Safe and Verifiable Design of Multithreaded Java Programs with CSP and FDR

G. S. Stiles
Electrical and Computer Engineering
Utah State University
Logan UT 84322-4120 USA
+1 435 797 2840
dyke@stiles.ece.usu.edu

ABSTRACT

Java may used to develop code for life-, safety-, and mission-critical embedded systems; multithreaded applications must be free from deadlock and livelock. These problems can be eliminated by basing designs on the structures and formalisms of Hoare's Communicating Sequential Processes (CSP). Java class libraries supporting CSP constructs now allow programmers to take advantage of these methods. CASE tools are available to automatically check CSP designs for deadlock and livelock and to validate implementations against specifications.

KEYWORDS

Java, CSP, deadlock, livelock, verification, CASE

INTRODUCTION

The use of Java in life-, safety-, and mission-critical embedded systems requires that the greatest care be given to avoid deadlock and livelock in multithreaded implementations. The recent appearance of Java class libraries with CSP constructs permits the design of Java programs which can be proven free of deadlock and livelock and to correctly implement specifications stated in CSP; CASE tools are available to automate these checks. Furthermore, based on twelve years of experience, we feel that the multithreaded or concurrent approach is the best way to design most programs. This method typically results in better understanding of the problem and the solution, and leads to much cleaner implementations. Below we briefly introduce CSP and the FDR CASE tools. This will be followed by the examination of a small example expressed in CSP and its analysis and verification with FDR.

CSP AND FDR

Hoare developed CSP [4] after his experience with monitors [3]. CSP is a tool for describing and reasoning about systems of concurrent processes which interact through some type of communication. Introductions can be found in Hoare's book [5] and texts by Roscoe [10] and Schneider [11]. Below we follow Roscoe [10].

CSP is designed to handle systems which communicate only via explicit messages; each process has access only to its own variables. Messages are controlled by handshaking: communication takes place only when both processes are ready; if one becomes ready before the other, it must wait. The set of all possible communications (*events*) which a system may engage in is designated as its *alphabet* Σ. Events may include true messages as well as such things as interrupts, clock ticks, etc.

The basic operation in CSP is *prefixing* ($\rightarrow$), which connects an event to a process; the process cannot do anything until the event occurs. Consider, e.g., a data acquisition system that is controlled by a process *ACQUIRE1*. This process does nothing until it is notified by an event *data_ready*. This is indicated in CSP by $data\_ready \rightarrow ACQUIRE1$. A perpetual version can be defined recursively: $ACQUIRE2 = data\_ready \rightarrow get\_data \rightarrow ACQUIRE2$. A process can be characterized by its *traces*, which is the set of the finite sequences of events in which the process may legitimately engage.

Deadlock occurs when there exists a cycle of ungranted requests to communicate from process to process. Livelock (divergence) exists when there is an internal cycle of communications between processes which can occur with no interaction from the outside world; once in such a state, such a system can refuse all further external communication. 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 livelock. Traces specify what a process may do (safety) - but do not require it to do anything. We impose a requirement that something must be done (liveness) by specifying what failures the process cannot exhibit. We also specify that a process not exhibit any divergences.

We compare processes by comparing their traces, failures, and divergences. Given a trace specification of a process *P* - which may contain several acceptable traces - we say that process *Q trace-refines P* if *Q* performs at least one trace of *P*. Since trace specifications do not force a process to do anything, we usually work with failures specifications. 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*, 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. (Below we test refinement of specifications by failures/divergences.) Freedom from deadlock is now defined as the absence of any failures containing the entire alphabet of a process.

Over the past decade a great deal of theoretical and practical work on CSP has been completed. For the purpose of designing real applications, the most interesting work has dealt with the prevention of deadlock at roughly two levels: within processes and between processes which themselves have already been shown to be deadlock-free. At the more fundamental level are the works by Roscoe and Dathi [9], Martin and Jassim [7], and Martin et al. [6]. Rules for reliably connecting systems of processes known to be deadlock-free are given in Welch et al. [12] and Martin and Welch [8]; these present a number of configurations, applicable to common applications, which are shown to be deadlock-free. Welch's papers argue convincingly that design of complex systems is far better approached from the CSP viewpoint; it more closely reflects the operation of the world, leads to easier designs, and to easier verification of those designs. This is largely due to the compositional nature of CSP systems: the behavior depends only on the interactions between processes, and not on their internal states.

Recently class libraries providing many CSP features for Java have appeared. The libraries were developed at Twente University in the Netherlands [1, 2, 13] and the University of Kent at Canterbury [14]. With these packages we can easily design multithreaded Java programs which adhere to the CSP standards.

The FDR package from Formal Systems (Europe) Ltd. [15] offers interactive tools for testing CSP programs for deadlock, livelock, determinism, and refinement of CSP specifications. Refinement is based on traces, failures, or failures/divergences.

EXAMPLE

We model two nodes (of a larger system) which communicate over a single link with two virtual channels. Data flow through the system in packets from Node 1 to Node 2. A packet cannot be sent along a virtual channel until the previous packet has been acknowledged. At times Node 1 must be ready to accept a packet from upstream on a channel which has no outstanding acknowledgements, and also ready to accept an acknowledgement from downstream. The fact that items may be ready to move in both directions at the same time threatens deadlock if the system is not designed correctly.

We begin with a design which can deadlock. We start on Node 1 with an input handler for each virtual channel; these handlers accept inputs from their respective buffers, pass them on to a multiplexer which transmits them over the link, and wait for acknowledgements; they then repeat.

Developing the CSP script for the processes is simple; each consists of only a few lines. The machine-readable CSP [10] for one of the input handlers is

```
UpHandler1 = upin1 ? x:RawData -> upout1 ! x -> upack1 ? y:InternalAcks ->
             UpHandler1
```

(Full CSP script and Java code are at [16]). This process accepts over channel `upin1` an item `x` which is of type `RawData`, sends it out over `upout1` to the mux, waits to accept an acknowledgement of type `InternalAcks` over `upack1`, and repeats. The mux code (`NaiveMux`) is composed of three components: two (`NInOut1` and `NInOut2`) accept data from input handlers, format it, and then pass it over the link; a third (`NupControl`) accepts an acknowledgement from the downstream node, sends the acknowledgement (e.g., `upack1`) to the appropriate input handler, and restarts the mux.

The three components are combined to form `NaiveMux` using the *external choice* parallel operator `[]`: when an input acceptable by any of the three processes appears, the corresponding process runs and then restarts `NaiveMux`:

```
NaiveMux =  NInOut1 [] NInOut2 [] NupControl
```

The entire Node 1 process is

```
NaiveUp =    ((((UpHandler1 [| {| upout1 |} |] NaiveMux) \ {| upout1 |})
             [|{| upout2 |} |] UpHandler2) \ {| upout2 |}) \ {| upack1, upack2 |}
```

`UpHandler1` and `NaiveMux` run in *shared parallel*, sharing all communications over channel `upout1`, and this combination shares with `UpHandler2` over `upout2`. The `\{| channel |}` (*hiding*) operator indicates that communications over the named channel are treated as internal and are not visible to the outside world. The only actions available to the environment are thus those via the two input channels `upin1` and `upin2` and the output (link) channel `uptodown`. The Node 2 processes are similar and eventually send the data out over channels `downout1` and `downout2`.

This system deadlocks if each node tries to send to the other at the same time. This occurs when one input has been passed to Node 2, and a second arrives before acknowledgement of the first makes it back to Node 1. When the CSP script is run through the deadlock checker, FDR reports the possibility of deadlock and provides the sequence of events which lead to it.

Deadlock can be prevented by providing processes which monitor the flow in both directions at all times. When this version is run through FDR the tool reports that it is indeed deadlock free - and free of divergence.

The next step is to verify that the system does what we want. When viewed from outside, the system should simply pass data unchanged from the inputs `upin1` and `upin2` to the outputs `downout1` and `downout2`, respectively. The system should thus be equivalent to two simple independent copy operations of the form

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

These processes are combined with the *interleaved parallel* operator (implying that they run independently since there are no common events) to form the specification for the system:

```
DCopy =     Copy1 ||| Copy2
```

FDR is used to verify that the system refines `DCopy`; the verification is a success, and we know that the system will faithfully pass the data. (`DCopy` can also be shown to refine the system and the two are thus equivalent.)

This last feature is important. We have verified that the data are 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 more complicated protocols which might provide error correction over a noisy channel, compression and decompression, or encryption and decryption. CSP has been of considerable use in modeling a variety of communications systems (e.g., [10]).

Once the CSP is designed, checked, and verified, it is easily be implemented in Java with the CSP constructs. If the Java run-time system does not provide context switching fast enough to support some applications, we may wish to have sequential rather than concurrent implementations. Concurrent CSP processes can be easily transformed - retaining their correctness - to sequential versions. A sequential version has been shown by FDR to be equivalent to the deadlock-free system above.

SUMMARY

We have described a method for designing safe and verifiable multithreaded programs based upon the formalisms of CSP and the CSP Java class libraries. The designs can be checked with CASE tools to confirm their freedom from deadlock and livelock and their compliance with formal specifications. The tools can check the equivalence of various implementations and verify that sequential versions provide the same performance.

ACKNOWLEDGEMENTS

Peter Welch and Andy Bakkers have been endless sources of inspiration and clear thinking. Thanks also to Gerald Hilderink for developing the CSP class library used in this project; Paul Austin created the Kent CSP class libraries.

REFERENCES

1. Hilderink, Gerald, Jan Broenink, Wiek Vervoort, and Andre Bakkers, Communicating Java threads, in Proc. WoTUG 20: Parallel Programming and Java, ed. A. Bakkers, IOS Press, Amsterdam, April 1997, pp. 48-76.
2. Hilderink, Gerald, Communicating Java threads reference manual, in Proc. WoTUG 20: Parallel Programming and Java, IOS Press, Amsterdam, April 1997, pp. 283-325.
3. Hoare, C. A., Monitors: an operating system structuring concept, *CACM*, 17(10), October 1974, pp. 549-557.
4. Hoare, C. A., Communicating sequential processes, *CACM*, 21(8), August 1978, pp. 666-677.
5. Hoare, C. A., *Communicating Sequential Processes*, Prentice-Hall, London, 1985.
6. Martin, J., I. East, and S. Jassim, Design rules for deadlock freedom, Transputer Communications 2(3), September 1994, pp. 121-133.
7. 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, April 1997, pp.1-16.
8. Martin, J. M. R., and P. H. Welch, A design strategy for deadlock-free concurrent systems, Transputer Communications 3(4), October, 1996, pp. 215-232.
9. Roscoe, A. W., and N. Dathi, The Pursuit of Deadlock Freedom, Technical Monograph PRG-57, Oxford University Computing Laboratory, 1986.
10. Roscoe, A. W., *The Theory and Practice of Concurrency*, Prentice-Hall, London, 1998.
11. Schneider, Steve, *Concurrency and Time*, in press.
12. 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. R. Grebe et al., IOS Press, Amsterdam, September 1993, pp. 981-1004.
13. _____, JavaPP, http://www.rt.el.utwente.nl/javapp/, July 21, 1998.
14. _____, Java Communicating Sequential Processes, http://www.hensa.ac.uk/parallel/languages/java/jcsp
15. _____, Formal Systems (Europe) Ltd.: http://www.formal.demon.co.uk/
16. _____, http://multi.ece.usu.edu/projects/parsys/examples.html#Java/CSP