1 Introduction

A concurrent datatype is ... A concurrent datatype offers encapsulation of concurrency and makes writing concurrent programs simpler.

For example, the MenWomen object is a concurrent datatype that captures a classical synchronization problem. In this problem, some processes need to pair with other processes by exchanging their identities. Figure 1 is an interface of a concurrent datatype for the MenWomen problem.

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

Figure 1: Interface of a MenWomen object

The safety and liveness properties are essential to the correctness of concurrent datatypes. The safety property states that the behaviour of the concurrent object should observe some invariant. For example, if a process with identity 1 calling manSync returns 2, then the process with identity 2 should call womanSync and return 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, a system with one process calling manSync and one process calling womanSync should not deadlock.

In this paper, we examine the above two correctness properties for various concurrent datatypes. In addition, we provide a few CSP implementations for objects commonly used in concurrent programming, which can be used in future CSP projects.

1.1 Linearization test

To verify the correctness of a concurrent datatype, one can carry out the Linearization test described in [TODO: Reference]. The linearization testing framework measures each call's starting and returning time to get a history of function calls and returns. 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 buggy if the framework can not find a valid synchronization point series.

In this remaining section we shall look at a few history from the MenWomen object. The timeline in Figure 2 visualizes the function call history of two process T1 and T2. T1 first calls manSync, then T2 calls womanSync. A synchronization occurs between T1 and T2. T1 returns the identity of process T2 then T2 returns the identity of process T1.

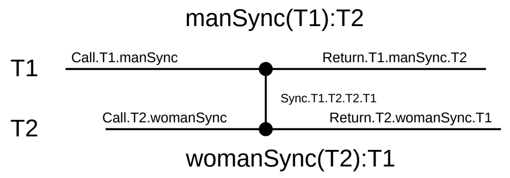


Figure 2: Visualized history of T1 calling manSync and T2 calling womanSync

In Figure 3, both processes calls manSync, and no synchronization is possible. Note that the liveness condition is not invalidated even if the system deadlocks in this case.

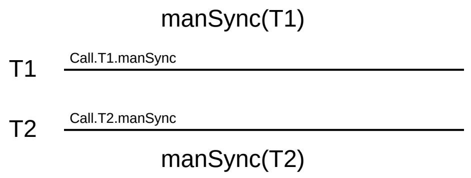


Figure 3: Visualized history of both T1 and T2 calling womanSync

Scheduling is one of the reasons validating a history can be complicated. In Figure 4, process T3 calls manSync first but gets descheduled. Then T1 calls manSync and synchronizes with T2 which later calls womanSync. The linearization framework usually needs to search a large state to find a valid series of synchronization points.

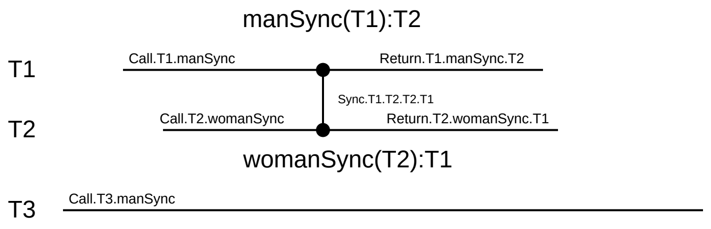


Figure 4: Visualized history of T3 get descheduled

1.2 Checking safety property using CSP

The history can be captured as a trace of a CSP system. In addition to performing the function body, each function call sends a Call event before the function body and a Return event after the function body. Figure 5 is the definition of Call and Return channel in CSP. The definition of the channel usually needs to include all the identity of the calling process, the function called, and its parameter.

```
--identity of the calling process
--function called by the process
channel Call : TypeThreadID.TypeOps
--identity of the calling process
--function called by the process
--return value of the function call
channel Return: TypeThreadID.TypeOps.TypeThreadID
```

Figure 5: Definition of Call and Return channel

To check the safety property, we check that a testing system built from some processes using the concurrent datatype refines a specification process built from the object definition in the CSP trace model.

A generic and scalable system is used for the testing system to generate possible histories of processes using concrete datatype. 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 generates all valid histories. The process uses the same number of linearizer processes synchronizing on events from the Sync channel. Event from the Sync channel should include information from all participating processes, and Figure 6 is the Sync channel definition for ... Each linearizer process repeatedly calls a function, synchronize with zero or some processes, and returns according to the calling argument and extra information from the synchronization point. To match the definition of the generic and scalable testing system, each linearizer process can also choose to terminate.

```
--Identity of thread calling ManSync
--Return of ManSync
--Identity of thread calling WomanSync
--Return of WomanSync
channel Sync: TypeThreadID.TypeThreadID.TypeThreadID.TypeThreadID
```

Figure 6: Definition of Sync channel

We shall see a concrete testing system and specification process in the Men-Women section.

1.3 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. Suppose all process calls manSync. Since a linearizer process calling manSync sends Return event only after synchronizing Sync event with another linearizer process calling womanSync, the linearizer will refuse to return any function call, which is a expected behavior. One can use a datatype-specific specification process that does not explicitly use any synchronization points. However, reusing the linearizer process is easier.

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 7 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 7, 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.

```
−−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 7: The shared variable module in CSP

Figure 8 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 8: Example of two processes using a shared variable

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 down 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 unraised. 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 simple in CSP. 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 [| chanset |] P) \
                    ( if hide then chanset else {})
  channel upChan, downChan: TypeThreadID
endmodule
```

Figure 9: The binary semaphore module in CSP

2.3 Monitor

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

Usually, code protected by the monitor is wrapped inside a synchronized block. In Figure 10, op1 uses synchronized to prevent race condition on the variable a. To run a synchronized block in CSPM, the process first needs to synchronize with the monitor process on a waitEnter event, which works as a certificate to run the code insider synchroninzed block. After running the code inside synchronized block, the process sends a Exit event to notify the monitor process that it is exiting.

Inside a synchronized block, the process can also perform wait, notify, and notifyAll. The method op2 in Figure 10 first increments b, wait to be notified, and decrements b. So the shared variable b here counts the number of processes

```
class MonitorExample {
  private var a = 0;
  def op1():Unit = synchronized{
    a+=1;
  }

  private var b = 0;
  def op2():Unit = synchronized{
    b+=1;
    wait ();
    b−=1;
  }

  def op3():Unit = synchronized{
    notify ();
  }

  def op4():Unit = synchronized{
    notifyAll ();
  }
}
```

Figure 10: A simple Scala class that uses a monitor internally

waiting to be notified. The method op3 notifies one waiting process and op4 notifies all waiting processes.

In the CSP implementation, a waiting process first synchronizes a Wait event with the monitor process to indicate that the process calls wait and no longer holds the running certificate. The process can then be waked up with waitNotify or spuriously by an event SpuiousWake from the monitor process. Finally, the process needs to reobtain the certificate WaitEnter to resume execution. In practice, the wait function is often guarded with a while loop to prevent spurious wakeup. Again CSPM does not have a built-in while loop, and a generic while statement is implemented in continuation pass style. (Maybe leave this continuation pass style while loop in appendix?)

Note that both WaitNotify and SpuriousWake events come from the monitor process. When a process calls notify or notifyAll, it needs to synchronize with the monitor process. notify and notifyAll are implemented in this way because usually, a process does not know how many processes are waiting. If a process calling notify synchronizes directly with a waiting process, then 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.

The monitor process has two states. The monitor process allows a different set of events in two different states. When there is no running process, the monitor can allow a free process to enter its synchronized block by synchronizing on a waitEnter event with the process. When there is a running process, the monitor process should respond to method calls from the running process, but the monitor should not allow another process to obtain the monitor lock. In either state, the monitor process can spuriously wake up a waiting process.

//Should I split some functions into respective paragraphs here. Or use line number.

```
module ModuleMonitor(TypeThreadID)
  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 → (
      if waiting={} then
        --do nothing if no thread is waiting
        active(cur, {})
      else
        --wakeup a process
        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}))
    )

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 → ((
          WaitNotify . me →
          WaitEnter . me →
          SKIP
        ) □ (
          SpuriousWake . me →
          WaitEnter . me →
          SKIP
        )
    )

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

Figure 11: The monitor module in CSP

3 MenWomen

For simplicity, a process calling ManSync is called a man process, and a process calling WomanSync is called a woman process.

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 12 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 12 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 13. 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 12: 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 13: Translated CSP code for the correct MenWomen object immplementation

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 the CSP implementation, a helper function is used. chaosP(P) runs the process P that terminate with SKIP any number of time. With the helper function, the process P only needs to non-deterministically choose to perform manSync or womanSync with its identity.

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.

Similarly, a linearizer process can non-deterministically choose to perform manSync, synchronize with another linearizer process calling womanSync, and return with the identity it received from the Sync event. Also, the linearizer

```
Thread(me)=chaosP(manSync(me) ⊓ womanSync(me))
System(All)=runWith(True,True,||| me:All • Thread(me))
```

Figure 14: Definition of CSP processes in the testing system

can choose to call womanSync or terminate. Linearizers(All) puts all linearizers processes in parallel.

The Linearizers function creates a combined linearizer process in three steps. First, it put all linearizer processes in parallel using Replicated Generalized Parallel in CSPM. Specifically, General Parallel uses three arguments, the linearizer identity set, the linearizer process, and a synchronization alphabet set. The synchronization alphabet set for a process identity is the set of Sync events where the process identity appears on the first or the third argument. If a linearizer process wants to send an event in its synchronization alphabet set, it must synchronize with all other linearizer processes whose synchronization alphabet sets include this event. Then Linearizers runs the paralleled process with the specification process of Sync event. For MenWomen object, the synchronization is stateless and only requires the return value to be the identity of the other process. Finally, like the testing system, all Sync events are hidden to provide all valid histories.

Figure 16 visualizes how the linearizer process can generate the trace that corresponds to the history described in Figure 2, where one process calls manSync and another process calls womanSync. And 17 visualizes the linearizer deadlocks as required if both processes calls manSync.

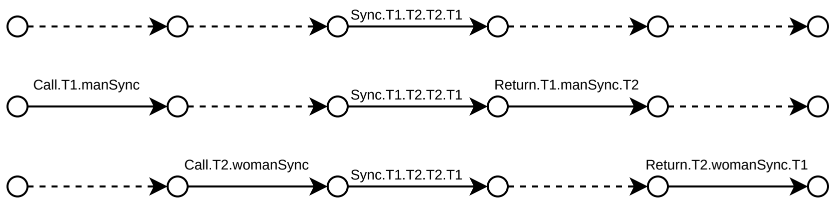


Figure 16: Linearizers generating the history where one process calls manSync and another process calling womenSync

```
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 15: Definition of linearizer process in CSP

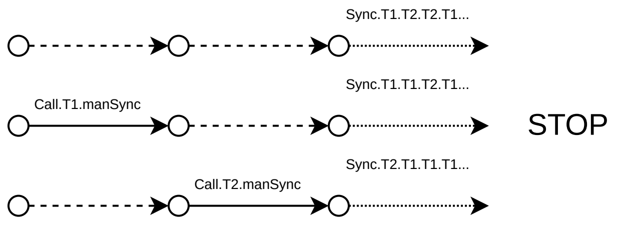


Figure 17: Linearizers generating the history where one process calls manSync and another process calling womenSync

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

3.3 A faulty version

We shall examine another MenWomen implementation in Figure 19. One key difference in this faulty MenWomen object is that it uses Option data in the

```
System2=System({T1,T2})
Spec2Thread=Linearizers({T1,T2})
 assert  Spec2Thread  ⊑_T  System2
 assert  Spec2Thread  ⊑_F  System2

System3=System({T1,T2,T3})
Spec3Thread=Linearizers({T1,T2,T3})
 assert  Spec3Thread  ⊑_T  System3
 assert  Spec3Thread  ⊑_F  System3
```

Figure 18: Part of liveness and safetyness test in CSP for MenWomen object

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 20. 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 a equivalent way to trigger the bug in Scala.

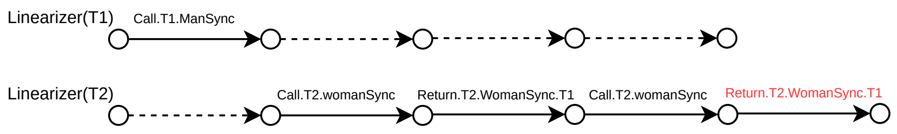


Figure 20: Trace

- 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.

```
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 19: A Faulty Scala implementation of MenWomen object

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 21 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.

```
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 21: A semaphore-based Scala implementation of the ABC object

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 22 So inside chaosP, the processes also choose an argument with General Non-Deterministic Choice.

```
Thread(me)=chaosP(
  ⊓x:TypeData • (
          SyncA(me,x)
     ⊓ SyncB(me,x)
     ⊓ SyncC(me,x)
  )
)
```

Figure 22: Definition of processes in the testing system.

Using the standard linearization testing technique, tthe Sync channel in Figure 23 is used to represent the synchronization of three involved threads. For example, the event $Sync.t_1.a.b.c.t_2.d.e.f.t_3.g.h.i$ represents the synchronizations of three threads, t_1, t_2, t_3, in which the first 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 last process t_3 calls cSync with g and returns (h, i). The Sync specification process, shown in the same figure, should then check that for each synchronization point, the return value of each functional call is the pair of arguments of the two other function call. Finally a linearizer process is defined in a similar format.

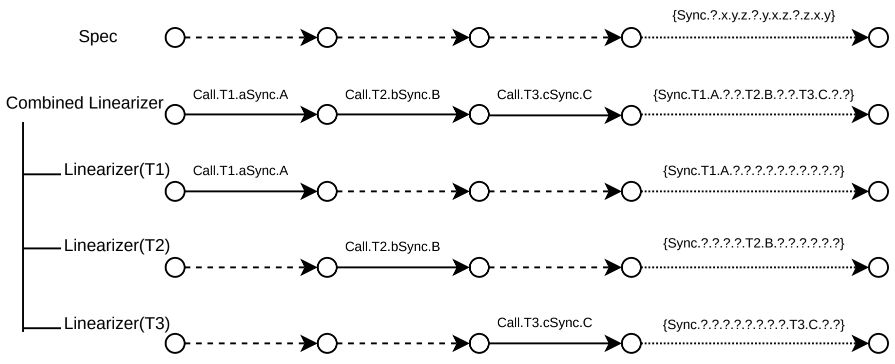
4.2.1 Speeding up model compilation

Consider the specification process with three processes. Let M be the size of the set of all possible arguments. Conside r the trace in Figure 26, where process T1 calls aSync with A, T2 calls bSync with B, T3 calls cSync with C. Then they synchronization.

```
−−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 23: Definition of Sync channel and specification of Sync event



Spec
{Sync.?.x.y.z.?.y.x.z.?.z.x.y}
Combined Linearizer
Call.T1.aSync.A
Call.T2.bSync.B
Call.T3.cSync.C
{Sync.T1.A.?.?.T2.B.?.?.T3.C.?.?}
Linearizer(T1)
Call.T1.aSync.A
{Sync.T1.A.?.?.?.?.?.?.?.?.?.?}
Linearizer(T2)
Call.T2.bSync.B
{Sync.?.?.?.?.T2.B.?.?.?.?.?.?}
Linearizer(T3)
Call.T3.cSync.C
{Sync.?.?.?.?.?.?.?.?.T3.C.?.?}

Figure 26: 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 27 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 the above optimization, the testing finishes quickly for both test cases.

```
−−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 24: Definition of linearizer process

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

assert Spec3Thread ⊑T System3
assert Spec3Thread ⊑F System3
```

Figure 25: Part of test for ABC object. This tests a system with three processes.

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

```
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 27: Simplified definition of Sync channel and part of simplified linearizer

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.

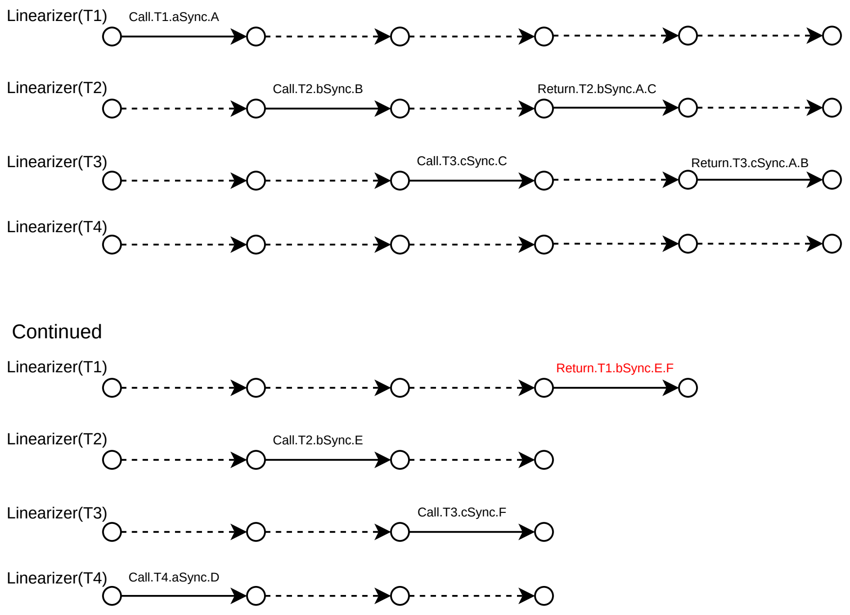


Figure 28: Trace that violates the specification

- 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 overwites 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.