

## Abstract

Concurrent objects are convenient tools for programmers. With concurrent objects, programmers can write code with multiple threads as if they are writing a single-threaded code. However, it is crucial to know the correctness of concurrent objects. In this thesis, we study a technique to justify the correctness of synchronization objects. We first present CSP models for common concurrent primitives according to their behaviours. We then systematically build several concurrent objects from their Scala sources. We make assertions of these synchronization objects with a technique derived from linearizability testing. We find these assertions can effectively find bugs in a concurrent datatype and provide a history to give more context for the developer.

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                            | <b>1</b>  |
| 1.1      | Thesis Overview . . . . .                      | 1         |
| 1.2      | Synchronization linearizability test . . . . . | 2         |
| 1.3      | Checking safety property using CSP . . . . .   | 3         |
| 1.4      | Checking liveness property using CSP . . . . . | 4         |
| 1.5      | Related work . . . . .                         | 4         |
| <b>2</b> | <b>Common Objects</b>                          | <b>5</b>  |
| 2.1      | Shared Variable . . . . .                      | 5         |
| 2.2      | Semaphore . . . . .                            | 5         |
| 2.3      | Monitor . . . . .                              | 7         |
| 2.3.1    | JVM Monitor . . . . .                          | 7         |
| 2.3.2    | Monitor Module . . . . .                       | 8         |
| <b>3</b> | <b>MenWomen</b>                                | <b>13</b> |
| 3.1      | Implementation . . . . .                       | 13        |
| 3.2      | Linearization Test . . . . .                   | 14        |
| 3.3      | A faulty version . . . . .                     | 17        |
| <b>4</b> | <b>ABC</b>                                     | <b>19</b> |
| 4.1      | Implementation . . . . .                       | 19        |
| 4.2      | Testing . . . . .                              | 19        |
| 4.2.1    | Speeding up model compilation . . . . .        | 22        |
| 4.3      | Faulty version . . . . .                       | 24        |
| 4.3.1    | Explanation of the error case . . . . .        | 24        |
| 4.4      | Explicit linearization point test . . . . .    | 25        |
| <b>5</b> | <b>Terminating Queue</b>                       | <b>28</b> |
| 5.1      | Implementation . . . . .                       | 28        |
| 5.2      | Linearization Testing . . . . .                | 30        |
| 5.3      | Faulty Implementation . . . . .                | 33        |
| 5.4      | Test with another queue . . . . .              | 34        |
| <b>6</b> | <b>Other objects</b>                           | <b>36</b> |
| <b>7</b> | <b>Conclusion</b>                              | <b>37</b> |
| <b>8</b> | <b>Reference</b>                               | <b>38</b> |
| <b>9</b> | <b>Appendix</b>                                | <b>39</b> |

# 1 Introduction

Concurrent objects are convenient tools for programmers. With concurrent datatypes, programmers can write code with multiple threads as if they are writing a single-threaded code. However, it is crucial to know the correctness of concurrent objects. If the implementation of a concurrent object is wrong, then code using the concurrent object is very likely to be faulty.

The **Channel** object is one synchronization object commonly used in Go. The channel object can be used to share data from one process to another process. A process can send data to another process by calling **send** with the data to share. Likewise, a process can receive data from other processes by calling **receive** function. In Go and Communicating Scala Object package, channels are unbuffered by default. If there is no process to receive the data, the sending process blocks until a process is willing to receive its data. Similarly, a receiving process blocks until a process sends some data.

```
trait Channel{  
  def send(data: Int): Int  
  def receive(): Int  
}
```

Figure 1: Interface of a Channel object

In this thesis, we shall study the correctness of synchronization objects. Each synchronization in a synchronization object involves multiple processes, whereas synchronization in concurrent datatypes like concurrent queue and concurrent only involves a single process.

There are two main properties to check for a synchronization object, the safety property and the liveness property. The safety property states that the history of the synchronization object should satisfy some conditions. For example, if one process sends **1** when no other process is sending, then a process calling **receive** should only receive **1**. The liveness property states that the concurrent object should not refuse to synchronize when synchronization is possible between one or more processes. For example, if a process calls **send** and a process **receive**, the system should be able to synchronize and should not deadlock.

## 1.1 Thesis Overview

In the remaining part of Section 1, we describe the correctness condition for a synchronization object and abstractly how to test these conditions in CSP using the linearization test technique.

In section 2, we build CSP modules for common concurrent primitives such as shared variables, monitors and semaphores.

Starting from section 3, we use the linearization test technique to distinguish correct and faulty implementation for several synchronization objects. We first implement the synchronization object in CSP according to its Scala source code. Then we write specifications for a system using the synchronization object and carry out the tests.

## 1.2 Synchronization linearizability test

To verify the correctness of a concurrent datatype, one can carry out the linearizability test described in the paper Testing for Linearizability [Low17]. The linearizability testing framework logs the orders of each function call and function return. The testing framework then justifies the observed history by attempting to find a series of synchronization points that obey the safety property. The concurrent datatype implementation is considered faulty if the framework can not find a valid synchronization point series.

In this remaining section, we shall look at a few examples of histories of systems using the `Channel` object. Figure 2 visualizes the history of a system with two processes. Process `T1` calls `send` with argument 1 and returns. Process `T2` calls `receive` and returns with 1. Each long horizontal line in the timeline represents a function call made by the corresponding process. The short vertical bars at the two ends of the long horizontal line indicate the function call's starting time and ending time. And the long vertical line between `T1` and `T2` represents the synchronization between the two processes.

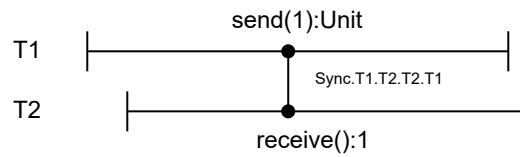


Figure 2: Visualized history of T1 calling `send(1)` and T2 calling `receive()`

Figure 3 shows a timeline similar to Figure 2, but `T2` returns 2 instead of 1. In this case, the linearizability test framework can not justify the return of process `T2`'s `receive`, and suggests the trace is generated by a faulty channel implementation.

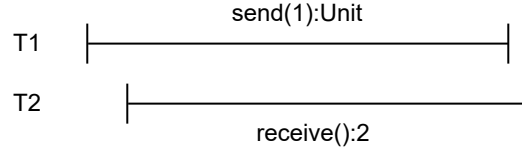


Figure 3: Visualized history of T1 sends 1 but T2 receives 2

In Figure 4, both processes calls `send`, and no synchronization is possible. Note that the liveness condition is not invalidated even if the system deadlocks in this case.

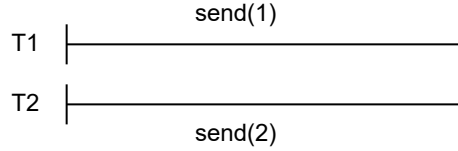


Figure 4: Visualized history of both T1 and T2 calling send

Scheduling is one of the reasons validating a history can be complicated. In Figure 5, process **T3** calls `send(3)` first but gets descheduled. Then **T1** calls `send(1)` and synchronizes with **T2** which later calls `receive`. The linearization framework usually needs to search a large state space to find a valid series of synchronization points.

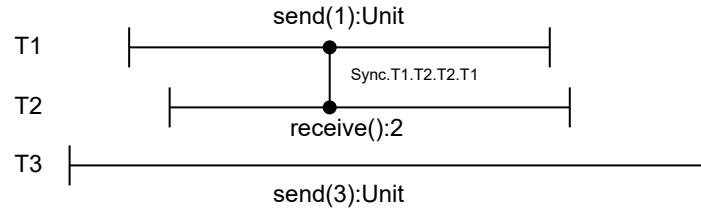


Figure 5: Visualized history of T3 get descheduled

### 1.3 Checking safety property using CSP

The history observed by the linearizability framework can be captured as a trace in CSP. A **Call** event in CSP represents the start of a function call in the observed history. A **Return** event represents the returning of a function call. For the safety property, we check that set of all possible histories of a testing system is a subset of all correct histories. In CSP, this corresponds

to an assertion that a testing system trace refines a specification of systems using the synchronization object.

A generic and scalable system is used as the testing system. Each process in the testing system can call any function from the concurrent object with any arguments allowed. Each process must be allowed to terminate. Otherwise, the testing system only models a system that runs forever, given that there is no deadlock. We shall see how this affects bug finding in a concurrent datatype in later objects.

The specification process is constructed using the linearization technique. On the high level, the specification process for the system internally uses **Sync** events to represent synchronization between processes. Inside the specification process, some sub-processes generate corresponding **Call** and **Return** event for every synchronization point. When all sub-processes are placed in parallel, the **Sync** event agrees. So the resulting specification system generates all possible histories.

We shall see a concrete implementation of a testing system and a specification process in the MenWomen section.

## 1.4 Checking liveness property using CSP

For liveness property, we check the same generic and scalable testing system refines the same specification process, but in the failure model. One could use a datatype-specific specification process that does not explicitly use any synchronization points. However, reusing the linearizer process is easier.

## 1.5 Related work

Testing for Linearizability [Low17] presents a framework to test concurrent datatypes. As described in earlier subsections, the testing framework uses observations of histories and is unlikely to exhaust all possible histories of a system or check a infinite size history. However, the linearization test can exhaust all finite and infinite histories of a system using FDR's state machine.

There are also runtime programming tools to detect race conditions and deadlocks in concurrent code. Thread Sanitizer [SI09] detects race conditions and deadlocks in C++ and Go when the program is running. Like linearizability testing, runtime programming tools is unlikely to exhaust all possible histories.

In Chapter 19 of Understanding Concurrent Systems [Ros11], the author describes a CSP model for shared variables and provides a tool to analyze shared variable programs. This project further supports analyzing program using more synchronizations, such as monitor and semaphores.

## 2 Common Objects

### 2.1 Shared Variable

The usage of shared variables is common in concurrent datatypes. For example, some concurrent datatypes may temporarily store the identity of a waiting process. However, CSP is more like a functional programming language and does not support mutable variables.

A recursive process in CSP can capture the behaviour of a shared variable. The recursive process holds the value of the variable in its parameter. At any time, the variable process is willing to answer a query for the variable value in channel `getValue`. Alternatively, the process can receive an update on the variable value in channel `getValue`, after which the function recurses with the new variable value.

Because it is natural for a concurrent datatype to use multiple shared variables, the global variable is implemented as a CSP module in Figure 6 to allow better code reuse. The module requires two parameters. `TypeValue` is the set of possible values for the variable, and `initialValue` is the value before any process modifies the variable. An uninitialized variable module is also available in the same Figure 6, with the only difference that the variable non-deterministically chooses an initial value from `TypeValue` at start time. `runWith` is a convenient helper function to run a given process `P` with the `Var` process. If the parameter `hide` is true, `runWith` function hides all events introduced by the shared variable. In later chapters, we will see how the `runWith` function helps reduce the code complexity of the synchronization object implementation.

Figure 7 is an example of two processes using a shared variable. The first line in the example creates a shared variable `VarA` with value ranging from 0 to 2 and initialized with 0. Process `P` increments `VarA` modulo 3 forever and process `Q` reads `VarA` forever. Process `P` interleaves with process `Q`, and the combined process is further synchronized with the variable `VarA` process. In the resulting process `System`, changes to `VarA` made by process `P` is visible to process `Q`.

### 2.2 Semaphore

A Semaphore is a simple but powerful concurrent primitive. This thesis shall describe and use a simplified binary semaphore from [Lea06], which removes interrupts and timeout operations.

A binary semaphore can either be raised or lowered. A `up` function call raises the semaphore regardless of the semaphore state. If a process calls the

```

--set of possible value for the variable
--inital value for the variable
module ModuleVariable(TypeValue, initialValue)
  Var(value) = getValue!value → Var(value)
             □ setValue?value → Var(value)
  chanset = {getValue, setValue}
exports
  --(Bool, Proc) → Proc
  runWith(hide,P) = if hide then (Var(initialValue) [|chanset|] P) \ chanset
                    else Var(initialValue) [|chanset|] P
  channel getValue, setValue: TypeValue
endmodule

module ModuleUninitVariable(TypeValue)
  Var(value) = getValue!value → Var(value)
             □ setValue?value → Var(value)
  chanset = {getValue, setValue}
exports
  runWith(hide,P) =
    if hide then (|~| x:TypeValue • Var(x)) [| chanset |] P) \ chanset
    else (|~| x:TypeValue • Var(x)) [| chanset |] P
  channel getValue, setValue: TypeValue
endmodule

```

Figure 6: The shared variable module in CSP

```

instance VarA = ModuleVariable({0..2},0)
P = VarA::getValue? a → VarA::setValue!((a+1)%3) → P
Q = VarA::getValue? a → Q
System = VarA::runWith(false,P|||Q)

```

Figure 7: Example of two processes using a shared variable



**down** method when the semaphore is raised, the semaphore becomes lowered. However, if the semaphore is unraised, the process waits until another process calls **up** and proceeds to put down the semaphore. Depending on the initial state of the semaphore, a binary semaphore can be further categorized as a mutex semaphore or a signalling semaphore.

Modelling a semaphore is straightforward in CSP. Figure 8 is the CSP semaphore module. A process may call **up** function or **down** function via channel **upChan** or channel **downChan** respectively. The semaphore is modelled by a process implemented by two mutually recursive functions **Semaphore(True)** and **Semaphore(False)**. The semaphore process representing an unraised state accepts a **upChan** event by any process and proceeds to the raised process. The semaphore process representing a raised state can either accept a **upChan** event and recurse to the raised process, or accept a **downChan** event and proceed to the unraised process.

Like the shared variable in the earlier subsection, the semaphore is encapsulated in a CSP module. To create a semaphore, one needs to supply two arguments. **TypeThreadID** is the set of identities of processes that use this semaphore. **initialState** is a boolean value indicating the starting state of the semaphore. If **initialState** is true, the semaphore is raised initially. Otherwise, the semaphore is lowered.

```

module ModuleSemaphore(TypeThreadID, initialState)
  --Raised
  Semaphore(True) = downChan ? id → Semaphore(False)
                  □ upChan ? id → Semaphore(True)

  --Unraised
  Semaphore(False) = upChan ? id → Semaphore(True)

  chanset = {upChan, downChan}
exports
  --runWith::(Bool,Proc) → Proc
  runWith(hide,P) = (Semaphore(initialState) || chanset || P) \
                    (if hide then chanset else {})
  channel upChan, downChan: TypeThreadID
endmodule

```

Figure 8: The binary semaphore module in CSP

## 2.3 Monitor

### 2.3.1 JVM Monitor

A Monitor is another powerful concurrent primitive. This thesis will also use a simplified monitor from [Lea06]. JVM Monitor provides two key features,

mutual exclusion and waiting.

Monitors can be used to prevent race conditions. At any time, only one process can run code inside a synchronized block that belongs to one monitor. The function `op1` in Figure 9 uses synchronized block to prevent race condition on variable `a`.

Inside a `synchronized` block, the process can also perform `wait`, `notify`, and `notifyAll`. When a process inside the synchronized block calls `wait`, the process suspends and waits for notification from other processes. Since a waiting process may be spuriously waked up, so a `wait` call is used with a while loop and a condition. In the MonitorExample of Figure 9, `op2` waits until there is 10 `op1` calls. In `op1`, a process calls `notifyAll` after incrementing the shared variable `a`. When there aren't 10 `op1` calls, process waiting in `op2` first wakes, finds the condition `a < 10` true, and returns to sleep.

```
class MonitorExample {  
  private var a = 0;  
  
  def op1():Unit = synchronized{  
    a=(a+1)%20;  
    notifyAll()  
  }  
  def op2():Unit = synchronized{  
    while(a<10) wait();  
  }  
}
```

Figure 9: A simple Scala class that uses a monitor internally

### 2.3.2 Monitor Module

The monitor process has two states and behaves differently in two states, captured by two processes in Figure 11.

When there is no running process, the behaviour of the monitor is captured by the CSP process `inactive`, with parameter `waiting` being the set of waiting processes. The monitor can allow a process to run synchronized code with a `waitEnter` event. Or, the monitor can spuriously wake a process with a `SpuriousWake` event, and the spuriously waken process behaves like a normally waken process.

When there is a running process, the behaviour of the monitor is captured by the CSP process `active`, with parameter `cur` being the identity of the process running `synchronized` block, and `waiting` being the set of the waiting process. The monitor process should respond to method calls from the running process `cur`, and the monitor should not allow another process to obtain the monitor lock. In `active` state, the monitor can also spuriously wake a waiting process.

On the client process side, most functions are implemented by simply synchronizing with the monitor on an event. For example, before entering the `synchronized` block, the process sends a `WaitEnter` event. The only exception is the `wait` function, as after being notified, the process needs to resume execution. The process first sends a `wait` event to tell the monitor that it is waiting and release the monitor lock. The monitor then receives a `waitNotify` or `spuriousWake` event, for being notified. Then the process reobtains the monitor lock with a `waitEnter` event.

The monitor module also provides a few useful macros. The `synchronized` function wraps a CSP code to run under the protection of the monitor. The `whilewait` function implements the common Scala pattern ‘while(cond) wait()’. The implementation uses a functional replacement for `while` statement in imperative programming languages. The `cond` parameter is a CSP function of type ‘(Proc, Proc) -> Proc’. The return process of `cond` first performs some events to check the condition of the while statement. If the condition is true, the return process continues to run the process in the first parameter, or the return process runs the process in the second parameter.

With these process side functions and macros, the Scala `MonitorExample` class in Figure 9 can be converted into the CSP code in Figure 10.

```
instance VarA = ModuleVariable({0..20},0)
instance Monitor = ModuleMonitor(TypeThreadID, False)

op1(me)=synchronized(me,
  --a=(a+1)%20;
  VarA::getValue?x → VarA::setValue!((x+1)%20) →
  --notifyAll();
  Monitor::notifyAll(me)
)

op2(me)=synchronized(me,
  --while(a<10) wait()
  Monitor::whileWait(me, \ ktrue,kfalse •
    VarA::getValue?x → if x<10 then ktrue else kfalse
  )
)
```

Figure 10: The CSP implementation of `MonitorExample` in Figure 9

There are two design choices worth mentioning in the implementation of the monitor.

First, note that both `WaitNotify` and `SpuriousWake` events come from the monitor process instead of directly synchronizing with the currently running process. When a process calls `notify` or `notifyAll`, it needs to synchronize with the monitor process. This is because the running process does not know how

many processes are waiting. If the monitor is implemented the other way, the notifying process will block if there is no waiting process. Similarly, a process calling `notifyAll` does not know how many processes it should wake up.

Secondly, a monitor can introduce divergence by repetitively spuriously waking up a waiting process, whose condition keeps unsatisfied and never changes. This is an unwanted behaviour in failure testing. So the monitor module has an extra parameter `disableSpurious` to disable spurious wakeups.

```

module ModuleMonitor(TypeThreadID, disableSpurious)
  channel Notify, NotifyAll, Exit, Wait,
    WaitNotify, WaitEnter, SpuriousWake: TypeThreadID

  chanset = { Notify, NotifyAll, Exit, Wait, WaitNotify, WaitEnter, SpuriousWake }

  --A list of event for every event e in s
  repeat(ch, s) = if s={} then SKIP else ch? a:s → repeat(ch, diff(s, {a}))

  --cur is current active running thread
  --waiting is a set of threads waiting to be notified
  active(cur, waiting) =
    --current running thread notify
    Notify.cur → (
      --do nothing if no thread is waiting
      if waiting={} then active(cur, {})
      --wakeup a process
      else WaitNotify? a:waiting →
        active(cur, diff(waiting, {a}))
    ) □ --current running thread notifyAll
    NotifyAll.cur → (
      repeat(WaitNotify, waiting);
      active(cur, {})
    ) □ --current running thread exit
    Exit.cur → (
      inactive(waiting)
    ) □ --current running thread wait
    Wait.cur → (
      inactive(union(waiting, {cur}))
    ) □ --spurious wakeup
    waiting ≠ {} & SpuriousWake? a:waiting → (
      active(cur, diff(waiting, {a}))
    )

  --when no active thread is running
  inactive(waiting) =
    --pick a thread that is ready to enter
    WaitEnter? a → (
      active(a, waiting)
    ) □
    --spurious wakeup
    waiting ≠ {} & SpuriousWake? a:waiting → (
      inactive(diff(waiting, {a}))
    )

```

Figure 11: The CSP Monitor Module - Part 1 - the monitor process

```

exports
  --Given a process that uses the monitor
  --Return the process synchronized with the monitor server process
  --If hide is true, monitor channels are hidden
  runWith(hideSpurious, hideInternal, P) =
    let hideset0 = if hideInternal then chanset else {} within
    let hideset1 = if hideSpurious then hideset0
                    else diff(hideset0, {SpuriousWake}) within
    (inactive({}) [|chanset|] P) \ hideset1

  --java-like synchronized function
  synchronized(me, P) = WaitEnter.me → P; Exit.me → SKIP

  enter(me) = WaitEnter.me → SKIP

  exit(me) = Exit.me → SKIP

  --notify()
  notify(me) = Notify.me → SKIP

  --notifyAll()
  notifyAll(me) = NotifyAll.me → SKIP

  --wait()
  wait(me) =
    Wait.me →
    if disableSpurious then (
      (WaitNotify.me → WaitEnter.me → SKIP)
      □ (SpuriousWake.me → WaitEnter.me → SKIP)
    ) else (
      (WaitNotify.me → WaitEnter.me → SKIP)
    )

  whileWait(me, cond) = while(cond)(wait(me);SKIP)
endmodule

```

Figure 12: The CSP Monitor Module - Part 2 - client process side functions

### 3 MenWomen

The MenWomen object is a classical problem from the concurrent programming course. In the problem, some processes need to pair off and share identities between the paired processes. Figure 13 is the interface of the MenWomen object. A process can call `manSync` from the object to pair with a process calling `womanSync`. Also a process can call `womanSync` from the object to pair with another process calling `manSync`. For simplicity, if a process is calling `manSync`, we shall call it a man process. Similarly, a process calling `manSync` is called a woman process.

```
trait MenWomenT{  
  def manSync(me: Int): Int  
  def womanSync(me: Int): Int  
}
```

Figure 13: Scala Interface of the MenWomen object

#### 3.1 Implementation

One way to implement the MenWomen object is to use a monitor and a shared variable indicating the stage of synchronization. Figure 14 is a Scala implementation of the MenWomen object with monitor.

- A man process enters the synchronization and waits until the current stage is 0. Then in stage 0, the man process sets the global variable `him` inside the `MenWomen` object to its identity. Then the man process notifies all processes so that a waiting woman process can continue. Finally, the man process waits for stage 2.
- A women process enters the synchronization and waits until the current stage is 1. The woman process sets the global variable `her` to its identity and returns the value of the global variable `him`.
- In stage 2, the waiting man process in stage 0 is wakened up by the woman process in stage 1. The man process notifies all waiting processes and returns the value of `her`.

The code snippet in Figure 14 is a Scala implementation of the MenWomen process using a monitor by Gavin Lowe. With the shared variable and monitor module, the Scala code is further translated to a CSP code in Figure 15. With the convention described in the introduction section, every function call begins with a Call event containing all parameters. And every function call ends with a Return event containing the return value.

```

class MenWomen extends MenWomenT{
  private var stage = 0
  private var him = -1
  private var her = -1

  def manSync(me: Int): Int = synchronized{
    while(stage != 0) wait()
    him = me; stage = 1; notifyAll()
    while(stage != 2) wait()
    stage = 0; notifyAll(); her
  }

  def womanSync(me: Int): Int = synchronized{
    while(stage != 1) wait()
    her = me; stage = 2; notifyAll();
  }
}

```

Figure 14: A correct MenWomen object implementation in Scala

### 3.2 Linearization Test

Recall that in the testing system, each process can call any function provided by the concurrent datatype or choose to terminate. In defining processes in the testing system, a helper function is used. `chaosP(P)` runs the process `P` forever or terminates after running a finite number of `P`. The processes in the testing system simply use `chaosP` with a process that non deterministically chooses to perform `manSync` or `womanSync` with their identities.

```

chaosP(P) = (P;chaosP(P)) □ SKIP
Thread(me)=chaosP(manSync(me) □ womanSync(me))

```

All processes in the testing system interleave with other processes and synchronize with the processes of shared variables and the process of the monitor. Since the testing system should only include `Call` and `Return` events, all other events are hidden using the first two boolean flags in `runWith` defined in earlier sections.

```

System(All)=runWith(True, True, ||| me:All • Thread(me))

```

`Sync` events represent the synchronization between a man process and a woman process, which is used internally in the testing specification. The CSP code below is the definition and specification of `Sync` channel.

```

channel Sync: TypeThreadID . TypeThreadID . TypeThreadID . TypeThreadID
Spec = Sync ? man ? woman:diff(TypeThreadID, {man}) ! woman ! man → Spec

```

A `Sync` event takes four parameters, the identity of the man process, the return value of the man process, the identity of the woman process, the return value of the woman process. For each synchronization, the spec



```

instance VarStage = ModuleVariable({0,1,2},0)
instance VarHim = ModuleUninitVariable(TypeThreadID)
instance VarHer = ModuleUninitVariable(TypeThreadID)
instance Monitor = ModuleMonitor(TypeThreadID)
manSync(me) =
  Call ! me! ManSync →
  Monitor::enter(me);
  Monitor::whileWait(me, \ ktrue,kfalse •
    VarStage::getValue? x →
    if x≠0 then ktrue else kfalse
  );
  VarHim::setValue! me →
  VarStage::setValue! 1 →
  Monitor::notifyAll(me);
  Monitor::whileWait(me, \ ktrue,kfalse •
    VarStage::getValue? x →
    if x≠2 then ktrue else kfalse
  );
  VarStage::setValue! 0 →
  Monitor::notifyAll(me);
  VarHer::getValue? ans →(
  Monitor::exit(me);
  Return ! me! ManSync! ans→
  SKIP
  )
womanSync(me)=
  Call ! me! WomanSync →
  Monitor::enter(me);
  Monitor::whileWait(me, \ ktrue,kfalse •
    VarStage::getValue? x →
    if x≠1 then ktrue else kfalse
  );
  VarHer::setValue! me →
  VarStage::setValue! 2 →
  Monitor::notifyAll(me);
  VarHim::getValue? ans →(
  Monitor::exit(me);
  Return ! me! WomanSync! ans→
  SKIP
  )

```

Figure 15: Translated CSP code for the correct MenWomen object implementation

process ensure two properties. First, synchronization occurs between two different processes. Second, the return value of each participating process is the identity of the other participating process.

A linearizer ensures two properties. The return value of a function call comes from the synchronization. And the synchronization point occurs sometime during the function call. The `Linearizer` function for `MenWomen` achieves this by two branches of CSP processes, each representing a method in the object. Each branch starts with a `Call` event, then a `Sync` event with its identity, and finally a `Return` event which computes from the `Sync` event. In addition to the two branches, the linearizer process should be able to `STOP` to match the behaviour in the testing system.

Finally, `Linearizer(All)` composites the `Sync` specification with the linearizer processes so that the `Sync` events are valid according to the `Sync` specification and `Sync` events are agreed by both participating process. The latter property is achieved using replicated generalized parallel. By the semantic of the replicated generalized parallel, when a process wants to perform an event, the process must synchronize all other processes which can perform this event. Figure 16 visualizes a trace generated by `1,T2Linearizers(T)` by the FDR debugging window. When a linearizer performs `Call` and `Return`, the linearizer does not synchronize with any process. When `linearizer(T1)` performs `Sync.T1.T2.T2.T1`, `linearizer(T2)` and `Spec` must also perform the `Sync` event.

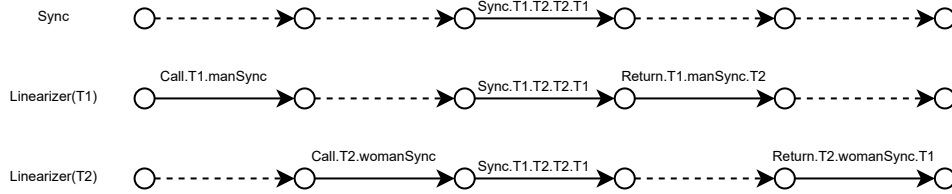


Figure 16: FDR Visualization of traces of a man process and a woman process synchronizing

All above properties suffice to show that `Linearizers(All)` generates all traces that correspond to a valid history and that `Linearizers` is a valid specification process for the testing system. Informally, for any valid history, a linearizability test can find a set of synchronization points, which corresponds to a trace of the specification for `Sync` events. Furthermore, the valid history can be obtained by adding respective call and return events around synchronization points.

Finally, we perform the test using trace refinement for safety property and failure refinement for liveness. As expected, the correct implementation

```

Lin(All,me)= (
  Call ! me!ManSync→
  Sync!me?mereturn?other?otherreturn →
  Return!me!ManSync!mereturn →
  Lin(All,me)
)□(
  Call ! me!WomanSync →
  Sync?other?otherreturn!me?mereturn →
  Return!me!WomanSync!mereturn →
  Lin(All,me)
)□STOP

LinEvents(All,me)=union({
  ev | ev←{Sync},
  let Sync.t1.a.t2.b=ev within
    countList(me,<t1,t2>)=1 and
    member(t1, All) and
    member(t2, All)
},{Call.me,Return.me})

Linearizers(All)=(|| me: All • [LinEvents(All,me)] Lin(All,me)) [|{Sync}|] Spec
\ {Sync}

```

Figure 17: Definition of linearizer process in CSP

passes all tests.

```

System2=System({T1,T2})
Spec2Thread=Linearizers({T1,T2})
assert Spec2Thread  $\sqsubseteq_T$  System2
assert Spec2Thread  $\sqsubseteq_F$  System2

```

### 3.3 A faulty version

We shall examine another MenWomen implementation in Figure 18. One key difference in this faulty MenWomen object is that it uses **Option** data in the shared variables to store the identity of the process calling **manSync** and the process calling **womanSync**.

The safeness property test shows that this implementation handles scheduling carelessly. This implementation fails the test  $\text{Spec2Thread} \sqsubseteq_T \text{System2}$ , and FDR provides a trace that violates the safeness specification, shown in Figure 19. FDR further allows the user to expand the hidden  $\tau$  events in the testing system, which are normally processes' interactions with shared variables and the monitor. By understanding the expanded trace in CSP, we can find a equivalent way to trigger the bug in Scala.

```

class FaultyMenWomen extends MenWomenT{
  private var him: Option[Int] = None
  private var her: Option[Int] = None

  def manSync(me: Int): Int = synchronized{
    while(him.nonEmpty) wait()
    him = Some(me); notifyAll()
    while(her.isEmpty) wait()
    val Some(res) = her
    her = None; notifyAll()
    res
  }

  def womanSync(me: Int): Int = synchronized{
    while(her.nonEmpty) wait()
    her = Some(me); notifyAll()
    while(him.isEmpty) wait()
    val Some(res) = him
    him = None; notifyAll()
    res
  }
}

```

Figure 18: A Faulty Scala implementation of MenWomen object

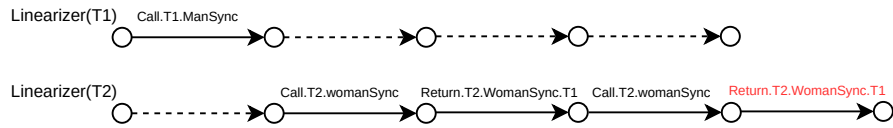


Figure 19: Trace

- A process **T1** calls **manSync**. On first line, since the shared variable **him** is initially **None**, **T1** skips the **wait**, set **him** to **Some(T1)** and waits for a process calling **womanSync**.
- A process **T2** calls **womanSync** and returns **T1**. At this stage, there is no waiting process waiting to run **womanSync** and the shared variable is not **None**. So **T2** does not wait at any point, notifies all waiting process, and returns.
- Before **T1** reenters the **synchronized** block, process **T2** calls **womanSync** again. **him** has not been reset by **T1** yet. So **T2** pairs with **T1** again, which should not be allowed.

## 4 ABC

With an ABC object, three processes can exchange data with each other two processes. More specifically, one process calling `aSync`, one process calling `bSync`, and one process calling `cSync` synchronizes. Then each of the three processes returns with the arguments of two other processes. For simplicity, we shall call a process calling `syncA` as an A-process, a process calling `syncB` as a B-process, and a process calling `syncC` as a C-process.

One of the challenges to check an ABC object is the huge number of states in the CSP model. In this section, we shall see how the linearizer process can be optimized and the efficiency of the explicit linearization point test.

### 4.1 Implementation

Figure 20 is a Scala implementation of a ABC object with semaphore. In each round of synchronization,

- Initially semaphore `aClear` is raised. An A-process acquires semaphore `aClear`, sets the shared variable `a` to its parameter, raises semaphore `bClear` and waits to acquire semaphore `aSignal`. A B-process and a C-process behaves similarly in turn, except they use different semaphores and shared variables.
- After a C-process raises semaphore `aSignal`, the A-process is able to continue. The A-process reads the shared variable `b` and `c`, raises the semaphore `bSignal`, and returns `b` and `c`. Likewise, B and C also take the value of two other shared variable and raise respective semaphores in turn.

Using the shared variable and semaphore module, it is easy to translate the Scala implementation to a CSP implementation.

### 4.2 Testing

Similar for the MenWomen object, a testing system is defined through any number of working processes and a specification is built from a `Sync` channel, linearizer processes, and a synchronization alphabet set.

One key difference of the ABC object is that processes can call with any argument from the set `TypeData`, whereas in `MenWomen` object, processes can only call with their identity. Inside `chaosP`, the processes also choose an argument with General Non-Deterministic Choice.

```

class ABC[A,B,C] extends ABCT[A,B,C]{
  // The identities of the current (or previous) threads.
  private var a: A = _
  private var b: B = _
  private var c: C = _

  // Semaphores to signal that threads can write their identities.
  private val aClear = MutexSemaphore()
  private val bClear, cClear = SignallingSemaphore()

  // Semaphores to signal that threads can collect their results.
  private val aSignal, bSignal, cSignal = SignallingSemaphore()

  def syncA(me: A) = {
    aClear.down      // (A1)
    a = me; bClear.up // signal to b at (B1)
    aSignal.down     // (A2)
    val result = (b,c)
    bSignal.up       // signal to b at (B2)
    result
  }

  def syncB(me: B) = {
    bClear.down      // (B1)
    b = me; cClear.up // signal to C at (C1)
    bSignal.down     // (B2)
    val result = (a,c)
    cSignal.up       // signal to c at (C2)
    result
  }

  def syncC(me: C) = {
    cClear.down      // (C1)
    c = me; aSignal.up // signal to A at (A2)
    cSignal.down     // (C2)
    val result = (a,b)
    aClear.up        // signal to an A on the next round at (A1)
    result
  }
}

```

Figure 20: A semaphore-based Scala implementation of the ABC object

```

instance VarA = ModuleUninitVariable(TypeData)
instance VarB = ModuleUninitVariable(TypeData)
instance VarC = ModuleUninitVariable(TypeData)

instance aClear = ModuleMutexSemaphore(TypeThreadID)
instance bClear = ModuleSignallingSemaphore(TypeThreadID)
instance cClear = ModuleSignallingSemaphore(TypeThreadID)
instance aSignal = ModuleSignallingSemaphore(TypeThreadID)
instance bSignal = ModuleSignallingSemaphore(TypeThreadID)
instance cSignal = ModuleSignallingSemaphore(TypeThreadID)

runWith(hide,p)=
  VarA::runWith(hide,
  VarB::runWith(hide,
  VarC::runWith(hide,
  aClear::runWith(hide,
  bClear::runWith(hide,
  cClear::runWith(hide,
  aSignal::runWith(hide,
  bSignal::runWith(hide,
  cSignal::runWith(hide,
    p
  )))))))

SyncA(me,avalue) =
  Call ! me ! ASync ! avalue →
  --aClear . down
  aClear::downChan ! me →
  --a = me
  VarA::setValue ! avalue →
  --bClear . up
  bClear::upChan ! me →
  --aSignal . down
  aSignal::downChan ! me →
  --(b,c)
  VarB::getValue ? b →
  VarC::getValue ? c →
  --bSignal . up
  bSignal::upChan ! me →
  --result →
  Return ! me ! ASync ! (b . c) →
  SKIP

...

```

Figure 21: Translated CSP Code for the correct ABC implementation

```

Thread(me)=chaosP(  $\sqcap$  x:TypeData • (
  SyncA(me,x)
 $\sqcap$  SyncB(me,x)
 $\sqcap$  SyncC(me,x)
))

```

The testing specification uses the same component, the **Sync** channel, linearizer processes, and `linEventns`. The definition of **Sync** channel is shown in Figure 22. The event  $Sync.t_1.a.b.c.t_2.d.e.f.t_3.g.h.i$  represents the synchronizations between three threads,  $t_1, t_2, t_3$ . Process  $t_1$  calls **aSync** with  $a$  and returns  $(b, c)$ . The second process  $t_2$  calls **bSync** with  $d$  and returns  $(e, f)$ . And the last process  $t_3$  calls **cSync** with  $g$  and returns  $(h, i)$ . The Sync specification process, shown in the same figure, checks that in each Sync events, the return value of each process is the pair of arguments of the two other function call.

Figure 23 is the definition of a linearizer process, written a similar format with the `MenWomen` object.

```

--thread identity calling ASync. ASync parameter a. ASync return pair (b,c)
--thread identity calling BSync. BSync parameter b. BSync return pair (a,c)
--thread identity calling CSync. CSync parameter c. CSync return pair (a,b)
channel Sync: TypeThreadID . TypeData . TypeData . TypeData .
              TypeThreadID . TypeData . TypeData . TypeData .
              TypeThreadID . TypeData . TypeData . TypeData

Spec = Sync ? aid ? a ? b ? c
      ? bid:diff(TypeThreadID,{aid})!b!a!c
      ? cid:diff(TypeThreadID,{aid,bid})!c!a!b
      → Spec

```

Figure 22: Definition of Sync channel and specification of Sync event

```

System3=System({T1,T2,T3})
Spec3Thread=Linearizers({T1,T2,T3})

assert Spec3Thread  $\sqsubseteq_T$  System3
assert Spec3Thread  $\sqsubseteq_F$  System3

```

#### 4.2.1 Speeding up model compilation

Consider the specification process with three processes. Let  $M$  be the size of the set of all possible arguments. Consider the trace in Figure 24, where process **T1** calls **aSync** with **A**, **T2** calls **bSync** with **B**, **T3** calls **cSync** with **C**. Then they synchronization.



```

--Linearizer for a process
Lin(All,me)=(
  --me synchronizes as thread A
  Call ! me!ASync? a →
  Sync! me! a? b? c? t2:diff(All,{me})? t2b? t2a? t2c? t3:diff(All,{me,t2})? t3c? t3a? t3b →
  Return! me!ASync! b! c →
  Lin(All,me)
) □ (
  --me synchronizes as thread B
  Call ! me!BSync? b →
  Sync? t2:diff(All,{me})? t2b? t2a? t2c! me! b? a? c? t3:diff(All,{me,t2})? t3c? t3a? t3b →
  Return! me!BSync! a! c →
  Lin(All,me)
) □ (
  --me synchronizes as thread C
  Call ! me!CSync? c →
  Sync? t2:diff(All,{me})? t2b? t2a? t2c? t3:diff(All,{me,t2})? t3a? t3b! me! c? a? b →
  Return! me!CSync! a! b →
  Lin(All,me)
)

```

Figure 23: Definition of linearizer process

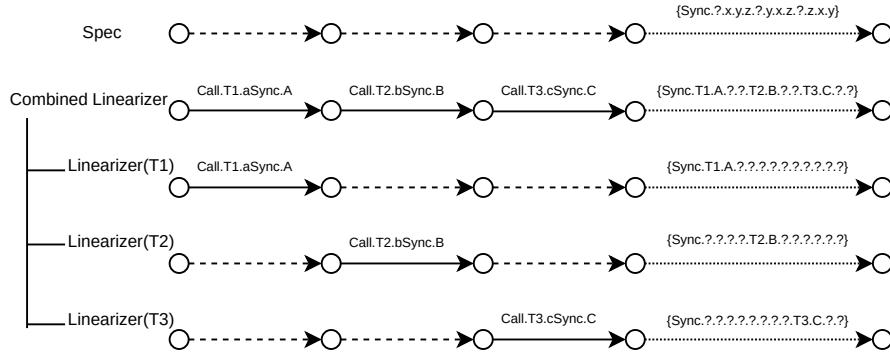


Figure 24: The set of possible Sync event for each CSP process

The last transition in the diagram is the **Sync** event between three processes. Above the edge is the set of possible **Sync** event that every process accepts. Each linearizer accepts  $3^2 * M^8$  possible **Sync** event. The combined linearizer accepts  $M^6$  sync event. However, according to the sync specification, only one **Sync** event is valid.

With the above analysis, it is tempting to reduce the redundancy in **Sync** event. Optimize the linearizer process by using the information from the specification process. Instead of choosing all possible remaining arguments, the individual linearizer could choose correct arguments according to the speci-

fication process. Figure 25 includes part of the simplified code. This change does not reduce the number of transitions in the resulting specification, but it helps FDR build the process faster.

With this optimization, the testing for less than 5 processes finishes quickly.

```

channel Sync: TypeThreadID . TypeThreadID . TypeThreadID .
                TypeData . TypeData . TypeData

Lin(All,me)= (
  Call ! me!ASync? a →
  Sync! me? t2:diff(All,{me})? t3:diff(All,{me,t2})! a? b? c →
  Return! me!ASync! b! c →
  Lin(All,me)
) ...

```

Figure 25: Simplified definition of Sync channel and part of simplified linearizer

### 4.3 Faulty version

Recall that in Java and Scala, raising a semaphore immediately allows another thread waiting to acquire the semaphore to continue. So it is essential to take a copy of the two other arguments before raising the semaphore.

On the other hand, what if the implementation of `syncA` does not take a copy of the argument? It turns out that the faulty `ABC` object passes tests for three processes but fails the linearisation test with at least four threads.

#### 4.3.1 Explanation of the error case

For the test `Spec4Thread  $\sqsubseteq_T$  System4`, FDR displays a trace of the testing system that violates the specification. From the trace, it seems that process `T1` synchronizes with `T2` and `T3` in the first round, and should return `(B,C)`, but `(E,F)`, the argument in the second round is returned. Expanding the  $\tau$  event and translating CSP traces into program traces makes it possible to see what goes wrong in the faulty version when there are four threads.

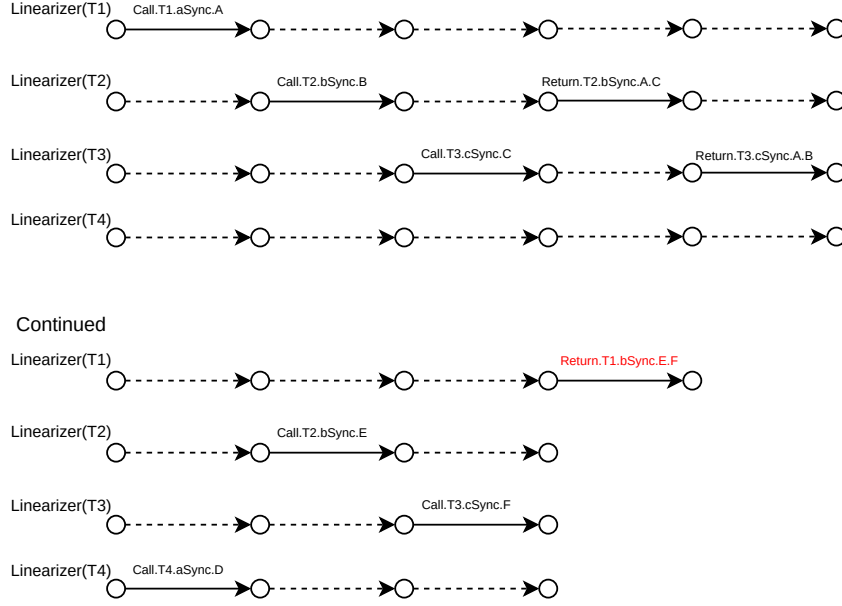


Figure 26: Trace that violates the specification

- In the first round of synchronization, process  $T_A$ ,  $T_B$ ,  $T_C$  call **aSync**, **bSync** or **cSync** respectively, and put down its argument in turn.
- Process  $T_A$  raises **bSignal**. Before  $T_A$  exits, the other two processes  $T_B$ ,  $T_C$  returns. Now  $T_A$  should return argument of  $T_B$  and  $T_C$ .
- Another round of synchronization starts. Thread  $T_D$ ,  $T_B$ ,  $T_C$  call **aSync**, **bSync** or **cSync** respectively, and overwrites the shared variable **a,b,c** in turn.
- Now  $T_A$  returns with  $(b, c)$  from the second round, which may not be the argument of **bSync** and **cSync** in the first round.

#### 4.4 Explicit linearization point test

In this section, we describe another fast testing technique. In explicit synchronization test, one first understands the implementation of the object and finds synchronization point, where synchronization occurs between processes. Then based on the observation, one writes the specification for the these synchronization point. Finally, one checks that a testing system with only LP events visible refines the specification for LP.

From the source code of the ABC object, synchronizations occur in six places. The first three synchronizations occur when processes set the respective shared variables before raising semaphores. The other synchronizations

occur when the processes read the shared variables after lowering semaphores. For every synchronization, the respective CSP lines are replaced with a **LP** event and the shared variable module is adapted to listen on the **LP** event. Figure 27 is the definition of LP events and the implementation of **SyncA**.

```

SyncA(me, avalue) =
  --aClear . down
  aClear :: downChan ! me →
  --a = me
  LP1 ! me ! avalue →
  --bClear . up
  bClear :: upChan ! me →
  --aSignal . down
  aSignal :: downChan ! me →
  --(b,c)
  LP4 ! me ! bvalue ! cvalue →
  --bSignal . up
  bSignal :: upChan ! me →
  SKIP

```

Figure 27: aSync function in explicit linearization point testing

The specification for the **LP** events is simple. As described earlier in the section, the synchronization between process occurs in a fixed order. Initially only A-process is able to proceed to set the shared variable, then the B-process and the C-process can proceed in turn. In the second round the A-process, B-process, C-process reads the shared variables in turn. Figure 4.4 is the specification process of **LP** events.

```

spec =
  LP1 ? t1 ! t1 →
  LP2 ? t2 ! t2 →
  LP3 ? t3 ! t3 →
  LP4 ? t1 ! t2 ! t3 →
  LP5 ? t2 ! t1 ! t3 →
  LP6 ? t3 ! t1 ! t2 →
  spec

```

Immediately we can see the specification process is much simpler than the process in linearization testing. But how fast is the explicit synchronization test? We use FDR to find the size of the state machine of the specification and testing system when three processes are involved. The sizes are organized in the table below.

| Linearization test |               |                | Explicit Synchronization test |                |
|--------------------|---------------|----------------|-------------------------------|----------------|
|                    | Specification | Testing System | Specification                 | Testing System |
| States             | $4 * 10^6$    | $4 * 10^5$     | $2 * 10^3$                    | $1 * 10^5$     |
| Transitions        | $1 * 10^7$    | $2 * 10^6$     | $1 * 10^4$                    | $6 * 10^5$     |

As we can see from the table, the state machine of the specification process is much smaller in the explicit synchronization test. It is no surprise because the specification process in the explicit synchronization test is a simple recursive process. The state machine of the testing system is slightly smaller because the **Call** and **Return** events are removed in the explicit linearization test, and code execution is never interrupted by another process's return.

However, the drawback of the explicit synchronization test is that it needs to be done on a per implementation basis. In the faulty implementation, the assumption on synchronization has changed. So the testing for correct implementation cannot be directly applied to the faulty implementation.

## 5 Terminating Queue

A terminating queue provides thread-safe enqueue and dequeue operations. Suppose a process dequeues when the queue is empty. The process blocks and waits for some process to enqueue. In addition, if all processes are dequeuing, no new element can be added to the queue and the whole system deadlocks. In this case, the queue terminates and returns `None` for every process. Figure 28 is the interface of a terminating queue.

```
trait TerminatingQueueT[A]{
  def enqueue(x: A): Unit
  def dequeue: Option[A]
}
```

Figure 28: Scala interface of terminating queue

A terminating queue is a stateful synchronization object, as earlier enqueue and dequeue operations affect later synchronizations. The timeline diagram below is invalid because the second `dequeue` call should return 1 instead of 2.

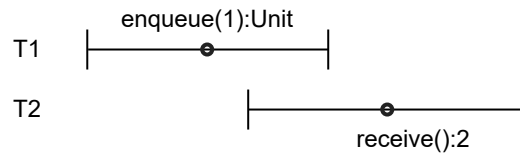


Figure 29: A faulty timeline of termianting queue

### 5.1 Implementation

The Scala implementation of the Terminating Queue wraps a Scala Queue with a monitor. Upon creation, the object is given `numWorkers`, the number of processes using the terminating queue. In addition to an internal queue `queue`, the terminating queue uses two more shared variables. The shared variable `waiting` is the number of processes waiting to dequeue an element from the queue. The shared variable `done` indicates if the queue has terminated. If the value of `done` is true, then any further function call should do nothing.

Enqueue is a trivial operation. In the synchronized block, the process adds the element to the internal queue and notifies a process waiting to dequeue an element. Dequeuing is also trivial when the internal queue is not empty. The process simply performs `dequeue` on the internal queue. When the queue is empty, if the value of `waiting` is `numWorkers-1`, then all processes are

now waiting to dequeue, so the queue should terminate. The process notifies all waiting processes and returns **None**. Otherwise, the process increments the counter **waiting** and waits. The waiting process may be wakened for two cases. In the first case, a new element is added to the queue. The process decrements **waiting** and returns the queue head as normal. In the second case, the process is wakened because the queue is terminating, and the process should return **None**.

```
class TerminatingQueue[A](numWorkers: Int){
  private val queue = new Queue[A]
  private var waiting = 0
  private var done = false

  def enqueue(x: A) = synchronized{
    if(!done){
      queue.enqueue(x)
      if(waiting > 0) notify()
    }
  }

  def dequeue: Option[A] = synchronized{
    if(!done && queue.isEmpty){
      if(waiting == numWorkers-1){ // System should terminate
        done = true; notifyAll()
      }
    }
    else{
      waiting += 1
      while(queue.isEmpty && !done) wait()
      waiting -= 1
    }
  }
  if(done) None else Some(queue.dequeue)
}
```

Figure 30: Scala implementation of terminating queue

A few workarounds are required to keep the state machine of the CSP model finite and analyzable for FDR. First, the range of variable **waiting** is limited to integers from 0 to **numWorkers** inclusive, instead of all  $2^{32}$  possible values from a Scala integer. However, while building the state machine, FDR attempts to set the value of **waiting** to  $-1$  and **numWorkers+1** and throws an error. Even though the value of **waiting** will never be  $-1$  or **numWorkers+1** in the correct implementation, FDR is unable to derive such information while building the state machine for **dequeue** when **waiting** is 0 or **numWorker**. As a workaround, we guard **waiting** settings with an if statement in Figure 32. Before setting the value of **waiting**, the process checks if the new value is in range. If not, the process diverges. In the testing system, we check that system is divergence free to show that **waiting** is always in range as expected.

Secondly, the Scala implementation uses a Scala Queue internally, which has unlimited capacity and cannot be modelled with finite states in CSP. To address this, we use a capacity-limited queue which behaves the same as the infinite queue when the queue is not full. However, if a process enqueues a new element when the internal queue is full, the process **STOP** or **DIV** according to what is required for testing. In Figure 31, we implement the capacity-limited queue with many functions, The interface may be slightly overkilling for a capacity-limited queue, but it is necessary for a more powerful finite-state queue described in later subsections.

```

--Maximum number of element in the queue
NQueueCapacity=3
--Set of states of the capacity-limited queue
TypeQueue={q|i←{0..NQueueCapacity},q←ArrangementInList(i,TypeData)}
--Returns if the queue is empty
qEmpty(q)=q=<>
--Returns an empty queue
qNewQueue=<>
--Given a queue q, return a set of element that can be enqueued to queue q
qValidEnqueue(q)=if length(q)≥NQueueCapacity then {} else TypeData
--Given an element x, return a set of queues that accept an enqueue of x
qCanEnqueue(x)={q|q←TypeQueue, length(q)<NQueueCapacity}
--Return the state of the queue after enqueueing x
qEnqueue(q,x)=q^<x>
--Return the set of all possible result and state pair after dequeueing
qDequeue(<>)={}
qDequeue(<qhead>^qtail)={(qhead,qtail)}

```

Figure 31: CSP Implementation of capacity-limited queue

## 5.2 Linearization Testing

We use similar approaches to construct the testing system and the specification. For the terminating queue object, there are three synchronizations between a process and the object. When a process enqueues or successfully dequeues, it synchronizes with the object and acts on the internal queue. For **enqueue** and **dequeue**, the synchronization is represented by a Sync event with the identity of the process, an object representing the function call, and the return value. When the queue shuts down and a process returns **None**, the process synchronizes with all other processes. In this case, the synchronization is represented by **SyncShutdown**.

Since a terminating queue is a stateful synchronization object, the specification of **Sync** event is different from the earlier **Sync** specification, in that the specification uses a parameter to keep the current queue. For both queues,



```

instance Monitor=ModuleMonitor(TypeThread, True)
instance VarQueue=ModuleVariable(TypeQueue,qNewQueue)
instance VarWaiting=ModuleVariable({0..5},0)
instance VarDone=ModuleVariable(Bool,false)

enqueue(NThread,me,x)=
  Call ! me! (EnqueueCall . x) →
  Monitor::synchronized(me,
    --if(!done)
    VarDone::getValue?done →
    if (not done) then (
      --queue.enqueue(x), block until the queue is not full
      VarQueue::getValue? q →
      if (not member(x,qValidEnqueue(q))) then QUEUE_ERROR_ACTION
      else (
        VarQueue::setValue! (qEnqueue(q,x)) →
        --if(waiting > 0) notify()
        VarWaiting::getValue? waiting →
        if (waiting>0) then Monitor::notify(me) else SKIP
      )) else SKIP
    );
  Return ! me! (EnqueueReturn) →
  SKIP

```

Figure 32: CSP implementation of the terminating queue - Enqueue Part

```

dequeue(NThread,me)=
  Call ! me!(DequeueCall) →
  Monitor::enter(me);
  --if(!done && queue.isEmpty)
  VarDone::getValue? done →
  VarQueue::getValue? q →
  ( if(not done and qEmpty(q)) then (
    --if(waiting = numWorkers-1)
    VarWaiting::getValue? x→
    if(x≥NThread) then DIV
    else if(x=NThread-1) then(
      --done=true
      VarDone::setValue!true→
      --notifyAll
      Monitor::notifyAll(me)
    ) else(
      --waiting+=1
      --if(x=NThread) then div else
      VarWaiting::setValue!(x+1) →
      --while(queue.isEmpty && !done) wait()
      Monitor::whileWait(me, \ ktrue,kfalse •
        VarQueue::getValue? q →
        VarDone::getValue? done →
        if(qEmpty(q) and not done) then ktrue else kfalse
      );
      --waiting -= 1
      VarWaiting::getValue? y→
      if y=0 then DIV else
      VarWaiting::setValue!(y-1)→
      SKIP
    )
  ) else SKIP);
  --if(done) None else Some(queue.dequeue)
  VarDone::getValue? done →
  if (done) then (
    Monitor::exit(me);
    Return ! me!(DequeueReturnNone) →
    --the thread should stop any work
    STOP
  ) else (
    VarQueue::getValue? q →
    if qEmpty(q) then DIV else
    □ (ans, qtail): qDequeue(q) •
    VarQueue::setValue! qtail →
    Monitor::exit(me);
    Return ! me!(DequeueReturn . ans) →
    SKIP
  )
)

runWith(hideSpurious,hide,P)=
  VarQueue::runWith(hide,
  VarWaiting::runWith(hide,
  VarDone::runWith(hide,
  Monitor::runWith(hideSpurious,hide,
  P
  )))

```

Figure 33: CSP implementation of the termianting queue - Dequeue Part

the **Sync** specification should ensure that elements are added and removed in a First-In-First-Out order, and all enqueue and dequeue operation are valid for the queue. **Sync** specification does not check **SyncShutdown** however. Figure 34 shows the definition of **Sync** event and the specification.

```

channel Sync: TypeThread . TypeCallParam . TypeReturnParam
channel SyncShutDown

Spec(q)=
  (qValidEnqueue(q)≠{} & □ x:qValidEnqueue(q) • (
    Sync ? t ! (EnqueueCall . x) ! EnqueueReturn →
    Spec(qEnqueue(q,x))
  ))□
  (qDequeue(q)≠{} & □ (x,newq):qDequeue(q) • (
    Sync ? t ! (DequeueCall) ! (DequeueReturn . x) →
    Spec(newq)
  ))□(
    qEmpty(q) & SyncShutDown → STOP
  )

```

Figure 34: Definition of Sync event and the specification

In addition to the safeness and liveness we usually test, we first check that the shared variable **waiting** is actually in the range 0 to **numWorkers**. We set spurious wakeup visible and invalid **enqueue** to cause **STOP**. If the testing system never diverges, it must be that a process always set a value from 0 to **numWorkers** for the variable **waiting**.

For safeness and liveness testing, we use the normal setup, in which the spurious wakeup is invisible and the process diverges when it performs an invalid **enqueue** operation.

### 5.3 Faulty Implementation

In this section, we use the linearization test to distinguish the correct implementation with three faulty implementations. In the first faulty implementation, **enqueue** operation always adds **A** to the internal queue regardless of the parameter of **enqueue**. In the second faulty implementation, **enqueue** method **notify** does not wake up a process waiting to dequeue. The third implementation has a miss-by-one error. In **dequeue**, A process checks **waiting = numWorker** before incrementing **waiting**. The result is shown in the table below.

|         | Divergence Free | Trace Refinement | Failure Refinement |
|---------|-----------------|------------------|--------------------|
| Correct | Pass            | Pass             | Pass               |
| Faulty1 | Pass            | Fail             | Fail               |
| Faulty2 | Pass            | Pass             | Fail               |
| Faulty3 | Pass            | Pass             | Fail               |

With no surprise, the correct implementation passes all tests. The first faulty implementation fails all tests. The second faulty implementation fails only when spurious wakeups are disabled, as the missing `notify` can be compensated by some "coincidental" spurious wakeup. It shows that when testing an object that internally uses a monitor, one should test it with spurious wake and without spurious wakeups from the monitor. The third faulty implementation shows a similar pattern. Because of the miss-by-one error, the last process will not terminate the queue. As a result, all processes refuse to return. However, as spurious wakeups can cause divergence, the failure model is unable to capture this error.

## 5.4 Test with another queue

In this section, we apply the same tests with a different finite-state queue, which has infinite capacity but requires the element enqueued in  $A^*BC^*$  form. The queue is adapted from [Low22]. If the test succeeds, the queue should not duplicate an element, reorder the enqueued element or miss an element with infinite data.

When the  $A^*BC^*$  queue accepted some **A** but no **B**, the queue can enqueue an **A** or a **B**, but not a **C**. When the queue accepted a **B** or a **C**, the queue can only enqueue a **C** element.

Dequeue operation can be non-deterministic in this queue. For example, the state `QAsB0C` represents a queue with zero or more **A** followed by a **B**. A queue in state `QAsB0C` accepts an dequeue of **A** and an dequeue of **B**. The state of the  $A^*BC^*$  queue is further split into empty states and non-empty states. For example, the state `Q0C` is an empty state and rejects all dequeue operations. As shown in Figure 35, the implementation of the  $A^*BC^*$  queue is written in the same interface as the capacity-limited queue.

When the divergence free test, trace refinement test and the failure refinement test is applied on a system using the  $A^*BC^*$  queue, the test result shows a same pattern as earlier test on capacity limited queue.

```

datatype TypeQueue= Q0A | QAAs | QAsB0C | QAsBCCs | Q0C | QCCs
qEmpty(Q0A)=True
qEmpty(Q0C)=True
qEmpty(_)=False

qNewQueue=Q0A

qEnqueue(Q0A, A)=QAAs
qEnqueue(Q0A, B)=QAsB0C
qEnqueue(QAAs, A)=QAAs
qEnqueue(QAAs, B)=QAsB0C
qEnqueue(QAsB0C, C)=QAsB0C
qEnqueue(QAsBCCs, C)=QAsB0C
qEnqueue(Q0C, C)=QCCs
qEnqueue(QCCs, C)=QCCs

qValidEnqueue(Q0A)={A,B}
qValidEnqueue(QAAs)={A,B}
qValidEnqueue(_)= {C}

qCanEnqueue(x)={q | q←TypeQueue, member(x,qValidEnqueue(q))}

qDequeue(QAAs)= {(A,Q0A),(A,QAAs)}
qDequeue(QAsB0C)={ (A,QAsB0C),(B,Q0C)}
qDequeue(QAsBCCs)={ (A,QAsBCCs),(B,QCCs)}
qDequeue(QCCs)={ (C,Q0C),(C,QCCs)}
qDequeue(_)= {}

```

Figure 35: CSP Implementation of capacity-limited queue

## 6 Other objects

There are several synchronization object we tested in the project, but testing on these object are not presented in this project due to the word limit. In this section we briefly describe their interface. Interested readers can refer to the appendix section for the CSP code of these objects.

Barrier is another synchronization object commonly used in concurrent programming. A `sync` call returns only after all processes using the barrier calls `sync`.

```
trait BarrierT{
  def sync: Unit
}
```

With the exchanger object, processes can exchange data with another process. Like the `MenWomen` object, processes pair up and return the data of the other process.

```
trait ExchangerT[A]{
  /** Exchange x with another thread. */
  def exchange(x: A): A
}
```

Filterchan is like a channel but a process can specify what data it wants to receive by providing a function as a parameter in `receive`.

```
trait FilterChanT[A]{
  /** Send x on the channel. */
  def send(x: A): Unit

  /** Receive a value that satisfies p. */
  def receive(p: A => Boolean): A
}
```

## 7 Conclusion

The synchronization test technique can be used to test the correctness of a synchronization object given its interface and expected behaviour. In the thesis, we describe and apply the linearization test to many synchronization objects, and these tests can distinguish between the correct and faulty implementation of the object. We show how to optimize the linearization test by reducing the redundancy in the linearizer process and compare the optimized linearization test with the explicit linearization point test. With all the techniques, we apply the linearization test to a complicated object.

The linearization test has some drawbacks. The complexity of the testing system usually grows exponentially regarding the number of processes and the size of the variable used. As a result, we usually use a small set of processes and variable values, which usually suffices to find a counterexample trace in a faulty system. Besides, sometimes reducing the variable size has no side effects. In the MenWomen object section, the variable `stage` is declared as a Scala integer, but only value 0,1,2 is used.

Also, some of the synchronization objects tested in the thesis are more like artificially created objects for concurrent teaching.

For future work, one can build CSP models for synchronization objects from other sources, such as JVM Source code. When more CSP modules become available, it will also be interesting to build a compiler from Java to CSP to enable large scale automatic testing of synchronization objects. Finally, one can look into improving linearization testing further to allow efficient testing of large systems.

## 8 Reference

### References

- [Lea06] Douglas Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2006.
- [Low17] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.
- [Low22] Gavin Lowe. Parameterized verification of systems with component identities, using view abstraction. *International Journal on Software Tools for Technology Transfer*, 24(2):287–324, Apr 2022.
- [Ros11] A.W. Roscoe. *Understanding Concurrent Systems*. 01 2011.
- [SI09] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.



## 9 Appendix