSP semantics are **compositional** - which means that we only have to reason at each layer of the network in order to design, understand, code, and maintain it.



3-Jul-01

Copyright P.H.Welch

47

Deterministic Processes

So far, our parallel systems have been **deterministic**:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, does not introduce non-determinism.

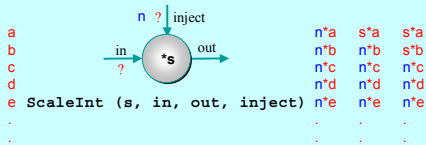
This gives a firm foundation for exploring real-world models which cannot always behave so simply.

3-Jul-01

Copyright P.H.Welch

48

Another Control Process



```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
  [PRI]
  in?a --> out!s*a --> SKIP
  );
ScaleInt (s, in, out, inject)
```

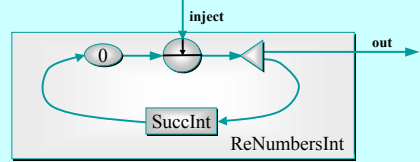
Note: [] is the (external) choice operator of CSP.
[PRI] is a prioritised version - giving priority to the event on its left.

3-Jul-01

Copyright P.H.Welch

55

Some Resettable Networks



This is a resettable version of the `NumbersInt` process.

If nothing is sent down `inject`, it behaves as before.

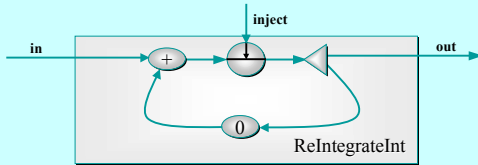
But it may be reset to count from *any* number at *any* time.

3-Jul-01

Copyright P.H.Welch

56

Some Resettable Networks



This is a resettable version of the `IntegrateInt` process.

If nothing is sent down `inject`, it behaves as before.

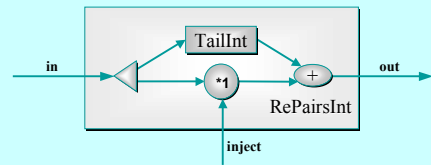
But its running sum may be reset to *any* number at *any* time.

3-Jul-01

Copyright P.H.Welch

57

Some Resettable Networks



This is a resettable version of the `PairsInt` process.

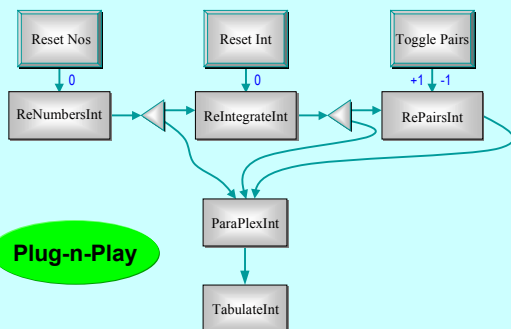
By sending `-1` or `+1` down `inject`, we can toggle its behaviour between `PairsInt` and `DifferentiateInt` (a device that cancels the effect of `IntegrateInt` if pipelined on to its output).

3-Jul-01

Copyright P.H.Welch

58

A Controllable Machine

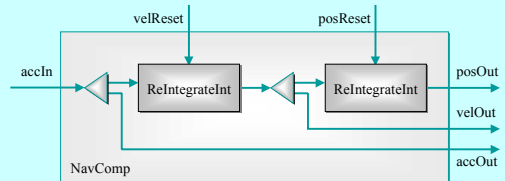


3-Jul-01

Copyright P.H.Welch

59

An Inertial Navigation Component



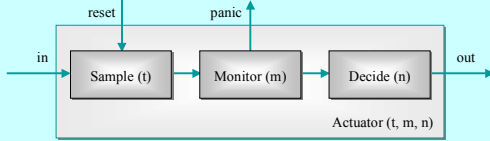
- `accIn`: carries *regular* accelerometer samples;
- `velReset`: velocity *initialisation* and *corrections*;
- `posReset`: position *initialisation* and *corrections*;
- `posOut/velOut/accOut`: *regular* outputs.

3-Jul-01

Copyright P.H.Welch

60

Final Stage Actuator



- **Sample (t)** : every t time units, output the *latest* input (or **null** if none); the value of t may be reset;
- **Monitor (m)** : copy input to output counting **nulls** - if m in a row, send panic message and terminate;
- **Decide (n)** : copy non-**null** input to output and *remember* last n outputs - convert **nulls** to a *best guess* depending on those last n outputs.

3-Jul-01

Copyright P.H.Welch

61

Putting CSP into practice ...

JCSP

<http://www.cs.ukc.ac.uk/projects/ofa/jcsp/>

3-Jul-01

Copyright P.H.Welch

62

3-Jul-01

Copyright P.H.Welch

63

CSP for Java (JCSP)

- A **process** is an object of a class implementing the **CSProcess** interface:

```

interface CSProcess {
    public void run();
}
  
```

- The behaviour of the process is determined by the body given to the `run()` method in the implementing class.

3-Jul-01

Copyright P.H.Welch

64

JCSP Process Structure

```

class Example implements CSProcess {
    ... private shared synchronisation objects
        (channels etc.)
    ... private state information

    ... public constructors
    ... public accessors (gets) / mutators (sets)
        (only to be used when not running)

    ... private support methods (part of a run)
    ... public void run() (process starts here)
}
  
```

3-Jul-01

Copyright P.H.Welch

65

Two Sets of Channel Classes (and Interfaces)

Object channels
- carrying (references to) arbitrary Java objects

int channels
- carrying Java ints

3-Jul-01

Copyright P.H.Welch

66

Channel Interfaces and Classes

- Channel interfaces are what the processes see. Processes only need to care what kind of data they carry (`ints` or `objects`) and whether the channels are for output, input or **ALTING** (i.e. choice) input.
- It will be the network builder's concern to choose the actual channel **classes** to use when connecting processes together.

3-Jul-01

Copyright P.H.Welch

67

int Channels

- The int channels are convenient and secure.
- As with **occam**, it's difficult to introduce race hazards.
- For completeness, JCSP should provide channels for carrying all of the Java primitive data-types. These would be trivial to add. So far, there has been no pressing need.

3-Jul-01

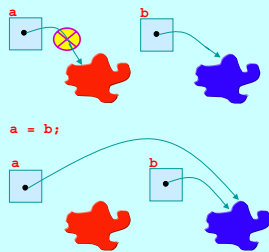
Copyright P.H.Welch

68

Object Aliasing - Danger !!

Java objects are referenced through variable names.

`Thing a = ..., b = ...;`



`a` and `b` are now *aliases* for the same object!



3-Jul-01

Copyright P.H.Welch

69

Object Channels - Danger !!

- Object channels expose a danger not present in **occam**.
- Channel communication only communicates the object reference.

```
Thing t = ...
c.write(t); // c!t
... use t
```

c

```
Thing t;
t = (Thing) c.read(); // c?t
... use t
```

3-Jul-01

Copyright P.H.Welch

70

Object Channels - Danger !!

- After the communication, each process has a reference (in its variable `t`) to the **same** object.
- If **one** of these processes modifies that object (in its `... use t`), the **other** one had better forget about it!

```
Thing t = ...
c.write(t); // c!t
... use t
```

c



```
Thing t;
t = (Thing) c.read(); // c?t
... use t
```

3-Jul-01

Copyright P.H.Welch

71

Object Channels - Danger !!

- Otherwise, **occam's** parallel usage rule is violated and we will be at the mercy of **when** the processes get scheduled for execution - a **RACE HAZARD!** 
- We have design patterns to prevent this. 

```
Thing t = ...
c.write(t); // c!t
... use t
```

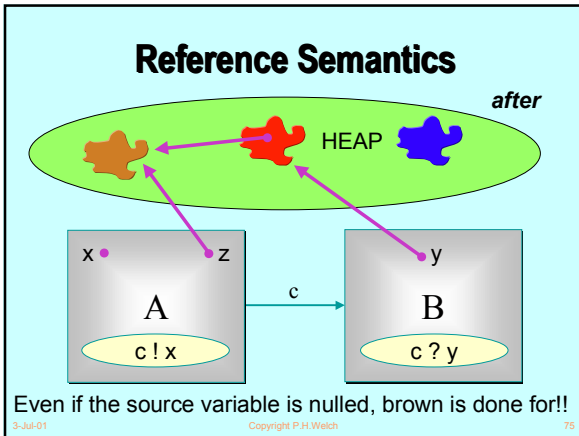
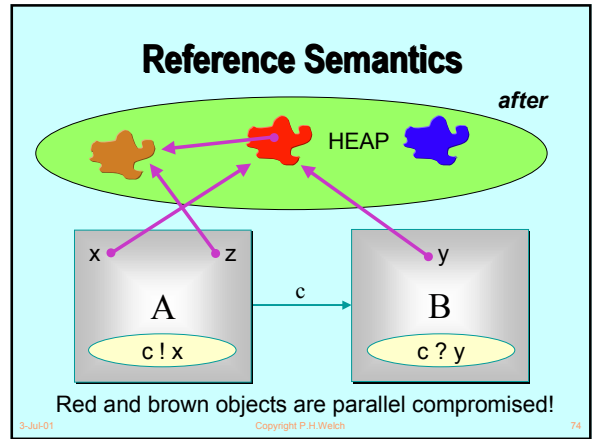
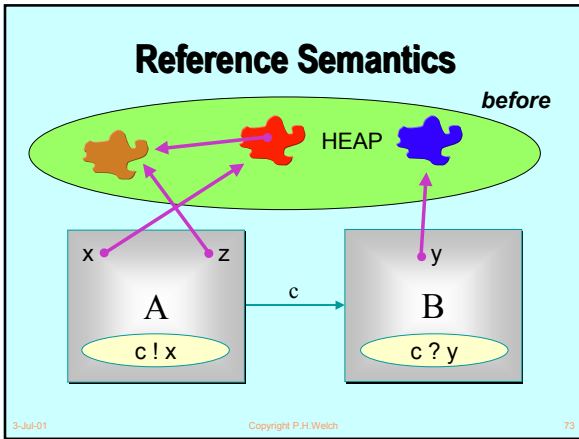
c

```
Thing t;
t = (Thing) c.read(); // c?t
... use t
```

3-Jul-01

Copyright P.H.Welch

72



Classical occam

Different in-scope variables *implies* different pieces of data (*zero aliasing*).

Automatic guarantees against *parallel race hazards* on data access ... and against *serial aliasing accidents*.

Overheads for *large* data communications:

- space (needed at both ends for both copies);
- time (for copying).

3-Jul-01 Copyright P.H.Welch 76

Java / JCSP

Hey ... it's Java ... so *aliasing* is endemic.

No guarantees against *parallel race hazards* on data access ... or against *serial aliasing accidents*. We must look after ourselves.

Overheads for *large* data communications:

- space (*shared* by both ends);
- time is O(1).

3-Jul-01 Copyright P.H.Welch 77

Object and Int Channels (interfaces)

```

interface ChannelOutput {
    public void write (Object o);
}

interface ChannelOutputInt {
    public void write (int o);
}

interface ChannelInput {
    public Object read ();
}

interface ChannelInputInt {
    public int read ();
}

abstract class
    AltIntChannelInput
    extends Guard
    implements ChannelInput {

abstract class
    AltIntChannelInputInt
    extends Guard
    implements ChannelInputInt {

```

3-Jul-01 Copyright P.H.Welch 78

Channel Interfaces

- These are what the processes see - they only care what kind of data they carry (ints or objects) and whether the channels are for output, input or **ALting** (i.e. choice) input.
- It will be the network builder's concern to choose the actual channel **classes** to use when connecting processes together.
- Let's review some of the *Legoland* processes - this time in JCSP.

3-Jul-01

Copyright P.H.Welch

79

JCSP Process Structure

```
class Example implements CSProcess {
    ... private shared synchronisation objects
        (channels etc.)
    ... private state information

    ... public constructors
    ... public accessors (gets)/mutators (sets)
        (only to be used when not running)

    ... private support methods (part of a run)
    ... public void run() (process starts here)
}
```

reminder

3-Jul-01

Copyright P.H.Welch

80



```
class SuccInt implements CSProcess {
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

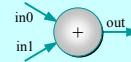
    public SuccInt (ChannelInputInt in,
                  ChannelOutputInt out) {
        this.in = in;
        this.out = out;
    }

    public void run () {
        while (true) {
            int n = in.read ();
            out.write (n + 1);
        }
    }
}
```

3-Jul-01

Copyright P.H.Welch

81



```
class PlusInt implements CSProcess {
    private final ChannelInputInt in0;
    private final ChannelInputInt in1;
    private final ChannelOutputInt out;

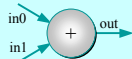
    public PlusInt (ChannelInputInt in0,
                  ChannelInputInt in1,
                  ChannelOutputInt out) {
        this.in0 = in0;
        this.in1 = in1;
        this.out = out;
    }

    ... public void run ()
}
```

3-Jul-01

Copyright P.H.Welch

82



```
class PlusInt implements CSProcess {
    ... private final channels (in0, in1, out)
    ... public PlusInt (ChannelInputInt in0, ...)

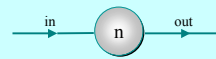
    public void run () {
        while (true) {
            int n0 = in0.read ();
            int n1 = in1.read ();
            out.write (n0 + n1);
        }
    }
}
```

Note: the inputs really need to be done in parallel - later!

3-Jul-01

Copyright P.H.Welch

83



```
class PrefixInt implements CSProcess {
    private final int n;
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

    public PrefixInt (int n, ChannelInputInt in,
                    ChannelOutputInt out) {
        this.n = n;
        this.in = in;
        this.out = out;
    }

    public void run () {
        out.write (n);
        new IdInt (in, out).run ();
    }
}
```

3-Jul-01

Copyright P.H.Welch

84

Process Networks

- We now want to be able to take instances of these *processes* (or components) and connect them together to form a network.
- The resulting network will itself be a *process*.
- To do this, we need to construct some real wires - these are instances of the *channel classes*.
- We also need a way to compose everything together - the `Parallel` constructor.

3-Jul-01

Copyright P.H.Welch

85

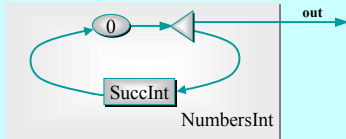
Parallel

- **Parallel** is a `CSPProcess` whose constructor takes an array of `CSPProcesses`.
- Its `run()` method is the parallel composition of its given `CSPProcesses`.
- The semantics is the same as for the **occam** `PAR` (or CSP `||`).
- The `run()` terminates when and only when all of its component processes have terminated.

3-Jul-01

Copyright P.H.Welch

86



```
class NumbersInt implements CSPProcess {
    private final ChannelOutputInt out;

    public NumbersInt (ChannelOutputInt out) {
        this.out = out;
    }

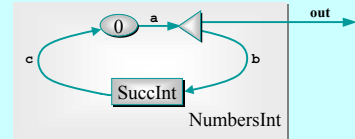
    ... public void run ()
}

```

3-Jul-01

Copyright P.H.Welch

87



```
public void run () {
    One2OneChannelInt a = new One2OneChannelInt ();
    One2OneChannelInt b = new One2OneChannelInt ();
    One2OneChannelInt c = new One2OneChannelInt ();

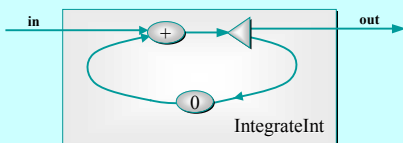
    new Parallel (
        new CSPProcess[] {
            new PrefixInt (0, c, a),
            new Delta2Int (a, out, b),
            new SuccInt (b, c)
        }
    ).run ();
}

```

3-Jul-01

Copyright P.H.Welch

88



```
class IntegrateInt implements CSPProcess {
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

    public IntegrateInt (ChannelInputInt in,
        ChannelOutputInt out) {
        this.in = in;
        this.out = out;
    }

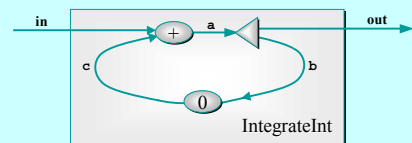
    ... public void run ()
}

```

3-Jul-01

Copyright P.H.Welch

89



```
public void run () {
    One2OneChannelInt a = new One2OneChannelInt ();
    One2OneChannelInt b = new One2OneChannelInt ();
    One2OneChannelInt c = new One2OneChannelInt ();

    new Parallel (
        new CSPProcess[] {
            new PlusInt (in, c, a),
            new Delta2Int (a, out, b),
            new PrefixInt (0, b, c)
        }
    ).run ();
}

```

3-Jul-01

Copyright P.H.Welch

90



```

class SquaresInt implements CSProcess {
    private final ChannelOutputInt out;

    public PairsInt (ChannelOutputInt out) {
        this.out = out;
    }

    ... public void run ()
}

```



```

public void run () {
    One2OneChannelInt a = new One2OneChannelInt ();
    One2OneChannelInt b = new One2OneChannelInt ();

    new Parallel (
        new CSProcess[] {
            new NumbersInt (a),
            new IntegrateInt (a, b),
            new PairsInt (b, out)
        }
    ).run ();
}

```

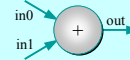
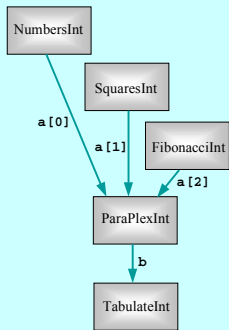
Quite a Lot of Processes

```

One2OneChannelInt[] a =
One2OneChannelInt.create (3);
One2OneChannel b =
new One2OneChannel ();

new Parallel (
    new CSProcess[] {
        new NumbersInt (a[0]),
        new SquaresInt (a[1]),
        new FibonacciInt (a[2]),
        new ParaPlexInt (a, b),
        new TabulateInt (b)
    }
).run ();

```



```

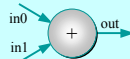
class PlusInt implements CSProcess {
    ... private final channels (in0, in1, out)
    ... public PlusInt (ChannelInputInt in0, ...)

    public void run () {
        while (true) {
            int n0 = in0.read ();
            int n1 = in1.read ();
            out.write (n0 + n1);
        }
    }
}

```

Change this!

Note: the inputs really need to be done in parallel - now!



This process does one input and terminates.

```

public void run () {
    ProcessReadInt readIn0 = new ProcessReadInt (in0);
    ProcessReadInt readIn1 = new ProcessReadInt (in1);

    CSProcess parRead =
    new Parallel (new CSProcess[] {readIn0, readIn1});

    while (true) {
        parRead.run ();
        out.write (readIn0.value + readIn1.value);
    }
}

```

Implementation Note

- As in the transputer (and KRoC **occam** etc.), a JCSP **Parallel** object runs its first (n-1) components in separate Java threads and its last component in its own thread of control.
- When a **Parallel.run()** terminates, the **Parallel** object parks all its threads for reuse in case the **Parallel** is run again.
- So processes like **PlusInt** incur the overhead of Java thread creation only during its first cycle.
- That's why we named the **parRead** process before loop entry, rather than constructing it anonymously each time within the loop.

Deterministic Processes

So far, our JCSP systems have been *deterministic*:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, does not introduce non-determinism.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.

3-Jul-01

Copyright P.H.Welch

97

Non-Deterministic Processes

In the real world, it is sometimes the case that things happen as a result of:

- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP (JCSP) addresses these issues *explicitly*.

Non-determinism does not arise by default.



3-Jul-01

Copyright P.H.Welch

98

Alternation* - the CSP Choice

```
public abstract class Guard {  
    ... package-only abstract methods (enable/disable)  
}
```

Five JCSP classes are (i.e. extend) Guards:

AltingChannelInput	(Objects)
AltingChannelInputInt	(ints)
AltingChannelAccept	(CALLs)
Timer	(timeouts)
Skip	(polling)

Only the 1-1 and any-1 channels extend the above (i.e. are *ALTable*).

* Alternation is named after the occam ALT ...

3-Jul-01

99

Ready/Unready Guards

- A **channel** guard is ready iff *data is pending* - i.e. a process at the other end has output to (or called) the channel and this has not yet been input (or accepted).
- A **timer** guard is ready iff *its timeout has expired*.
- A **skip** guard is *always* ready.

3-Jul-01

Copyright P.H.Welch

100

Alternation

For ALTING, a JCSP process must have a `Guard[]` array - this can be any mix of channel inputs, call channel accepts, timeouts or skips:

```
final Guard[] guards = {...};
```

It must construct an *Alternative* object for each such guard array:

```
final Alternative alt =  
    new Alternative (guards);
```

The ALT is carried out by invoking one of the three varieties of select methods on the alternative.

3-Jul-01

Copyright P.H.Welch

101

alt.select()

This blocks passively until one or more of the guards are ready. Then, it makes an **ARBITRARY** choice of one of these ready guards and returns the index of that chosen one. If that guard is a **channel**, the ALTING process must then **read** from (or **accept**) it.

alt.priSelect()

Same as above - except that if there is more than one ready guard, it chooses the one with the lowest index.

3-Jul-01

Copyright P.H.Welch

102

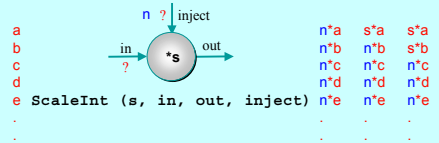
alt.fairSelect()

Same as above - except that if there are more than one ready guards, it makes a **FAIR** choice.

This means that, in successive invocations of **alt.fairSelect**, no ready guard will be chosen twice if another ready guard is available. At worst, no ready guard will miss out on *n* successive selections (where *n* is the number of guards).

Fair alternation is possible because an **Alternative** object is tied to one set of guards.

Another Control Process



```
ScaleInt (s, in, out, inject) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
ScaleInt (s, in, out, inject)
```

Note: [] is the (external) choice operator of CSP.
[PRI] is a prioritised version - giving priority to the event on its left.

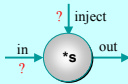
```
class ScaleInt implements CSProcess {
```

```
  private int s;
  private final ChannelInputInt in, inject;
  private final ChannelOutputInt out;
```

```
  public ScaleInt (int s, ChannelInputInt in,
                  ChannelInputInt inject,
                  ChannelOutputInt out) {
    this.s = s;
    this.in = in;
    this.inject = inject;
    this.out = out;
  }
```

```
  ... public void run ()
```

```
}
```



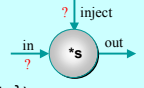
```
public void run () {
```

```
  final Alternative alt =
    new Alternative (new Guard[] {inject, in});

  final int INJECT = 0, IN = 1; // guard indices
```

```
  while (true) {
    switch (alt.priSelect ()) {
      case INJECT:
        s = inject.read ();
        break;
      case IN:
        final int a = in.read ();
        out.write (s*a);
        break;
    }
  }
}
```

Note these are in priority order.



Real-Time Sampler



- This process services any of 3 events (2 inputs and 1 timeout) that may occur.
- Its τ parameter represents a time interval. Every τ time units, it must output the *last* object that arrived on its *in* channel during the previous time slice. If nothing arrived, it must output a *null*.
- The length of the timeslice, τ , may be reset at any time by a new value arriving on its *reset* channel.



```
class Sample implements CSProcess {
  private final long t;
  private final AltingChannelInput in;
  private final AltingChannelInputInt reset;
  private final ChannelOutput out;

  public Sample (long t,
                AltingChannelInput in,
                AltingChannelInputInt reset,
                ChannelOutput out) {
    this.t = t;
    this.in = in;
    this.reset = reset;
    this.out = out;
  }
  ... public void run ()
}
```



```

public void run () {
    final Timer tim = new Timer ();
    final Alternative alt =
        new Alternative (new Guard[] {reset, tim, in});
    final int RESET = 0, TIM = 1, IN = 2; // indices

    Object sample = null;
    long timeout = tim.read () + t;
    tim.setAlarm (timeout);

    ... main loop
}

```

Note these are in priority order.



```

while (true) {
    switch (alt.priSelect ()) {
    case RESET:
        t = reset.read ();
        break;
    case TIM:
        out.write (sample);
        sample = null;
        timeout += t;
        tim.setAlarm (timeout);
        break;
    case IN:
        sample = in.read ();
        break;
    }
}

```

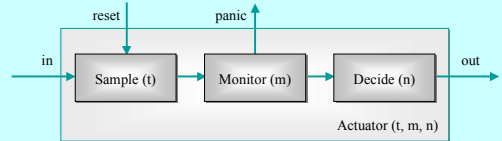


```

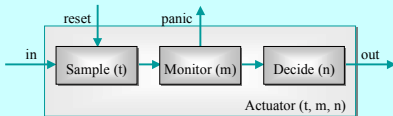
while (true) {
    switch (alt.priSelect ()) {
    case RESET:
        t = reset.read ();
        timeout = tim.read ();
    case TIM:
        out.write (sample);
        sample = null;
        timeout += t;
        tim.setAlarm (timeout);
        break;
    case IN:
        sample = in.read ();
        break;
    }
}

```

Final Stage Actuator



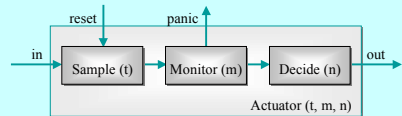
- **Sample (t)** : every t time units, output the *latest* input (or null if none); the value of t may be reset;
- **Monitor (m)** : copy input to output counting nulls - if m in a row, send panic message and terminate;
- **Decide (n)** : copy non-null input to output and remember last n outputs - convert nulls to a best guess depending on those last n outputs.



```

class Actuator implements CSProcess {
    ... private state (t, m and n)
    ... private interface channels (in, reset, panic and out)
    ... public constructor (assign parameters t, m, n, in, reset, panic and out to the above fields)
    ... public void run ()
}

```



```

public void run ()
{
    final One2OneChannel a = new One2OneChannel ();
    final One2OneChannel b = new One2OneChannel ();

    new Parallel (
        new CSProcess[] {
            new Sample (t, in, reset, a),
            new Monitor (m, a, panic, b),
            new Decide (n, b, out)
        }
    ).run ();
}

```

Shared Channels

- So far, all our channels have been point-to-point, zero-buffered and synchronised (i.e. standard CSP primitives);
- JCSP also offers multi-way shared channels (in the style of `occam3` and the `KRoC` shared channel library);
- JCSP also offers buffered channels of various well-defined forms.

3-Jul-01

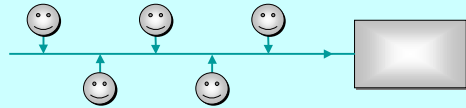
Copyright P.H.Welch

115

One2OneChannel



Any2OneChannel

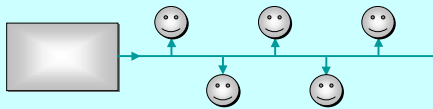


3-Jul-01

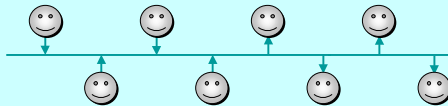
Copyright P.H.Welch

116

One2AnyChannel



Any2AnyChannel



No ALTing!

3-Jul-01

Copyright P.H.Welch

117

Object Channel classes

```
class One2OneChannel
  extends AltingChannelInput
  implements ChannelOutput;
```

```
class One2AnyChannel
  implements ChannelInput,
  ChannelOutput;
```

```
class Any2OneChannel
  extends AltingChannelInput
  implements ChannelOutput;
```

```
class Any2AnyChannel
  implements ChannelInput,
  ChannelOutput;
```

3-Jul-01

Copyright P.H.Welch

118

int Channel classes

```
class One2OneChanneInt
  extends AltingChannelInputInt
  implements ChannelOutputInt;
```

```
class One2AnyChannelInt
  implements ChannelInputInt,
  ChannelOutputInt;
```

```
class Any2OneChannelInt
  extends AltingChannelInputInt
  implements ChannelOutputInt;
```

```
class Any2AnyChannelInt
  implements ChannelInputInt,
  ChannelOutputInt;
```

3-Jul-01

Copyright P.H.Welch

119

Graphics and GUIs

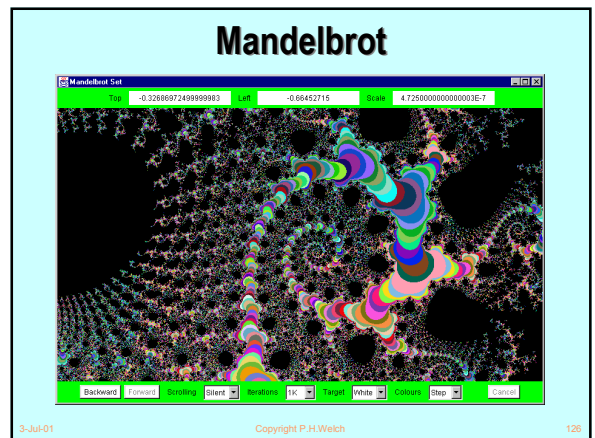
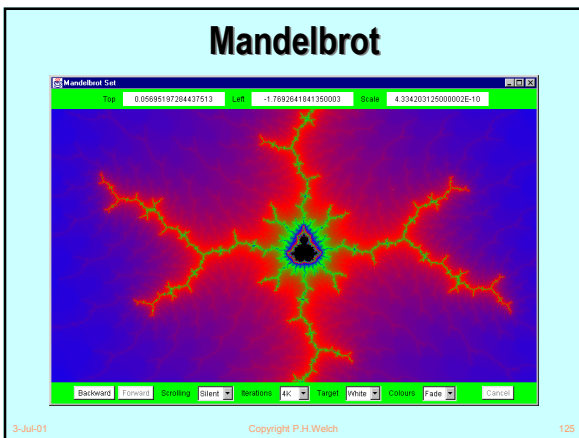
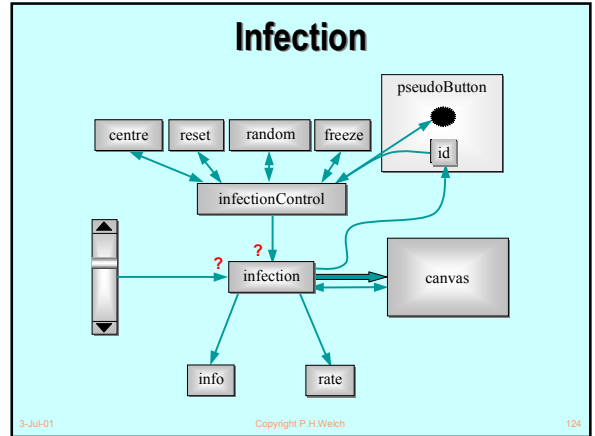
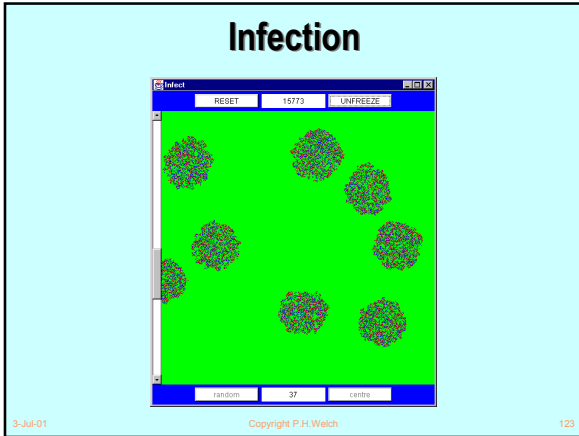
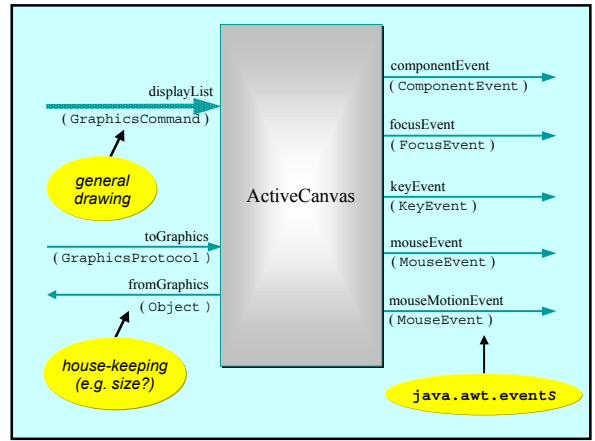
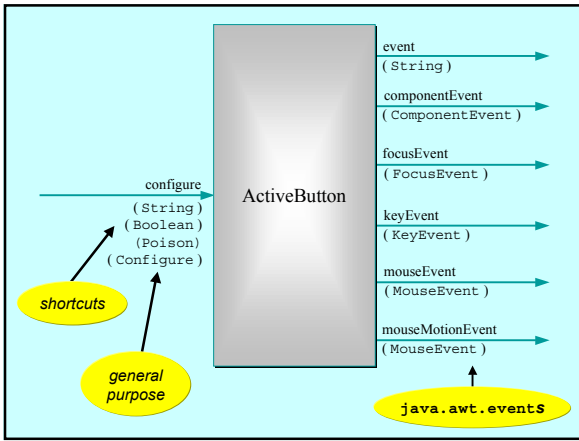
`jcsp.awt = java.awt + channels`

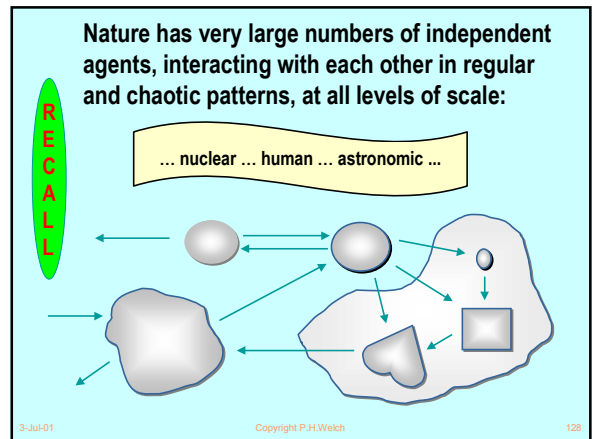
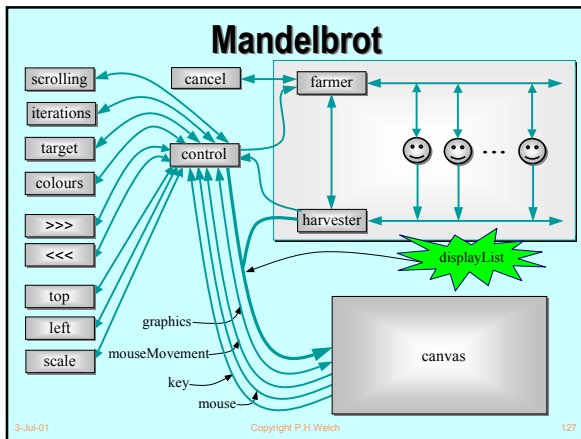
- GUI events → channel communications
- Widget configuration → channel communications
- Graphics commands → channel communications

3-Jul-01

Copyright P.H.Welch

120





Good News!

The good news is that we can worry about each process on its own. A process interacts with its environment *through its channels*. It does not interact directly with other processes.

Some processes have *serial* implementations - these are just like traditional serial programs.

Some processes have *parallel* implementations - i.e. networks of sub-processes.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. This will scale!

3-Jul-01 Copyright P.H.Welch 129

Other Work

- A CSP model for the Java monitor mechanisms (**synchronized**, **wait**, **notify**, **notifyAll**) has been built.
- This enables *any* Java threaded system to be analysed in CSP terms - e.g. for formal verification of freedom from deadlock/livelock.
- Confidence gained through the formal proof of correctness of the JCSP channel implementation:
 - ◆ a JCSP channel is a non-trivial monitor - the CSP model for monitors transforms this into an even more complex system of CSP processes and channels;
 - ◆ using FDR, that system has been proven to be a refinement of a single CSP channel and *vice versa* - **Q.E.D.**

3-Jul-01 Copyright P.H.Welch 130

Other Work

- Higher level synchronisation primitives (e.g. JCSP *CALL channels*, *barriers*, *buckets*, ...) that capture good patterns of working with low level CSP events.
- Proof rules and design tool support for the above.
- CSP kernels and their binding into JVMs to support JCSP (or *CoreJCSP* ... ?).
- **Communicating Threads for Java (CTJ):**
 - ◆ this is another Java class library based on CSP principles;
 - ◆ developed at the University of Twente (Netherlands) with special emphasis on real-time applications - it's excellent;
 - ◆ CTJ and JCSP share a common heritage and reinforce each other's on-going development - we do talk to each other!

3-Jul-01 Copyright P.H.Welch 131

Distributed JCSP (soon)

- Network channels + plus simple brokerage service for letting JCSP systems find and connect to each other transparently (from anywhere on the Internet).
- Virtual channel infrastructure to support this. All application channels auto-multiplexed over *single* (auto-generated) TCP/IP link between any two JVMs.
- Channel Name Server (CNS) provided. Participating JCSP systems just need to know where this is. More sophisticated brokers are easily bootstrapped on top of the CNS (using JCSP).
- **Killer Application Challenge:**
 - ◆ second generation Napster (no central control or database) ...

3-Jul-01 Copyright P.H.Welch 132

Summary

WYSIWYG

Plug-n-Play

- CSP has a **compositional** semantics.
- CSP concurrency can **simplify** design:
 - ◆ data encapsulation within processes does not break down (unlike the case for objects);
 - ◆ channel interfaces impose clean decoupling between processes (unlike method interfaces between objects)
- JCSP enables direct Java implementation of CSP design.

3-Jul-01

Copyright P.H.Welch

133

Summary



- CSP kernel overheads are sub-100-nanosecond (KRoC/CCSP). *Currently*, JCSP depends on the underlying Java threads/monitor implementation.
- Rich mathematical foundation:
 - ◆ 20 years mature - recent extensions include simple priority semantics;
 - ◆ higher level design rules (e.g. *client-server*, *resource allocation priority*, *IO-par*) with formally proven guarantees (e.g. freedom from deadlock, livelock, process starvation);
 - ◆ commercially supported tools (e.g. FDR).
- We don't need to be mathematically sophisticated to take advantage of CSP. It's built-in. Just use it!

3-Jul-01

Copyright P.H.Welch

134

Summary

- Process Oriented Design (processes, syncs, alts, parallel, layered networks).
- WYSIWYG:
 - ◆ each process considered individually (own data, own control threads, external synchronisation);
 - ◆ leaf processes in network hierarchy are ordinary *serial* programs - all our past skills and intuition still apply;
 - ◆ *concurrency* skills sit happily alongside the old serial ones.
- Race hazards, deadlock, livelock, starvation problems: we have a rich set of design patterns, theory, intuition and tools to apply.



3-Jul-01

Copyright P.H.Welch

135

Conclusions

- We are **not** saying that Java's threading mechanisms need changing.
- Java is sufficiently flexible to allow **many** concurrency paradigms to be captured.
- JCSP is just a **library** - Java needs no language change to support CSP.
- CSP rates serious consideration as a basis for any real-time specialisation of Java:
 - ◆ **quality** (robustness, ease of use, scalability, management of complexity, formalism);
 - ◆ **lightness** (overheads do not invalidate the above benefits - they encourage them).

3-Jul-01

Copyright P.H.Welch

136

Acknowledgements

- Paul Austin - the original developer of JCSP (p_d_austin@hotmail.com).
- Andy Bakkers and Gerald Hilderink - the CTJ library (bks@el.utwente.nl, G.H.Hilderink@el.utwente.nl).
- Jeremy Martin - for the formal proof of correctness of the JCSP channel (Jeremy.Martin@comlab.ox.ac.uk)
- Nan Schaller (ncs@cs.rit.edu), Chris Nevison (chris@cs.colgate.edu) and Dyke Stiles (dyke.stiles@ece.usu.edu) - for pioneering the teaching.
- The **WoTUG** community - its workshops, conferences and people.

3-Jul-01

Copyright P.H.Welch

137

URLs

- **CSP** www.comlab.ox.ac.uk/archive/csp.html
- **JCSP** www.cs.ukc.ac.uk/projects/ofa/jcsp/
- **CTJ** www.rt.el.utwente.nl/javapp/
- **KRoC** www.cs.ukc.ac.uk/projects/ofa/kroc/
- **java-threads@ukc.ac.uk**
www.cs.ukc.ac.uk/projects/ofa/java-threads/
- **WoTUG**
wotug.ukc.ac.uk/

3-Jul-01

Copyright P.H.Welch

138