Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2013

Life of occam-Pi

Peter H. WELCH

School of Computing, University of Kent, UK

`p.h.welch@kent.ac.uk`

Abstract. This paper considers some questions prompted by a brief review of the history of computing. Why is programming so hard? Why is concurrency considered an *"advanced"* subject? What's the matter with *Objects*? Where did all the Maths go? In searching for answers, the paper looks at some concerns over fundamental ideas within *object orientation* (as represented by modern programming languages), before focussing on the concurrency model of communicating processes and its particular expression in the occam family of languages. In that focus, it looks at the history of occam, its underlying philosophy (*Ockham's Razor*), its semantic foundation on Hoare's CSP, its principles of *process oriented design* and its development over almost three decades into occam-π (which blends in the concurrency dynamics of Milner's π-calculus). Also presented will be an urgent need for rationalisation – occam-π is an experiment that has demonstrated significant results, but now needs time to be spent on careful review and implementing the conclusions of that review. Finally, the future is considered. In particular, is there a future?

Keywords. process, object, local reasoning, global reasoning, occam-pi, concurrency, compositionality, verification, multicore, efficiency, scalability, safety, simplicity

Introduction

Let's try a few questions to which the answer might be: *"occam, obviously"*. Which language most grievously irritates programmers who want to hack code quickly? Quite possibly – though it might also answer the question about which language *really* supports programmers who want to hack code quickly. Which language has concurrency built into its core design with a formal semantics founded on process algebra? Definitely – to the extent that it almost *is* a process algebra, with laws and proof rules and a model checker to enable formal verification to be managed as part of the program writing experience (by programmers who are not mathematicians and who do this as a matter of course).

Which language has concurrency safety rules built into its design that enable the compiler to prohibit data race hazards (even when the concurrency is highly dynamic, with process networks forming and breaking up dynamically)? Which language has a concurrency model that is deterministic *by default*, but which can build non-deterministic systems (when necessary) *by design*? Which language has the fastest and most scalable multicore scheduler on the planet (and which is proving highly effective in large-scale complex system simulations and the study of emergent behaviour)? Which language can be introduced to undergraduate CS students in *Fresher* week so that they can be programming LEGO robots to navigate an environment (reacting to bumps and following tracks) using concurrent processes for sensor data gathering, brain logic and motor drivers ... in 90 minutes? Which language had large-scale industrial use in embedded applications twenty years ago ... but has now been mostly forgotten? Yes, yes, yes, yes and (sadly) yes. Enough! Let's look at some history.

1. Some History

1.1. A Long Long Time Ago[1] *...*

Computers were invented: calculating machines that were programmable and *Turing complete* ... meaning they could do anything any future computer could do (which was pretty cool).

Babbage's *Analytical Engine* [1] (1837) was probably the first, but was too far ahead of its time and handicapped by lack of funding and a bad (in retrospect) engineering decision over number representation (decimal). For working machines, we had to wait for a better understanding of the theory (Russell's *Principia Mathematica* [2,3] (1910-13), Gödel's *Incompleteness Theorem* [4] (1931), Church's λ*-calculus* [5] (1936) and Turing's *Computable Numbers* [6] (1936)) and seriously better funding (the *Second World War*). Built, programmed and operated in secret, they arrived: the Zuse Z3 [7] (Germany, 1941), Colossus [8] (Britain, 1943) and, probably, several others. Colossus, in particular, had significant impact on war outcomes.

Following the war, the pace picked up as the commercial potential became obvious – in the USA, at least (ENIAC [9], 1946). Ideas for hardware and software developed at a rate unprecedented for any other technology. A new and unfortunate race, concerning the phenomenon known as *Moore's Law*, started and continues to this day. Paraphrasing David May [10]: *"Hardware capabilities double every 18 months. Unfortunately, software overheads respond by matching this ..."*. So, we seem to be making progress – but is anything useful happening? There is cause for concern.

Edsger Dijkstra made a similar point: *"In the beginning when we had no computers, we had no problems. Then when we had small computers, we had small problems. Now that we have big computers, we have big problems ..."* [11]. Of course, now (2013) we have tiny computers again – mass-produced and in most everyone's pockets. Thanks to the ingenuity of our hardware engineers, their astonishing power is barely related to their physical size; and thanks to the ingenuity of our software engineers, we have astonishing problems. Our laptops/tablets/phones are millions of times more powerful than the computers that guided the Apollo astronauts to the moon. But what are they doing most of the time ... why, so often, won't they respond to us ... why the spinning wheel (or hour-glass) of death?

1.2. Bad News on the Doorstep ...

Something went wrong. A message in the preceding section is that mathematics is probably important. The pioneers of computing were mathematicians and engineers, who relied on (or were inventing) established mathematical models of the materials they were engineering. Such foundations seem largely ignored today. Walk into a class of (say) second-year Computer Science undergraduates in any (maybe, almost any) university and ask them what a *loop invariant* is ... or *recursion invariant* (if they are into functional programming) or *class invariant* (if they are into object-orientation). For the past three years, I have done that in my university and received mostly blank stares. This is not an uncommon reaction, but gentle further questioning revealed that they really didn't know. Some, thank goodness, did express curiosity though. I fear the result would be the same in most programming rooms (bedrooms/garages) of industry where most of the software we (have to) use gets written.

Nailing *invariants* in a system is one of the most powerful forms of analysis, whether in mathematics or programming. We must be forming invariants in our head whenever we write a loop (or recursive function or class); otherwise we would not know what to write and our programming would be guesswork. So, why not declare those invariants *explicitly*

[1]With special acknowledgements to Don McLean.

in our code, where they could also play a crucial role in verification (of which there is much more to say later)? We don't need to be mature mathematicians to write invariants – just good engineers. And we need to be good engineers to develop computer systems.

Perhaps invariance *is* taught in a *theory* course somewhere? But that would not be good enough: invariance is a fundamental concept in programming and should be taught alongside all the other fundamentals (variables, assignment, messages, events, sequence, parallel, conditionals, choice, loops, functions, parameters, recursion, ...). Missing invariance, however, is only one symptom of things that are wrong.

Dijkstra suggested that only well-trained mathematicians should be allowed to program computers. That is not my opinion ... but I believe that in failing to engineer certain crucial mathematics into the tools provided for programming, we have failed as engineers and should not be surprised when things go very wrong. Addressing such failure is long overdue.

1.3. A Generation Lost in Space ...

Where does concurrency fit in? A computer system – to be of use in this world – needs to model that part of the world for which it is to be used. In our world, things happen *at the same time*. If our modelling reflects that natural concurrency, the system should be *simpler*.

Yet concurrency is thought to be an "advanced" topic, much harder than sequential computing which, therefore, needs to be taught and mastered first. This *sequential-computing-first* tradition makes no sense. Concurrency is a powerful tool for *simplifying* the description of systems. Performance spins out from this, but is not the primary focus. This has radical implications on how we educate people for computer science *and* on how we practice it.

Of course, we need a model of concurrency that is mathematically rich and clean, yields no engineering surprises and scales well with system complexity. *Communicating Process Architectures*, based on the ideas and mathematics of Hoare's CSP [12,13,14,15,16] and Milner's π-calculus [17], fulfills this need. Traditional approaches to concurrency (e.g. shared data and locks) conflict with some basic assumptions of, and our intuition for, sequential programming. They are, not surprisingly, very difficult to use. The *'S'* in CSP, on the other hand, stands for *Sequential* and there is no conflict.

Matters have now reached a crisis because the nature of modern architecture (e.g. instruction-reordering processors, multicore/manycore nodes and networks thereof) *requires* concurrency in the systems that ride on them to be effective. This has forced the issue.

A key insight, which goes against much received wisdom, is that concurrency is a necessary concept for *all* stages of specification, design, verification, implementation and maintenance of computer systems. Concurrency ideas are as fundamental as, for example, those of sequence, looping or procedural abstraction and we must be confident in taking advantage of them. If concurrency remains treated as an advanced concept, too difficult to be used as a matter of routine and only reached for to raise system performance (e.g. supercomputing applications) or responsiveness (e.g. embedded systems), then our systems will be based on a model of the world that does not match reality and will be unnecessarily complicated ... and slow ... and wrong.

1.4. Some Happy News ...

Concurrency raises a few issues not present in sequential logic and they must be addressed. The ones we are, perhaps, most frightened by are race hazard, deadlock, livelock and process starvation.

The first of these concerns concurrent update and observation of shared data, which leads to data corruption and unplanned non-determinism. This is what *locks* of various forms were invented to prevent ... but for which, in all but the simplest of scenarios, super-hero programmers are required for their correct and efficient application. In the communicating process model, only immutable data may be shared and so the problems simply do not arise. Of course, it requires well-engineered programming languages to ensure that this rule of the model is enforceable and enforced. Providing that assurance, without unduly limiting their areas of application or compromising performance, is an engineering challenge that is solvable (and to which the original designers of occam [18,10] have made a massive contribution). Finding the super-hero programmers on the scale needed for programming future systems with locks is, however, not possible.

A classic example of overlooked deadlock is given by Dijkstra's *Dining Philosophers* [19]. It illustrates a common problem in concurrency: how to share resources safely between competing consumers. The problem arises when there are not enough resources to go around and consumers sometimes need exclusive use of two or more. Simple *process oriented* solutions can be found in Hoare's 1985 book [13]. The story of the dining philosophers was derived from a real deadlock that happened, very occasionally, in an early multiprocessor operating system. The resources being shared were a limited supply of memory buffers, used for smoothing the reading and writing of files. Today – almost 30 years later – memory is not so scarce! Yet, operating system (or specific application) deadlock is rampant. We have been made to grow tolerant of our laptop/tablet/phone (or one of its apps) freezing on us. We have been, and still are, making the same mistakes again and again. Will we accept this forever?

Livelock – where a process demands processing resource but refuses to deliver or accept anything (e.g. a kill signal) – is also sadly common. Why does my laptop sometimes go into a spasm of 100% core activity, refuse to respond to keyboard or mouse events and force a powerdown? As I type these words, why does my web browser insist on sustained consumption of 50% of one of my cores, even though the pages open are all static, and only a full system reboot restores sanity (for a while)? Yet, livelock is addressed by CSP. We do not have to flounder in uncertainty. Systems can, and should, be designed to be free from livelock and that absence can, and should, be formally verified (Sections 3.2.2 and 3.4.5).

Process starvation is where some parts of the system fail to progress compared with other parts. This is always due to unfair servicing of their requests for resources, either by application processes managing those resources or by the run-time system scheduler managing access to the processors. The former is a particularly easy trap to fall into when using locks (e.g. the *"Wot, No Chickens?"* example, [20,21], in which the approved way for programming Java monitors leads to permanent starvation of one process despite a total processor loading of less than 1%). Under the communicating process model, the former is simply managed by sharing the ends of the application service channels (i.e. strict FIFO policing of *interleaved* use of the channels) or by well-known *fair* choice algorithms by the servers. The latter is rarely a problem for communicating processes, so long as the communication channels are *synchronised* or, occasionally, have small finite buffers that block when full. This is because processes that need resource (either to receive or deliver information) will block, leaving the run-time scheduler no choice but to schedule other processes including, eventually, those that will supply or accept the information that will unblock the first set – unless, of course, the system is mis-designed or mis-programmed. In these circumstances, process scheduling is self-driven, leaving little for the scheduler to consider. *[Note:* if the communication primitives are *asynchronous* and supported by unlimited buffering (e.g. Erlang mailboxes), process scheduling to avoid starvation will have to be more clever ... and expensive.*]*

Summary: communicating processes with synchronised message passing avoids, just by its nature, the dangers of race hazard and process starvation. Deadlock and livelock are easy to design against and formally verify that the design achieves their absence. Why, then, is this not state of the art? Why do we accept these repeated failures in concurrent systems, especially as modern processor architecture and our ambition for systems with ever more complex behaviours and scale mean that all systems will have to be concurrent?

2. We Need to Talk About Objects

Originally, objects were supposed to be autonomous entities, each containing a cohesive collection of data and algorithms for managing that data. A system is a collection of objects interacting by sending each other messages. Objects are typed through a hierarchy of classes, where sub-classes represent specialisation (or, if you prefer, extension) from the super-class.

The first two sentences of the preceding paragraph actually describes the model of communicating processes discussed in this paper. We claim that processes and channels fulfills the original aims for objects in ways that the objects and methods of modern "object-oriented" languages fail.

A serious problem with modern objects is that they are not *autonomous* – i.e. they cannot, in general, be understood on their own; their semantics are system sensitive. We sometimes tell our students that objects are water-tight containers of data, interacting with other objects through formal public interfaces. Unfortunately, objects leak information.

One problem is that an object does not declare the external resources its implementation needs (e.g. other classes of object, specific methods, monitor locks); its public interface only guides how it may be used by others (which does not necessarily cover how it is *affected* by others). This *information hiding* is sometimes a good thing – why should the user of an object know what other resources it uses? But that information is needed to understand how the whole system is put together. UML class entity relation diagrams show calling dependencies between classes, which is marginally interesting, but does not say anything about the dynamic network of objects at runtime. To understand a system, we need that information and, to find it, must hunt through the whole source code.

Of course, processes and channels give us concurrent systems; whereas objects and methods, in their modern guise, execute sequentially (ignoring, for now, special mechanisms for launching threads). The issues raised in the following subsections relate just to sequential programming with objects.

2.1. Hidden References ("not a word was spoken")

Suppose an object, `A`, has reference to two other objects, `B` and `C`. Consider somewhere in `A`'s (Java) coding:

```
B.reset ();
C.something ();
```

After the `C.something` method returns, can we rely on `B` still being in its *reset* state? `C` may have a reference, or knows someone who knows someone who knows someone who has a reference, to `B` and the call to `C` may have altered `B` – so no! Did the documentation of `C`'s class note that it might change something in an object of `B`'s class? If that was viewed as an implementation detail, then no. So, we have to search source code to find the network of objects in our system to discover if `B` might have been kicked! Local analysis of just the above two lines is insufficient. Such hidden side-effects happen often in OO code and the result is mystery.

An alarming postscript to the above scenario is that the side-effect on `B` can happen *even if* `B` *is held in a (supposedly) private field of* `A`*, was constructed by* `A` *and has never had its reference given away by* `A`! The last of the 'someones' in the second sentence of the previous paragraph may be `A` itself, who certainly knows about `B` (i.e. the side-effect was by a *callback*, on which more is said in Sections 2.2, 2.3 and 2.4).

Given the semantics of heap objects and multiple aliasing (which is part of the power of object-orientation, albeit a dangerous one), the above is, perhaps, not surprising and good engineers will discipline their designs to limit and control its occurrence. When it happens though, strong documentation is needed to guide the reader through the relevant trail of source code and unmask the mystery of those two lines. This is evidence of weak engineering in our programming technology – we should not have to be doing this.

2.2. Globalisation and Loss of Privacy ("the day the music died")

This example was produced more than ten years ago by Tom Locke – to whom I remain very grateful. I continue to be shocked each time I see it.

Suppose an object, `A`, has reference to an object `C` and a *private* integer field, `count`. The latter does not live on the heap and cannot be shared with other objects. Surely local reasoning about `count` will work? Consider somewhere (else) in `A`'s coding:

```
count = 42;
C.something ();
```

After the `C.something` method returns, can we rely on `count` still having the value `42`? Alas, no! Although no object other than `A` may directly change the value of `count`, `C` may have a reference, or knows someone who knows someone who knows someone who has a reference, back to `A` – who can change it for them. Again, we have to start searching a lot more code than the above two lines.

The conclusion is that local fields of objects must be regarded as global, regardless of any notions of *privacy* declared for them. In the past, proliferation of global variables meant that local reasoning about code was unsound. *Structured programming* led us safely out of that mire. *Object orientation* seems to have taken us back in.

2.3. Global Reasoning Required (the Counter Example)

This section gives a short complete example of a class whose semantics cannot be determined from its code, although it looks as they they should! The code seems perfectly innocent, obeying simple principles of good object orientation:

```
public class Counter {

  private int count = 0;

  private Logger logger;

  public Counter (Logger logger) {
    this.logger = logger;
  }

  public int getCount () {
    return count;
  }

  public void increment () {     // adds 1 to count (are you sure?)
    count++;
    logger.log (count);          // log this event
  }

}
```

Inspecting this code, we would hope that an invocation of the `increment` method would increment the value of the `count` field of the invoked object by one (as indicated by the comment). For example, if `counter` is a `Counter` object and we see the code:

```
x = counter.getCount ();  counter.increment ();  y = counter.getCount ();
```

would we not expect `y` to get the value `x + 1`? We should not need to look up what a `Logger` object does with its `log` method: primitive data types are passed by *value* in Java, so that call cannot change `count`. Unfortunately, none of these sentiments are reliable!

Let's look at the `Logger` class:

```
public class Logger {

  public void log (int count) {
    System.out.println ("Log " + System.currentTimeMillis () + ": " + count);
  }

}
```

Well, that seems mostly harmless ... so, are we safe?

Unfortunately, no! Because of the *inheritance* mechanism – one of the most promoted features of object-orientation – we have to look further. `Logger` may not be the class of the object given to `Counter`. It could be a sub-class and that could do anything ... like a callback! For example:

```
import java.util.Random;

public class Blogger extends Logger {

    private Counter counter = null;

    private boolean awake = true;  // flag to ignore callbacks

    public void setCounter (Counter counter) {
      this.counter = counter;
    }

    private Random random = new Random ();

    public void log (int count) {
      if (awake) {
        if (counter != null) {
          awake = false;           // callbacks will be ignored
          for (int i = 0; i < random.nextInt (100); i++) {
            counter.increment ();
          }
          awake = true;            // callbacks accepted again
        }
        super.log (count);
      }
    }

}
```

Now, if `Counter` is constructed with a `Blogger`, rather than a `Logger`, its behaviour will become somewhat unexpected *(to a reader of just its code)* should the `Blogger`'s `setCounter` method ever get invoked with a reference back to the `Counter`. For example:

```
Blogger blogger = new Blogger ();
Counter counter = new Counter (blogger);

for (int i = 0; i < 10; i++) {
  counter.increment ();              // adds 1
}
```

```
blogger.setCounter (counter);

for (int i = 0; i < 10; i++) {
  counter.increment ();              // adds a random amount!
}
```

The point is that the `Counter` class seems very simple, clean and innocent. Yet its semantics cannot be determined in isolation – we have to be aware of the network of objects in which it is placed. This need for non-local analysis is a serious trap for the unwary object orienteer. If a class as simple as `Counter` can have its semantics corrupted, what about everything else? If a system does not decompose into parts that can be understood in isolation and that understanding not violated when those parts are put together, then complex systems will always be unsafe. And we have not begun to talk about concurrency and objects.

Of course, `Counter` could be coded to protect itself from callbacks, using the same *ignore flag* trick employed by `Blogger`. But that is something of a distraction to its purpose and we would rather not have to do it.

2.4. *Class Invariants ("do you believe in rock & roll?")*

In Section 1.2, class *invariants* were mentioned as something that might be important. They are not in many textbooks. At my institution, we do not currently teach them ... although there are plans! I fear they may not be currently much used in practice.

Class invariants are predicates expressing properties of, and relationships between, data held within an object that are always true before and between method calls. All constructors and initialisation code must establish them. The implementor of a method can rely on them at the start of the method and must re-establish them before the end.

Unfortunately, the above description is incomplete – another trap for the unwary! The implementor has to work harder to ensure that invariants are maintained. The problem, again, is callback. *Any* callout from a method to another object has the potential for callback. Discovering if this is so for a particular system requires non-local analysis. Discovering if this so for a class whose objects are to be used in systems not known to the designer of the class (e.g. `Counter`) is not possible. Precautions, therefore, must be taken.

Should a callback happen and a class invariant be invalid at the point of callout, the assumptions of the method called back will not be honoured and it will be operating on a corrupt object – with unknown consequences. If the implementor of the method calling out does not know whether this will happen, the class invariants must be re-established *before* calling out. One way to do this is to program the *ignore flag* trick used by the `Blogger` class in Section 2.3 – the invariants then become extended through being implied by that flag. However, if there is a callback, there is probably a reason for it and simply doing nothing may not be an option; so the original invariants must be properly re-established. How often in this care taken?

So, we are faced with either non-local analysis (which makes complex system development and maintenance ultimately impossible) or non-trivial added complexity to any method that invokes other objects (which has the same result).

In Section 3.3, we argue that the communicating process model for computation does not suffer from these problems. Local-only reasoning is sufficient to establish the semantics of processes, with no complex trickery needed. Those semantics are not violated by sequential or concurrent composition. Large scale and complex systems can be built and maintained with confidence. And, of course, their natural concurrency means they can directly take advantage of distributed and multicore platforms. So, there will be correctness and there will be performance.

3. Process Oriented Design and occam-π

We call the programming discipline underlying Communicating Process Architectures *Process Orientation (PO)*. A process is a *component* that encapsulates a cohesive collection of data and algorithms for managing that data. Unlike an object, its data and algorithms are strictly *private* – the outside world can neither see its data nor execute its code. Each process is *alive*, its algorithms executed by threads of control that stay always inside the process.

A process *variable* changes its value if and only if that process makes that change. None of its state can change behind its back, which gives a simple and intuitive semantics for sequential operation. A major, but largely unremarked, advantage of this is that PO code directly and safely exploits modern instruction-reordering processors – no special care needs to be taken by PO programmers.

Processes interact through synchronised message-passing over named channels, or through multiway synchronisation on named barriers. Networks can be built dynamically through the runtime construction of processes, channels and barriers; and by the communication of channel and barrier *ends* through channels. These mechanisms have direct representation in CSP and the π-calculus and (can) have extremely lightweight implementation – a few tens of cycles – on modern architectures.

Synchronised communication has several benefits. Section 1.4 mentions the force it provides for scheduling, simplifying the work of the process scheduler to enable *fair* progression of all parts of the system. A semantic power is the knowledge it gives to the sender of a message: if the message has gone, the sender knows the receiver has received. A corollary is the semantic power it gives to a receiver: if the receiver is not in a state to take a message from some source, it may refuse *(simply by not having the code to accept)*. There is no need for complex logic involving *condition variables* [22] or Java *monitors* (`wait` and `notify`) [21] or for runtime expensive logic such as *busy polling*.

For these reasons, we claim that processes are *autonomous reusable components* and, of course, they plug into and build concurrent systems.

3.1. A Diagram Language (in 4 + 2 Pictures)[2]

The starting point for process oriented design is to sketch some process networks. Networks have structure - networks within networks - and the design can be done from any level and in any direction. Figure 1 shows the small diagram language for process-oriented design.

Figure 1(a) shows part of a network, with three processes connected by 3 *point-to-point* channels to each other and 2 channels to the rest of the network. Figure 1(b) shows a single *client* process with access, through a channel, to a bank of *servers*. The client does not care which server takes any request it sends. The servers *share* the reading end of the channel and, when not busy with an earlier task, form a queue to receive new instructions (*interleaving*, in CSP terms, their use of the channel). Figure 1(c) shows the opposite: multiple clients sharing the writing end of a channel, with a single server at the other. Again, clients who seek access interleave their use of the channel (disciplined by a queue) to reach the server. Channels may be shared at both ends (not shown), where (needy) clients queue up to seek service and (idle) servers queue up, separately, to provide. In these examples, a client must not care which server it gets and a server must not care which client.

The channels in these diagrams are unidirectional. However, they may also represent *bundles* of channels, so that communication may be two-way. If bundle-*ends* are shared, a process acquiring them (by reaching the front of the queue) has exclusive use of all the channel-ends inside, so that an extended two-way conversation can be made. In the case of such sharing, the arrowhead points to the *server* end.

[2]Four paragraphs in this section are based on paragraphs from Sections 3.1.1 and 3.1.2 of [23].

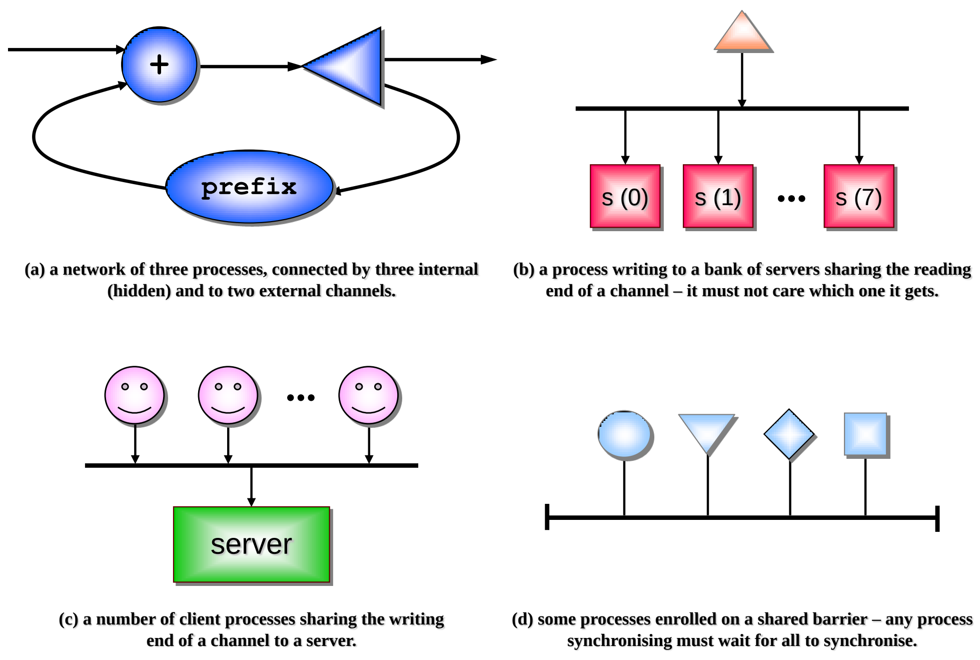
+
prefix
s (0)
s (1)
s (7)
(a) a network of three processes, connected by three internal (hidden) and to two external channels.
(b) a process writing to a bank of servers sharing the reading end of a channel – it must not care which one it gets.
server
(c) a number of client processes sharing the writing end of a channel to a server.
(d) some processes enrolled on a shared barrier – any process synchronising must wait for all to synchronise.

Figure 1. Process oriented design: components and connectors (*from [24]*). There are no rules for the shapes of component icons (*processes*) – whatever is meaningful to the designer is allowed.

Figure 1(d) shows four processes synchronising on a barrier. This is a multi-process event: if one process offers to synchronise (i.e. tries to get over the barrier), it must wait for *all* processes connected to the barrier to do the same. The last process to synchronise releases all the others.

occam-π [25,26,27,28,29,24,30,23] has a simple syntax that directly captures these network structures. Figure 1(a) maps to:

```
CHAN REAL64 a, b, c:                -- channels are strongly typed
PAR
  adder (in?, c?, a!)               -- these processes
  delta (a?, b!, out!)              -- can be listed
  prefix (b?, c!)                   -- in any order
```

where only the 3 completely used channels (`a`, `b` and `c`) are declared. The 2 remaining channels (`in` and `out`) connect to the rest of the network and so must have wider scope (i.e. they are declared global to this piece of code). *Notation:* the expression `c!` is the writing-end of the channel named `c` and `c?` is its reading-end.

We can abstract those missing channels into parameters for a complete process component. The parameter channels represent external connections that must be wired into the network using this component; the internally declared channels represent internal wiring that is hidden from the external network:

```
PROC integrate (CHAN REAL64 in?, out!)
  ...  above five lines
:
```

Visually (figure 2), this abstraction of a network into a reusable component corresponds to embedding the network (with loose ends) on to a named chip (with I/O pins). The `integrate` component can now be used in forming higher-level networks – in the same way as `adder`, `delta` and `prefix` were used in its own construction.

For completeness, we show occam-π coding for the other three diagrams of Figure 1. The server-bank network in (b) may be set up by:

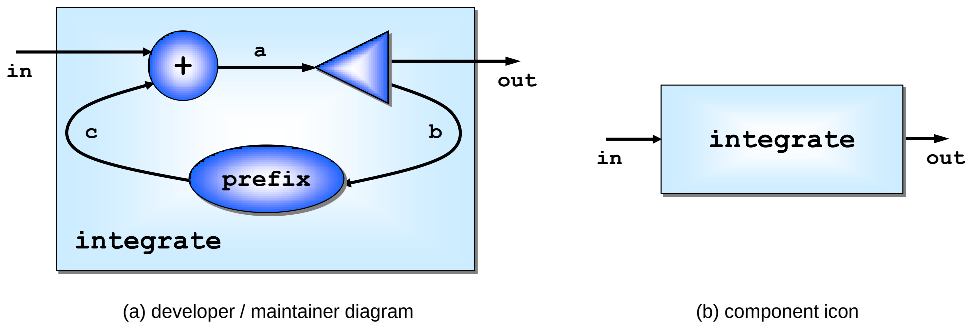

Figure 2. Process oriented design: a network of processes is also a process.

```
SHARED ? CHAN SOME.SERVICE c:         -- only the reading-end is shared
PAR
  triangle (c!)                       -- one writer process
  PAR i = 0 FOR 8
    s (i, c?)                         -- many servers, sharing the reading-end
```

The many-client system in (c) is:

```
SHARED ! CHAN ANOTHER.SERVICE c:      -- only the writing-end is shared
PAR
  PAR i = 0 FOR n.clients
    smiley (i, c!)                    -- many clients, sharing the writing-end
  server (c?)                         -- one reader process
```

The network of processes in (d), synchronising on a shared barrier is:

```
BARRIER b:
PAR ENROLL b
  circle (b)
  triangle2 (b)
  diamond (b)
  square (b)
```

We have given these examples of occam-π syntax because they show how network structure is explicitly declared. There are no equivalent mechanisms in *threads and locks* programming. Erlang networks are represented implicitly by process variables holding references to (the mailboxes of) other processes. Discovering the networks involved means searching disparate pieces of code. Discovering *structure* in the networks is harder still. The concepts are one-dimensional – there is no such thing as *threads-within-a-thread* nor, in Erlang, *processes-within-a-process*. Yet, without network structure, there would be no universe.

The iconography in Figures 1 and 2 reflects hardware design: *chips* and *wires*. The physical constraints implied by hardware mechanics reflect needed semantic rules underlying process autonomy (layered construction, localisation and inviolability of state, interaction only if explicitly connected). Software has no need for such constraints: we can let threads of logic quietly interfere with and depend upon each other's state – just because we can! Even though we may try to retrofit controls (e.g. *locks*) to try and avoid the worst consequences, the freedom afforded means mistakes will be made and we exploit it at our peril.

The hardware discipline that process-oriented design imposes on software has important benefits for simplicity and performance. Engineers – whatever their field[3]– have long experience of network (or *circuit*) diagrams, with intuition for their semantics long established. The underlying formal semantics of CSP and π-calculus align with that intuition so that we get no surprises, at least at this low-level, from our concurrency.

[3]Sadly, it seems that software engineering may be an exception to this.

3.2. CSP Semantics (in one page)

This section presents no CSP syntax or examples, but outlines the key properties of its three most common semantic models (*traces*, *failures* and *divergences*) together with the concept of *refinement*. Why this is important for the specification and verification of future computer systems – and why *verification* is important – is left for the reader to consider.

We note that the FDR model-checker [31] enables machine-checked verification of the refinements described below, so long as the systems being tested do not have too many states. Model checking is an art that needs to become accessible to systems designers and programmers as normal practice. Assistance for this is proposed in Section 3.4.5. By setting up and asking the right questions, surprisingly large systems (e.g. those with more states than atoms in the universe [32], though not infinitely many) can be verified.

3.2.1. Safety (traces refinement)

An *event* (e.g. channel communication, barrier synchronisation) happens when, and only when, all processes relevant to it choose to engage. A process trace is a finite sequence of events in which a process *may* engage.

`P` *trace-refines* `Q` means the traces of `P` are also traces of `Q` – anything `P` may do, so may `Q`. Turning this round, if there is something `Q` cannot do, `P` cannot do it either. Now, if `Q` is a *specification*, then `P` is safe in the sense that `P` cannot exhibit behaviour (presumably 'bad') disallowed by `Q`.

This is not enough – e.g. the `STOP` process trace-refines anything, since it does nothing (not even terminate); but it's probably not what the specifiers had in mind!

3.2.2. Liveness (failures refinement)

A process *state* is what a process has become after executing one of its traces (and may be represented by that trace). An event is *external* to a process if other processes may engage on it. A state is *stable* if there is no *internal* (i.e. hidden) event on which it may engage. A stable *resolution* of a state is a stable state reached by zero or more internal events only. A process *failure* is a state paired with a set of external events on which a stable resolution of that state refuses to engage. A *deadlocked* system has reached a stable state where *all* pairings of that state (i.e. the trace that led it there) with external events are failures.

`P` *failure-refines* `Q` means (`P` trace-refines `Q`) *and* (the failures of `P` are also failures of `Q`). So, if a <*trace, event-set*> is not a failure of `Q`, it is not a failure of `P` either. Now, if `Q` is a *specification*, then `P` fulfills its liveness conditions: if the specification (`Q`) says that *in this state you will react to one of these events* (i.e. there is no failure here), the implementation (`P`) *will react* (and the reaction will be safe because of trace refinement).

3.2.3. Liveness (failures-divergences refinement)

A process state is *divergent* if an infinite sequence of *internal* events is possible from that state. This is usually a bad thing. A divergent state is, of course, unstable but *may* recover to a stable state. However, a divergent state is considered so dangerous that further consideration of process behaviour is not worth pursuing. All divergent process states, together with all subsequent states (even if stable), are written off and deemed to be the same. This means that everything is a failure with them and slightly changes the meaning of *failure-refines* in the following definition.

`P` *failure-divergence-refines* `Q` means (`P` failure-refines `Q`) and (the divergences of `P` are also divergences of `Q`). Now, if `Q` is a *specification* with no divergences (which would be usual), then the implementation (`P`) also has no divergences.

If divergence-freedom is verified, *failures* and *failures-divergences* refinement are the same – so only the former need be used for further analysis.

3.3. Local-only Reasoning (the Counter Example)

This section reviews issues raised in Section 2 as they relate to the communicating process model.

3.3.1. Expect Processes to Change (They are Alive)

Section 2.1 refers to a sequence of two method invocations on two different objects, where the second one has a hidden side-effect on the object reset by the first. If we replace the objects with processes running concurrently and the method invocations with channel communications, the surprise does not arise:

```
SEQ
  to.B ! reset
  to.C ! something
```

The process at the other end of the `to.B` channel (presumably `B`) is alive. Having been reset, it is free to move on – so we have no expectation that it is still in its reset state, even before the execution of the second line.

If the action in the second line causes process `C` (at the other end of the `to.C` channel) to interact (directly or indirectly) with `B`, that can be seen as a possibility by looking at the network diagram (or its explicit coding) for this part of the system. Only if connections are visible and we are concerned, need we consider the behaviour of `C`.

So, if the `B` process changes since we reset it, there is no surprise. The problem with the object code is that the system is purely sequential and the `B` object – which cannot do anything on its own – changes without any visible sign to the reader of that code.

3.3.2. Local Reasoning for Local Data

Section 2.2 shows a disturbing inability to secure control over the value of a supposedly *private* integer field. If that integer is data local to a process, control is assured: the value of a variable changes if and only if the process owning it makes a change:

```
SEQ
  count := 42
  to.C ! something
```

Local reasoning on just these three lines of code is sufficient. They are executed in sequence. The value of `count` is set once and not changed subsequently. Therefore, its value after the sequence ends is `42`.

No further analysis is needed. If the `C` process (after receiving `something`) tries to communicate back to this process so that `count` will be changed, that cannot yet have happened simply because there is no code in the above lines to accept such a communication – so, if one is being attempted, it will be blocked.

3.3.3. Counter as a Process

Section 2.3 develops this further, presenting a simple `Counter` class whose semantics cannot be determined from its code – they depend on the context in which it is used.

If `Counter` becomes a *process*, its semantics are locally determined. Figure 3 shows an icon for this `Counter` process. The `answer` and `log channels` carry integers (the value of the internal `count`). The channel `ask` carries two kinds of question and, in occam-π, is best described with a *protocol*:

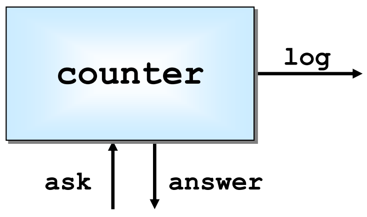

Figure 3. Process oriented design: process semantics does not depend on context.

```
PROTOCOL COUNTER.ASK
  CASE
    increment
    get.count
:
```

The process almost writes itself:

```
PROC counter (CHAN COUNTER.ASK ask?, CHAN INT answer!, log!)
  INITIAL INT count IS 0:
  WHILE TRUE
    ask ? CASE
      increment                     -- adds 1 to count
        SEQ
          count := count + 1
          log ! count               -- log this event
      get.count
        answer ! count
:
```

Unlike the case for the object code (Section 2.3), we can immediately deduce that the response to the `increment` request is to add 1 to the internal `count`. The behaviour of the surrounding processes (Figure 4) does not matter.

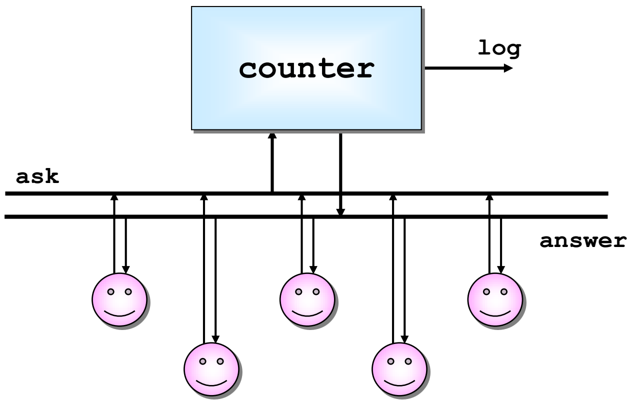

Figure 4. Process oriented design: some context (on which the semantics of `counter` do not depend).

If the `logger` process (on the end of the `log` channel) is, mysteriously, one of the *smiley faces* and calls back, its request would be queued until this process can take it. The `increment` request that triggered the `log` message will complete *before* any callback is serviced.

Of course, should `logger` never accept the `log` message, the servicing of the `increment` request would also never complete. Also, if `logger` held the `log` message in an occam-π *extended input* and tried to call back (which breaks a design rule for extended input), there would be deadlock. Neither of these scenarios is the concern of the `Counter` process. Its semantics are that the `ask` messages are processed correctly and that their servicing continues so long as its responses (on the `answer` and `log` channels) are accepted. Ensuring the latter is the responsibility of the network builder.

3.3.4. Process Invariants

No new concept is needed. Data in processes have no special semantics (cf. object fields, class fields, method variables). We just need the standard notions of *loop* and *recursion* invariants.

3.4. Rationalising occam-π *(Unfinished Business)*

occam-π [25,26,33,34] has been an experiment in language design and implementation for the communicating process model, starting around the mid-1990s from the occam2.1 language developed by INMOS (later SGS-Thomson Microelectronics, who kindly open-sourced their compiler and libraries for the Transputer). In parallel, libraries enabling this model for a range of standard languages (including CTJ [35,36,37,38] and JCSP [39,40, 41,42] for Java, CCSP [43,44,45,46,47] for C, C++CSP [48,49] for C++, CHP [50,51] for Haskell, CSO [52] for Scala and PyCSP [53], python-csp [54] and Hydra [55] for Python) have been and are being developed. The work started at Kent but has spread to researchers at many institutions around the world. Numerous papers – too many to reference more than a tiny fraction – have been written and published, not least in the CPA conference series.

Applications are an essential driver for the research. For occam-π, these have recently included small platform targets (e.g. the Arduino [56,57,58]), operating systems (RMoX [59, 60]), the engineering of nano-scale machines (TUNA [61,62,63]) and the modelling and control of complex systems with emergent behaviours (CoSMoS [23,64,65,66,67,68]).

occam-π is a careful blending of the dynamics of Milner's π-calculus (mobile channels and processes) with Hoare's CSP underpinning of classical occam, together with significant extension of CSP capabilities (e.g. barriers [27], shared channel-ends [28]). Supporting this, the lightest (by some margin) multicore scheduler available today has been researched and developed, on which applications automatically and effectively scale with the number of cores supplied [46,45]. We also have a fast algorithm for the resolution of general CSP choice (between barriers, input/output guards and timeouts) [69,70], though this has only so far been exploited by JCSP (the CSP library for Java) and an experimental compiler for CSP [71].

However, the occam-π extensions are showing signs of strain and it is now time to look to rationalise and integrate the lessons learnt into a leaner and improved design. Whether this should lead to a new language family or one that is obviously still occam should be considered by its community of stakeholders. The rest of this section outlines a few ideas.

3.4.1. Recursive Union Types

These are an old idea – long overdue for occam. Pascal and C had them back in the 1970s, although they were somewhat unsafe in that they allowed data to be written as one of the variant types and interpreted as another. Securely typed versions underpin modern functional programming languages and were proposed (in 1991) for the (sadly unimplemented) occam3 language [72]. More recently, they are a key element of the proposed *Guppy* [73] follow-up language from occam-π.

Here is described one of the *occam Enhancement Proposals*: OEP-156 (2006) [74]. Its syntax reflects that of *variant (*CASE*) protocols* in occam-π, whose purpose it is intended to replace (Section 3.4.4). Two examples:

```
DATA TYPE FOO                              DATA TYPE COLOUR
  CASE                                       CASE
    sugar, BOOL, REAL64, []BYTE                red
    salt, BYTE, BYTE                           green
    pepper                                     blue
:                                          :
```

Each element type of the union is introduced by a user-chosen *tag* word, followed by zero, one or more component fields (i.e. a *record*). The tags must be different. In the second example, there are only tags and the union type corresponds to an *enumerated* type.

Union types are first class – their values may be freely assigned, communicated, abbreviated, passed as parameters and returned from functions, subject to the normal rules for strong typing. Example literals have the same syntax as those for RECORD types:

```
[sugar, TRUE, 99.99, "Krakatoa"]                [red]
[salt, 42, 'A']                                 [green]
[pepper]                                        [blue]
```

In case of ambiguity arising from different union types having identical variants (same-named tags and same-typed fields), the type of the literals must be resolved with the standard occam syntax for type decoration. For example:

```
[pepper] (FOO)                                  [blue] (COLOUR)
```

Processing values of a union variable requires a CASE process to determine the variant. For example, if x if a variable of type FOO:

```
CASE x
  sugar, BOOL b, REAL64 r, []BYTE s
    ...  b, r and s abbreviate the component fields
    ...  optional follow-up (can change x variant, abbreviations not in scope)
  salt, BYTE m, n
    ...  m and n abbreviate the component fields
    ...  optional follow-up (can change x variant, abbreviations not in scope)
  pepper
    ...  can change x variant (no field abbreviations)
```

Thus, values of one variant cannot be processed as another. The optional process following the response to some variants is only needed if the variant has to be changed. This is not allowed in the first response to the variant if there are field abbreviations, since those would be invalidated. The abbreviations are not in scope for the follow-up process.

Recursive union types are allowed. For example:

```
DATA TYPE TREE
  CASE
    node, TREE, FOO, INT, TREE     -- left, data, count, right
    empty
:
```

Here is a process for inserting an element of FOO, which the code assumes has a (user-defined) total ordering with the usual comparison operators, in the above tree:

```
PROC insert (VAL FOO x, TREE t)
  CASE t
    node, TREE left, FOO y, INT count, TREE right
      IF
        x < y
          insert (x, left)
        x = y
          count := count + 1
        x > y
          insert (x, right)
    empty
      t := [node, [empty], x, 1, [empty]]     -- variant changed
:
```

This is standard recursive tree walking. Note, however, that in line with all dynamically allocated occam-π elements, no *"null pointer exceptions"* can occur at runtime – a major win. This is because the *definedness tracking* by the compiler either ensures that all variables and fields have assigned values before they can be observed or demands the explicit insertion (by the programmer) of runtime checks.

For convenience of recursive declarations generally, we propose to change the historic occam language rule that defined names do not come into scope until *after* their declaration (i.e. after the closing colon). Instead, we adopt the standard practice for modern languages that all names in the same block of declarations are in each other's scope. This requires the compiler to make two passes through each declaration block, but that is not a problem.

A more serious issue for recursive data structures is whether to allow *branch-and-join* or *cycles*. Both these introduce *aliasing* (i.e. multiple references to the same data), which is otherwise not present in occam-π. Indeed, this is essential for the static analysis (by the compiler) that eliminates data race hazards in occam-π systems (Section 1.4) – and must not be lost! *Separation logic* [75] may rescue us here. Otherwise, we may need to allow only *tree* structures to be built, which do not introduce aliasing, through a combination of compiler and (hopefully not) runtime checks. Trees also avoid unpredictable overheads of garbage collection, which would remain linearly proportional to the size of the structure being recovered.

Recursive union types make new fields of application practical for occam-π systems. For example, the compiler for the revised language could be written in itself. Novel and simple parallel approaches to compiling are open to research – to be efficiently consumed by multicore processors.

3.4.2. Unify Static and Dynamic Allocation

occam-π introduces *mobile* data types. Like any occam-π data, there can only be one owner at a time. When assigned or communicated, non-mobile data is copied but mobile data is *moved* – meaning that the source variable loses it to the target (i.e. only one copy ever exists). Mobile data is constructed through dynamic allocation from heap storage. If source and target are in the same memory space (i.e. have access to the same heap), movement is by copying the reference to the target and *undefining* the source variable (meaning that it cannot be looked at again until it is *redefined* by assignment or input). Otherwise, movement has to be by copying (to the heap in the target's memory space), but still with *undefining* the source variable (so that the semantics of movement are the same whether it is within or between memory spaces).

Arrays with unspecified sizes can be declared mobile and allocated to runtime determined sizes (which may change during the life of a variable). Currently, the only way to get dynamically sized arrays in occam-π is to make them mobile. The syntax for declaring and constructing such arrays is, therefore, different and somewhat more complex than the syntax for declaring known-sized arrays. The position is even more complicated for runtime sized arrays of channels!

We propose to simplify this, leaving the question of static versus dynamic allocation to be handled invisibly by the compiler and runtime. Data will just be data. Some data, if the logic in the application demands that at most one copy exists, can be declared `MOBILE`.

For non-recursive processes, stack allocation can be arranged – as at present – by the compiler *before* runtime. Recursive process stack allocation happens dynamically, when any level is started. The same is true for processes replicated in parallel by a runtime found value.

Items with compiler-known *small* size (less than 8 or, perhaps, 16 bytes) are allocated on their process stack. Everything else is dynamically allocated on the heap, with references on the stack. *[Note:* currently, compiler-known large items are pre-allocated in *Vectorspace*, following the mechanism from the Transputer implementation of occam. Putting all large items on the heap, regardless of whether their sizes are pre-known, simplifies things.*]*

The programmer is blind to the above. In particular, there is the same syntax for arrays, regardless of whether its size is known to the compiler:

```
[n]THING t:        -- n may be a run-time value
```

Array size is no longer part of the type. An array variable declared with one size may be assigned to an array with another size (same type, of course). An array variable may be declared without size, but must then be assigned (either by assignment or incoming communication) to an actual array value before being used.

Assignment and communication are handled in the most efficient way. Regardless of any declared mobility, stack items – always small – are assigned/communicated by *copying* and heap items by *reference*. In the latter case, this would normally be by copying the reference held on the stack. However, if compiler usage analysis of the sending process shows the assigned/communicated data is used later by that process, a reference to a (deeply) cloned copy is sent. This is Brown's algorithm from his paper at CPA 2009 [76].

If data is explicitly declared MOBILE, the above decision is different: if compiler usage analysis of the sending process shows the assigned/communicated data is used later by that process, this is a semantic error and the compilation fails (reporting the error). This is the current algorithm for occam-π mobiles.

3.4.3. Remove occam2.1 *Protocols*

Having proposed extensions to the language, the principles of *Ockham's Razor* suggest there may be some things to remove. We propose removing occam2.1 PROTOCOLS. These are not real *protocols* in the usual meaning of that term for computer networks. They define a flexible range of *message structures* that are no longer needed, thanks to more powerful types.

The *sequential protocol* is simply replaced by occam2.1 RECORD types. A reason this was not done before is that there was one semantic win for this protocol over record types: a sequential protocol message is a *sequence* of communications, so that an earlier item of received data could be used to address the location of a later one. For example:

```
in ? i; A[i]
```

Fortunately, this capability is won back through *session protocols* (see Section 3.4.4), with no loss of syntactic clarity or runtime efficiency.

The *counted array protocol* is replaced by the revised array type (Section 3.4.2), whose size is not part of the type but carried as part of its value.

The *variant (*CASE*) protocol* is replaced by *union (*CASE*)* types. Before the decision to allocate all large items of data on the heap, there was one pragmatic win for variant protocols over such unions. Where the receiving process logic meant that only small varieties of message were to be accepted, space for the larger variants did not have to be (pre-)allocated – whereas a variable of a union type would have needed space reserved for its largest variant. Now, however, union types with large variants will always be on the heap, so that only space for its reference needs to be reserved in the receiving process.

These proposals mean that communication consists of copying only standard-sized amounts of data (8 or, perhaps, 16 bytes for references and primitive types) – except, of course, when communicating across a memory frontier. An additional benefit from this is that buffered channels that are *generic* across all types become easy to build – a memory frontier is not a problem (since the buffering can be placed on either side or both). Currently, implementing a buffered channel carrying (say) a variant protocol requires special code for each variant, which is tedious to write and inefficient to run (compared with the new proposal).

3.4.4. Introduce Session Protocols

These were proposed by Adam Sampson [77,78]. They are communication protocols in the sense normally understood (i.e. *patterns* of communication). They are associated with a single channel, which may have *shared* ends. The channel is directed (in the same sense as a current channel record is directed), but may be used in both directions (possibly at the same time). We call them *session* protocols, reflecting the idea of *session types* [79,80,81,82]. As the channel is used, the type of message it carries (and its direction) may change.

The simplest session protocol is one data type, sent one way, once. This corresponds to a classical channel. The very simple example in Section 3.4.3 becomes syntactic sugar for:

```
SEQ
  in ? i
  in ? A[i]
```

where the `in` channel is engaged in part of a session specifying that an integer is followed by whatever type is held by elements of the `A` array.

Structured sessions consist of separately typed messages flying in (nested) `SEQ`, `ALT` and/or `PAR`, including repetition (through recursion or looping). This declared structure is the session protocol – syntactic details are not yet settled. The compiler checks that all code operating on the channel conforms, tracking use across all processes and procedures. Channel parameters carrying a session protocol will have to declare which (named) part of the protocol their `PROC` implements. The intention is to eliminate all protocol errors through compiler checks.

Session protocols *may* let us drop channel records (i.e. channel bundles) from the language. These are mainly used for two-way conversations that would be more safely handled by a session protocol (and with less syntactic clutter).

3.4.5. Verification within the Language

This is a proposal to make formal verification of occam-π programs manageable within the language [83,84,32].

The language is extended with qualifiers on types and processes (to indicate relevance for verification and/or execution) and assertions about refinement (including deadlock, livelock and determinism). The compiler abstracts a set of CSP equations and assertions, delegates their analysis to the FDR model checker and reports back in terms related to the occam-π source. The full (FDR) range of CSP assertions is accessible, with no knowledge of CSP syntax required by the programmer (but who will need at least the semantic understanding outlined in Section 3.2). Programs are proved just by writing and compiling programs.

Model-checking is a skilled art. There is more to be done than simply asking the compiler to verify, for example, deadlock-freedom (although that may be a useful starting point). The skill lies in *asking the right questions* and setting up the necessary auxiliary processes needed to frame them. By enabling this to be done as part of normal programming, the intention is to let formal verification become everyday practice [30,83,84]. We suggest that this is urgently needed.

3.4.6. Call Channels and Process Mobility

`CALL` channels were proposed for occam3 [72]. They are syntactic sugar for a simple and common session protocol: *lend-return*, where a process lends some of its resources (i.e. anything in its scope) to a process (at the end of the call-channel) and waits for that process to signal it has finished. The borrowing process may change some of the resource it has been lent. This may be implemented by *lend* and *return* communications over the channel *(the session)*, though obvious and safe shortcuts can be made if lender and borrower are in the same memory space. Syntax for declaring and calling the channel reflects that for declaring and calling a procedure (`PROC`). The syntax for accepting the call reflects the implementation body of a procedure, except that it happens at the places of acceptance and can be different each time. The semantics are those for classical *extended rendezvous* (e.g. Ada task entries).

A proposal extending `CALL` channels to allow *variant* interfaces has been made [85,86]. The syntax is different from the occam3 proposal, being expressed as a *protocol* with which channels may be typed. The interfaces in the protocol still have procedural syntax, but using them gives explicit reminder of the synchronisations involved (`!`, by the lender, and `??` by the borrower). The *double question-mark* is to be consistent with the notion of *extended input* [87] (see Section 3.4.7) in occam-π, which already provides the semantics of extended rendezvous (but without the return of resources). Variant call channels have long been available in the JCSP library.

We have highlighted variant call channels because they provide a direct way to show the *duality* between mobile channels and mobile processes in a future occam-π (or derivative). But we need to re-consider the form of mobile processes currently implemented in occam-π. At present, they work efficiently and allow whole networks (and networks of networks) to be moved, but are somewhat tedious to program because they can only have a *single* interface and that interface allows only channel-end parameters. Also, the mobile network (or solitary process) must be explicitly programmed to *freeze* before it can be moved. We are proposing a new model, [86], that gives mobile processes *variant* interfaces, allows *any* kinds of parameter, greatly simplifies their use and can be moved at any time – even when executing (except during rendezvous with a current host). It is these that are the *dual* of (mobile) variant call channels. A similar proposal for mobile processes has been made for the ProcessJ language by Pedersen and Sowders [88] and is further developed by Pedersen and Smith this year [89].

Almost all of the large dynamic models built so far with occam-π have used *channel*, rather than *process*, mobility. These have a difficulty for migrating processes across memory boundaries, since it is not easy for the runtime system to decide whether a process needs to be moved across such a boundary to follow a migrating *channel-end* whose other end it is holding (and we don't try)! Instead, the programmer must intervene (e.g. [90,91]). However, if a *process* is moved across a memory boundary, that's a pretty good hint to the runtime that it ought to move it (and we think we know how this may be arranged).

3.4.7. and More ...

There are many more ideas that need attention. Not least are several from the occam3 language (in addition to the `CALL` channels just discussed). There are an interesting new range of process declarations (`INITIAL`, `FINAL`, `RESOURCE` and `SERVER`), parametrised `MODULE` types and *libraries*[4]. The reader is invited to look again at the reference manual [72].

The *extended input* process of occam-π was mentioned in Section 3.4.6. This holds the sender of a message while the receiving process responds, rescheduling the sender only when the receiver chooses – the sender is unaware of the extended rendezvous. It has some surprising and powerful uses (e.g. for inserting processes into a channel for monitoring/black-boxing traffic *without* changing the semantics of synchronised communication over the channel). Implementation has been achieved with *zero* overhead for communications that do not use it (i.e. no checking has to be made by a sending process).

Brown and Ritson have proposed the converse: *extended output*, where the receiver process – once committed – is held while the sender decides what to send [93,94]. This also has

[4]For large software projects, package support built into the language would provide great assistance and is more-or-less expected in modern languages. Currently, occam-π uses traditional mechanisms (pre-processor [92], conditional compilation, separate compilation, makefiles).

interesting applications for hiding processes. But it also allows *fishing* by processes looking after sensors in real-time applications, so that they send only the latest available data once a receiver has been *caught*. It is also a useful primitive for modelling/implementing higher level synchronisation mechanisms (such as Teig's *x-channels* [95,96]).

The *fast* algorithm for resolving choice between arbitrary CSP events (e.g. barriers), mentioned at the start of this Section (3.4), is linear with respect to the number of events offered by each process (and needs no *back-tracking*). JCSP provides barrier and output *guards* with a cost that does not impact choices that restrict themselves to the set classically allowed by occam (inputs, timeouts and skips). In occam-π, we might also like to be able to write:

```
ALT
  SYNC bar
    ...  over the barrier, carry on
  out ! n
    ...  message taken, continue
  in ? x
    ...  message arrived, process it
  tim ? AFTER timeout
    ...  timed out, respond
```

However, the fast algorithm works for shared-memory multicore, but does not easily span multiple memory spaces. There may be a way to combine McEwan's *two-phase commit* protocol [97] (for distributed memory synchronisation) with our fast resolution (for shared memory) – more research is needed.

Finally, generic types have long been needed within the language. The code for buffering INTs is no different from that for REAL64s, nor for anything else! With the replacement of classical occam PROTOCOLs by session protocols (Section 3.4.4), union types (Section 3.4.1) and the revised array types (Section 3.4.2), this will be simpler to devise and achieve. Whether this should extend to a hierarchy of *types-with-associated-operations* (e.g. the data type FOO with operators <, = and > from Section 3.4.1), though not *classes*, should be considered.

4. Conclusions

Much received opinion is that object orientation and/or functional programming has solved the problems of management of state and that the problems of concurrency can be finessed with better use of traditional techniques. We do not subscribe to this view.

There is exciting, interesting and important work to do in language engineering for concurrent systems – by which we mean, of course, *all* future systems. Whether this takes the form of a rationalised and extended occam-π or completely new families (such as Guppy and ProcessJ) does not matter. All and more may happen. The important element is that lessons from the occam family are carried forward, with particular regard to *engineered-in* safety against avoidable concurrency errors (such as race hazard), safety against avoidable sequential errors (such as misplaced aliasing and callbacks), semantics that are composable and not context-sensitive (so that only local reasoning is needed), the ability to reason about systems intuitively as well as formally, and simplicity and (therefore) performance. Such engineering cannot happen without the close involvement of those developing the theory and those responsible for applications. We need to keep talking.

The future for software systems rebuilt on simple, secure and stable foundations, so that our energies as designers and programmers can focus on creative ideas, is there to be taken. For the future of our communities, environment and curiosity, it needs to be taken. Unless we take it, which means *work* and *funding*, it will not be there.

Acknowledgements

The opinions reported here are those of the author. The ideas, tools and applications around which they formed are the result of decades of work by highly skilled, insightful and respected colleagues all round the world. Special thanks are owed to Fred Barnes, Carl Ritson and Adam Sampson at Kent, to our partners on the TUNA and CoSMoS projects [67], to all past, present and future members of the the occam-π [33] and KRoC [34] (or derivatives) development teams and to the IFIP Working Group 2.4 for their patient listening, constructive criticism and valuable advice over the past ten years. Finally, some giants of the field (on whose shoulders we gratefully stand) must be acknowledged: Tony Hoare and Bill Roscoe (CSP), Robin Milner (π-calculus) and David May (classical occam).

References

[1] Wikipedia. Analytical Engine, 1837. http://en.wikipedia.org/wiki/Analytical_engine.

[2] Alfred N. Whitehead and Bertrand Russell. *Principia Mathematica (vols. I-III).* Cambridge University Press, 1910-1913.

[3] Wikipedia. Principia Mathematica, 1910-1913. http://en.wikipedia.org/wiki/Principia_Mathematica.

[4] Wikipedia. Gödel's incompleteness theorems, 1931. http://en.wikipedia.org/wiki/Godel%27s_incompleteness_theorems.

[5] Wikipedia. Lambda Calculus, 1936. http://en.wikipedia.org/wiki/Lambda_calculus.

[6] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, October 1937. doi:10.1112/plms/s2-42.1.230.

[7] Wikipedia. Z3 Computer, 1941. http://en.wikipedia.org/wiki/Z3_(computer).

[8] Wikipedia. Colossus Computer, 1943. http://en.wikipedia.org/wiki/Colossus_computer.

[9] Wikipedia. ENIAC, 1946. http://en.wikipedia.org/wiki/ENIAC.

[10] M. D. May. CSP, occam and Transputers. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 75–84. Springer Verlag, April 2005.

[11] Ngi. Nieuwe Ontwikkelingen voor Software Development, 2005. https://www.ngi.nl/Afdelingen/Informatie-Systemen/Verslagen/Nieuwe-ontwikkelingen-voor-Software-Development.html.

[12] Charles A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[13] Charles A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[14] Andrew W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[15] Steve Schneider. *Concurrent and Real-time Systems — The CSP Approach.* Wiley and Sons Ltd., UK, Baffins Lane, Chichester, UK, 1999. ISBN: 0-471-62373-3.

[16] Andrew W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.

[17] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999. ISBN-10: 0521658691, ISBN-13: 9780521658690.

[18] Dick Pountain and David May. *A Tutorial Introduction to occam Programming.* McGraw-Hill, Inc., New York, NY, USA, 1987.

[19] E.W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, June 1971. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF.

[20] P.H. Welch. Wot, No Chickens? Java Threads mailing list archive, September 1996. http://www.cs.kent.ac.uk/projects/ofa/java-threads/0.html.

[21] P.H. Welch. Java Threads in the Light of occam/CSP. In P.H. Welch and A.W.P. Bakkers, editors, *Proceedings of WoTUG 21*, volume 52 of *Concurrent Systems Engineering*, pages 259–284, Amsterdam, The Netherlands, April 1998. WoTUG, IOS Press. ISBN: 90-5199-391-9. http://www.cs.kent.ac.uk/pubs/1998/702/content.pdf.

[22] Charles A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[23] Peter H. Welch, Kurt Wallnau, Adam T. Sampson, and Mark Klein. To Boldly Go: an occam-π Mission to Engineer Emergence. *Natural Computing*, 11(3):449–474, September 2012. http://dx.doi.org/10.1007/s11047-012-9304-2.

[24] Peter H. Welch. Concurrency Design and Practice. `http://www.cs.kent.ac.uk/projects/ofa/sei-cmu/`, November 2007. A course on concurrency and process-oriented design, based on occam-π.

[25] P. H. Welch and F. R. M. Barnes. Communicating Mobile Processes: introducing occam-π. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[26] F. R. M. Barnes and P. H. Welch. Communicating Mobile Processes. In Ian East, Jeremy Martin, Peter H. Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*, volume 62, WoTUG-27 of *Concurrent Systems Engineering Series, ISSN 1383-7575*, pages 201–218, Amsterdam, The Netherlands, September 2004. IOS Press. ISBN: 1-58603-458-8.

[27] P. H. Welch and F. R. M. Barnes. Mobile Barriers for occam-π: Semantics, Implementation and Application. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures 2005*, volume 63, WoTUG-28 of *Concurrent Systems Engineering Series*, pages 289–316, Amsterdam, The Netherlands, September 2005. IOS Press. ISBN: 1-58603-561-4.

[28] P. H. Welch and F. R. M. Barnes. A CSP Model for Mobile Channels. In *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 17–33, Amsterdam, The Netherlands, September 2008. IOS Press. ISBN: 978-1-58603-907-3.

[29] P. H. Welch. *An occam-π Quick Reference Guide*. Programming Languages and Systems Research Group, University of Kent, `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference/`, 2011.

[30] Peter H. Welch and Jan B. Pedersen. Santa Claus: Formal Analysis of a Process-Oriented Solution. *ACM Trans. Program. Lang. Syst.*, 32(4):14:1–14:37, April 2010. `http://doi.acm.org/10.1145/1734206.1734211`.

[31] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement (FDR2 User Manual)*. Oxford University Computing Laboratory. `http://www.cs.ox.ac.uk/projects/concurrency-tools/`, 2.94 edition, May 2012. `http://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf`.

[32] Peter H. Welch and Neil C.C. Brown. Self-Verifying Dining Philosophers. Presentation to IFIP Working Group 2.4, September 2011. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/IFIP_WG24`.

[33] A. T. Sampson, C. G. Ritson, M. C. Jadud, F. R. M. Barnes, and P. H. Welch. *occam-π Home Page*. Programming Languages and Systems Research Group, University of Kent, `http://occam-pi.org/`, 2010.

[34] Frederick R. M. Barnes, Peter H. Welch, Jim Moores, and David C. Wood. *The KRoC Home Page*. Programming Languages and Systems Research Group, University of Kent, `http://www.cs.kent.ac.uk/projects/ofa/kroc/`, 2010.

[35] G.H. Hilderink, J.F. Broenink, and A. W. P. Bakkers. Communicating Java Threads. In *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50, pages 48–76, University of Twente, Netherlands, 1997. IOS Press, Netherlands.

[36] G.H. Hilderink, J.F. Broenink, and A. W. P. Bakkers. Communicating Threads for Java. In *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–261. Proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, Keele, United Kingdom, 1999.

[37] A. W. P. Bakkers, G.H. Hilderink, and J.F. Broenink. A Distributed Real-time Java System Based on CSP. In *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 229–241. Proceedings of the 22nd World Occam and Transputer User Group Technical Meeting, Keele, United Kingdom, 1999.

[38] M. M. Bezemer, R. J. W. Wilterdink, and J.F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In P.H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J.F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV. WoTUG-33.

[39] P. H. Welch and N. C. C. Brown. The JCSP (CSP for Java) Home Page, 2011. `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`.

[40] P.H. Welch. Process Oriented Design for Java: Concurrency for All. In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002. Keynote Tutorial.

[41] P. H. Welch, N. C. C. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process*

Architectures 2007, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370, Amsterdam, The Netherlands, July 2007. IOS Press. ISBN: 978-1-58603-767-3.

[42] Peter H. Welch, Neil C. C. Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, March 2010. The DOI should redirect to `http://onlinelibrary.wiley.com.chain.kent.ac.uk/doi/10.1002/cpe.1471/abstract`.

[43] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M. Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, The Netherlands, April 1999. WoTUG, IOS Press. ISBN: 90-5199-480-X. `http://www.cs.kent.ac.uk/pubs/1999/753`.

[44] James Moores. *The Design and Implementation of occam/CSP Support for a Range of Languages and Platforms*. PhD thesis, The University of Kent at Canterbury, Canterbury, Kent. CT2 7NF, December 2000.

[45] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer, June 2009. `http://www.cs.kent.ac.uk/pubs/2009/2928/`.

[46] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. *Science of Computer Programming*, 77(6):727–740, 2012. Revised from LNCS (5521) paper with same name.

[47] Carl G. Ritson. *Scalable Support for Process-Oriented Programming*. PhD thesis, School of Computing, University of Kent, March 2013. (Awarded July, 2013).

[48] N. C. C. Brown and P. H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, ISSN 1383-7575, pages 139–156, Amsterdam, The Netherlands, September 2003. IOS Press. ISBN: 1-58603-381-6.

[49] N. C. C. Brown. *C++CSP Home Page*. Programming Languages and Systems Research Group, University of Kent, `http://www.cs.kent.ac.uk/projects/ofa/c++csp/`, 2010.

[50] N. C. C. Brown. *Communicating Haskell Processes Home Page*. Programming Languages and Systems Research Group, University of Kent, `http://www.cs.kent.ac.uk/projects/ofa/chp/`, 2010.

[51] Neil C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan, Gardner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 67–83, Amsterdam, The Netherlands, September 2008. WoTUG, IOS Press.

[52] Bernard Sufrin. Communicating Scala Objects. In *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 35–54, Amsterdam, The Netherlands, sep 2008. IOS Press.

[53] Brian Vinter, John Markus Bjrndalen, and Rune Mllegard Friborg. PyCSP Revisited. In *Communicating Process Architectures 2009*, volume 67, WoTUG-32 of *Concurrent Systems Engineering Series*, pages 263–276, Amsterdam, The Netherlands, nov 2009. IOS Press.

[54] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert Newman. CSP as a Domain-Specific Language Embedded in Python and Jython. In *Communicating Process Architectures 2009*, volume 67, WoTUG-32 of *Concurrent Systems Engineering Series*, pages 293–309, Amsterdam, The Netherlands, nov 2009. IOS Press.

[55] Waide B. Tristram and Karen Bradshaw. Hydra: A Python Framework for Parallel Computing. In *Communicating Process Architectures 2009*, volume 67, WoTUG-32 of *Concurrent Systems Engineering Series*, pages 311–324, Amsterdam, The Netherlands, nov 2009. IOS Press.

[56] M.C. Jadud. Parallel Programming for the Rest of Us, 2013. `http://concurrency.cc/`.

[57] Christian L. Jacobsen, Matthew C. Jadud, Omer Kilic, and Adam T. Sampson. Concurrent Event-driven Programming in occam-π; for the Arduino. In *Communicating Process Architectures 2011*, volume 68, WoTUG-33 of *Concurrent Systems Engineering Series*, pages 177–193, Amsterdam, The Netherlands, June 2011. IOS Press. ISBN: 978-1-60750-773-4.

[58] Ian Armstrong, Michael Pirrone-Brusse, A. Smith, and Matthew C. Jadud. The Flying Gator: Towards Aerial Robotics in occam-π. In *Communicating Process Architectures 2011*, volume 68, WoTUG-33 of *Concurrent Systems Engineering Series*, pages 329–340, Amsterdam, The Netherlands, June 2011. IOS Press. ISBN: 978-1-60750-773-4.

[59] F.R.M. Barnes. RMoX: an occam-pi Operating-System, January 2005. Available at: `http://rmox.net`.

[60] F.R.M. Barnes and C.G. Ritson. Process-Oriented Device Driver Development. *Concurrency and Computation: Practice and Experience*, 22(8):995–1006, June 2010.

[61] S. Stepney, P.H. Welch, F.A.C. Pollack, J.C.P. Woodcock, S. Schneider, H.E. Treharne, and A.L.C. Cavalcanti. TUNA: Theory Underpinning Nanotech Assemblers (feasibility study), January 2005. EPSRC grant EP/C516966/1. Available from: `http://www.cs.york.ac.uk/nature/tuna/index.htm`.

[62] C. G. Ritson and P. H. Welch. A Process-Oriented Architecture for Complex System Modelling. *Concurrency and Computation: Practice and Experience*, 22:965–980, March 2010.

[63] C. Ritson and P.H.Welch. 3D Blood Clotting (TUNA), 2006. `https://www.cs.kent.ac.uk/research/groups/sys/wiki/3D\_Blood\_Clotting/`.

[64] P. H. Welch, A. T. Sampson, and D. C. Wood. *"To Boldly Go: an occam-π Mission to Engineer Emergence" – supplementary materials*. Programming Languages and Systems Research Group, University of Kent, `http://projects.cs.kent.ac.uk/projects/naco-boldly/`, 2011.

[65] Paul S. Andrews, Adam T. Sampson, John Markus Bjørndalen, Susan Stepney, Jon Timmis, Douglas N. Warren, and Peter H. Welch. Investigating Patterns for the Process-oriented Modelling and Simulation of Space in Complex Systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, August 2008.

[66] Fiona A.C. Polack, Paul S. Andrews, and Adam T. Sampson. The Engineering of Concurrent Simulations of Complex Systems. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 217–224. IEEE Press, 2009.

[67] Susan Stepney, Peter H. Welch, Fiona Polack, Jon Timmis, Fred R. M. Barnes, and Andy Tyrrell. CoSMoS Home Page, 2010. EPSRC EP/E053505/1 and EP/E049419/1. `http://www.cosmos-research.org/about.html`.

[68] S. Stepney, P.H. Welch, J. Timmis, C. Alexander, F.R.M. Barnes, M. Bates, F.A.C. Polack, and A. Tyrrell. CoSMoS: Complex Systems Modelling and Simulation infrastructure, April 2007. EPSRC grants EP/E053505/1 and EP/E049419/1. URL: `http://www.cosmos-research.org/about.html`.

[69] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating Complex Systems. In Michael G. Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117. IEEE, August 2006.

[70] P. H. Welch, N. C. C. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh. Alting Barriers: Synchronisation with Choice in Java using CSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, 2010.

[71] F.R.M. Barnes. Compiling CSP. In P.H. Welch, J. Kerridge, and F.R.M. Barnes, editors, *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering Series*, pages 377–388, Amsterdam, The Netherlands, September 2006. IOS Press. ISBN: 1-58603-671-8.

[72] Geoff Barrett. *occam3 Reference Manual*. INOS Ltd., March 1992. `http://www.wotug.org/occam/documentation/oc3refman.pdf`.

[73] F.R.M. Barnes. Towards a New Language for Concurrent Programming. *CPA 2011 Fringe:* slides at `http://www.wotug.org/papers/CPA-2011/Barnes11/Barnes11-slides.pdf`, August 2011. Guppy Home Page: `http://frmb.org/guppy.html`.

[74] P.H. Welch. Union Data Types. *occam Enhancement Proposal 156*, April 2006. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/156`.

[75] J.C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *Proceedings of the 17th. Annual Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, July 2002. WoTUG, IEEE Press. ISBN: 0-7695-1483-9. `http://dx.doi.org/10.1109/LICS.2002.1029817`.

[76] Neil C.C. Brown. Auto-Mobiles: Optimised Message-Passing. In *Communicating Process Architectures 2009*, volume 67, WoTUG-32 of *Concurrent Systems Engineering Series*, pages 225–238, Amsterdam, The Netherlands, nov 2009. IOS Press.

[77] Adam T. Sampson. Two-Way Protocols for occam-π. In Peter H. Welch, S. Stepney, F.A.C Polack, Frederick R. M. Barnes, Alistair A. McEwan, G. S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 85–97, Amsterdam, The Netherlands, sep 2008. IOS Press.

[78] Adam T. Sampson. *Process-Oriented Patterns for Concurrent Software Engineering*. PhD thesis, University of Kent, October 2010.

[79] K. Honda. Types for Dyadic Interaction. In *CONCUR93: Proceedings of the International Conference on Concurrency Theory*, page 509523. Springer-Verlag, 1993. LNCS 715.

[80] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE 94: Parallel Architectures and Languages Europe*, pages 398–413. Springer-Verlag, 1994. LNCS 817.

[81] Nobuko Yoshida and Vasco T. Vasconcelos. Language Primitives and Type Discipline for Structured Communication-based Programming. In *Proceedings of the European Symposium on Programming 1998*, pages 122–138. Springer-Verlag, 1998. LNCS 1381.

[82] Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session Types for Inter-Process Communication. Technical report, Department of Computing Science, University of Glasgow, 2003. TR-2003-133.

[83] Peter H. Welch, Jan Baekgaard Pedersen, Frederick R. M. Barnes, Carl G. Ritson, and Neil C.C. Brown. Adding Formal Verification to occam-π. In *Communicating Process Architectures 2011*, volume 68, WoTUG-33 of *Concurrent Systems Engineering Series*, pages 379–379 *(Endnote Presentation)*, Amsterdam, The Netherlands, jun 2011. IOS Press. ISBN: 978-1-60750-773-4. `http://www.wotug.org/papers/CPA-2011/Welch11/Welch11-slides.pdf`.

[84] Peter H. Welch, Jan B. Pedersen, Frederick R.M. Barnes, and Carl G. Ritson. Self-Verifying Concurrent Programming. Presentation to IFIP Working Group 2.4, September 2011. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/IFIP_WG24`.

[85] P.H. Welch and E. Bonnici. Variant Call Protocols. *occam Enhancement Proposal 178*, August 2010. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/178`.

[86] Eric Bonnici and Peter H. Welch. Mobile Processes and Call Channels with Variant Interfaces (a Duality). In *Communicating Process Architectures 2011*, volume 68, WoTUG-33 of *Concurrent Systems Engineering Series*, pages 377–377 *(Fringe Presentation)*, Amsterdam, The Netherlands, June 2011. IOS Press. ISBN: 978-1-60750-773-4.

[87] F.R.M. Barnes. Extended Rendezvous. *occam Enhancement Proposal 110*, December 2001. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/110`.

[88] Jan Baekgaard Pedersen and Matthew Sowders. Static Scoping and Name Resolution for Mobile Processes with Polymorphic Interfaces. In Peter H. Welch, Adam T. Sampson, Jan Baekgaard Pedersen, Jon Kerridge, Jan F. Broenink, and Frederick R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 71–85, jun 2011.

[89] J.B.Pedersen and M.L.Smith. ProcessJ: A Possible Future of Process Oriented Design. In *Communicating Process Architectures 2013*. Open Channel Publishing, August 2013.

[90] John Markus Bjørndalen and Adam T. Sampson. Process-Oriented Collective Operations. In *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, pages 309–328, Amsterdam, The Netherlands, September 2008. WoTUG, IOS Press. `http://www.cosmos-research.org/docs/cpa2008-poco-slides.pdf`.

[91] Adam T. Sampson, John Markus Bjørndalen, and Paul S. Andrews. Birds on the Wall: Distributing a Process-Oriented Simulation. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 225–231. IEEE Press, May 2009.

[92] Frederick R. M. Barnes. occam-π Pre-processing Support, 2006. `http://frmb.org/occ21-extensions.html#preproc`.

[93] N.C.C. Brown and C.G. Ritson. Extended Outputs. *occam Enhancement Proposal 142*, March 2006. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/142`.

[94] Neil C.C. Brown. How to Make a Process Invisible. In *Communicating Process Architectures 2008*, volume 66, WoTUG-31 of *Concurrent Systems Engineering Series*, pages 445–445 *(Fringe Presentation)*, Amsterdam, The Netherlands, sep 2008. IOS Press.

[95] Oyvind Teig. XCHANs: Notes on a New Channel Type. In *Communicating Process Architectures 2012*, pages 155–170. Open Channel Publishing, August 2012.

[96] Peter H. Welch. An occam Model of XCHANs, 2013. `https://www.cs.kent.ac.uk/research/groups/plas/wiki/An_occam_Model_of_XCHANs`.

[97] A.A. McEwan. *Concurrent Program Development*. DPhil thesis, The University of Oxford, 2006.