, ***International Journal of Computers and Applications*, 23, (3)**, pp. 159-165, ISSN: 1206-212X

Safe and Verifiable Design of Concurrent Java Programs

P. H. Welch
Computing Laboratory, University of Kent at Canterbury, CT2 7NF, England
+44 1227 823629; FAX: +44 1227 762811; P.H.Welch@ukc.ac..uk

G. H. Hilderink and A. W. P. Bakkers
Faculty of Electrical Engineering, Control Laboratory, University of Twente, 7500 AE Enschede, The Netherlands
+31-53-4892606, FAX: +31-53-4892223; G.H.Hilderink@el.utwente.nl; A.W.P.Bakkers@el.utwente.nl

G. S. Stiles
Department of Electrical and Computer Engineering, 4120 Old Main Hill, Utah State University, Logan UT 84322-4120;
+1-435-797-2840; FAX: +1-435-797-3054; dyke.stiles@ece.usu.edu

Abstract

The design of concurrent programs has a reputation for being difficult, and thus potentially dangerous in safety-critical real-time and embedded systems. The recent appearance of Java, whilst cleaning up many insecure aspects of OO programming endemic in C++, suffers from a deceptively simple threads model that is an insecure variant of ideas that are over 25 years old [1]. Consequently, we cannot directly exploit a range of new CASE tools -- based upon modern developments in parallel computing theory -- that can verify and check the design of concurrent systems for a variety of dangers such as deadlock and livelock that otherwise plague us during testing and maintenance and, more seriously, cause catastrophic failure in service.

Our approach uses recently developed Java class libraries based on Hoare's Communicating Sequential Processes (CSP); the use of CSP greatly simplifies the design of concurrent systems and, in many cases, a parallel approach often significantly simplifies systems originally approached sequentially. New CSP CASE tools permit designs to be verified against formal specifications and checked for deadlock and livelock. Below we introduce CSP and its implementation in Java and develop a small concurrent application. The formal CSP description of the application is provided, as well as that of an equivalent sequential version. FDR is used to verify the correctness of both implementations, their equivalence, and their freedom from deadlock and livelock.

Keywords: concurrency, multithreading, Java, CSP, formal methods

I. Introduction

Java was originally designed to support the development of embedded systems [e.g., 2], but its overwhelming success in WWW-based applications has overshadowed those intentions. The interest of the embedded and real-time communities in Java has grown steadily, however, and it is difficult to find a recent publication in those areas which does not have one or more articles on Java [e.g., 3, 4, 5]. The initial concerns of speed, memory management, and scheduling - among others - now appear to be surmountable via compilation, dedicated hardware, and improved schedulers [2, 3, 4, 5].

Many real-time and embedded applications are naturally - and most conveniently - designed using concurrency. Unfortunately, the design of concurrent or multithreaded programs has gained the reputation of being extremely difficult and, in many cases, dangerous, due primarily to the possibility of deadlock, livelock, race hazards, or starvation, phenomena that are not encountered in single-threaded programs. Lea, in his text on concurrent Java [6], emphasizes a concern for the apparent difficulty: "Liveness considerations in concurrent software development introduce context dependencies that can make the construction of reusable components harder than in strictly sequential settings." Two approaches he suggests for design sound labor intensive: "*Top-down* (*safety first*): Initially design methods and classes assuming full synchronization (when applicable), and then remove unnecessary synchronization as needed to obtain liveness and efficiency...*Bottom up* (*liveness first*): Initially design methods and classes without concern for synchronization policies, then add them via composites, subclassing, and related layering techniques..." Lea gives extensive design patterns to assist this work, but it is not easy for one to master them and feel confident in that mastery.

Even those intimately connected with the development of Java seem reluctant to employ more than a single thread. In the web-available documentation for the Swing API we find the admonition "If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug. In general, they just aren't necessary for strictly GUI work, such as updating component properties" [7]. Oaks and Wong [8], also associated with Sun, are more enthusiastic about the use of multithreading, but do note that "Deadlock between threads competing for the same set of locks is the hardest problem to solve in any threaded program. It's a hard enough problem, in fact, that we will not solve it – or even attempt to solve it." A bit later they state "Nonetheless, a close examination of the source code is the only option presently available to determine if deadlock is a possibility..." and add that no tools exist for detecting deadlock in Java programs. Developing concurrent Java programs is further complicated by the fact that the synchronization primitives provided with Java are themselves not secure. See, e.g., Brinch Hansen [1].

We feel, based on fifteen years of experience, that the concurrent approach is the best way to design most programs. Done properly (e.g., using CSP), from the start, this method typically results in better understanding of the problem and the solution, and leads to much cleaner implementations. (If poor system support for concurrency requires a sequential implementation, a sequential version can be derived rigorously from the concurrent version.) A tremendous amount of work has been done on and with CSP in recent years, and the present states of the language and the tools offer the Java programmer excellent facilities for the design and analysis of multithreaded programs. Furthermore, Java designs based on the CSP class libraries, and following the CSP paradigm, now can be verified against formal specifications and checked for deadlock and livelock with CASE tools – prior to implementation. Other afflictions, such as race hazards and starvation, can also be handled.

In the following sections we shall first briefly introduce CSP and FDR. This will be followed by the complete development in CSP of a larger example dealing with a virtual channel system: the first concurrent version will deadlock (and shown to by FDR), and the second will be shown to be deadlock- and livelock-free and to precisely implement a CSP specification of the behavior of the system. A summary follows. (A very concise summary of this work was presented at a recent OOPSLA workshop [9].)

II. CSP Notation

Hoare developed CSP [10] after his experience with monitors [11]. CSP is a *calculus* or *algebra* for describing and reasoning about systems of concurrent processes that interact through some type of communication. (Excellent introductions are found in Hoare's delightful book [12] and new texts by Roscoe [13] and Schneider [14].) Much of the detail below follows Roscoe [13].

CSP is designed to handle systems that communicate only via explicit messages. Each process has access to only its own, private, variables and communicates only via explicit messages. The messages themselves are controlled by handshaking - that it, the communication takes place only when both processes are ready; if one become ready before the other, it must wait (or *block*) until the other is ready. The set of all possible communications that a system may engage in is designated as an *alphabet* Σ. The term "communication" may include true messages as well as other events - such as interrupts, clock ticks, etc. Events may be composite - that is, they may be able to be broken down into simpler events - as we could break the sale of an object into an offer, and acceptance, the transfer of the money to the seller, and finally the transfer of the object to the buyer. In general, events may occur between two or more processes. An event may occur only when all processes engaging in that event are ready. When all processes are ready, that event (or perhaps some other event which also happens to be ready) must occur. The basic operator in CSP is *prefixing*; this operator ($\rightarrow$) connects an event to a process; the process cannot do anything until the event occurs. Consider, for example, a data acquisition system that is controlled by a process *ACQUIRE1*. This process does not do anything until it is notified by an event *data_ready;* this fact is indicated in CSP by *data_ready* ® *ACQUIRE1*. Concurrent and distributed systems, e.g., real-time systems and communication systems, frequently have processes that run continuously. This aspect is easily represented in CSP by recursion. Continuing the example, we can define a simple recursive process which repeatedly waits for notification that data is ready and then acquires a sample: $ACQUIRE2 = data\_ready \rightarrow get\_data \rightarrow ACQUIRE2$. The process *ACQUIRE2* engages in event *data_ready*, then event *get_data*, and then repeats. Mutual recursions may be established using two or more processes which call each other.

Current versions of CSP support several different concurrency operators, but we shall require only three below. In the first, *external choice*, the environment can select one of possibly several actions by offering a particular event. In the composite process $(a_0 \rightarrow P_0)$ [] $(a_1 \rightarrow P_1)$, e.g., the environment can force the execution of P_0 by providing a_0 as input, or P_1 by providing a_1. (We use the machine-readable CSP syntax; see, e.g., [13].) Note that this operator offers the choices in parallel, but once one process is selected, it alone will run. The second operator is *sharing* parallelism. Given two processes P_0 and P_1, and a channel c perhaps referenced in both processes, the composite $P0 \; _{\{c\}}\|_{\{c\}} \; P1$ (or in machine-readable format P_0 [| {| c |} |] P_1 for use with FDR) represents a system where P_0 and P_1 run simultaneously and must synchronize on all events involving the channel c. The last operator is *interleaving*; if we have the same two processes, and assume they have alphabets A_0 and A_1, respectively, in the interleaved combination P_0 ||| P_1 the two processes run simultaneously and independently; if an event occurs which appears in both alphabets, one of the processes is chosen nondeterministically to participate and the other does not.

III. Characterization of Processes

Processes may be characterized partly by their *traces*, the sets of the finite sequences of events that a process may legitimately engage in. The traces which *ACQUIRE2* may exhibit are {<>, *<data_ready>*, *<data_ready, get_data>*^{n}, *<data_ready, get_data>*^{n}^*<data_ready>* | *n* ≥ 1}, where <> is the empty sequence and ^ is the concatenation operator.

Given these definitions, characterizations of deadlock and livelock (divergence) are straight-forward. Deadlock occurs when there exists a cycle of committed but unanswered attempts to communicate from process to process: each process is blocked, waiting for the next process in the cycle. Livelock exists when a set of processes gets into an infinite sequence of communications entirely with each other that cannot be interrupted; once in such a state, the process can refuse all further communication from the outside world. Note that, externally, both problems appear the same: both refuse any further external communication. Internally, however, the causes differ significantly and must be thoroughly understood.

Processes may also be characterized by their *failures* and *divergences*. The failures of a process are tuples formed from a trace and the set of events a process may refuse to engage in after that trace. Divergences are those sets of traces after which a process may diverge.

The set of a process's traces specify which sequences of events the process may engage in - if it chooses to do anything at all; the traces thus specify what a process may do - but do not require it to do anything. We impose a requirement that something must be done by specifying what failures the process cannot possess. We would also typically specify that a process may not exhibit any divergences.

We can compare processes by comparing their traces, failures, and divergences. Given some trace specification of a process *P* - which may contain several acceptable traces - we say that process *Q trace-refines P* if the traces of *Q* are included in the traces of *P*. Since trace specifications are weak in the sense that they do not force a process to do anything, we will usually work with failures specification. Given the traces and failures of *P*, process *Q failures-refines P* if the traces of *Q* are included in the traces of *P* and the failures of *Q* are included in *P*. The net result is that *Q* must accept (or reject) every event that *P* does. This concept is extended to *failures/divergences-refinement*, where the same applies to the failures and divergences of the two processes, with the added qualification that the failures are extended to include the set of all tuples formed from the divergences and the corresponding events which are refused due to the divergence. In the example below we test refinement of specifications by failures/divergences.

With these definitions, deadlock can be made precise as the absence of any failures containing the entire alphabet of a process. A process is deterministic if its divergences is empty and, for all trace sequences *t*^<*x*>, (*t*, {*x*}) is not in its failures; the last condition guarantees that the process does not diverge and cannot choose to either accept or reject an event.

Over the past decade a great deal of theoretical and practical work on CSP has been completed. For our purposes, the most interesting work has dealt with the prevention of deadlock. These deal with the problem at roughly two levels: within processes and between processes which themselves have already been shown to be deadlock-free (note that the same basic problem exists at both levels). At the more fundamental level are the works by Roscoe and Dathi [15], Martin and Jassim [16], and Martin et al. [17]. Rules for reliably connecting systems of processes known to be deadlock-free are given in Welch et al. [18] and Martin and Welch [19]. Welch's papers argue forcefully that design of complex systems is far better approached from the concurrent viewpoint; it more closely reflects the operation of the world, leads to easier designs, and to easier verification of those designs. With the CSP approach, the internal states of processes are hidden from each other and interactions among processes are explicitly specified. If the processes themselves are appropriately written, this results in components whose behavior depends only on their interactions with other components. The net result is that the working of a system of interconnected components can be clearly determined. Furthermore, once verified, a complex composition of processes can often be represented by a much simpler process with the same response to external interactions, thus simplifying the verification of systems in which that composite is

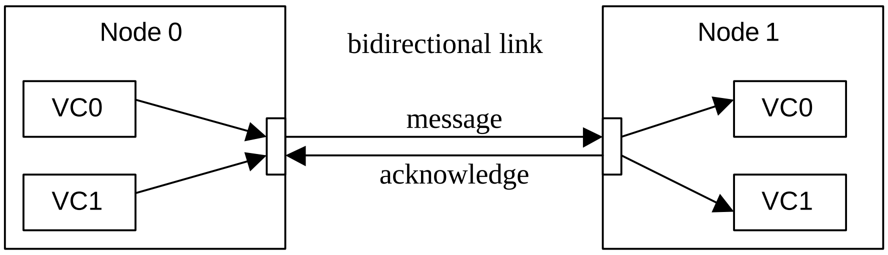

Figure 1. A Virtual Channel System

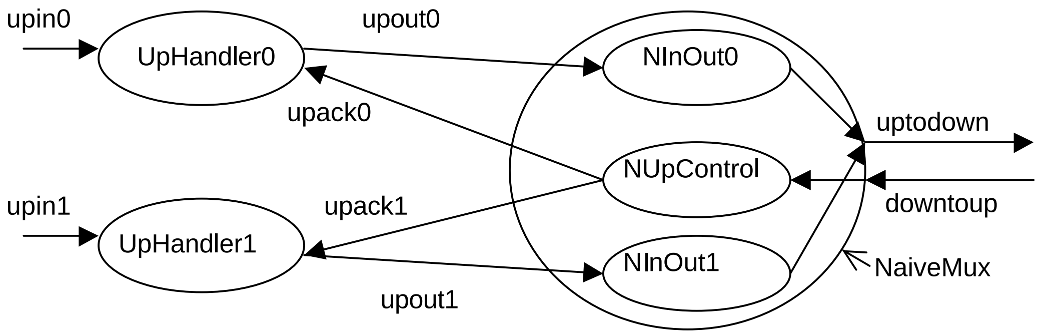

Figure 2. The Node 0 process diagram for the system which deadlocks.

embedded.

These results - which are easily understandable and implemented - form the basis for designing reliable concurrent systems. We may begin with small modules, which may be simple enough that they are clearly free of problems, or may require the application of tools to confirm the absence of deadlock and livelock. Once these modules have been verified, we can then form larger reliable systems if we follow the appropriate rules when connecting them together (e.g., [19]). This process is not all that different from constructing large digital logic circuits from small, proven standardized components.

The programming language occam is based on CSP and is now available on a number of platforms (e.g., [20, 21]). Several versions of C have been available that provide CSP-like constructs (e.g., [22]). One of the most interesting developments has been the appearance of class libraries for Java which provide many CSP features. These developments have taken place at Twente University in the Netherlands [23, 24, 25] and at the University of Kent at Canterbury [26, 27, 28]. With these packages we can now easily develop multithreaded Java programs that adhere to CSP standards. Multithreaded GUI programming can, after all, be made simple.

The other significant recent development is the appearance of commercial tools which can check CSP programs for deadlock, livelock, nondeterminacy, and correct implementations of specifications - also expressed in CSP. The FDR package from Formal Systems (Europe) Ltd. [29] provides an interactive (or batch) system for testing CSP programs for deadlock, livelock, determinism, and refinement. Refinement can be based on traces, failures, and failures/divergences. FDR also provide a tool called ProBE (Process Behavior Explorer) which allows the user to step through the events of a process; this is particularly useful for debugging. Once the CSP scripts are written a simple point & click returns the results of the analysis. And, as we see next, development of the scripts is often very straightforward.

IV. Example

We shall model a simple communications system based on virtual channels. We assume that we have two nodes which must communicate over a single link providing communication in both directions. To manage multiple conversations, we provide virtual channels over the link. Deadlock of the entire system is prevented by providing a separate buffer for each virtual channel on each node and appropriate routing. A handler on the input of each virtual channel insures that a packet is not received unless the buffer for that channel is empty. The system is diagrammed in Figure 1.

Data flows through the system in packets from left to right. A packet cannot be sent along a virtual channel until the previous packet has been acknowledged. The routing system must thus handle flow in both directions: data packets from left to right, and acknowledgements from right to left. At any time the system must be ready to accept a packet from upstream (Node 0) on a channel which has no outstanding acknowledgements, and also ready to accept an acknowledgement from downstream for an as yet unacknowledged packet. The fact that items may be ready to move in both directions at the same time offers ample opportunity for deadlock if the system is not designed correctly.

We begin with a design that deadlocks (the complete script is at [28]). The design is most easily handled by breaking it into small components and then combining them in parallel. We start on Node 0 with an input handler for each virtual channel; these accept inputs from their respective sources, pass them on to a multiplexer which passes them over the link, and wait for acknowledgements; they then repeat. The Node 0 process diagram is shown in Figure 2.

The diagram makes the relationship of the processes and the channels clear, and developing the CSP script for each process is simple; each consists of only a few lines. The entire process is formed from the parallel combinations of the components. The machine-readable CSP script for one of the handlers is

```
UpHandler0 =
    upin0?x:RawData -> upout0!x ->
    upack0?y:InternalAcks -> UpHandler0
```

This is a process which accepts over channel `upin0` an item `x` which is of type `RawData`, sends it out over channel `upout0` to the mux, waits to accept an acknowledgement of type `InternalAcks` over channel `upack0`, and then repeats. The code for the other input handler is similar. The mux code is composed of three components, two of which accept data from an input handler, format it, and then pass it over the link, and a third which is an acknowledgement controller. These components are combined using the external choice operator; when an input acceptable by any of the three processes appears, the corresponding process runs and then restarts the mux. (This is not a particularly good implementation - but it does test the deadlock checker.) The first of the components is

```
 NInOut0 = upout0?x:RawData ->
          uptodown!0.x -> NaiveMux
```

This component accepts the raw data, appends a "0" to it to indicate virtual channel 1, sends it over the output link channel `uptodown` to the other node. The acknowledgement controller is

```
NUpControl = downtoup?x:LinkAcks ->
          if x == 0
             then upack0!0 -> NaiveMux
             else upack1!0 -> NaiveMux
```

This process accepts an input from the downstream node of type `LinkAcks` (which identifies the virtual channel to be acknowledged), sends the acknowledgement signal (the value 0) to the appropriate input handler, and restarts the mux. The mux is specified by

```
NaiveMux =
     NInOut0 [] NInOut1 [] NUpControl
```

and the entire Node 1 process by

```
NaiveUp = ((((UpHandler0
          [| {| upout0, upack0 |} |]
          NaiveMux)
          \ {| upout0, upack0 |})
          [| {| upout1, upack1 |} |]
          UpHandler1)
          \ {| upout1, upack1 |})
```

`UpHandler0` and NaiveMux share all communications over channel `upout0` and `upack0`, and this combination shares with `UpHandler1` over `upout1` and `upack1`. The \ {| channel |} notation indicates that communications over the named channels are treated as internal and are not visible to the outside world. In this process the only actions available to the environment are thus those via the two input channels `upin0` and `upin1` and the output channel `uptodown`.

The downstream (Node 1) processes are similar. A demux on the channel from Node 0 is formed from the external choice between one process that accepts inputs over the channel and a second that accepts acknowledgements from the destination buffers and passes them upstream over the channel `downtoup`. Again, this is not a good design, since this process will not be able to accept data moving in both directions at the same time - and will deadlock. The demux passes the incoming data on to `DownHandler0` or `DownHandler1`, which accept the data and respond with acknowledgements that work their way back upstream. The entire downstream process is formed by the sharing combination of these processes, and the internal channels are again hidden. The complete system is formed from the sharing combination of the upstream and downstream processes, with the channels between the two nodes hidden.

A little analysis of this system will show that it will deadlock when both nodes are trying to send data to the other. This will occur when one input has been passed to Node 1, and a second input arrives before the acknowledgement of the first makes it back to Node 0. At this point Node 0 is trying to send its data downstream, and Node 1 is trying to send its acknowledgement upstream. Since neither is listening, we have a cycle of ungranted requests and the system deadlocks. When the complete CSP script is run through the deadlock checker, FDR reports the occurrence of deadlock and provides the sequence of events that lead to it.

The deadlock arises *not* from the parallelism in the design but from there *not being enough* parallelism in the design! `NaiveMux` is essentially a serial process operating its channels one at a time. We need to multiplex data on `uptodown` and demultiplex acknowledgements from the `downtoup` in parallel. Figure 3 shows the process structure for the revised Node 1 system.

A similar modification appears on Node 1. The entire CSP (and Java) listings are available at [30]. We again emphasize that each of the individual CSP processes is typically only a few lines, and that the composite processes are also only a few lines. Once the diagrams have been created and the processes and channels labeled, a student with little training can develop the CSP scripts and Java fairly quickly.

When this version is run through FDR the tool reports that it is indeed deadlock free - and is free of divergence. The next step is to verify that the system does precisely what we want. When viewed from outside, the system should simply pass data unchanged from the inputs `upin0` and `upin1` to the outputs `downout0` and

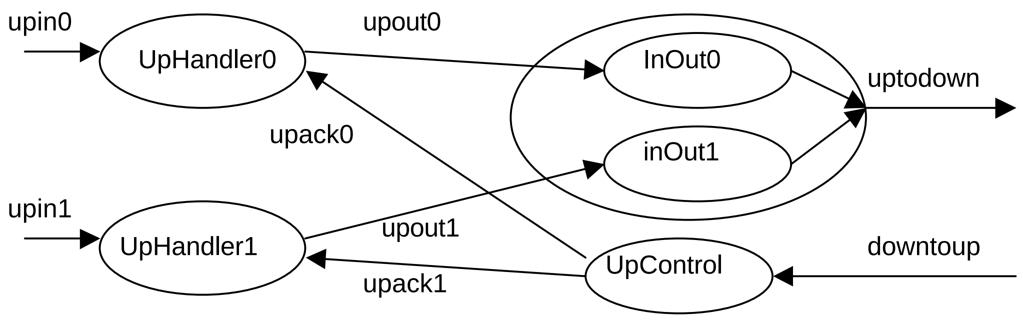

Figure 3. The Node 0 process diagram for the deadlock-free system.

`downout1`, respectively. The connections should thus be equivalent to simple independent copy operations, operating in parallel over dual links:

```
Copy0 = upin0?x:RawData ->
        downout0!x -> Copy0

Copy1 = upin1?x:RawData ->
        downout1!x -> Copy1
```

These processes are then interleaved (implying that they run independently) to form the specification for the system:

```
   DCopy = Copy0 ||| Copy1
```

FDR is then used to verify that the system refines `DCopy`; the verification is a success, and we know that the system will faithfully pass the data.

This last feature is extremely important. We have verified that the data are simply replicated from the input to the output, in spite of the modifications undergone as it is moved by the routing system. This same approach can be used to verify the operations of far more complicated protocols that might provide error correction over a noisy channel, compression and decompression, or encryption and decryption. CSP has, in fact, been of considerable use in modeling a variety of communications systems (see, e.g., [13]).

Once we have the system designed, checked, and verified, it can easily be implemented in Java using the CSP constructs. The CSP scripts for both concurrent versions, plus the complete Java implementation, can be found at [30]. (A number of additional runnable Java/CSP examples are available presently at the sites listed in the References [25, 26].)

Since current Java run-time environments may not provide context switching fast enough to support some applications, we may wish to have sequential rather than concurrent implementations. Fortunately concurrent CSP processes can be easily transformed - retaining their correctness - to sequential versions; this can be done automatically with a tool such as Mathematica. We have implemented a sequential version of the deadlock-free system presented above, and have used FDR to verify that it also satisfies the same specifications. The CSP script for this version is also at [30].

Currently available occam run-time environments provide an existence proof that CSP primitives can be implemented with overheads well under one microsecond. The nearly-released `HotSpot' ™ Java run-time environment from Sun might, hopefully, approach these levels of performance for its multithreading and these will automatically apply to Java CSP class libraries.

IV. Summary

We described a method for creating reliable, safe, and provably correct multithreaded implementations in Java. This approach is based upon CSP and uses CSP class libraries developed for Java. CSP designs can be checked with commercially available CASE tools to verify freedom from deadlock and livelock and compliance with formal specifications. The tools can also check the equivalence of various implementations, and can verify that sequential versions provide the same performance. The CSP approach itself, which emphasizes the analysis of problems in terms of concurrent processes interacting only through explicit messages, often leads to better understanding of the problems and cleaner designs. We have found that students respond very quickly to this approach and can develop fairly complex applications after little training. By restricting our concurrent Java programs to message passing we avoid the problems inherent in the use of lower-level synchronization primitives, and gain the formal support, safety, and twenty years of experience offered by CSP - a reasonable trade by any standards.

V. References

[1] Brinch Hansen, Per, Java's insecure parallelism, *ACM SIGPLAN Notices* 34(4), pp. 38-45, April 1999.
[2] Nilsen, Kevin, Adding real-time capabilities to Java, *Comm. ACM* 41(6), pp. 49-56, June, 1998.
[3] Dibble, Peter, The reality of real-time Java, *Computer Design*, pp. 70-76, Aug. 1998.

[4] Hedenstedt, Joakim, Java in embedded systems: two approaches, *Computer Design*, p. 74, Aug. 1998.
[5] Ivanovic, Vladimir, From desktop Java to embedded Java, *Real Time Computing*, pp. 147-147.
[6] Lea, Doug, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, Reading, MA, 1997.
[7] Muller, Hans, and Kathy Walrath, Threads and Swing, http://java.sun.com:80/products/jfc/swingdoc-archive/threads.html, April 3, 1998.
[8] Oaks, Scott, and Henry Wong, *Java Thread*s, O'Reilly, Sebastopol, CA, 1997.
[9] Stiles, G. S., Safe and verifiable design of multithreaded Java programs via CSP and FDR, *Proc. Workshop on Formal Underpinnings of Java, OOPSLA '98,* Oct. 1998, to appear in Princeton University Technical Report
[10] Hoare, C. A., Communicating sequential processes, *CACM*, 21(8), pp. 666-677, August 1978.
[11] Hoare, C. A., Monitors: an operating system structuring concept, *CACM*, 17(10), pp. 549-557, October 1974.
[12] Hoare, C. A., *Communicating Sequential Processes*, Prentice-Hall, London, 1985.
[13] Roscoe, A. W., *The Theory and Practice of Concurrency,* Prentice-Hall, London, 1998.
[14] Schneider, Steve, *Concurrency and Time*, Wiley, in press.
[15] Roscoe, A. W., and N. Dathi, *The Pursuit of Deadlock Freedom*, Technical Monograph PRG-57, Oxford University Computing Laboratory, 1986.
[16] Martin, J., I. East, and S. Jassim, Design rules for deadlock freedom, *Transputer Comm*. 2(3), pp. 121-133, September 1994.
[17] Martin, J. M. R., and S. A. Jassim, A tool for proving deadlock freedom, in *Proc. WoTUG20: Parallel Programming and Java*, ed. A. Bakkers, IOS Press, Amsterdam, pp.1-16, April 1997.
[18] Welch, P. H., G. R. R. Justo, and C. Willcock, High-level paradigms for deadlock-free high-performance systems, in *Transputer Applications and Systems '93*, ed. Grebe et al., IOS Press, Amsterdam, pp. 981-1004, September 1993.
[19] Martin, J. M. R., and P. H. Welch, A design strategy for deadlock-free concurrent systems, *Transputer Communications* 3(4), pp. 215-232, October, 1996.
[20] Galletly, John, *occam 2 – including occam 2.1*, UCL Press, London, 1996.
[21] occam-for-all Team, http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/index.html
[22] ______, http://www.alphadata.co.uk/softhome.htm
[23] Hilderink, G., J. Broenink, W. Vervoort, and A. Bakkers, Communicating Java threads, in *Proc. WoTUG 20: Parallel Programming and Java*, ed. A. Bakkers, pp. 48-76, IOS Press, Amsterdam, April 1997.
[24] Hilderink, Gerald, Communicating Java threads reference manual, in *Proc. WoTUG 20: Parallel Programming and Java*, pp. 283-325, IOS Press, Amsterdam, April 1997.
[25] JavaPP, http://www.rt.el.utwente.nl/javapp/, July 21, 1998.
[26] Java Communicating Sequential Processes (JCSP), http://www.hensa.ac.uk/parallel/languages/java/jcsp/
[27] Welch, P. H., Java threads in the light of occam/CSP, in *Proc. WoTUG 20: Parallel Programming and Java*, ed. A. Bakkers, pp. XX-YY, IOS Press, Amsterdam, April 1997.
[28] Welch, P. H., Parallel and distributed computing in education, in *VECPAR'98 (Third International Conference on Vector and Parallel Processing) - Selected Papers*, Jose M. L. M. Palma, Jack Dongarra and Vicente Hernandez (editors), Lecture Notes in Computer Science, Springer-Verlag (to appear), Porto, Portugal, June 1998.
[29] Formal Systems (Europe) Ltd.: http://www.formal.demon.co.uk/
[30] _____, http://multi.ece.usu.edu/projects/parsys/examples.html#Java/CSP