

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 primitive 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

1	Introduction	3
1.1	Thesis Overview	3
1.2	Synchronization linearizability test	4
1.3	Checking safety property using CSP	5
1.4	Checking liveness property using CSP	6
1.5	Related work	6
2	Common Objects	7
2.1	Shared Variable	7
2.2	Semaphore	9
2.3	Monitor	9
2.3.1	JVM Monitor	9
2.3.2	Monitor Module	11
3	MenWomen	15
3.1	Implementation	15
3.2	Linearization Test	18
3.3	A faulty version	20
4	ABC	23
4.1	Implementation	23
4.2	Testing	23
4.2.1	Speeding up model compilation	26
4.3	Faulty version	29
4.3.1	Explanation of the error case	29
5	Terminating Queue	30
5.1	Implementation	30
5.2	Linearization Testing	33
5.3	Linerization Point Test	33
5.4	Faulty Implementation	33
6	Conclusion	34
7	Reference	35

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, the 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 MenWomen 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 between 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 [?]. The linearizability testing framework logs the orders of each function call and function return. Then for the observed history, the testing framework attempt to find a series of synchronization point that obeys 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.

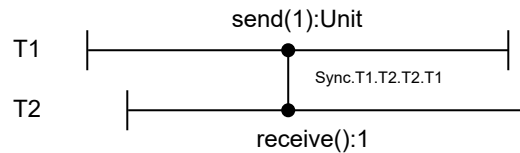


Figure 2: Visualized history of T1 calling `send(1)` and T2 calling `receive()`

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.

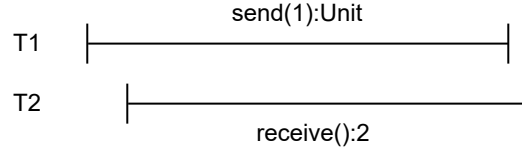


Figure 3: Visualized history of T1 sends 1 but T2 receives 2

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.

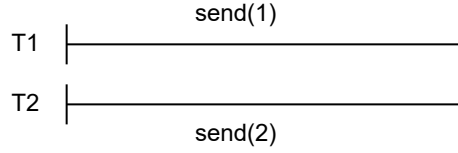


Figure 4: Visualized history of both T1 and T2 calling send

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 to find a valid series of synchronization points.

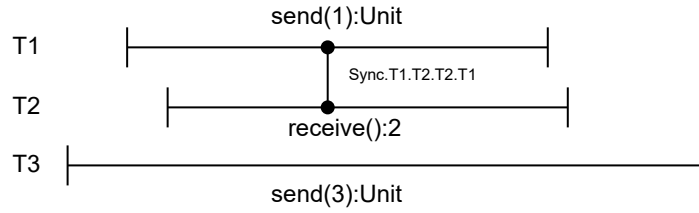


Figure 5: Visualized history of T3 get descheduled

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 [?] presents a framework to test concurrent datatypes. However, because the testing framework uses observations of histories, it is unlikely to exhaust all possible histories of a system.

In Chapter 19 of Understanding Concurrent Systems [?], the author describes a CSP model for shared variables and provides a tool to analyze shared variable programs. But the tool lacks support for objects frequently used in concurrent programming, such as monitors and semaphores.

There are also runtime programming tools to detect race conditions and deadlocks in concurrent code. Thread Sanitizer [?] detects race conditions and deadlocks in C++ and Go.

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`.

```
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

```

--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

2.2 Semaphore

A Semaphore is a simple but powerful concurrent primitive. This thesis shall describe and use a simplified binary semaphore from [TODO: Reference], 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 `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.

2.3 Monitor

2.3.1 JVM Monitor

A Monitor is another powerful concurrent primitive. This thesis will also use a simplified monitor from [TODO:reference]. 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

```

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 [] chanset [] P) \
    (if hide then chanset else {})
  channel upChan, downChan: TypeThreadID
endmodule

```

Figure 8: The binary semaphore module in CSP

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 `spuiousWake` 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`

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


```

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

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))
```

Figure 16: Definition of helper function `chaosP` and processes in the testing system

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))
```

Figure 17: Definition of CSP processes in the testing system

Figure 18 is the definition and specification of `Sync` event used internally in the testing specification. 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, we check 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.

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

Figure 18: Definition of `Sync` channel

A linearizer process alone generates all possible `Call` and `Return` events with `Sync` events in between given the identity of process. For `MenWomen` object, the linearizer is implemented with three non-deterministic choice branches, shown in Figure 22. In the first branch, the linearizer calls `manSync`, synchronize with another linearizer process calling `womanSync`, and returns with the identity it received from the `Sync` event. In the second branch, the linearizer performs `womanSync`. In the last branch, the linearizer terminates. This construction ensures two properties: the return value of a function call comes

from the synchronization, and the synchronization point occurs sometime during the function call.

Linearizers(All) creates a linearizer process for every process and puts these linearizers in parallel using replicated generalized parallel in CSPM. 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. In the case of **MenWomen**, when a linearizer for **T1** wishes to perform a **Call** or a **Return** event, the linearizer does not need to synchronize with any processes because no linearizer can call using others' identity. However, when **linearizer(T1)** wishes to perform a **Sync** event such as **Sync.T1.T2.T2.T1**, **linearizer(T1)** needs to synchronize with **linearizer(T2)** because the latter process can perform this **Sync** event. Such construction ensures one more property: a **Sync** event must be performed by corresponding processes at the same time. The combined process is then put to run in parallel with specification for the **Sync** event, shown earlier in Figure 18, to ensure all **Sync** event represents valid synchronizations. Finally, like the testing system, all **Sync** events are hidden, leaving **Call** and **Return** in **Linearizers**.

Figure 19 visualizes trace generated by **1,T2Linearizers(T)** in a FDR way. 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.

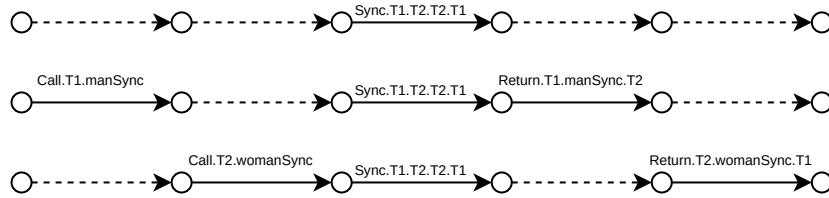


Figure 19: FDR Visualization of traces of a man process and a woman process synchronizing

Figure 20 is another FDR visualization of traces generated by **Linearizers**. In this example, all two linearizers choose to call **manSync** and no **Sync** event is possible between the two linearizers.

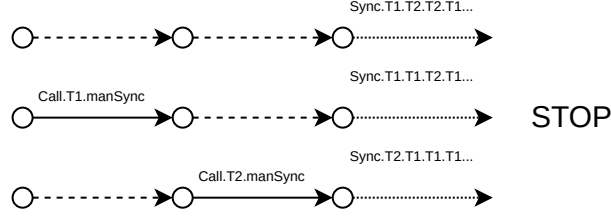


Figure 20: FDR Visualization of traces of two man processes deadlocking

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 passes all tests.

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

Figure 21: Part of liveness and safetyness test in CSP for MenWomen object

3.3 A faulty version

We shall examine another MenWomen implementation in Figure 23. 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 `Spec2Thread \sqsubseteq_T System2`, and FDR provides a trace that violates the safeness specification, shown in Figure 24. FDR further allows the user to expand the hidden τ 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 an equivalent way to trigger the bug in Scala.

```

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 22: Definition of linearizer process in CSP

```

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 23: A Faulty Scala implementation of MenWomen object

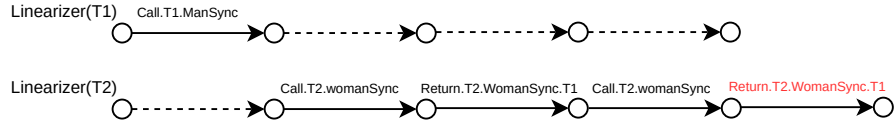


Figure 24: Trace

- 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 25 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. As shown in Figure 27 So 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 25: 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 26: 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)
))

```

Figure 27: Definition of processes in the testing system.

The testing specification uses the same component, the **Sync** channel, linearizer processes, and linEventns. The definition of **Sync** channel is shown in Figure 28. The event *Sync.t₁.a.b.c.t₂.d.e.f.t₃.g.h.i* represents the synchronizations between three threads, *t₁*, *t₂*, *t₃*. Process *t₁* calls **aSync** with *a* and returns (*b*, *c*). The second process *t₂* calls **bSync** with *d* and returns (*e*, *f*). And the last process *t₃* 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 29 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 28: 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

```

Figure 30: Part of test for ABC object. This tests a system with three processes.

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 31, where

```

--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 29: Definition of linearizer process

process T1 calls `aSync` with A, T2 calls `bSync` with B, T3 calls `cSync` with C. Then they synchronization.

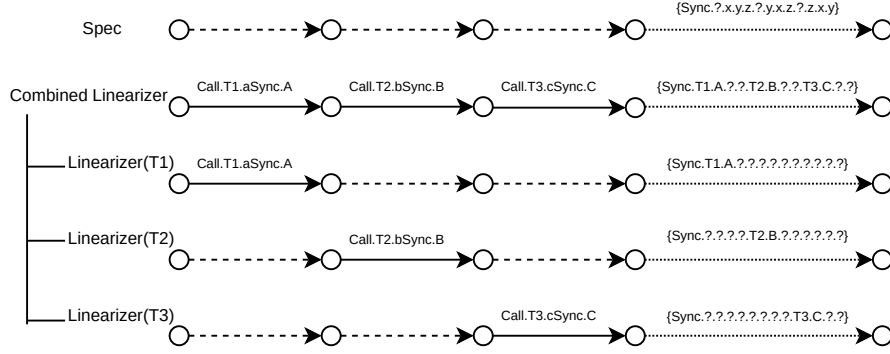


Figure 31: The set of possible Sync event for each CSP process

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 specification process. Figure 32 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 32: 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 τ 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.

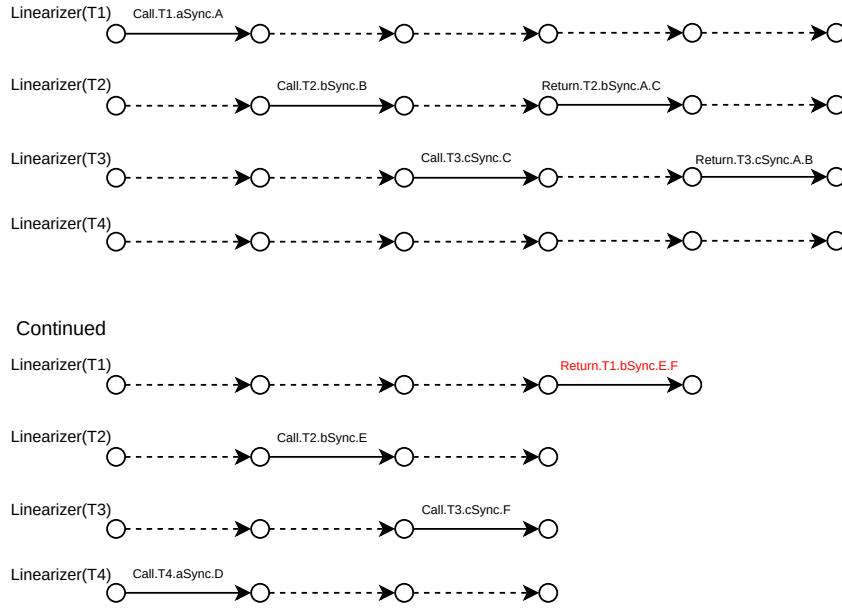


Figure 33: Trace that violates the specification

- 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.

5 Terminating Queue

A terminating queue provides thread-safe enqueue and dequeue operations. If a process dequeues when the queue is empty, the process blocks and waits for some process to enqueue. In addition, if all processes dequeue when the queue is empty, the queue terminates and returns **None**. Figure 34 is the interface of a terminating queue.

```

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

```

Figure 34: Scala interface of terminating queue

A terminating queue is a stateful synchronization object, as earlier enqueue and dequeue operation affect later synchronizations. The first diagram below

5.1 Implementation

The Scala implementation of the Terminating Queue wraps a Scala Queue with a monitor. Upon creation, the object is given the number of processes using the queue **numWorkers**. In addition to a 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 false, 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. Dequeueing is also trivial when the internal queue is not empty. The process simply perform dequeue on the internal queue. When the queue is empty, if the value of **waiting** is $n - 1$, then all processes are now waiting to dequeue, so the queue should terminate. The process notifies all waiting process and returns **None**. Otherwise the process increments the counter **waiting** and waits. The waiting process may be waken because a new

element is added to the queue, all n processes are waiting, and spurious wake up. In the first case, the process decrements `waiting` and returns the queue head as normal. In the latter case, the process returns `None`.

//6559

There are a few workarounds required for the CSP implementation, because the set of possible value should be finite so that the communication graph FDR computes is finite and analyzable. For this reason, the range of variable `waiting` is limited to integers from 0 to `numWorkers` inclusive. However, while building the communication graph, FDR attempts to set the value of `waiting` to `-1` and `numWorkers+1` and errors. Even though the value of `waiting` will never be `-1` or `numWorkers+1` in both the Scala implementation and CSP implementation, FDR is unable to derive such information while building the communication graph for one process.

To workaround this, we guard `waiting` settings with a if statement. A CSP process diverges if it attempts to set a value outside 0 to `numWorkers`. In the testing system, we check that system is divergence free to show that `waiting` works as expected.

Second, the Scala implementation uses a Scala Queue internally, which does not have a capacity limit and results to a infinite choices of queues in CSP. To address this, we use two finite-state queues that captures some properties of a infinite queue. Both queue has extra restrictions on the enqueue operations. When the constraint is violated, the process `STOP` or `DIV` according to what is required in testing. Figure ref is the implementation of the two queue, and the two queue are written under the same interface to allow code reuse.

The first queue is a capacity-limited queue and tests , which blocks a process when a process tries to the queue has more element than capacity. The first queue tests traces where the size of queue never exceeds some number.

The second queue is an infinite queue but it only allows enqueue operation in the form of A^*BC^* . The idea comes from [TODO:Reference] and a system with this queue tests that the terminating queue does not miss or duplicate an element in the queue. The transition of the queue is presented in Figure ref. If a process calls dequeue, the queue non deterministically chooses one dequeue outcome and returns to the process. For example, in the state A^*B0C where the queue has zero or more A, followed by one B and no C, the queue can non deterministically dequeue an A. Or the queue can choose to dequeue an B and transites the the state $Q0C$.

```

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 35: Scala implementation of terminating 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 ways a process synchronizes using the object. When a process enqueues or dequeues, it synchronizes with the object and perform action on the internal queue. In the enqueue and dequeue case, the synchronization is represented by a `Sync` event with the identity of the process, an object representing the function call the return value. When the queue shutdowns 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 with earlier `Sync` specification. For both queue, the `Sync` specification should ensure that elements are added and removed in a FIFO order, and all enqueue and dequeue operation are valid for the queue. `Sync` specification does not check `SyncShutdown` however. Because In addition the `Sync` specification should ensure no enqueue operation is allowed when the queue is full. For the second queue, the `Sync` specification should ensure the element enqueued are in the form of A^*BC^* . Figure ref is the specification for the first queue, which is implemented by a process holding the queue in the parameter. Figure ref is the specification for the second queue. The specification is splitted to several phases, with `Spec1` specifying the synchronization before B is added, `Spec2` specifying synchronization before B is dequeued, `Spec3` specifying the

[TODO: Update Code] Before testing the safety and liveness property, we check that the shared variable `waiting` is actually in the range 0 to `numWorkers`, by checking the testing system is divergence free when spurious wakeup are visible and invalid enqueue causes a process to `STOP` instead of `DIV`. For safety and liveness property, process diverges when they perform a invalid enqueue, so that they reject any operations

5.3 Linearization Point Test

5.4 Faulty Implementation

In this section we use the linearization test to distinguish the correct implementation with three faulty implementation of the terminating queue. In the first faulty implementation, element `A` is always added to the internal queue regardless of the parameter of enqueue function. In the second faulty implementation, `enqueue` method does not use `notify` to wake up waiting dequeuing processes. In the third implementation has a miss-by-one error.

The termination condition becomes that value of `waiting` is `numWorkers` before incrementing the variable.

The four implementation is tested under different combinations, and the result is listed in the table below.

	Divergence Free	Trace Refinement		Failure Test	
		Disable Spurious	Allow Spurious	Disable Spurious	Allow Spurious
Correct	Pass	Pass	Pass	Pass	Pass
Faulty1	Pass	Fail	Fail	Fail	Fail
Faulty2	Pass	Fail	Pass	Fail	Pass
Faulty3	Pass	Fail	Pass	Fail	Pass

With no surprise, the correct implementation passes all tests. The first faulty implementation fails all tests. The second faulty implementation fails only when spurious wakeup are disabled, as waiting processes can "coincidentally" spuriously wake up after one process enqueues. It shows that when testing a object that internally uses a monitor, one should test the object with spurious wake and without spurious wake. The third faulty implementation shows a similar pattern. With the new condition, 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.

6 Conclusion

In the thesis we describe and apply linearization test to a number of synchronization objects, and these tests are able to distinguish correct and faulty implementation of the object. We also use explicit linearization point test and find explicit linearization points are faster because they do not use `Call` and `Return` event. We show optimize linearization test by reducing the redundancy in linearizer process and sync event. With all the knowledge we apply test to a complicated object. There are also more object we test for the project, but not presented in thesis. Interested readers can see the appendix section for the CSP code and assertion for these objects.

Linearization test has some drawbacks. Limited by the complexity of testing system, the CSP model often use a different and smaller set of values compared to the actual implementation. For example, function converted from Scala synchronization object often accept a small set of data instead of the whole integer set. In the earlier section, the variable `waiting` in terminating queue also used the exact range instead of the whole set of integer. This also applies to the number of processes in the system. When a system uses more than five processes, the refinement checking often takes a long time. However, a faulty system often fails at small scale and small ranges of values, so this is less likely to be a problem.

Also, some of the synchronization object tested in the thesis are more like artificially created objects for concurrent teaching. For future work, one can build CSP models for concurrent and synchronization objects from Java source code. And a compiler from Java to CSP will benefit automatic testing of synchronization objects.

7 Reference