http://www.cs.bris.ac.uk/~alan/Java/ieeelet.html

To:
Edward A. Parrish
The Editor
IEEE Computer
(computer@wpi.edu)

Dear Sir,

Ted Lewis' article ("If Java is the Answer, What Was the Question?", Binary Critic, March 1997) makes stimulating reading. We cheered his sentiments on the failures of *Universal Common Language* and the awful legacy of maintenance mountains. We echoed his approval of the bold application, by the designers of Java, of Occam's Razor to C/C++ (e.g. the dispatching of functions, structures and explicit pointers) and shared his frustration that they didn't quite cut it far enough (e.g. the retention of side-effecting operators).

But then we came to his main example on `Threadbare Java', where we felt that his warnings ("Java atomic procedures and lightweight threads can lead programmers down the road to ruin") needed some qualification! He is absolutely right, of course, but there may be something that can be done to rescue things ... like hiding the Java primitives under a layer of abstraction that presents something simpler and safer to the user.

Ted complains of a "missed golden opportunity", by the designers of Java threads, "to make future systems more reliable". He illustrates this with a small two-threaded example which "from the programmer's point of view ... (has) nothing wrong with the code", but whose execution *sometimes* deadlocks! Now, intermittent system deadlock is about the worst nightmare of any engineering team needing to get a product delivered by a week last Tuesday. Common practice seems to be to ignore it and rely on customer patience and a re-boot mechanism (< ctl > + < alt > + < del > ?); but that's not much use if the deadlock is crashing your car ...

What beggars belief is that we are still getting into such a mess in 1997. Ted is spot on in saying that worrying about such synchronisation problems when writing application code is far too late. So why are we still doing this?!! For twenty years or so, we have known that problems like deadlock, livelock and thread starvation must be confronted and eliminated at design time and that this implies that the ways in which threaded components synchronise with their environments (i.e. other threaded components) must be burnt into their specifications. We need a theory (and it has to be a *mathematical* theory) of synchronising processes through which we can make and refine the necessary specifications and for which we can develop system building methods that can

be proven safe (*before* we incorporate them into tools and/or programming languages). This theory exists.

Ted mentions *Path Pascal* [1], which enables a useful measure of discipline to be specified on the order in which synchronisations can take place. But surely the major candidate for sorting out this mess is Hoare's *Communicating Sequential Processes* (CSP)? First published in 1978 [2], CSP had evolved through two generations [3][4] to the accompaniment of a wealth of academic literature, software tools, programming languages and industrial practice. Even so, the mainstream computer world seems content to pass it by.

The standard web tutorial on Java [5] frustratingly states: "The Java language and runtime system support thread synchronisation through the use of monitors, which were first outlined in C.A.R.Hoare's article *Communicating Sequential Processes*" and they refer to the CACM reference [2]. Sadly, this is not quite right! The Java White Paper [6] gets the reference right, quoting the earlier work on monitors [7] developed by Hoare (and others) in the late '60s and early '70s. But Hoare climbed away from this and broke entirely *new* ground in the late '70s with CSP.

One crucial benefit of CSP is that its thread semantics are compositional (i.e. WYSIWYG), whereas monitor thread semantics are context-sensitive (i.e. non-WYSIWYG and that's why they hurt!). Example: to write and understand *one* synchronized method in a (Java) class, we need to write and understand *all* the synchronized methods in that class *at the same time* -- we can't knock them off one-by-one! This does not scale!! We have a combinatorial explosion of complexity!!!

With CSP, threads can refuse individual events if they are not in a state to accept them. They do not wait on shared condition variables and do not rely on other methods to fix things up for them. Each method has its own contract and looks after itself. This type of logic *does* scale.

Java threads *should* have been built upon CSP ... but they used the wrong paper of Hoare and built it upon monitors. Fortunately, there is a way to build the CSP model on top of Java threads. We call it JavaPP (*Java(TM) Plug-and-Play*) [8].

We just need to create a class library for CSP events, channels and alternation (and, possibly, a higher-level kit bag of buffers, multicasters, multiplexors, routers etc.). The implementors of the CSP primitives need to get (carefully) involved in those difficult non-WYSIWYG Java primitives (e.g. synchronized, wait, notify), but nobody else does. The Java language is unchanged, but application programmers work solely with the CSP classes to glue their threads together. They can inherit all the CSP design and analysis methods and tools. Multi-threaded systems become structured to reflect real world system hierarchies. Components become automatically thread-safe and reusable. The

nasty accidents of deadlock, livelock and starvation can be ruled out by design. System complexity can be ramped up with linear increase in effort. And no run-time overheads are imposed that wouldn't be needed anyway to prevent race hazards. Of course, without changing the Java language, we cannot match the full security rigour (or even the performance) achieved by a CSP-aware language such as occam [9], but these are significant wins.

The above work has been done. Initial results were presented at a WoTUG workshop in England last September [10] and further refined at the WoTUG-20 conference in the Netherlands in April [11]. A one-day tutorial will be presented this coming June at the PDPTA'97 conference in Las Vegas [12].

How does the particular deadlock outlined in Ted's article fare in the world of *JavaPP*? We are a bit puzzled by the code fragments provided, but from the textual description this is a classic example of *deadlock-by- multiply-acquired-resource*. There is an equally classic design solution called *Resource Allocation Priority* (RAP) [13] (whereby the resources are given an arbitrary ordering and processes needing them all are obliged to acquire them sequentially and in that order). So, the *JavaPP* design would have specified a RAP pattern on those shared buffers and the threads would have to play the game -- design tools can be built to capture such rules and check that they are followed.

There are other design rules meeting other design aims that guarantee freedom from deadlock, livelock and starvation for CSP-conforming systems. Examples include a precise definition of client-server communications (for irregular patterns of synchronisation such as occur in GUIs and other reactive systems) and *cyclic-PO*, *I/O-PAR* and *I/O-SEQ* (for regular patterns such as occur in scientific computations, control applications, multimedia processing etc.) [13][14][15]. There are mature tools for analysing and refining general CSP specifications [16] and for supporting directly a range of design rules [17][18] -- all of which can be exploited within *JavaPP*.

Threads under the CSP discipline are a powerful mechanism for managing complexity in systems. The opportunities they afford for increased performance (through multiprocessors) are an excellent, but secondary, bonus. It's time to change the culture: *Keep-It-Simple-Stupid* no longer means stick with one thread until you are forced otherwise. *KISS* means using threads because they are natural -- and nature does have a way of coming up with simple solutions to complex problems. Java's native thread mechanisms missed the trick that makes this possible and they need repairing -- JavaPP is one such repair.

Yours etc.

Signed:

Alastair Allen (University of Aberdeen, UK)
Andre Bakkers (University of Twente, Netherlands)

Richard Beton (Roke Manor Research Limited, UK)
Alan Burke (Aurigor Telecom Systems, Canada)
Alan Chalmers (University of Bristol, UK)
Barry Cook (University of Keele, UK)
Michael Goldsmith (Formal Systems (Europe) Ltd, UK)
Gerald Hilderink (University of Twente, Netherlands)
Ruth Ivimey-Cook (Advanced RISC Machines Ltd, UK)
Adrian Lawrence (University of Oxford, UK)
Jeremy Martin (University of Oxford, UK)
Nan Schaller (Rochester Institute of Technology, USA)
Dyke Stiles (Utah State University, USA)
Oyvind Teig (Autronica, Norway)
Paul Walker (4 Links, UK)
Peter Welch (University of Kent, UK)

References:

[1] R.H.Campbell and R.B.Kolstad: `An overview of Path Pascal's design', and `Path Pascal User Manual', ACM SIGPLAN Notices, 15(9), pp. 13-24, 1980.

[2] C.A.R.Hoare: `Communicating Sequential Processes', CACM 21(8), pp. 666-677, 1978.

[3] C.A.R.Hoare: `Communicating Sequential Processes', Prentice-Hall, 1984.

[4] Oxford University Computer Laboratory: `The CSP Archive'.

[5] JavaSoft: `Threads of Control (Monitors)'.

[6] Java White Papers: `The Java language: an overview'.

[7] C.A.R.Hoare: `Monitors: an operating system structuring concept', CACM 17(10), pp. 549-557 (1974).

[8] JavaPP: `Java Plug-and-Play'.

[9] occam Language definition and compilers.

[10] WoTUG: `Java Threads Workshop'.

[11] WoTUG-20: `Parallel Programming and Java', edited by A.W.P.Bakkers, IOS Press, Amsterdam, ISBN 90-5199-336-6, 1997.

[12] PDPTA'97: `Java(TM) Plug-and-Play', June 29th., 1997, Las Vegas.

[13] A.W.Roscoe and N.Dathi: `The Pursuit of Deadlock Freedom', Technical Monograph PRG-57, Oxford University Computing Laboratory, 1986.

[14] P.H.Welch and G.R.R.Justo and C.Willcock: `High-Level Paradigms for Deadlock-Free High-Performance Systems', in Transputer Applications and Systems '93, edited by R.Grebe et al., IOS Press, Amsterdam, ISBN 90-5199-140-1, 1993.

[15] J.M.R.Martin and I.East and S.Jassim: `Design Rules for Deadlock Freedom', Transputer Communications 2(3), pp. 121-133, John Wiley and Sons Ltd., ISSN 1070-454X, 1994.

[16] Formal Systems (Europe) Ltd.: `FDR2 User Manual and Tutorial', 3 Alfred Street, Oxford, OX1 4EH, England, 1997.

[17] D.J.Beckett and P.H.Welch: `A Strict occam Design Tool', in Proceedings of UK Parallel '96, edited by C.R.Jesshope et al., pp. 53-69, Springer-Verlag, ISBN 3-540-76068-7, 1996.

[18] J.M.R.Martin and P.H.Welch: `A Design Strategy for Deadlock-Free Concurrent Systems', Transputer Communications, 3(4), John Wiley and Sons Ltd. (in press).