



Discovering and correcting a deadlock in a channel implementation

Gavin Lowe¹

¹Department of Computer Science, University of Oxford, Oxford, UK.

Abstract. We investigate the cause of a deadlock in the implementation of a channel in a message-passing concurrency API. We model the channel implementation using the process algebra CSP, and then use the model checker FDR to find the cause of the deadlock. The bug is rather subtle, and arguably infeasible to spot by hand. We then propose a straightforward fix to the bug, and use CSP and FDR to verify this fix.

Keywords: Message-passing concurrency, Channel implementation, Model checking, CSP.

1. Introduction

Communicating Scala Objects (CSO) [Suf08] is a library of message-passing concurrency primitives for the Scala programming language, based on CSP [Ros10]. It has been used for teaching concurrent programming at Oxford for around twelve years. I was therefore surprised to find a deadlock involving a CSO channel.

This paper describes how I discovered the cause of the deadlock, by modelling the channel in CSP [Ros10], and then analysing it using the model checker FDR [GRABR14]. The bug is rather subtle: I was certainly unable to spot it by examining the code; even when one has seen the error, the code does not look incorrect; and the trace leading to the deadlock contains 37 events, making it possibly the longest error trace I have come across. It seems that the deadlock occurs only under certain use cases, on specific architectures: this helps to explain why it was not discovered earlier. Nevertheless, the bug has a straightforward fix, which can be verified using CSP and FDR.

I consider the main contribution of this paper to be the discovery and correction of the bug. In addition, the paper serves as a reminder of the difficulty of low-level concurrent programming, and the benefits to be gained from a formal analysis. Further, the paper provides a tutorial in the use of CSP for such analyses.

I found the bug within the context of a bag-of-tasks pattern: a single controller process distributes computational tasks to worker processes, along a shared channel. In particular, I tested the program on systems with a random number of workers and with randomly sized computational tasks. This testing produced the deadlock, typically within a couple of minutes. It is my experience that testing using randomness in this way, rather than sticking to expected use cases, can help to discover corner cases containing bugs. By contrast, with the fixed implementation, no deadlock was found despite several days of testing.

The rest of the paper is structured as follows. In Sect. 2, we describe the implementation of the CSO channel. In Sect. 3 we give a brief overview of the fragment of CSP that we use subsequently. In Sect. 4 we present the CSP model of the channel, and describe our analysis, leading to the cause of the deadlock. In Sect. 5 we correct the implementation, and describe the verification of that fix. We sum up in Sect. 6.

Correspondence and offprint requests to: G. Lowe, E-mail: gavin.lowe@cs.ox.ac.uk

```

1 class OneOne[T]{
2   private val reader, writer = new AtomicReference[Thread]
3   private val full = new AtomicBoolean(false)
4   private var buffer: T = _
5
6   def !(value: T) = {
7     writer.set(Thread.currentThread); buffer = value; full.set(true)
8     LockSupport.unpark(reader.get)
9     while(full.get) LockSupport.park(this)
10    writer.set(null)
11  }
12
13  def ?(): T = {
14    reader.set(Thread.currentThread)
15    while(!full.get) LockSupport.park(this)
16    val result = buffer; full.set(false)
17    LockSupport.unpark(writer.getAndSet(null))
18    reader.set(null); result
19  }
20 }

```

Fig. 1. The implementation of a `OneOne` channel

Related work CSP has been used to analyse concurrent programs on a number of previous occasions. Welch and Martin [WM00] present a model of Java multi-threading (in particular, monitors), including a model of a channel within their own concurrency API. I [Low11] use CSP to derive and verify a generalisation of the alternation construct within CSO. Lawrence [Law05] describes the use of CSP and FDR in an industrial setting, for the analysis of a system for connection pooling. Mota and Sampaio [MS01] analyse a subset of the control system of a satellite, modelled in CSP-Z. CSP and FDR have been widely used to analyse security protocols (e.g. [Low96]). Hopkins and Roscoe [RH07] describe a compiler for compiling from a simple shared-variable language into CSP.

Various approaches have been used to detect deadlocks or to prove deadlock freedom. Roscoe et al. [RD87, BR91] study deadlocks based on cycles of ungranted requests, i.e. when there is a cycle $p_1, p_2, \dots, p_n, p_1$ of processes, where each process is trying to communicate with the next, but that next process is refusing that communication. Martin [Mar96] extends these ideas, seeking to eliminate false positives via local analysis. Antonino et al. [AGRR16a, AGRR16b, AGRR17] use approximation techniques, combined with various techniques designed to capture invariants, in order to strengthen the approach further.

2. The implementation of `OneOne` channels

`OneOne` channels in CSO provide communication between one sender and one receiver. The class `OneOne` is polymorphic in the type of data sent: channels of type `OneOne[T]` pass data of type `T`. If `c` is such a channel and `v` is of type `T`, then the operation `c!v` sends `v` on `c`; the expression `c?()` receives a value on `c`, and returns that value. This communication is *synchronous*: the send must wait until a receive is available.

While `OneOne` channels must be used by a single sender and receiver, CSO provides other types of channels that may be used by multiple senders and receivers. These are implemented as a wrapper around a `OneOne` channel, ensuring that at most one sender and one receiver use the `OneOne` channel concurrently.

The implementation of a `OneOne` channel (in Scala) is sketched in Fig. 1. In the interests of brevity, we have simplified the implementation: we have removed details concerning channels being closed (which means they can no longer be used), and have removed some other operations using channels.

The implementation uses the Java `LockSupport` class [Orac].¹ This class provides a means for threads to be suspended and resumed. It provides two operations, `park` and `unpark`, which act like a wait/signal mechanism. If a thread `t` calls `LockSupport.park(b)` (where `b` is the synchronization object responsible for this thread parking, normally the current object), then it will normally suspend and wait until another thread calls `LockSupport.unpark(t)`. However, `t` may spuriously wake up without any corresponding `unpark`; hence calls to `park` are normally guarded

¹ Scala runs on the Java Virtual Machine, so Scala code may freely call Java code.

within a **while** loop, to recheck the relevant condition. In addition, the mechanism used is permit-based: if a thread calls `unpark(t)` and `t` is not yet parked, a “permit” is stored; when `t` does call `park`, it immediately resumes, consuming the permit.

Line 1 of Fig. 1 defines the class `OneOne` to be polymorphic in the type `T`; this is the type of data passed by the channel.

Each `OneOne` channel has four variables. The variables `reader` and `writer` store the current reading and writing threads (if any). These are `AtomicReferences` [Orab]: these ensure a sequentially consistent memory, and also provide various atomic operations, including `getAndSet` which atomically updates the variable and returns the previous value. The variable `buffer` stores the current (or last) value sent.² The variable `full` records whether `buffer` holds a value that has not yet been claimed by a `?` operation. This is an `AtomicBoolean` [Oraa]: this again ensures a sequentially consistent memory.³

The `!` operation⁴ stores its own `Thread` object in `writer`, stores the value to be sent in `buffer`, sets `full` to true (to indicate the value in `buffer` is valid), and attempts to unpark a waiting reader. It then waits for the reader to empty the buffer, so as to complete the synchronisation: it parks, rechecking whether `full` has been cleared each time it awakens. Finally, it clears the `writer` variable, and returns.

The `?` operation stores its own `Thread` object in `reader`. It then waits until the buffer is filled: it parks, rechecking `full` each time it awakens. It reads the value written to `buffer`, clears `full`, clears the `writer` variable, and unparks the writer. Finally it clears `reader`, and returns the value read from `buffer`.

As noted earlier, this implementation suffers from a deadlock. The reader might like to try to spot the cause of the deadlock before reading on. It is far from obvious: I was not able to find it by hand despite trying for several hours.

3. Overview of CSP

In this section we give a brief overview of the syntax for the fragment of CSP that we will be using in this paper. We then review the relevant aspects of CSP semantics, and the use of the model checker FDR in verification. For more details, see [Ros10].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. A CSP model is defined in a *script*, using machine-readable CSP, denoted as CSP_M . A script includes definitions of processes (often parameterised), other values, and functions. In particular, CSP_M includes a powerful functional language as a sub-language, roughly equivalent to Haskell without type classes, but with support for sets and mappings. New enumerated types may be introduced using the keyword **datatype**; for example

datatype `T` = `A` | `B`

introduces a new type `T` containing two values `A` and `B`.

Processes communicate via atomic *events*. Events often involve passing values over channels; for example, the event `c.3` represents the value `3` being passed on channel `c`. Channels may be declared using the keyword **channel**; for example, **channel** `c` : `T` declares `c` to be a channel that passes values of type `T`. The notation $\llbracket c \rrbracket$ represents the set of events over channel `c`; with `c` and `T` as above, $\llbracket c \rrbracket = \{c.A, c.B\}$.

The simplest process is **STOP**, which represents a deadlocked process that cannot communicate with its environment.

The process `a → P` offers its environment the event `a`; if the event is performed, the process then acts like `P`. The process `c?x → P` is initially willing to input a value `x` on channel `c`, i.e. it is willing to perform any event of the form `c.x`; it then acts like `P` (which may use `x`). Similarly, the process `c?x:X → P` is willing to input any value `x` from set `X` on channel `c`, and then to act like `P`. The process `c!v → P` outputs value `v` on channel `c`. Inputs and outputs may be mixed within the same communication, for example `c?x!v → P`.

The process `P □ Q` can act like either `P` or `Q`, the choice being made by the environment: the environment is offered the choice between the initial events of `P` and `Q`. By contrast, `P ⊓ Q` may act like either `P` or `Q`, with the choice being made internally or nondeterministically, not under the control of the environment. The process

² The “_” initialises the variable to the default value of type `T`, for example `null` for a reference type.

³ The variable `buffer` is declared as a **var**, meaning it is mutable; by contrast, the other variables are declared as **vals**, meaning they are immutable, in each case an immutable reference to a mutable object implementing the atomic variable.

⁴ Operations or functions are declared in Scala using the **def** keyword. Parameters are typed using Pascal-style syntax.

$\sqcap x:X \bullet P(x)$ is an indexed nondeterministic choice, with the choice being made over the processes $P(x)$ for x in X . The process $c\$x \rightarrow P$ nondeterministically chooses a value x and offers the event $c.x$; this is equivalent to $\sqcap x:T \bullet c.x \rightarrow P$ where T is the type of c . The process **if** b **then** P **else** Q represents a conditional. The process $b \& P$ is a guarded process, that makes P available only if b is true; it is equivalent to **if** b **then** P **else** **STOP**.

The process $P \parallel A \parallel Q$ runs P and Q in parallel, synchronising on events from A . The process $P \parallel\!\!\parallel Q$ interleaves P and Q , i.e. runs them in parallel with no synchronisation. The process $\parallel\!\!\parallel x:X \bullet P(x)$ represents an indexed interleaving.

The process $P \setminus A$ acts like P , except the events from A are hidden, i.e. turned into internal τ events.

A *trace* of a process is a sequence of (visible) events that a process can perform. We say that P is refined by Q in the traces model, written $P \sqsubseteq_T Q$, if every trace of Q is also a trace of P . FDR can test such refinements automatically, for finite-state processes. Typically, P is a specification process, describing what traces are acceptable; this test checks whether Q has only such acceptable traces.

Traces refinement tests can only ensure that no “bad” traces can occur: they cannot ensure that anything “good” actually happens; for this we need the stable failures or failures-divergences models. A *stable failure* of a process P is a pair (tr, X) , which represents that P can perform the trace tr to reach a stable state (i.e. where no internal events are possible) where X can be refused (i.e. where none of the events of X is available). In particular, if X is the set of all events, this represents a deadlock. We say that P is refined by Q in the stable failures model, written $P \sqsubseteq_F Q$, if every trace of Q is also a trace of P , and every stable failure of Q is also a stable failure of P .

We say that a process *diverges* if it can perform an infinite number of internal (hidden) events without any intervening visible events. If $P \sqsubseteq_F Q$ and Q is divergence-free, then if P can stably offer an event a , then so can Q ; hence such tests can be used to ensure Q makes useful progress.

4. The CSP model

In this section we present the CSP model we used to analyse the OneOne channel. The model is available via <http://www.cs.ox.ac.uk/people/gavin.lowe/OneOne/>.

The model is parameterised by two types: the type T of data passed on the channel, and the type ThreadIDType of thread identities.

```
datatype T = A | B                -- The type of data
datatype ThreadIDType = Null | W | R -- The type of thread identities
ThreadID = diff(ThreadIDType, {Null}) -- Real thread identities
```

We choose small values for these types here. The type ThreadIDType includes a special value `Null`, modelling the Scala value `null`; the sub-type ThreadID models real thread identities.⁵

The CSP model of the variables is in Fig. 2. Most of the variables have a very similar structure, so we present a generic variable `Var(getChan, setChan)(value)`. Here `value` represents the current value of the variable. The value can be obtained by thread t using event `getChan.t.value`, or set to a new value `value1` using event `setChan.t.value1`. The `full`, `reader` and `buffer` variables are modelled by instantiating the `getChan` and `setChan` channels appropriately. The model of the `writer` variable needs to also support the `getAndSet` operation: an event `getAndSetWriter.t.value.value1` represents thread t reading the old value `value` and replacing it with `value1`.

The CSP model of `LockSupport` is also in Fig. 2. The process `LockSupport` is parameterised by two sets: `waiting` stores those threads that are currently parked; `permits` stores permits for those threads that have been unparked but have not yet parked. The auxiliary process `LockSupport1` captures the behaviour excluding spurious wake-ups. A thread t parking is modelled by event `park.t`: if there is a permit for t , then it is immediately resumed, via event `wakeUp.t`; otherwise t is added to `waiting`.⁶ A thread $t1$ calling `unpark(t)` is modelled by the event `unpark.t1.t`: if t is waiting, it is resumed; otherwise, if $t \neq \text{Null}$, t is added to `permits`. Finally, the process `LockSupport` may act like `LockSupport1`, or may nondeterministically cause a waiting thread (if there is one) to receive a spurious wake-up. As an aside, we believe that this CSP model captures the behaviour of `LockSupport` in a way that is clearer than its informal description [Orac].

The CSP model of the threads is in Fig. 3. We introduce extra events `endSend.t` and `endReceive.t` to represent thread t ending a send or receive, respectively; we will use these for specification purposes later.

⁵ The function `diff` gives the difference between two sets.

⁶ The function `member` tests whether a value is a member of a set; the function `union` gives the union of two sets.

Discovering and correcting a deadlock in a channel implementation

```

-- A generic variable
Var(getChan, setChan)(value) =
  getChan ? t ! value → Var(getChan, setChan)(value)
  □ setChan ? t ? value1 → Var(getChan, setChan)(value1)

-- The full, reader and buffer variables
channel getFull, setFull : ThreadID . Bool
channel getReader, setReader : ThreadID . ThreadIDType
channel getBuffer, setBuffer : ThreadID . T
Full = Var(getFull, setFull)( false )
Reader = Var(getReader, setReader)(Null)
Buffer = Var(getBuffer, setBuffer)(A)

-- The writer variable
channel getWriter, setWriter : ThreadID . ThreadIDType
channel getAndSetWriter : ThreadID . ThreadIDType . ThreadIDType
Writer0(value) =
  getWriter ? t ! value → Writer0(value)
  □ setWriter ? t ? value1 → Writer0(value1)
  □ getAndSetWriter ? t ! value ? value1 → Writer0(value1)
Writer = Writer0(Null)

-- All variables
AllVars = Full ||| Reader ||| Writer ||| Buffer

-- LockSupport
channel park, wakeUp : ThreadID
channel unpark : ThreadID . ThreadIDType

LockSupport0 = LockSupport({}, {})
LockSupport1(waiting, permits) =
  park ? t → (
    if member(t, permits)
    then wakeUp . t → LockSupport(waiting, diff(permits, {t}))
    else LockSupport(union(waiting, {t}), permits) )
  □
  unpark ? _ ? t → (
    if member(t, waiting)
    then wakeUp . t → LockSupport(diff(waiting, {t}), permits)
    else if t ≠ Null
    then LockSupport(waiting, union(permits, {t}))
    else LockSupport(waiting, permits) )
LockSupport(waiting, permits) =
  if waiting = {} then LockSupport1(waiting, permits)
  else ( LockSupport1(waiting, permits)
    □ wakeUp$t:waiting → LockSupport(diff(waiting, {t}), permits) )

```

Fig. 2. The CSP model of the variables and LockSupport

The process `Send(me, value)` corresponds to thread `me` sending `value`. This is a natural translation of the Scala code from Fig. 1. The subsidiary process `SendWait` corresponds to the code from line 9 onwards, recursing back to re-check if `full` is set each time it resumes after parking. A receiving thread is modelled similarly. We model a system with two threads: thread `W` repeatedly sends a nondeterministically chosen value; and thread `R` repeatedly receives.

The complete system is also in Fig. 3. We combine the processes in parallel, synchronising the threads with the other processes on the natural alphabets.

```

channel endSend, endReceive : ThreadID . T

Send(me, value) =
  setWriter . me . me → setBuffer . me . value → setFull . me . true →
  getReader . me ? r → unpark . me . r → SendWait(me, value)
SendWait(me, value) =
  getFull . me ? f →
  if f then park . me → wakeUp . me → SendWait(me, value)
  else setWriter . me . Null → endSend . me . value → Thread(me)

Receive(me) = setReader . me . me → ReceiverWait(me)
ReceiverWait(me) =
  getFull . me ? f →
  if not(f) then park . me → wakeUp . me → ReceiverWait(me)
  else
    getBuffer . me ? result → setFull . me . false → getAndSetWriter . me ? w ! Null →
    unpark . me . w → setReader . me . Null → endReceive . me . result → Thread(me)

Thread(me) = if me = W then  $\square$  value: T • Send(me, value) else Receive(me)

AllThreads =  $\square \square \square$  id: ThreadID • Thread(id)

syncSet = { getFull, setFull, getBuffer, setBuffer, getReader, getWriter,
            setReader, setWriter, getAndSetWriter, park, wakeUp, unpark }
System = AllThreads [] syncSet [] (AllVars  $\square \square \square$  LockSupport0)
assert System : [deadlock free] — fails

```

Fig. 3. CSP model of the threads and the complete system

4.1. Analysis

We can use FDR to test whether the system is deadlock-free. However, this test fails (in about 0.2 seconds). FDR discovers a trace of 37 events, involving two runs of the sender and three of the receiver; we explain the trace below (we have slightly altered the trace below from the one found by FDR, reordering some independent events, to help understanding).

1. The sender runs for the first time; sets `writer = W`; stores its value; sets `full = true`; and reads `reader = null`.
`<setWriter . W . W, setBuffer . W . B, setFull . W . true, getReader . W . Null,`
2. The receiver runs for the first time; sets `reader`; reads `full = true` (so does not park); reads buffer; and sets `full = false`.
`setReader . R . R, getFull . R . true, getBuffer . R . B, setFull . R . false,`
3. The sender continues: calls `unpark(null)`; reads `full = false`; clears `writer`; and returns.
`unpark . W . Null, getFull . W . false, setWriter . W . Null, endSend . W . B,`
4. The sender runs for the second time: sets `writer`; and stores its value.
`setWriter . W . W, setBuffer . W . B,`
5. The receiver continues: does `writer.getAndSet(null)`, clearing `writer` and obtaining `W`; calls `unpark(W)`; clears `reader`; and returns.
`getAndSetWriter . R . W . Null, unpark . R . W, setReader . R . Null, endReceive . R . B,`
6. The sender continues: sets `full = true`; reads `reader = null`; calls `unpark(null)`; reads `full = true`; calls `park`, but immediately resumes because of the `unpark` in the previous item; rereads `full = true`; and parks again.
`setFull . W . true, getReader . W . Null, unpark . W . Null, getFull . W . true, park . W, wakeUp . W,`
`getFull . W . true, park . W,`

7. The receiver runs for a second time: sets `reader`; reads `full = true` (so doesn't park); reads `buffer`; sets `full = false`; calls `writer.getAndSet(null)` and obtains `null`; calls `unpark(null)`; clears `reader`; and returns.

```
setReader.R.R, getFull.R.true, getBuffer.R.B, setFull.R.false, getAndSetWriter.R.Null.Null,
unpark.R.Null, setReader.R.Null, endReceive.R.B,
```

8. The receiver runs for a third time: sets `reader`; finds `full = false`; and parks.

```
setReader.R.R, getFull.R.false, park.R).
```

The resulting system is deadlocked: both threads are parked, and no other thread will unpark them.

The problem is that the `unpark` in the first run of the receiver (item 5) has an effect on the *second* run of the sender (item 6), whereas it should have been targetted towards the first run of the sender. Further, this run of the receiver clears the `writer` variable, without the sender actually returning. This means that the second run of the receiver (item 7) reads `null` from `writer`, and so passes `null` to `unpark`; this fails to unpark the second run of the sender from its second park (item 6).

The cause of the problem is reasonably straightforward to spot given the trace. The `getAndSetWriter.R.W.Null` event in item 5 stands out as having incorrectly cleared the `writer` variable.

5. Correcting the error

Given the discussion at the end of the previous section, a possible fix to the error would seem to be for the receiver not to clear the `writer` variable, i.e. to change line 17 of Fig. 1 to

```
LockSupport.unpark(writer.get)
```

In this section we show that the proposed fix does indeed work.

It is straightforward to adapt the model from the previous section corresponding to this change. FDR then fails to find any deadlock on the system.

However, we can prove a stronger property of the channel. We can check that the senders and receivers are loosely synchronised: each completes at most one run more than the other. More importantly, we can verify that corresponding runs of the sender and receiver agree upon the value sent on the channel. We do this by hiding all events other than `endSend` and `endReceive` events:

```
System1 = System \ diff(Events, [| endSend, endReceive |])
```

Concentrating on just the `endSend` and `endReceive` events, the following specification process captures the patterns of events that we expect.

```
Spec = endSend$w$v → endReceive$r!v → Spec □ endReceive$r$w → endSend$w!v → Spec
```

We can then use FDR to verify that $\text{Spec} \sqsubseteq_F \text{System1}$.

However, the above turns out not to be enough: the above refinement does *not* hold in the failures-divergences model, because `System1` can diverge. The reason for the divergence is that a process that has parked might repeatedly receive a spurious wake-up, recheck the relevant condition, and then park again. This is not the sort of behaviour we should be concerned about: spurious wake-ups are rare and, while we need to guard against them, we should not be concerned about the possibility of threads spinning in this way.

We should, however, check that this divergence is not masking a more serious one: we want to be sure that the only divergences are those caused by repeated spurious wake-ups. We do this by introducing a new event `spurious` that indicates that the following wake-up is spurious: we adapt the `LockSupport` process as follows.

```
LockSupport(waiting, permits) =
  if waiting = {} then LockSupport1(waiting, permits)
  else ( LockSupport1(waiting, permits)
        □ spurious → wakeUp$t:waiting → LockSupport(diff(waiting, {t}), permits) )
```

We can then use FDR to verify the following process is divergence-free.

```
System2 = System \ diff(Events, [| endSend, endReceive, spurious |])
```

This verifies that every divergence in `System1` indeed involves repeated spurious wake-ups, and hence that the system correctly implements a channel.

So far, we have considered only a system with a single sender and receiver. Recall that CSO has channels that can be used by multiple senders or receivers; these are implemented by wrapping a `OneOne` channel, and using two locks to ensure at most a single sender and single receiver access the `OneOne` at a time. We can extend our model to capture such channels as follows. We can arrange for each sender and receiver to perform an additional event to indicate that they are starting an execution, using the following channels.

`channel startSend, startReceive : ThreadID`

Recall that we already have similar events indicating the end of an execution. We can then model the locks as follows

`SendLock = startSend?t → endSend.t?v → SendLock`
`ReceiveLock = startReceive?t → endReceive.t?v → ReceiveLock`

We can add these processes into the system, synchronising appropriately.

We can then consider systems with multiple threads. It is clear that if all threads choose to act as senders (or receivers) then the system will deadlock. We therefore consider systems where one thread always sends, one always receives, and the others nondeterministically decide whether to send or receive. We have verified that each of the above refinement checks hold with such a system containing a total of six threads. (The checks complete in slightly over five minutes in total.)

The above analysis does not guarantee that the refinements also hold for larger numbers of threads, although it gives us high confidence. This is an instance of the *parameterised verification problem*, where one wants to verify all members of a parameterised family of systems; here, the parameter is the number of threads. In [Low18] we used techniques from view abstraction and symmetry reduction to tackle the parameterised verification problem for trace-based properties. We subsequently [Low19] extended this work to verify deadlock freedom. We have applied these techniques to verify the implementation satisfies the trace property, deadlock freedom and divergence freedom for an arbitrary number of threads (capturing divergence freedom as a trace property).

6. Conclusions

In this paper we have demonstrated how to use CSP and its model checker FDR to discover the reasons behind a deadlock in the implementation of a channel in a message-passing concurrency API. The bug was rather subtle, and required a particular interleaving of events, involving five operation calls. These sorts of errors are almost infeasible for humans to spot by hand. On the other hand, it is very easy for a tool like FDR to explore a model and so find the relevant trace. Further, it can then be used to verify a proposed fix. This paper, therefore, serves as a reminder of the difficulty of low-level concurrent programming, and of the importance of formal analysis.

As noted at the end of Sect. 4, the cause of the error was fairly straightforward to extract from the trace. Perhaps the biggest difficulty was that the raw trace produced by FDR contained many “context switches”, where successive events were performed by different threads. Reordering independent events to reduce the number of context switches gave a trace that was easier to understand. It would be useful if this reordering could be done automatically by the model checker. We leave this as future work.

Acknowledgements I would like to thank Bernard Sufrin for useful discussions involving CSO (over many years), and in particular discussions concerning this work. I would also like to thank the anonymous referees for their useful comments.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

- [AGRR16a] Antonino P, Gibson-Robinson T, Roscoe AW (2016) Efficient deadlock-freedom checking using local analysis and SAT solving. In: Proceedings of IFM, vol 9681, pp 345–360. LNCS
- [AGRR16b] Antonino P, Gibson-Robinson T, Roscoe AW (2016) Tighter reachability criteria for deadlock freedom analysis. In: Proceedings of FM, vol 9995. LNCS
- [AGRR17] Antonino P, Gibson-Robinson T, Roscoe AW (2017) The automatic detection of token structures and invariants using SAT checking. In: Proceedings of TACAS, vol 10206, pp 249–265. LNCS
- [BR91] Brookes SD, Roscoe AW (1991) Deadlock analysis in networks of communicating processes. *Distrib Comput* 4:209–230
- [GRABR14] Gibson-Robinson T, Armstrong P, Boulgakov A, Roscoe AW (2014) FDR3—a modern refinement checker for CSP. In: Proceedings of Tools and algorithms for the construction and analysis of systems (TACAS), vol 8413. LNCS
- [Law05] Lawrence J (2005) Practical applications of CSP and FDR to software design. In: Communicating sequential processes: the first 25 years. Lecture notes in computer science, vol 3525, pp 151–174. Springer, Berlin
- [Low96] Lowe G (1996) Breaking and fixing the Needham–Schroeder public-key protocol using FDR. In: Proceedings of TACAS. Lecture notes in computer science, vol 1055, pp 147–166. Springer, Berlin. Also in *Software—Concepts and Tools* (1996) 17:93–102
- [Low11] Lowe G (2011) Implementing generalised alt—a case study in validated design using CSP. In: Communicating process architectures, pp 1–34
- [Low18] Lowe G (2018) View abstraction for systems with component identities. In: Proceedings of the international symposium on formal methods (FM 2018), vol 10951, pp 502–522. Springer, Berlin
- [Low19] Lowe G (2019) Parameterised verification of systems with component identities, using view abstraction. Submitted for publication
- [Mar96] JMR Martin (1996) The design and construction of deadlock-free concurrent systems. Ph.D. thesis, University of Buckingham
- [MS01] Mota A, Sampaio A (2001) Model-checking CSP-Z: strategy, tool support and industrial application. *Science of computer programming* 40(1):59–96
- [Oraa] Oracle AtomicBoolean (Java SE9 & JDK 9). <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/AtomicBoolean.html>. Accessed 11 Jul 2019
- [Orab] Oracle AtomicReference (Java SE9 & JDK 9). <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/atomic/AtomicReference.html>. Accessed 11 Jul 2019
- [Orac] Oracle LockSupport (Java SE9 & JDK 9). <https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/locks/LockSupport.html>. Accessed 11 Jul 2019
- [RD87] Roscoe AW, Dathi N (1987) The pursuit of deadlock freedom. *Inf Comput* 75(3):289–327
- [RH07] Roscoe AW, Hopkins D (2007) SVA, a tool for analysing shared-variable programs. *Proceedings of AVoCS 2007*:177–183
- [Ros10] Roscoe AW (2010) Understanding concurrent systems. Springer, Berlin
- [Suf08] Sufriin B (2008) Communicating scala objects. In: Proceedings of communicating process architectures (CPA)
- [WM00] Welch P, Martin J (2000) A CSP model for Java multithreading. In: Proceedings of the IEEE international symposium on software engineering for parallel and distributed systems, pp 114–122

Received 31 August 2018

Accepted in revised form 26 June 2019 by Michael Butler