Submission to : *'Journeys in Non-Classical Computation'* (GC7)
Title : *'Through the Concurrency Gateway'*
From : Peter Welch, University of Kent Computing Laboratory

Massive Parallelism – *breaking the von Neumann paradigm:*

The real world exhibits concurrency at all levels of scale – from atomic, through human, to astronomic. This concurrency is endemic. Central points of control do not remain stable for long. Most of the novel paradigms identified in the GC7 description paper hint at something stronger – namely that central points of control actively *work against* the logic and efficiency of whatever it is that is we are trying to control/model/understand and that, in the long term, we must give up on it. The case in this submission is that *it is necessary to give up on it now*, that it is possible to do so, that it will be profitable to do so and that pushing through this particular gateway, by the mainstream computing community, will help set up the mindset for the much grander challenges outlined for this 'journey'.

In present day computer engineering, concurrency is not considered a fundamental concept – to be used everyday with the same fluency as we might use object-orientation or while-loops. It is taught, almost universally, only as an advanced topic and only to be used when there are no other ways to obtain specific performance targets. Examples include the reduction of response times to external interrupts/commands (whilst long running background computations are continuing), or the speed-up of completion times for large-scale scientific or engineering calculations (through the use of multi-processors).

Standard concurrency technologies are based on multiple threads of execution plus various kinds of locks to control the sharing of data between them. Get the locking wrong and systems will mysteriously corrupt themselves or deadlock. Received wisdom from decades of practice is that concurrency is very hard, and we are advised to steer well clear if at all possible [1].

In addition to these logical problems, there are also performance problems. Standard thread management imposes significant overheads in the form of additional memory demands (to maintain thread state) and run time (to allocate and garbage-collect thread state, to switch processor context between states, to recover from cache misses resulting from switched contexts, and to execute the protocols necessary for the correct and safe operation of locks). Even when using only *`lightweight'* threads, applications need to limit their implementations to only a few hundred threads per processor – beyond which performance catastrophically collapses (usually as a result of memory thrashing).

Modern computing already faces a dilemma: it is driven by ever-increasing demands for system functionality, performance, responsiveness, inter-operability, dynamics, safety, and security. Yet our standard concurrency models and tools, which ought to be fundamental in addressing these demands, throw up serious new problems that act against them. As a result, concurrency is used on a relatively small scale, where its analysis is (just) manageable and the performance benefits outweigh the overheads.

But the problems – and our ambitions, even now – are much bigger than this. For example, air traffic control over the UK requires the management of far greater concurrency than standard practice will directly and safely and simply allow. Common web services need to be able to conduct business with tens of thousands of clients simultaneously. Modelling even the simplest biological organisms quickly takes us into consideration of millions of concurrently active, autonomous, and interacting, agents. Limited by such constraints, we have to compromise on the degree of concurrency in our application design and implementation. Those compromises add significant complexity that, combined with the semantic instability of the concurrency mechanisms we do practice, lead to mistakes and the poor quality, late delivery and over-budget systems that are accepted as normal – for now – by our industry and its customers.

This submission suggests some ways for leaving these constraints behind.

Hypothesis to be Tested:

All computer systems have to model the real world, at some appropriate level of abstraction, if they are to receive information (data, signals, *etc.*) and feedback useful information (reports, control, *etc.*). To make that modelling easier, we should expect concurrency to play a fundamental rôle in the design and implementation of systems, reflecting the reality of the environment in which they are embedded. This does not currently seem to be the case.

Our thesis is that computer science has taken at least one wrong turn. Concurrency should be a natural way to design any system above a minimal level of complexity. It should simplify and hasten the construction, commissioning, and maintenance of systems; it should not introduce the hazards that are evident in modern practice; it should be employed as a matter of routine. Natural mechanisms should map into simple engineering principles with low cost and high benefit. Our hypothesis is that this is possible.

We propose a computational framework, based on established ideas of process algebra, to test the truth of the above hypothesis. It will be accessible from current computing environments (platforms, operating systems, languages) but will provide a foundation for novel ones in the future. It will integrate the best ideas from Hoare's CSP[2] and Milner's π-calculus[3], though this will require additional work on the theory.

CSP has a *compositional* and *denotational* semantics, which means that it allows modular and incremental development (*refinement*) even for concurrent components. In turn, this means that we get no surprises when we run processes in parallel (since their points of interaction have to be explicitly handled by all parties to these interactions). This is simply not the case for standard threads-and-locks concurrency, which have no formal denotational semantics and by which we get surprised all the time.

However, we need some extensions to describe certain new dynamics – and this is where we turn to the π-calculus. Specifically, we want to allow networks of processes to *evolve*, to change their topologies, to cope with growth and decay without losing semantic or structural integrity. We want to address the *mobility* of processes, channels and data and understand the relationships between these ideas. We want to retain the ability to reason about such systems, preserving the concept of refinement.

The framework has to provide highly efficient practical realisations of this extended model. Its success in opening up the long term horizons of GC7 will be a long term test of the above hypothesis. Shorter term tests will be the development of demonstrators (relevant to a broad range of computer applications – including those that are of concern to GC1, GC4 and GC6) with the following characteristics:

- they will be as complex as needed – and no more (*e.g.* through the *concurrency in the design* being directly delivered by the *concurrency in the implementation*);
- they will be scalable both in performance and function; *[Note: by functional scalability, we mean that the cost of incremental enhancement depends only on the scale of that enhancement – not upon the scale of the system being enhanced. The latter is the present state-of-the-art and is a major reason behind system delay and eventual failure.]*
- they will be amenable to formal specification and verification;
- notwithstanding the above, the concurrency models (and mechanisms) in their design (and implementation) will be practical for everyday use by non-specialists – concurrency becomes a fundamental element in the toolkit of every professional computer engineer;
- they will make maximum use of the underlying computation platform (through significantly reduced overheads for the management of concurrency – including the response times to interrupts).

Current State of our Framework:

Over the past ten years, our group[4] at Kent has been devoted to laying the foundations for such a framework. We have developed – and released as open source – concurrency packages for the Java (JCSP), C (CCSP), C++ (C++CSP) and J# (J#CSP) programming languages [5-9]. Despite their names, they all provide the mobile dynamics fed in from the π-calculus (although it is easy

to mis-program them, since their base languages do not have a clue as to what is happening). We have also advanced the original CSP programming language, occam, to do the same – but with some major safety and performance benefits (because the base language does know what is happening). An overview of the current state and potential of this language (christened, for the moment, occam-π) is given below. More detailed overviews of this work (on *JCSP* and occam-π) can be found on-line at [10].

occam-π is a sufficiently small language to allow experimental modification and extension, whilst being built on a language of proven industrial strength. It integrates the best features of CSP and the π-calculus, focussing them into a form whose semantics is intuitive and amenable to everyday engineering by people who are not specialised mathematicians – the mathematics is built into the language design, its compiler, run-time system and tools *(so that users benefit automatically from that foundation)*. The new dynamics broadens its area of direct application to a wide field of industrial, commercial and scientific practice.

occam-π runs on modern computing platforms and has much of the flexibility of Java and C, whilst at the same time retaining all the safety guarantees of *classical* occam (e.g. against aliasing and parallel usage errors) and the lightness of its concurrency mechanisms. It supports the dynamic allocation of processes, data and channels, their *movement* across channels and their automatic de-allocation (without the need for garbage collection, which otherwise invalidates real-time guarantees) [11-15]. We have extended the range of static safety checks so that aliasing errors and race hazards are not possible in occam-π systems, despite the new dynamics. This means that subtle side-effects between component processes cannot exist, which impacts (positively) on the general scalability and dependability of systems. The *mobility* and *dynamic construction* of processes, channels and data opens up a wealth of new design options that will let us follow nature more closely – with network structures *evolving* at run-time. Apart from the logical benefits derived from such directness and flexibility, there will be numerous gains for application efficiency.

Performance overheads for all occam-π concurrency mechanisms are mostly unit time, with the order of between 50 and 150 *nanoseconds* on modestly powered PCs (1GHz). Memory overheads are also very light: no more than 8 words per process. This means that dynamic systems evolving hundreds of thousands of (non-trivial) processes are already practical on single processors. Those processes can be implementing complex behaviour with time and space overheads for managing the concurrency minimal (less than 10%). Further, occam-π networks can naturally span many machines – the concurrency model does not change between internal and external concurrency. Application networks up to millions of serious processes then become viable (e.g. on modest clusters of laptops). The continuing progress of Moore's Law likely over the next few years means that networks of tens of millions of (non-trivial) processes will become possible.

A formal *denotational* semantics for occam-π mobile processes, based on Hoare and Jifeng's *Unified Theory of Programming* [16], has been drafted by Jim Woodcock (also at Kent) and one of his students (Xingbei Tang). However, these mobiles have not yet been implemented by the occam-π compiler and kernel, although we believe this will be straightforward. This contrasts to the status of mobile *channel-ends*, which are fully supported by the current system but for which a denotational semantics is still being researched.

One Model Application:

With colleagues at Royal Holloway and at York, we are interested in questions about dependability and evolution for novel embedded networks that may become viable during the next 10-20 years: *Nanite Assemblers*. These are active devices that manipulate their world (*e.g.* a human body) at the nanoscopic level, but which cause macroscopic effects (*e.g.* through cooperating to assemble artefacts 'cell' by 'cell'). In order to be effective, vast numbers of such nanites are needed, and these numbers may grow exponentially as they assemble copies of themselves. We need the capabilities to design, model, program and control complex and dynamic networks of these exotic machines, and to give credible assurance that they behave properly. As with swarm intelligence and ant colony algorithms, much of the interesting behaviour of a host of nanites comes from *emergent* properties.

Hierarchical networks of communicating processes are particularly suitable for these problems. But the languages used to support modelling and simulation must be simple, formal, and dynamic, and have a high-performance implementation. The models of such complex system must be as simple as possible. The models must be amenable to manipulation and formal reasoning. The topologies of these networks will evolve dramatically, as they support growth and decay that comes from nanites moving, splitting, and combining. Individual nanites must not only be mobile, they must also be aware of their own location and the proximity of their neighbours. Finally, simulations will require very large numbers of processes, so their implementation had better have very low overheads.

A good candidate for modelling and programming such systems is occam-π: it is robust and lightweight, and has sound theoretical support. We know how to construct systems to the order of 10^6 processes on modest processor resources, exhibiting rich behaviours in useful run-times. This is enough to make a start on our journey.

References and URLs:

[1] H. Muller and K. Walrath. *Threads and Swing (final section): Why did we implement Swing this way?* www.java.sun.com/products/jfc/tsc/articles/threads/threads1.html. 2000.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[3] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[4] P.H. Welch et al. *Concurrency Research Group*. www.cs.kent.ac.uk/research/groups/crg/. 2004.

[5] P.H. Welch. *CSP for Java (JCSP) Home Page*. www.cs.ukc.ac.uk/projects/ofa/jcsp/. 2004.

[6] P.H. Welch, J. R. Aldous and J. Foster. *CSP Networking for Java (JCSP.net)*. Computational Science – ICCS 2002, pp. 695--708. Springer-Verlag, April 2002.

[7] P.H. Welch and B. Vinter. *Cluster Computing and JCSP Networking*. Communicating Process Architectures 2002, *Concurrent Systems Engineering Series* (60) pp. 203-- 222. IOS Press, 2002.

[8] N.C.C. Brown and P.H. Welch. *An Introduction to the C++CSP Library*. Communicating Process Architectures 2003, *Concurrent Systems Engineering Series* (61) pp. 139--156. IOS Press, 2003.

[9] Quickstone Technology Ltd. *xCSP Home Page*. www.quickstone.com/xcsp/. 2004.

[10] P.H. Welch. *IFIP WG2.4 (Peter Welch's page)*. www.cs.kent.ac.uk/projects/ofa/ifip/. 2004.

[11] P.H. Welch and F.R.M. Barnes. *KRoC Home Page*. www.cs.kent.ac.uk/projects/ofa/kroc/. 2004.

[12] P.H. Welch and F.R.M. Barnes. *Prioritised Dynamic Communicating and Mobile Processes*. IEE Proceedings Software, 150(2), April 2003. www.iee.org/Publish/Journals/ProfJourn/Proc/SEN/Preprints.cfm

[13] P.H. Welch and F.R.M. Barnes. *Mobile Data, Dynamic Allocation, and Zero Aliasing: an occam Experiment*. CPA 2001, *Concurrent Systems Engineering Series* (59) pp. 243-- 264. IOS Press, 2001.

[14] M. Schweigler, F.R.M. Barnes and P.H. Welch. *Flexible, Transparent, and Dynamic occam Networking with KRoC.net*. Communicating Process Architectures 2003, *Concurrent Systems Engineering Series* (61) pp. 199-- 224. IOS Press, 2003.

[15] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. *RMoX: a Raw Metal occam Experiment*. Communicating Process Architectures 2003, *Concurrent Systems Engineering Series* (61) pp. 269--288. IOS Press, 2003.

[16] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1999.