Parallel and Distributed Computing in Education (Invited Talk)

Peter H. Welch

Computing Laboratory, University of Kent at Canterbury, CT2 7NF.
P.H.Welch@ukc.ac.uk

Abstract. The natural world is certainly not organised through a central thread of control. Things happen as the result of the actions and interactions of unimaginably large numbers of independent agents, operating at all levels of scale from nuclear to astronomic. Computer systems aiming to be of real use in this real world need to model, at the appropriate level of abstraction, that part of it for which it is to be of service. If that modelling can reflect the natural concurrency in the system, it ought to be much simpler

Yet, traditionally, concurrent programming is considered to be an advanced and difficult topic – certainly much harder than serial computing which, therefore, needs to be mastered first. But this tradition is wrong.

This talk presents an intuitive, sound and practical model of parallel computing that can be mastered by undergraduate students in the first year of a computing (major) degree. It is based upon Hoare's mathematical theory of *Communicating Sequential Processes* (CSP), but does not require mathematical maturity from the students – that maturity is pre-engineered in the model. Fluency can be quickly developed in both message-passing and shared-memory concurrency, whilst learning to cope with key issues such as race hazards, deadlock, livelock, process starvation and the efficient use of resources. Practical work can be hosted on commodity PCs or UNIX workstations using either Java or the occam multiprocessing language. Armed with this maturity, students are well-prepared for coping with real problems on real parallel architectures that have, possibly, less robust mathematical foundations.

1 Introduction

At Kent, we have been teaching parallel computing at the undergraduate level for the past ten years. Originally, this was presented to first-year students *before* they became too set in the ways of serial logic. When this course was expanded into a full unit (about 30 hours of teaching), timetable pressure moved it into the second year. Either way, the material is easy to absorb and, after only a few (around 5) hours of teaching, students have no difficulty in grappling with the interactions of 25 (say) threads of control, appreciating and eliminating race hazards and deadlock.

Parallel computing is still an immature discipline with many conflicting cultures. Our approach to educating people into successful exploitation of parallel mechanisms is based upon focusing on parallelism as a powerful tool for *simplifying* the description of systems, rather than simply as a means for improving their performance. We never start with an existing serial algorithm and say: 'OK, let's parallelise that!'. And we work solely with a model of concurrency that has a semantics that is *compositional* – a fancy word for WYSIWYG – since, without that property, combinatorial explosions of complexity always get us as soon as we step away from simple examples. In our view, this rules out low-level concurrency mechanisms, such as spin-locks, mutexes and semaphores, as well as some of the higher-level ones (like monitors).

Communicating Sequential Processes (CSP)[1–3] is a mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects. Developed by Tony Hoare in the light of earlier work on monitors, CSP has a compositional semantics that greatly simplifies the design and engineering of such systems – so much so, that parallel design often becomes easier to manage than its serial counterpart. CSP primitives have also proven to be extremely lightweight, with overheads in the order of a few hundred nanoseconds for channel synchronisation (including context-switch) on current microprocessors [4, 5].

Recently, the CSP model has been introduced into the Java programming language [6–10]. Implemented as a library of packages [11, 12], JavaPP[10] enables multithreaded systems to be designed, implemented and reasoned about entirely in terms of CSP synchronisation primitives (channels, events, etc.) and constructors (parallel, choice, etc.). This allows 20 years of theory, design patterns (with formally proven good properties – such as the absence of race hazards, deadlock, livelock and thread starvation), tools supporting those design patterns, education and experience to be deployed in support of Java-based multithreaded applications.

2 Processes, Channels and Message Passing

This section describes a simple and structured multiprocessing model derived from CSP. It is easy to teach and can describe arbitrarily complex systems. No formal mathematics need be presented – we rely on an intuitive understanding of how the world works.

2.1 Processes

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. Each process is alive, executing its own algorithms on its own data. Because those algorithms are executed by the component in its own thread (or threads) of control, they express the behaviour of the component from its own point of view[1]. This considerably simplifies that expression.

A *sequential process* is simply a process whose algorithms execute in a single thread of control. A *network* is a collection of processes (and is, itself, a process). Note that recursive hierarchies of structure are part of this model: a network is a collection of processes, each of which may be a sub-network or a sequential process.

But how do the processes within a network interact to achieve the behaviour required from the network? They can't see each other's data nor execute each other's algorithms – at least, not if they abide by the rules.

2.2 Synchronising Channels

The simplest form of interaction is synchronised message-passing along channels. The simplest form of channel is zero-buffered and point-to-point. Such channels correspond very closely to our intuitive understanding of a wire connecting two (hardware) components.

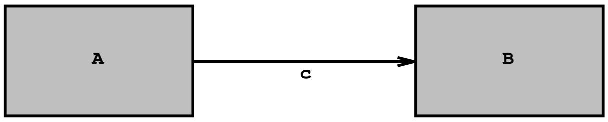

Fig. 1. A simple network

In Figure 1, `A` and `B` are processes and `c` is a channel connecting them. A wire has no capacity to hold data and is only a medium for transmission. To avoid undetected loss of data, channel communication is synchronised. This means that if `A` transmits before `B` is ready to receive, then `A` will block. Similarly, if `B` tries to receive before `A` transmits, `B` will block. When both are ready, a data packet is transferred – directly from the state space of `A` into the state space of `B`. We have a synchronised distributed assignment.

2.3 Legoland

Much can be done, or simplified, just with this basic model – for example the design and simulation of self-timed digital logic, multiprocessor embedded control systems (for which occam[13–16] was orignally designed), GUIs etc.

Here are some simple examples to build up fluency. First we introduce some elementary components from our 'teaching' catalogue – see Figure 2. All processes are cyclic and all transmit and receive just numbers. The `Id` process cycles

[1] This is in contrast with simple 'objects' and their 'methods'. A method body normally executes in the thread of control of the invoking object. Consequently, object behaviour is expressed from the point of view of its environment rather than the object itself. This is a slightly confusing property of traditional 'object-oriented' programming.

through waiting for a number to arrive and, then, sending it on. Although inserting an `Id` process in a wire will clearly not affect the data flowing through it, it *does* make a difference. A bare wire has no buffering capacity. A wire containing an `Id` process gives us a one-place FIFO. Connect 20 in series and we get a 20-place FIFO – sophisticated function from a trivial design.

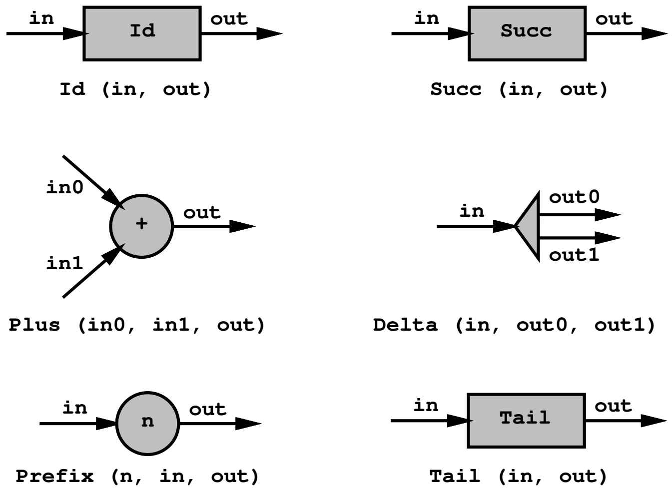

Fig. 2. Extract from a component catalogue

`Succ` is like `Id`, but increments each number as it flows through. The `Plus` component waits until a number arrives on each input line (accepting their arrival in either order) and outputs their sum. `Delta` waits for a number to arrive and, then, broadcasts it in parallel on its two output lines – both those outputs must complete (in either order) before it cycles round to accept further input. `Prefix` first outputs the number stamped on it and then behaves like `Id`. `Tail` swallows its first input without passing it on and then, also, behaves like `Id`. `Prefix` and `Tail` are so named because they perform, respectively, prefixing and tail operations on the streams of data flowing through them.

It's essential to provide a practical environment in which students can develop executable versions of these components and play with them (by plugging them together and seeing what happens). This is easy to do in occam and now, with the JCSP library[11], in Java. Appendices A and B give some of the details. Here we only give some CSP pseudo-code for our catalogue (because that's shorter than the real code):

```
Id (in, out) = in ? x --> out ! x --> Id (in, out)

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

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

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

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

Tail (in, out) = in ? x --> Id (in, out)
```

[Notes: 'free' variables used in these pseudo-codes are assumed to be locally declared and hidden from outside view. All these components are *sequential* processes. The process (`in ? x --> P (...)`) means: "wait until you can engage in the input event (`in ? x`) and, then, become the process `P (...)`". The input operator (`?`) and output operator (`!`) bind more tightly than the `-->`.]

2.4 Plug and Play

Plugging these components together and reasoning about the resulting behaviour is easy. Thanks to the rules on process privacy[2], race hazards leading to unpredictable internal state do not arise. Thanks to the rules on channel synchronisation, data loss or corruption during communication cannot occur[3]. What makes the reasoning simple is that the parallel constructor and channel primitives are deterministic. Non-determinism has to be explicitly designed into a process and coded – it can't sneak in by accident!

Figure 3 shows a simple example of reasoning about network composition. Connect a `Prefix` and a `Tail` and we get two `Id`s:

```
(Prefix (in, c) || Tail (c, out))  =  (Id (in, c) || Id (c, out))
```

Equivalence means that no environment (i.e. external network in which they are placed) can tell them apart. In this case, both circuit fragments implement a 2-place FIFO. The only place where anything different happens is on the internal wire and that's undetectable from outside. The formal proof is a one-liner from the definition of the parallel (`||`), communications (`!`, `?`) and *and-then-becomes* (`-->`) operators in CSP. But the good thing about CSP is that the mathematics engineered into its design and semantics cleanly reflects an intuitive human feel for the model. We can see the equivalence at a glance and this quickly builds confidence both for us and our students.

[2] No external access to internal data. No external execution of internal algorithms (methods).

[3] Unreliable communications over a distributed network can be accommodated in this model – the unreliable network being another active process (or set of processes) that happens not to guarantee to pass things through correctly.

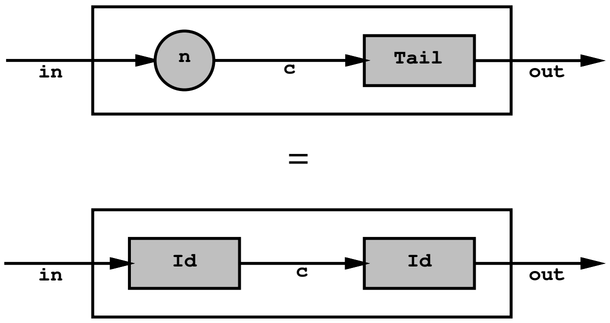

Fig. 3. A simple equivalence

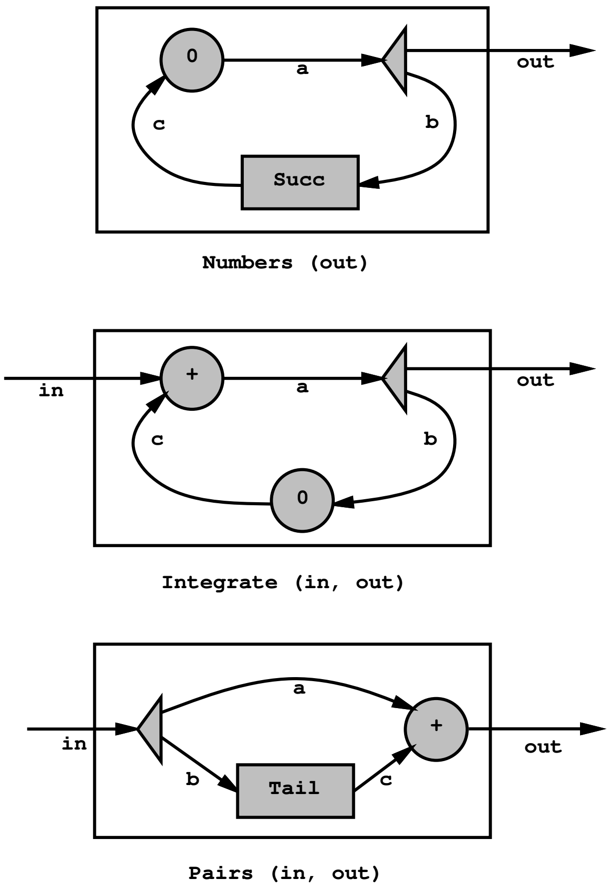

Fig. 4. Some more interesting circuits

Figure 4 shows some more interesting circuits with the first two incorporating feedback. What do they do? Ask the students! Here are some CSP pseudo-codes for these circuits:

```
Numbers (out)
  = Prefix (0, c, a) || Delta (a, out, b) || Succ (b, c)

Integrate (in, out)
  = Plus (in, c, a) || Delta (a, out, b) || Prefix (0, b, c)

Pairs (in, out)
  = Delta (in, a, b) || Tail (b, c) || Plus (a, c, out)
```

Again, our rule for these pseudo-codes means that `a`, `b` and `c` are locally declared channels (hidden, in the CSP sense, from the outside world). Appendices A and B list `occam` and Java executables – notice how closely they reflect the CSP.

Back to what these circuits do: `Numbers` generates the sequence of natural numbers, `Integrate` computes running sums of its inputs and `Pairs` outputs the sum of its last two inputs. If we wish to be more formal, let `c<i>` represent the `i`'th element that passes through channel `c` – i.e. the first element through is `c<1>`. Then, for any `i >= 1`:

```
Numbers:    out<i> = i - 1
Integrate:  out<i> = Sum {in<j> | j = 1..i}
Pairs:      out<i> = in<i> + in<i + 1>
```

Be careful that the above details only *part* of the specification of these circuits: how the values in their output stream(s) relate to the values in their input stream(s). We also have to be aware of how flexible they are in synchronising with their environments, as they generate and consume those streams. The base level components `Id`, `Succ`, `Plus` and `Delta` each demand one input (or pair of inputs) before generating one output (or pair of outputs). `Tail` demands two inputs before its first output, but thereafter gives one output for each input. This effect carries over into `Pairs`. `Integrate` adds 2-place buffering between its input and output channels (ignoring the transformation in the actual values passed). `Numbers` will always deliver to anything trying to take input from it.

If necessary, we can make these synchronisation properties mathematically precise. That is, after all, one of the reasons for which CSP was designed.

2.5 Deadlock – First Contact

Consider the circuit in Figure 5. A simple stream analysis would indicate that:

```
Pairs2:    a<i> = in<i>
Pairs2:    b<i> = in<i>
Pairs2:    c<i> = b<i + 1> = in<i + 1>
Pairs2:    d<i> = c<i + 1> = in<i + 2>
Pairs2:  out<i> = a<i> + d<i> = in<i> + in<i + 2>
```

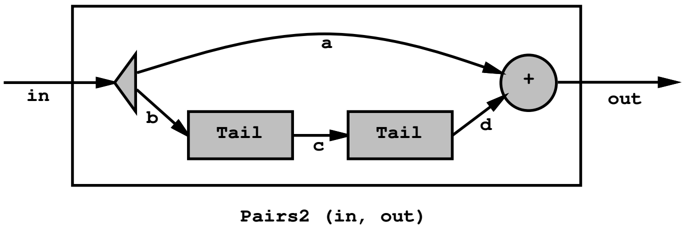

Fig. 5. A dangerous circuit

But this analysis only shows what would be generated *if* anything were generated. In this case, nothing is generated since the system deadlocks. The two `Tail` processes demand three items from `Delta` before delivering anything to `Plus`. But `Delta` can't deliver a third item to the `Tail`s until it's got rid of its second item to `Plus`. But `Plus` won't accept a second item from `Delta` until it's had its first item from the `Tail`s. Deadlock!

In this case, deadlock can be designed out by inserting an `Id` process on the upper (`a`) channel. `Id` processes (and FIFOs in general) have no impact on stream contents analysis but, by allowing a more decoupled synchronisation, *can* impact on whether streams actually flow. Beware, though, that adding buffering to channels is not a general cure for deadlock.

So, there are always two questions to answer: what data flows through the channels, assuming data does flow, and are the circuits deadlock-free? Deadlock is a monster that must – and can – be vanquished. In CSP, deadlock only occurs from a cycle of committed attempts to communicate (input or output): each process in the cycle refusing its predecessor's call as it tries to contact its successor. Deadlock potential is very visible – we even have a deadlock primitive (`STOP`) to represent it, on the grounds that it is a good idea to know your enemy!

In practice, there now exist a wealth of design rules that provide formally proven guarantees of deadlock freedom[17–22]. Design tools supporting these rules – both constructive and analytical – have been researched[23, 24]. Deadlock, together with related problems such as livelock and starvation, need threaten us no longer – even in the most complex of parallel system.

2.6 Structured Plug and Play

Consider the circuits of Figure 6. They are similar to the previous circuits, but contain components other than those from our base catalogue – they use components we have just constructed. Here is the CSP:

```
Fibonacci (out)
  = Prefix (1, d, a) || Prefix (0, a, b) ||
    Delta (b, out, c) || Pairs (c, d)

Squares (out)
  = Numbers (a) || Integrate (a, b) || Pairs (b, out)
```

```
Demo (out)
  = Numbers (a) || Fibonacci (b) || Squares (c) ||
    Tabulate3 (a, b, c, out)
```

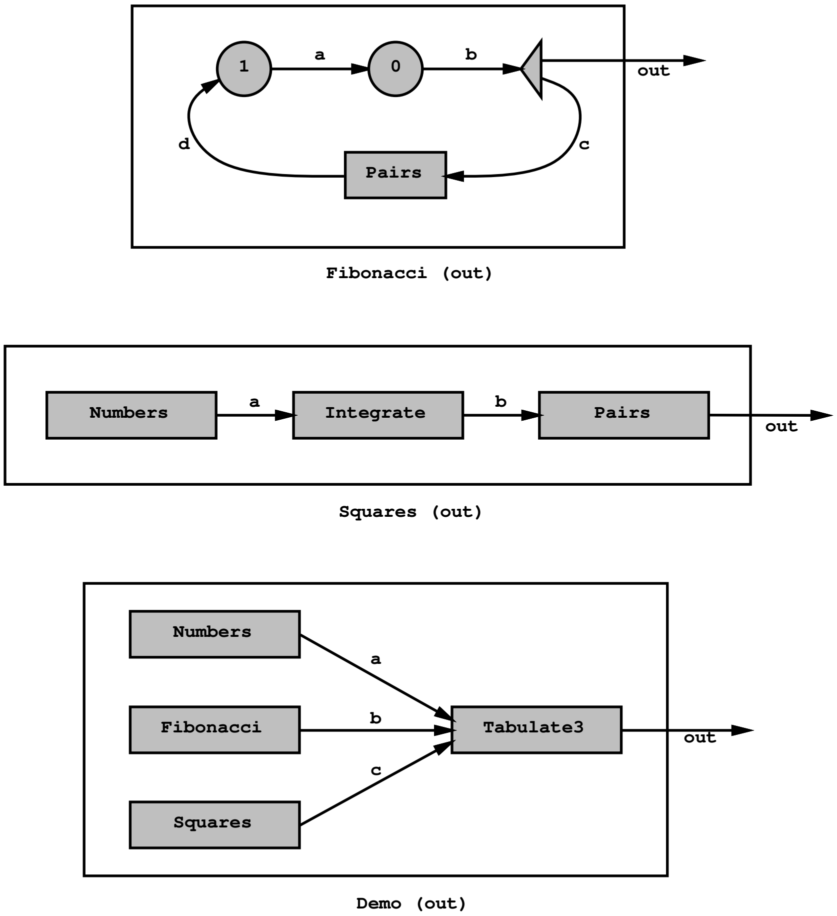

Fig. 6. Circuits of circuits

One of the powers of CSP is that its semantics obey simple composition rules. To understand the behaviour implemented by a network, we only need to know the behaviour of its nodes – not their implementations.

For example, `Fibonacci` is a feedback loop of four components. At this level, we can remain happily ignorant of the fact that its `Pairs` node contains another three. We only need to know that `Pairs` requires two numbers before it outputs anything and that, thereafter, it outputs once for every input. The two `Prefixes` initially inject two numbers (`0` and `1`) into the circuit. Both go into `Pairs`, but only one (their sum) emerges. After this, the feedback loop just contains a single circulating packet of information (successive elements of the Fibonacci sequence). The Delta process taps this circuit to provide external output.

Squares is a simple pipeline of three components. It's best not to think of the nine processes actually involved. Clearly, for i >= 1:

```
Squares:   a<i> = i - 1
Squares:   b<i> = Sum {j - 1 | j = 1..i} = Sum {j | j = 0..(i - 1)}
Squares: out<i> = Sum {j | j = 0..(i - 1)} + Sum {j | j = 0..i} = i * i
```

So, Squares outputs the increasing sequence of squared natural numbers. It doesn't deadlock because Integrate and Pairs only add buffering properties and it's safe to connect buffers in series.

Tabulate3 is from our base catalogue. Like the others, it is cyclic. In each cycle, it inputs in parallel one number from each of its three input channels and, then, generates a line of text on its output channel consisting of a tabulated (15-wide, in this example) decimal representation of those numbers.

```
Tabulate3 (in0, in1, in2, out)
  = ((in0 ? x0 - SKIP) || (in1 ? x1 - SKIP) || (in2 ? x2 - SKIP));
    print (x0, 15, out); print (x1, 15, out); println (x2, 15, out);
    Tabulate3 (in0, in1, in2, out)
```

Connecting the output channel from Demo to a text window displays three columns of numbers: the natural numbers, the Fibonacci sequence and perfect squares.

It's easy to understand all this – thanks to the structuring. In fact, Demo consists of 27 threads of control, 19 of them permanent with the other 8 being repeatedly created and destroyed by the low-level parallel inputs and outputs in the Delta, Plus and Tabulate3 components. If we tried to understand it on those terms, however, we would get nowhere.

Please note that we are not advocating designing at such a fine level of granularity as normal practice! These are only exercises and demonstrations to build up fluency and confidence in concurrent logic. Having said that, the process management overheads for the occam Demo executables are only around 30 microseconds per output line of text (i.e. too low to see) and three milliseconds for the Java (still too low to see). And, of course, if we are using these techniques for designing real hardware[25], we will be working at much finer levels of granularity than this.

2.7 Coping with the Real World – Making Choices

The model we have considered so far – parallel processes communicating through dedicated (point-to-point) channels – is *deterministic*. If we input the same data in repeated runs, we will always receive the same results. This is true regardless of how the processes are scheduled or distributed. This provides a very stable base from which to explore the real world, which doesn't always behave like this.

Any machine with externally operatable controls that influence its internal operation, but whose internal operations will continue to run in the absence of that external control, is not deterministic in the above sense. The scheduling of that external control will make a difference. Consider a car and its driver heading for a brick wall. Depending on when the driver applies the brakes, they will end up in very different states!

CSP provides operators for internal and external choice. An external choice is when a process waits for its environment to engage in one of several events – what happens next is something the environment can determine (e.g. a driver can press the accelerator or brake pedal to make the car go faster or slower). An internal choice is when a process changes state for reasons its environment cannot determine (e.g. a self-clocked timeout or the car runs out of petrol). Note that for the combined (parallel) system of car-and-driver, the accelerating and braking become internal choices so far as the rest of the world is concerned.

occam provides a constructor (`ALT`) that lets a process wait for one of many events. These events are restricted to channel input, timeouts and `SKIP` (a *null* event that has always happened). We can also set pre-conditions – run-time tests on internal state – that mask whether a listed event should be included in any particular execution of the `ALT`. This allows very flexible internal choice within a component as to whether it is prepared to accept an external communication[4]. The JavaPP libraries provide an exact analogue (`Alternative.select`) for these choice mechanisms.

If several events are pending at an `ALT`, an internal choice is normally made between them. However, occam allows a `PRI ALT` which resolves the choice between pending events in order of their listing. This returns control of the operation to the environment, since the reaction of the `PRI ALT`ing process to multiple communications is now predictable. This control is crucial for the provision of real-time guarantees in multi-process systems and for the design of hardware. Recently, extensions to CSP to provide a formal treatment of these mechanisms have been made[26, 27].

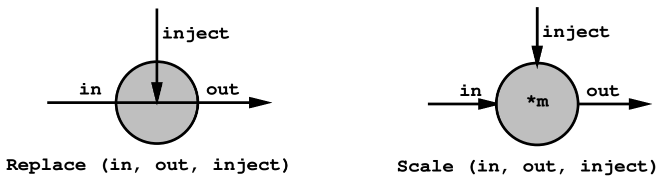

Fig. 7. Two control processes

[4] This is in contrast to *monitors*, whose methods cannot refuse an external call when they are unlocked and have to *wait* on condition variables should their state prevent them from servicing the call. The close coupling necessary between sibling monitor methods to undo the resulting mess is not WYSIWYG[9].

Figure 7 shows two simple components with this kind of control. `Replace` listens for incoming data on its `in` and `inject` lines. Most of the time, data arrives from `in` and is immediately copied to its `out` line. Occasionally, a signal from the `inject` line occurs. When this happens, the signal is copied `out` but, *at the same time*, the next input from `in` is waited for and discarded. In case both `inject` and `in` communications are on offer, priority is given to the (less frequently occurring) `inject`:

```
Replace (in, inject, out)
  = (inject ? signal --> ((in ? x --> SKIP) || (out ! signal --> SKIP))
     [PRI]
     in ? x --> out ! x --> SKIP
    );
    Replace (in, inject, out)
```

`Replace` is something that can be spliced into any channel. If we don't use the `inject` line, all it does is add a one-place buffer to the circuit. If we send something down the `inject` line, it gets injected into the circuit – replacing the next piece of data that would have travelled through that channel.

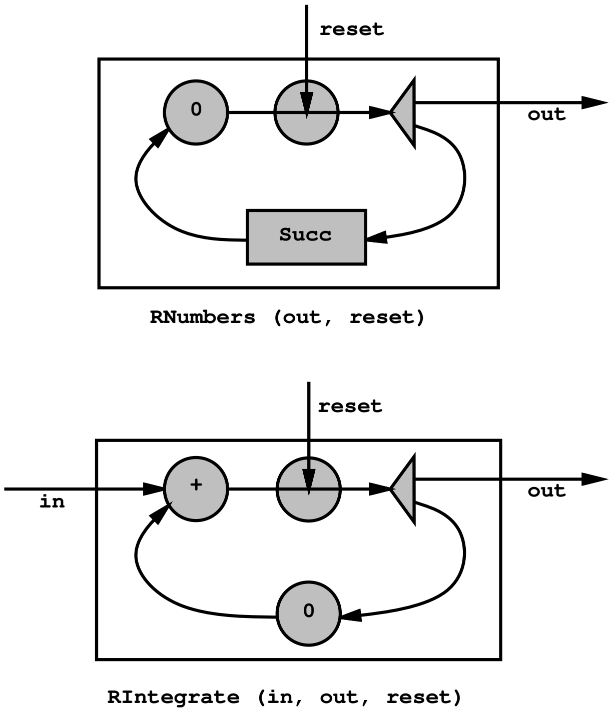

Fig. 8. Two controllable processes

Figure 8 shows `RNumbers` and `RIntegrate`, which are just `Numbers` and `Integrate` with an added `Replace` component. We now have components that are resettable by their environments. `RNumbers` can be reset at any time to continue its output sequence from any chosen value. `RIntegrate` can have its internal running sum redefined.

Like `Replace`, `Scale` (figure 7) normally copies numbers straight through, but scales them by its factor `m`. An `inject` signal resets the scale factor:

```
Scale (m, in, inject, out)
  = (inject ? m --> SKIP
      [PRI]
      in ? x --> out ! m*x --> SKIP
    );
    Scale (m, in, inject, out)
```

Figure 9 shows `RPairs`, which is `Pairs` with the `Scale` control component added. If we send just `+1` or `-1` down the reset line of `RPairs`, we control whether it's adding or subtracting successive pairs of inputs. When it's subtracting, its behaviour changes to that of a *differentiator* – in the sense that it undoes the effect of `Integrate`.

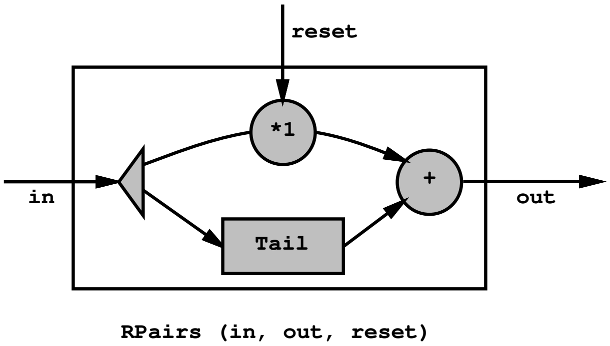

Fig. 9. Sometimes `Pairs`, sometimes `Differentiate`

This allows a nice control demonstration. Figure 10 shows a circuit whose core is a resettable version of the `Squares` pipeline. The `Monitor` process reacts to characters from the `keyboard` channel. Depending on its value, it outputs an appropriate signal down an appropriate reset channel:

```
Monitor (keyboard, resetN, resetI, resetP)
  = (keyboard ? ch -->
        CASE ch
          'N': resetN !  0 --> SKIP
          'I': resetI !  0 --> SKIP
          '+': resetP ! +1 --> SKIP
          '-': resetP ! -1 --> SKIP
    );
    Monitor (keyboard, resetN, resetI, resetP)
```

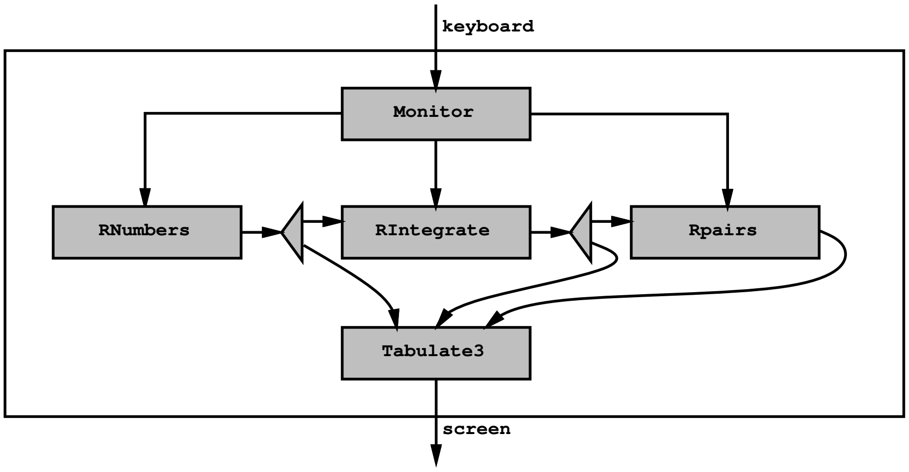

Fig. 10. A user controllable machine

When Demo2 runs and we don't type anything, we see the inner workings of the Squares pipeline tabulated in three columns of output. Keying in an 'N', 'I', '+' or '-' character allows the user some control over those workings[5]. Note that after a '-', the output from RPairs should be the same as that taken from RNumbers.

2.8 A Nastier Deadlock

One last exercise should be done. Modify the system so that output freezes if an 'F' is typed and unfreezes following the next character.

Two 'solutions' offer themselves and Figure 11 shows the *wrong* one (Demo3). This feeds the output from Tabulate3 back to a modified Monitor2 and then on to the screen. The Monitor2 process PRI ALTs between the keyboard channel and this feedback:

```
Monitor2 (keyboard, feedback, resetN, resetI, resetP, screen)
  = (keyboard ? ch -->
       CASE ch
         ...  deal with 'N', 'I', '+', '-' as before
         'F': keyboard ? ch --> SKIP
     [PRI]
     feedback ? x --> screen ! x  --> SKIP
    );
    Monitor2 (keyboard, feedback, resetN, resetI, resetP, screen)
```

[5] In practice, we need to add another process after Tabulate3 to slow down the rate of output to around 10 lines per second. Otherwise, the user cannot properly appreciate the immediacy of control that has been obtained.

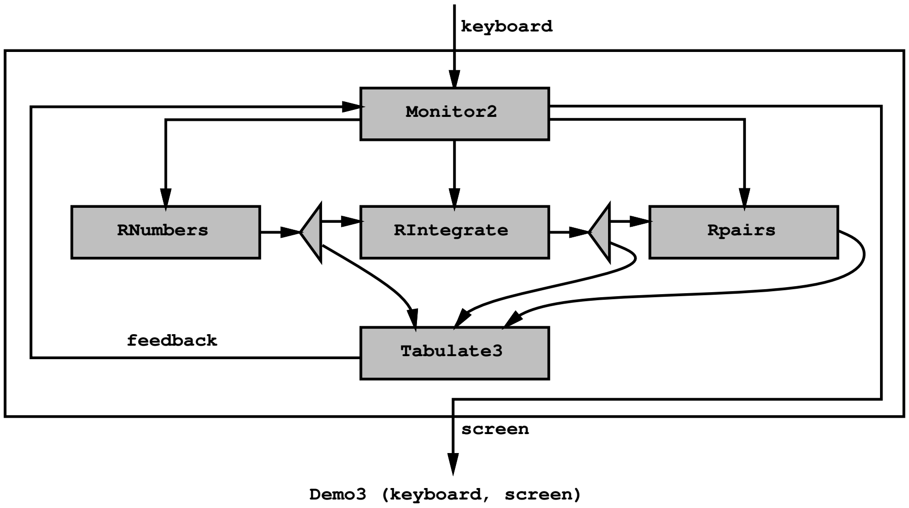

Fig. 11. A machine over which we may lose control

Traffic will normally be flowing along the `feedback-screen` route, interrupted only when `Monitor2` services the `keyboard`. The attraction is that if an '`F`' arrives, `Monitor2` simply waits for the next character (and discards it). As a side-effect of this waiting, the `screen` traffic is frozen.

But if we implement this, we get some worrying behaviour. The freeze operation works fine and so, *probably*, do the '`N`' and '`I`' resets. *Sometimes*, however, a '`+`' or '`-`' reset deadlocks the whole system – the `screen` freezes and all further `keyboard` events are refused!

The problem is that one of the rules for deadlock-free design has been broken: *any data-flow circuit must control the number of packets circulating!* If this number rises to the number of sequential (i.e. lowest level) processes in the circuit, deadlock always results. Each node will be trying to output to its successor and refusing input from its predecessor.

The `Numbers`, `RNumbers`, `Integrate`, `RIntegrate` and `Fibonacci` networks all contain data-flow loops, but the number of packets concurrently in flight is kept at one[6].

In `Demo3` however, packets are continually being generated within `RNumbers`, flowing through several paths to `Monitor2` and, then, to the `screen`. Whenever `Monitor2` feeds a reset back into the circuit, deadlock is possible – although not certain. It depends on the scheduling. `RNumbers` is always pressing new packets into the system, so the circuits are likely to be fairly full. If `Monitor2` generates a reset when they are full, the system deadlocks. The shortest feedback loop is from `Monitor2`, `RPairs`, `Tabulate3` and back to `Monitor2` – hence, it is the '`+`' and '`-`' inputs from `keyboard` that are most likely to trigger the deadlock.

[6] Initially, `Fibonacci` has two packets, but they combine into one before the end of their first circuit.

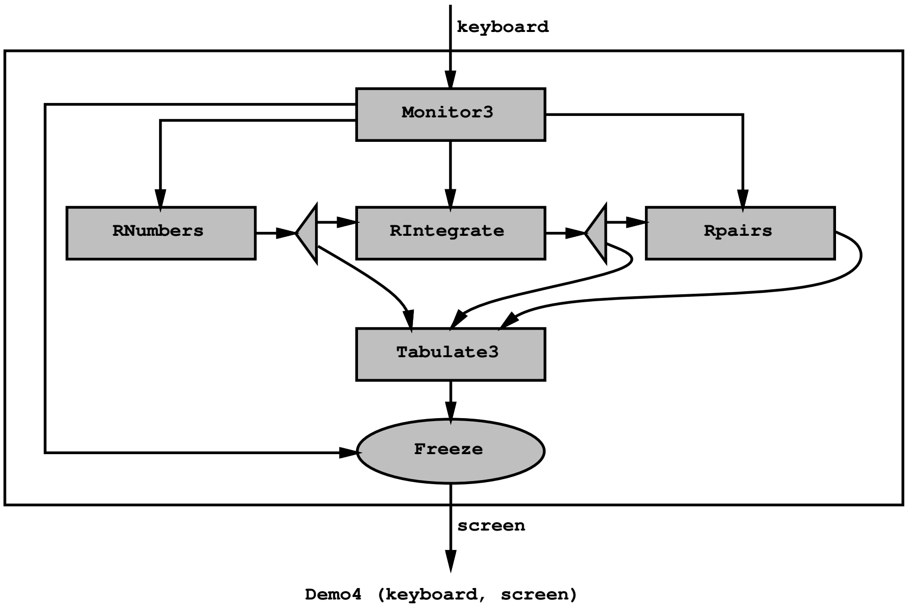

Fig. 12. A machine over which we will not lose control

The design is simply fixed by removing that feedback at this level – see `Demo4` in Figure 12. We have abstracted the freezing operation into its own component (and catalogued it). It's never a good idea to try and do too many functions in one sequential process. That needlessly constrains the synchronisation freedom of the network and heightens the risk of deadlock. Note that the idea being pushed here is that, unless there are special circumstances, *parallel design is safer and simpler than its serial counterpart!*

`Demo4` obeys another golden rule: *every device should be driven from its own separate process.* The `keyboard` and `screen` channels interface to separate devices and should be operated concurrently (in `Demo3`, both were driven from one sequential process – `Monitor2`). Here are the driver processes from `Demo4`:

```
Freeze (in, freeze, out)
  = (freeze ? x --> freeze ? x --> SKIP
     [PRI]
     (in ? x --> out ! x --> SKIP
    );
    Freeze (in, freeze, out)

Monitor3 (keyboard, resetN, resetI, resetP, freeze)
  = (keyboard ? ch -->
       CASE ch
         ...  deal with 'N', 'I', '+', '-' as before
         'F': freeze ! ch --> keyboard ? ch --> freeze ! ch --> SKIP
    );
    Monitor3 (keyboard, resetN, resetI, resetP, freeze)
```

2.9 Buffered and Asynchronous Communications

We have seen how fixed capacity FIFO buffers can be added as active processes to CSP channels. For the occam binding, the overheads for such extra processes are negligible.

With the JavaPP libraries, the same technique *may* be used, but the channel objects can be directly configured to support buffered communications – which saves a couple of context switches. The user may supply objects supporting *any* buffering strategy for channel configuration, including normal blocking buffers, overwrite-when-full buffers, infinite buffers and black-hole buffers (channels that can be written to but not read from – useful for masking off unwanted outputs from components that, otherwise, we wish to reuse intact). However, the user had better stay aware of the semantics of the channels thus created!

Asynchronous communication is commonly found in libraries supporting inter-processor message-passing (such as PVM and MPI). However, the concurrency model usually supported is one for which there is only *one* thread of control on each processor. Asynchronous communication lets that thread of control launch an external communication and continue with its computation. At some point, that computation may need to block until that communication has completed.

These mechanisms are easy to obtain from the concurrency model we are teaching (and which we claim to be general). We don't need anything new. Asynchronous sends are what happen when we output to a buffer (or buffered channel). If we are worried about being blocked when the buffer is full or if we need to block at some later point (should the communication still be unfinished), we can simply spawn off another process[7] to do the send:

```
(out ! packet --> SKIP |PRI| someMoreComputation (...));
continue (...)
```

The `continue` process only starts when both the `packet` has been sent and `someMoreComputation` has finished. `someMoreComputation` and sending the `packet` proceed concurrently. We have used the priority version of the parallel operator (`|PRI|`, which gives priority to its left operand), to ensure that the sending process initiates the transfer before the `someMoreComputation` is scheduled. Asynchronous receives are implemented in the same way:

```
(in ? packet --> SKIP |PRI| someMoreComputation (...));
continue (...)
```

2.10 Shared Channels

CSP channels are strictly point-to-point. occam3[28] introduced the notion of (securely) *shared* channels and channel structures. These are further extended in the KRoC occam[29] and JavaPP libraries and are included in the teaching model.

[7] The occam overheads for doing this are less than half a microsecond.

A channel structure is just a record (or object) holding two or more CSP channels. Usually, there would be just two channels – one for each direction of communication. The channel structure is used to conduct a two-way conversation between two processes. To avoid deadlock, of course, they will have to understand protocols for using the channel structure – such as who speaks first and when the conversation finishes. We call the process that opens the conversation a *client* and the process that listens for that call a *server*[8].

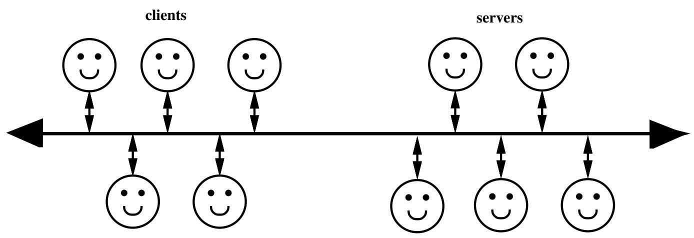

Fig. 13. A many-many shared channel

The CSP model is extended by allowing multiple clients and servers to share the same channel (or channel structure) – see Figure 13. Sanity is preserved by ensuring that only one client and one server use the shared object at any one time. Clients wishing to use the channel queue up first on a client-queue (associated with the shared channel) – servers on a server-queue (also associated with the shared channel). A client only completes its actions on the shared channel when it gets to the front of its queue, finds a server (for which it may have to wait if business is good) and completes its transaction. A server only completes when it reaches the front of its queue, finds a client (for which it may have to wait in times of recession) and completes its transaction.

Note that shared channels – like the choice operator between multiple events – introduce scheduling dependent non-determinism. The order in which processes are granted access to the shared channel depends on the order in which they join the queues.

Shared channels provide a very efficient mechanism for a common form of choice. Any server that offers a non-discriminatory service[9] to multiple clients should use a shared channel, rather than `ALT`ing between individual channels from those clients. The shared channel has a constant time overhead – `ALT`ing is linear on the number of clients. However, if the server needs to discriminate between its clients (e.g. to refuse service to some, depending upon its internal state), `ALT`ing gives us that flexibility. The mechanisms can be efficiently combined. Clients can be grouped into equal-treatment partitions, with each group clustered on its own shared channel and the server `ALT`ing between them.

[8] In fact, the client/server relationship is with respect to the channel structure. A process may be both a server on one interface and a client on another.

[9] Examples for such servers include window managers for multiple animation processes, data loggers for recording traces from multiple components from some machine, etc.

For deadlock freedom, each server must guarantee to respond to a client call within some bounded time. During its transaction with the client, it must follow the protocols for communication defined for the channel structure *and* it may engage in separate client transactions with other servers. A client may open a transaction at any time but may not interleave its communications with the server with any other synchronisation (e.g. with another server). These rules have been formalised as CSP specifications[21]. Client-server networks may have plenty of data-flow feedback but, so long as no cycle of client-server relations exist, [21] gives formal proof that the system is deadlock, livelock and starvation free.

Shared channel structures may be stretched across distributed memory (e.g. networked) multiprocessors[15]. Channels may carry all kinds of object – including channels and processes themselves. A shared channel is an excellent means for a client and server to find each other, pass over a private channel and communicate independently of the shared one. Processes will drag pre-attached channels with them as they are moved and can have local channels dynamically (and temporarily) attached when they arrive. See David May's work on *Icarus*[30, 31] for a consistent, simple and practical realisation of this model for distributed and mobile computing.

3 Events and Shared Memory

Shared memory concurrency is often described as being 'easier' than message passing. But great care must be taken to synchronise concurrent access to shared data, else we will be plagued with race hazards and our systems will be useless. CSP primitives provide a sharp set of tools for exercising this control.

3.1 Symmetric Multi-Processing (SMP)

The private memory/algorithm principles of the underlying model – and the security guarantees that go with them – are a powerful way of programming shared memory multiprocessors. Processes can be automatically and dynamically scheduled between available processors (*one object code fits all*). So long as there is an excess of (runnable) processes over processors and the scheduling overheads are sufficiently low, high multiprocessor efficiency can be achieved – with guaranteed no race hazards. With the design methods we have been describing, it's very easy to generate *lots* of processes with most of them runnable most of the time.

3.2 Token Passing and Dynamic CREW

Taking advantage of shared memory to communicate between processes is an extension to this model and must be synchronised. The shared data does not belong to any of the sharing processes, but must be globally visible to them – either on the stack (for occam) or heap (for Java).

The JavaPP channels in previous examples were only used to send data *values* between processes – but they can also be used to send objects. This steps outside the automatic guarantees against race hazard since, unconstrained, it allows parallel access to the same data. One common and useful constraint is only to send *immutable* objects. Another design pattern treats the sent object as a *token* conferring permission to use it – the sending process losing the token as a side-effect of the communication. The trick is to ensure that only one copy of the token ever exists for each sharable object.

Dynamic CREW (Concurrent Read Exclusive Write) operations are also possible with shared memory. Shared channels give us an efficient, elegant and easily provable way to construct an active *guardian* process with which application processes synchronise to effect CREW access to the shared data. Guarantees against starvation of writers by readers – and vice-versa – are made. Details will appear in a later report (available from [32]).

3.3 Structured Barrier Synchronisation and SPMD

Point-to-point channels are just a specialised form of the general CSP multi-process synchronising *event*. The CSP parallel operator binds processes together with events. When *one* process synchronises on an event, *all* processes registered for that event must synchronise on it before that first process may continue. Events give us structured multiway barrier synchronisation[29].

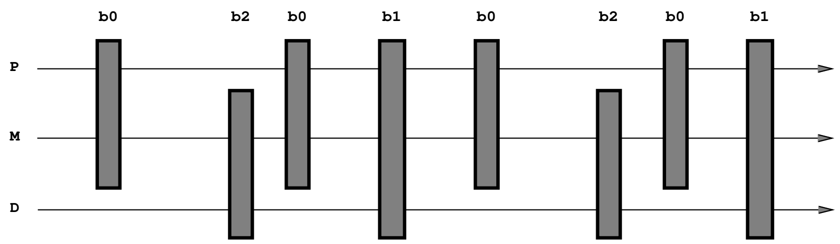

Fig. 14. Multiple barriers to three processes

We can have many event barriers in a system, with different (and not necessarily disjoint) subsets of processes registered for each barrier. Figure 14 shows the execution traces for three processes (`P`, `M` and `D`) with time flowing horizontally. They do not all progress at the same – or even constant – speed. From time to time, tha faster ones will have to wait for their slower partners to reach an agreed barrier before all of them can proceed. We can wrap up the system in typical SPMD form as:

```
|| <i = 0 FOR 3>
   S (i, ..., b0, b1, b2)
```

where b0, b1 and b2 are events. The replicated parallel operator runs 3 instances of S in parallel (with i taking the values 0, 1 and 2 respectively in the different instances). The S process simply switches into the required form:

```
S (i, ..., b0, b1, b2)
  = CASE i
      0 : P (..., b0, b1)
      1 : M (..., b0, b1, b2)
      2 : D (..., b1, b2)
```

and where P, M and D are registered only for the events in their parameters. The code for P has the form:

```
P (..., b0, b1)
   = someWork (...); b0 --> SKIP;
     moreWork (...); b0 --> SKIP;
     lastBitOfWork (...); b1 --> SKIP;
     P (..., b0, b1)
```

3.4 Non-Blocking Barrier Synchronisation

In the same way that asynchronous communications can be expressed (section 2.9), we can also achieve the somewhat contradictory sounding, but potentially useful, *non-blocking* barrier synchronisation.

In terms of serial programming, this is a two-phase commitment to the barrier. The first phase declares that we have done everything we need to do this side of the barrier, but does not block us. We can then continue for a while, doing things that do not disturb what we have set up for our partners in the barrier and do not need whatever it is that they have to set. When we need their work, we enter the second phase of our synchronisation on the barrier. This blocks us only if there is one, or more, of our partners who has not reached the first phase of its synchronisation. With luck, this window on the barrier will enable most processes most of the time to pass through without blocking:

```
doOurWorkNeededByOthers (...);
barrier.firstPhase ();
privateWork (...);
barrier.secondPhase ();
useSharedResourcesProtectedByTheBarrier (...);
```

With our lightweight CSP processes, we do not need these special phases to get the same effect:

```
doOurWorkNeededByOthers (...);
(barrier --> SKIP |PRI| privateWork (...));
useSharedResourcesProtectedByTheBarrier (...);
```

The explanation as to why this works is just the same as for the asynchronous sends and receives.

3.5 Bucket Synchronisation

Although CSP allows choice over general events, the occam and Java bindings do not. The reasons are practical – a concern for run-time overheads[10]. So, synchronising on an event commits a process to wait until everyone registered for the event has synchronised. These multi-way events, therefore, do not introduce non-determinism into a system and provide a stable platform for much scientific and engineering modelling.

Buckets[15] provide a non-deterministic version of events that are useful for when the system being modelled is irregular and dynamic (e.g. motor vehicle traffic[33]). Buckets have just two operations: `jump` and `kick`. There is no limit to the number of processes that can jump into a bucket – where they all block. Usually, there will only be one process with responsibility for kicking over the bucket. This can be done at any time of its own (internal) choosing – hence the non-determinism. The result of kicking over a bucket is the unblocking of all the processes that had jumped into it[11].

4 Conclusions

A simple model for parallel computing has been presented that is easy to learn, teach and use. Based upon the mathematically sound framework of Hoare's CSP, it has a compositional semantics that corresponds well with out intuition about how the world is constructed. The basic model encompasses object-oriented design with active processes (i.e. objects whose methods are exclusively under their own thread of control) communicating via passive, but synchronising, wires. Systems can be composed through natural layers of communicating components so that an understanding of each layer does not depend on an understanding of the inner ones. In this way, systems with arbitrarily complex behaviour can be safely constructed – free from race hazard, deadlock, livelock and process starvation.

A small extension to the model addresses fundamental issues and paradigms for shared memory concurrency (such as token passing, CREW dynamics and bulk synchronisation). We can explore with equal fluency serial, message-passing and shared-memory logic and strike whatever balance between them is appropriate for the problem under study. Applications include hardware design (e.g. FPGAs and ASICs), real-time control systems, animation, GUIs, regular and irregular modelling, distributed and mobile computing.

occam and Java bindings for the model are available to support practical work on commodity PCs and workstations. Currently, the occam bindings are

[10] Synchronising on an event in occam has a unit time overhead, regardless of the number of processes registered. This includes being the last process to synchronise, when all blocked processes are released. These overheads are well below a microsecond for modern microprocessors.

[11] As for events, the `jump` and `kick` operations have constant time overhead, regardless of the number of processes involved. The bucket overheads are slightly lower than those for events.

the fastest (context-switch times under 300 nano-seconds), lightest (in terms of memory demands), most secure (in terms of guaranteed thread safety) and quickest to learn. But Java has the libraries (e.g. for GUIs and graphics) and will get faster. Java thread safety, in this context, depends on following the CSP design patterns – and these are easy to acquire[12].

The JavaPP JCSP library[11] also includes an extension to the Java AWT package that drops channel interfaces on all GUI components[13]. Each item (e.g. a `Button`) is a process with a `configure` and `action` channel interface. These are connected to separate internal handler processes. To change the text or colour of a `Button`, an application process outputs to its `configure` channel. If someone presses the `Button`, it outputs down its `action` channel to an application process (which can accept or refuse the communication as it chooses). Example demonstrations of the use of this package may be found at [11]. Whether GUI programming through the process-channel design pattern is simpler than the listener-callback pattern offered by the underlying AWT, we leave for the interested reader to experiment and decide.

All the primitives described in this paper are available for KRoC occam and Java. Multiprocessor versions of the KRoC kernel targeting NoWs and SMPs will be available later this year. SMP versions of the JCSP[11] and CJT[12] libraries are automatic if your JVM supports SMP threads. Hooks are provided in the channel libraries to allow user-defined network drivers to be installed. Research is continuing on portable/faster kernels and language/tool design for *enforcing* higher level aspects of CSP design patterns (e.g. for shared memory safety and deadlock freedom) that currently rely on self-discipline.

Finally, we stress that this is *undergraduate* material. The concepts are mature and fundamental – *not* advanced – and the earlier they are introduced the better. For developing fluency in concurrent design and implementation, no special hardware is needed. Students can graduate to real parallel systems once they have mastered this fluency. The CSP model is neutral with respect to parallel architecture so that coping with a change in language or paradigm is straightforward. However, even for uni-processor applications, the ability to do safe and lightweight multithreading is becoming crucial *both* to improve response times *and* simplify their design.

The experience at Kent is that students absorb these ideas very quickly and become very creative[14]. Now that they can apply them in the context of Java, they are smiling indeed.

[12] Java active objects (processes) do not invoke each other's methods, but communicate only through shared passive objects with carefully designed synchronisation properties (e.g. channels and events). Shared use of *user*-defined passive objects will be automatically thread-safe so long as the usage patterns outlined in Section 3 are kept – their methods should not be `synchronized` (in the sense of Java monitors).

[13] We believe that the new Swing GUI libraries from Sun (that will replace the AWT) can also be extended through a channel interface for secure use in parallel designs – despite the warnings concerning the use of Swing and multithreading[34].

[14] The JCSP libraries used in Appendix B were produced by Paul Austin, an undergraduate student at Kent.

References

1. C.A. Hoare. Communication Sequential Processes. *CACM*, 21(8):666–677, August 1978.
2. C.A. Hoare. *Communication Sequential Processes.* Prentice Hall, 1985.
3. Oxford University Computer Laboratory. *The CSP Archive.* <URL: http:// www.comlab.ox.ac.uk/ archive/ csp.html>, 1997.
4. P.H. Welch and D.C. Wood. KRoC – the Kent Retargetable occam Compiler. In B. O'Neill, editor, *Proceedings of WoTUG 19*, Amsterdam, March 1996. WoTUG, IOS Press. <URL:http:// www.hensa.ac.uk/ parallel/ occam/ projects/ occam-for-all/ kroc/>.
5. Peter H. Welch and Michael D. Poole. occam for Multi-Processor DEC Alphas. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 189–198, Amsterdam, Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press.
6. Peter Welch et al. *Java Threads Workshop – Post Workshop Discussion.* <URL:http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/>, February 1997.
7. Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 48–76, 1997. (See reference [5]).
8. G.H. Hilderink. Communicating Java Threads Reference Manual. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 283–325, 1997. (See reference [5]).
9. Peter Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A. Bakkers, editors, *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, Netherlands, April 1998. World occam and Transputer User Group (WoTUG), IOS Press. ISBN 90-5199-391-9.
10. Alan Chalmers. *JavaPP Page – Bristol.* <URL:http://www.cs.bris.ac.uk/ ~alan/ javapp.html/>, May 1998.
11. P.D. Austin. *JCSP Home Page.* <URL:http://www.hensa.ac.uk/parallel/languages/ java/jcsp/>, May 1998.
12. Gerald Hilderink. *JavaPP Page – Twente.* <URL:http://www.rt.el.utwente.nl/ javapp/>, May 1998.
13. Ian East. *Parallel Processing with Communication Process Architecture.* UCL press, 1995. ISBN 1-85728-239-6.
14. John Galletly. *occam 2 – including occam 2.1.* UCL Press, 1996. ISBN 1-85728-362-7.
15. occam-for-all Team. *occam-for-all Home Page.* <URL:http://www.hensa.ac.uk/ parallel/occam/occam-for-all/>, February 1997.
16. Mark Debbage, Mark Hill, Sean Wykes, and Denis Nicole. Southampton's Portable occam Compiler (SPoC). In R. Miles and A. Chalmers, editors, *Progress in Transputer and occam Research, Proceedings of WoTUG 17*, Concurrent Systems Engineering, pages 40–55, Amsterdam, Netherlands, April 1994. World occam and Transputer User Group (WoTUG), IOS Press. <URL:http://www.hensa.ac.uk/ parallel/ occam/ compilers/ spoc/>.
17. J.M.R. Martin and S.A. Jassim. How to Design Deadlock-Free Networks Using CSP and Verification Tools – a Tutorial Introduction. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 326–338, 1997. (See reference [5]).

18. A.W. Roscoe and N. Dathi. The Pursuit of Deadlock Freedom. Technical Report *Technical Monograph PRG-57*, Oxford University Computing Laboratory, 1986.
19. J. Martin, I. East, and S. Jassim. Design Rules for Deadlock Freedom. *Transputer Communications*, 2(3):121–133, September 1994. John Wiley & Sons, Ltd. ISSN 1070-454X.
20. P.H. Welch, G.R.R. Justo, and C. Willcock. High-Level Paradigms for Deadlock-Free High-Performance Systems. In Grebe et al., editors, *Transputer Applications and Systems '93*, pages 981–1004, Amsterdam, 1993. IOS Press. ISBN 90-5199-140-1.
21. J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996. John Wiley & Sons, Ltd. ISSN 1070-454X.
22. A.W. Roscoe. *Model Checking* CSP*, A Classical Mind*. Prentice Hall, 1994.
23. J.M.R. Martin and S.A. Jassim. A Tool for Proving Deadlock Freedom. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 1–16, 1997. (See reference [5]).
24. D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In *Proceedings of UK Parallel '96*, pages 53–69, London, July 1996. BCS PPSIG, Springer-Verlag. ISBN 3-540-76068-7.
25. M. Aubury, I. Page, D. Plunkett, M. Sauer, and J. Saul. Advanced Silicon Prototyping in a Reconfigurable Environment. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 81–92, 1998. (See reference [9]).
26. A.E. Lawrence. Extending CSP. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 111–132, 1998. (See reference [9]).
27. A.E. Lawrence. HCSP: Extending CSP for Co-design and Shared Memory. In *Architectures, Languages and Patterns, Proceedings of WoTUG 21*, pages 133–156, 1998. (See reference [9]).
28. Geoff Barrett. occam3 reference manual (draft). <URL:http:// www.hensa.ac.uk/ parallel/occam/documents/>, March 1992. (unpublished in paper).
29. Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In *Parallel Programming and Java, Proceedings of WoTUG 20*, pages 104–129, 1997. (See reference [5]).
30. David May and Henk L Muller. Icarus language definition. Technical Report CSTR-97-007, Department of Computer Science, University of Bristol, January 1997.
31. Henk L. Muller and David May. A simple protocol to communicate channels over channels. Technical Report CSTR-98-001, Department of Computer Science, University of Bristol, January 1998.
32. D.J. Beckett. *Java Resources Page.* <URL:http://www.hensa.ac.uk/parallel/ languages/java/>, May 1998.
33. Kang Hsin Lu, Jeff Jones, and Jon Kerridge. Modelling Congested Road Traffic Networks Using a Highly Parallel System. In A. DeGloria, M.R. Jane, and D. Marini, editors, *Transputer Applications and Systems '94*, volume 42 of *Concurrent Systems Engineering*, pages 634–647, Amsterdam, Netherlands, September 1994. The Transputer Consortium, IOS Press. ISBN 90-5199-177-0.
34. Hans Muller and Kathy Walrath. Threads and swing. <URL:http://java.sun.com/ products/jfc/swingdoc-archive/threads.html>, April 1998.

Appendix A: occam Executables

Space only permits a sample of the examples to be shown here. This first group are from the 'Legoland' catalogue (Section 2.3):

```
PROC Id (CHAN OF INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x
:
```

```
PROC Succ (CHAN OF INT in, out)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      out ! x PLUS 1
:
```

```
PROC Plus (CHAN OF INT in0, in1, out)
  WHILE TRUE
    INT x0, x1:
    SEQ
      PAR
        in0 ? x0
        in1 ? x1
      out ! x0 PLUS x1
:
```

```
PROC Prefix (VAL INT n, CHAN OF INT in, out)
  SEQ
    out ! n
    Id (in, out)
:
```

Next come four of the 'Plug and Play' examples from Sections 2.4 and 2.6:

```
PROC Numbers (CHAN OF INT out)
  CHAN OF INT a, b, c:
  PAR
    Prefix (0, c, a)
    Delta (a, out, b)
    Succ (b, c)
:
```

```
PROC Integrate (CHAN OF INT in, out)
  CHAN OF INT a, b, c:
  PAR
    Plus (in, c, a)
    Delta (a, out, b)
    Prefix (0, b, c)
:
```

```
PROC Pairs (CHAN OF INT in, out)
  CHAN OF INT a, b, c:
  PAR
    Delta (in, a, b)
    Tail (b, c)
    Plus (a, c, out)
:
```

```
PROC Squares (CHAN OF INT out)
  CHAN OF INT a, b:
  PAR
    Numbers (a)
    Integrate (a, b)
    Pairs (b, out)
:
```

Here is one of the controllers from Section 2.7:

```
PROC Replace (CHAN OF INT in, inject, out)
  WHILE TRUE
    PRI ALT
      INT x:
      inject ? x
        PAR
          INT discard:
          in ? discard
          out ! x
      INT x:
      in ? x
        out ! x
:
```

Asynchronous receive from Section 2.9:

```
SEQ
  PRI PAR
    in ? packet
    someMoreComputation (...)
  continue (...)
```

Barrier synchronisation from Section 3.3:

```
PROC P (..., EVENT b0, b2)
  ...  local state declarations
  SEQ
    ...  initialise local state
    WHILE TRUE
      SEQ
        someWork (...)
        synchronise.event (b0)
        moreWork (...)
        synchronise.event (b0)
        lastBitOfWork (...)
        synchronise.event (b1)
:
```

Finally, non-blocking barrier synchronisation from Section 3.4:

```
SEQ
  doOurWorkNeededByOthers (...)
  PRI PAR
    synchronise.event (barrier)
    privateWork (...)
  useSharedResourcesProtectedByTheBarrier (...)
```

Appendix B: Java Executables

These examples use the JCSP library for processes and channels[11]. A process is an instance of a class that implements the `CSProcess` interface. This is similar to, but different from, the standard `Runable` interface:

```
package jcsp.lang;

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

For example, from the 'Legoland' catalogue (Section 2.3):

```
import jcsp.lang.*;        // processes and object carrying channels
import jcsp.lang.ints.*;   // integer versions of channels

class Succ implements CSProcess {

  private ChannelInputInt in;
  private ChannelOutputInt out;

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

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

class Prefix implements CSProcess {

  private int n;
  private ChannelInputInt in;
  private ChannelOutputInt out;

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

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

JCSP provides a `Parallel` class that combines an array of `CSProcess`es into a `CSProcess`. It's execution is the parallel composition of that array. For example, here are two of the 'Plug and Play' examples from Sections 2.4 and 2.6:

```
  class Numbers implements CSProcess {

    private ChannelOutputInt out;

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

    public void run () {
      One2OneChannelInt a = new One2OneChannelInt ();
      One2OneChannelInt b = new One2OneChannelInt ();
      One2OneChannelInt c = new One2OneChannelInt ();
      new Parallel (
        new CSProcess[] {
          new Delta (a, out, b),
          new Succ (b, c),
          new Prefix (0, c, a),
        }
      ).run ();
    }
  }

  class Squares implements CSProcess {

    private ChannelOutputInt out;

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

    public void run () {
      One2OneChannelInt a = new One2OneChannelInt ();
      One2OneChannelInt b = new One2OneChannelInt ();
      new Parallel (
        new CSProcess[] {
          new Numbers (a),
          new Integrate (a, b),
          new Pairs (b, out),
        }
      ).run ();
    }
  }
```

Here is one of the controllers from Section 2.7. The processes `ProcessReadInt` and `ProcessWriteInt` just read and write a single integer (into and from a public `value` field) and, then, terminate:

```
class Replace implements CSProcess {

  private AltingChannelInputInt in;
  private AltingChannelInputInt inject;
  private ChannelOutputInt out;

  public Replace (AltingChannelInputInt in,
                  AltingChannelInputInt inject,
                  ChannelOutputInt out) {
    this.in = in;
    this.inject = inject;
    this.out = out;
  }

  public void run () {

    Alternative alt = new Alternative (new Guard[] {inject, in});
    final int INJECT = 0, IN = 1;          // Guard indices (prioritised)

    ProcessWriteInt forward = new ProcessWriteInt (out);  // a CSProcess
    ProcessReadInt discard = new ProcessReadInt (in);     // a CSProcess
    CSProcess parIO  = new Parallel (new CSProcess[] {discard, forward});

    while (true) {
      switch (alt.priSelect ()) {
        case INJECT:
          forward.value = inject.read ();
          parIO.run ();
        break;
        case IN:
          out.write (in.read ());
        break;
      }
    }
  }
}
```

JCSP also has channels for sending and receiving arbitrary `Objects`. Here is an asynchronous receive (from Section 2.9) of an expected `Packet`:

```
// set up processes once (before we start looping ...)

ProcessRead readObj = new ProcessRead (in);               // a CSProcess
CSProcess someMore = new someMoreComputation (...);
CSProcess async = new PriParallel (new CSProcess[] {readObj, someMore});

while (looping) {
  async.run ();
  Packet packet = (Packet) readObj.value
  continue (...);
}
```