4yp

Thomas Aston

April 15, 2024

1 Introduction

In a world reliant on computer systems, the correctness of those systems are vital. Indeed, simple programming errors can lead to major incidents; examples of these include an automated trader losing $460 million[4] and the inaugural Ariane 5 flight breaking up after launch due to an overflow error[3].

In order to achieve better performance, concurrency can often be introduced to improve the performance programs or systems - especially those with semi-independent tasks or components. Concurrent systems, be this on a single computer or distributed across a network, achieve these performance improvements at the cost of additional complexity as the design now needs to consider the interactions and exchange of data between threads. Each of different possible interactions between threads could potentially lead to a *race condition* in a poorly designed systems; this is where two non-independent actions can occur in an order which produces an incorrect or unwanted outcome. Race conditions can be very damaging in practice - the Therac-25 radiation therapy machine killed three of its patients by radiation overexposure as a result of a race condition[9]. This was caused by the operator quickly changing from the high radiation beam mode to select the lower radiation beam instead; the race condition

resulted in the machine erroneously still using the high radiation beam instead. This highlights the importance of thorough system validation; had more through system validation been completed, these deaths would have likely been avoided[1].

There are two main approaches to developing correct software: testing and verification. Though thorough unit testing can be effective in minimising software bugs[6], this form of testing is significantly less effective in concurrent contexts. Testing can only establish the presence of a bug, not the absence of any; this can be somewhat addressed by writing exhaustive tests to cover every possible edge case, however this imposes severe restrictions on the complexity of such systems. Writing exhaustive tests is near impossible for sufficiently complex concurrent systems. This is predominantly due to the sheer number of different interactions between independent threads: considering all possible interactions and testing them effectively are both challenging tasks. *Linearizability* testing is an effective alternative approach, although this this relies on the random testing of edge cases; clearly this is not exhaustive either[8].

By contrast, formal verification can be used to show that systems satisfy some desired properties[2]. This, however, is a complex process and it is often impossible to model check the complete behaviour of a large system simply due to the size of the resulting state space. We instead focus on modelling the concurrent interactions between threads; if the interactions between threads behave as expected then system validation can be reduced to checking the behaviour of the individual, sequential, threads. Focusing on these interactions allows us to decrease the size of a system to the extent where it is now feasible to model check these behaviours.

We choose to use the process algebra Communicating Sequential Processes (CSP) as our tool for modelling these interactions; CSP is a language for describing processes that can interact both with their environment and other parallel processes. As CSP is covered in the Part B Concurrency course[5] we assume familiarity with CSP as

described in [11].

1.1 Contributions

The contributions of this project are as follows:

- We produce a CSP model of a 2-thread synchronisation object and how they can be organised into an arbitrary n-thread barrier synchronisation object. We then use the CSP model to prove the correctness of the barrier synchronisation object.
- We model both the standard JVM monitor and the SCL[7] implementation of a monitor, proving that the SCL monitor provides the same mutual exclusion and correctness properties as the JVM monitor but without the same unwanted behaviours.
- We examine a variety of locks from [10] and a number of desirable properties of locks, using CSP refinement checks to determine which locks satisfy. We also discuss the feasibility of modelling infinite properties using CSP.

References

[1] Sara Baase. *A Gift of Fire, 4th Edition*. 2012. URL: https://www.philadelphia.edu.jo/academics/lalqoran/uploads/A-Gift-of-Fire-4thEd-2012.pdf.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] Independent Inquiry Board. *ARIANE 5: Flight 501 Failure*. Report. 1996. URL: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html.

[4] Henrico Dolfing. *Case Study 4: The $440 Million Software Error at Knight Capital*. 2019. URL: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html.

[5] Bill Roscoe Gavin Lowe. *Concurrency Lecture Slides*. 2020. URL: https://www.cs.ox.ac.uk/teaching/materials19-20/concurrency/.

[6] Gunnar Kudrjavets Laurie Williams and Nachiappan Nagappan. *On the Effectiveness of Unit Test Automation at Microsoft*. URL: https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf.

[7] Gavin Lowe. *Concurrent Programming Lecture Slides*. 2022. URL: https://www.cs.ox.ac.uk/teaching/materials22-23/concurrentprogramming/.

[8] Gavin Lowe. "Testing for linearizability". In: *Concurrency and Computation: Practice and Experience* 29.4 (2017), e3928. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3928.

[9] Jamie Lynch. *The Worst Computer Bugs in History: Race conditions in Therac-25*. Blog. 2017. URL: https://www.bugsnag.com/blog/bug-day-race-condition-therac-25/.

[10] Hanno Nickau and Gavin Lowe. *Concurrent Algorithms and Data Structures Lecture Notes*. 2023. URL: https://www.cs.ox.ac.uk/teaching/courses/2023-2024/cads/.

[11] Andrew (Bill) Roscoe. *Understanding Concurrent Systems*. Springer, 2010.