1 Linearization

Proving a concurrent datatype's correctness is usually more complicated than proving for a single-threaded data structure. In addition to the state explosion when more and more processes are involved, scheduling is one particular reason that proving is difficult. For example, two processes are enqueuing their data to a concurrent queue. The process calling earlier may be descheduled and interact with the queue after the other process.

The linearizability test from [TODO:reference] is one way to prove that a sequence of process interactions is correct using only trivially measurable information - pairs of calling and returning time of every function call made by every process. From the history of function calls, The linearizability test tries to find a synchronization point for each call, where the call appears to make an effect. In addition, the series of synchronization points should satisfy the specification of the concurrent datatype.

In order to show that a system using a concurrent datatype is correct, the linearizability test works by running the system many times and checking the traces. In this essay, we use CSP models and the linearizer framework to prove the correctness of a concurrent datatype.

CSP is ... Trace is ... Trace refinement is ... FDR is ...

In the CSP model, the history of all function call is represented using related events. Call.t.op.p represents a process calls function op with parameter p. Return.t.op.ans represents a process returns from calling op with return value ans.

In CSP, trace refinement is a valuable tool to analyze the history of function calls described in the linearizability test. If the testing system, which produces possible function calls and function returns in CSP, refines a CSP specification of the system, then the testing system cannot do anything incorrect and thus must be correct. One way to construct the specification is to use the linearizer framework.

The linearizer has two core components - the linearizer process and the specification process for the Sync event. The linearizer process generates endless pairs of Call and Return events with its synchronization point for a specific process. When all linearizer processes, together with the specification process for Sync, run in parallel, the combined process produces all possible histories in CSP. Figure 1 shows how to create the combined processes in FDR. In Section ... we shall see more concrete instances of the linearizer framework.

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

Figure 1: Creation of the specification from the linearizers

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, as shown in Figure 2. 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 2, 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.

```
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 2: CSP implementation of global variable process module

Figure 3 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 3: CSP Example of a process using a shared variable

2.2 Semaphore

Semaphore is a simple but powerful concurrent primitive. The essay 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 unraised. A down function call, usually by a process that previously acquired the semaphore, raises the semaphore regardless of the semaphore state. If a process calls the up method when the semaphore is raised, the semaphore becomes unraised. However, if the semaphore is unraised, the process waits until another process calls down and proceeds to put down the semaphore. Depending on the initial state of the semaphore, a binary semaphore can be further classified 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.

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

Figure 4: Implementation of a binary semaphore in CSP

2.3 Monitor

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

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

Figure 5: Description

Usually, code (protected by the monitor) is wrapped inside a synchronized block, as shown in Figure 5. But before the process runs code inside the synchronized block, the process needs to check if others have obtained the monitor lock. Also, an exiting process needs to release the monitor lock so that other processes can continue. In Figure 1, edge 1 represents a process entering the synchronized block, and edge 6 describes the transition of a process exiting the synchronized block. Inside the synchronized block, the process may call notify, notifyAll, wait function. A running process calling wait blocks execution and waits to be notified. When a running process notifies a waiting process, the waiting process becomes runnable and waits to obtain the monitor lock. Note that the waiting process can only continue execution after the current process exits its synchronized block, unlike that a process waiting to acquire the semaphore can immediately continue after another process raises the semaphore. All above transitions are represented by edge 2,3,4,5 in diagram 1 respectively.

In the CSP implementation of monitors, the monitor process does most of the work. Most monitor methods are implemented simply by sending an event to the monitor process, except for the wait. Recall that a running process calling wait first waits for notification and reobtains the monitor lock. So after sending the Wait event, the process waits for a notify event or a spurious wakeup event. Finally, the waiting process remains for a WaitEnter event, which gives the process the monitor lock. In practice, the wait function is often guarded with a while loop to prevent spurious wakeup. Again CSPM does not have a built-in

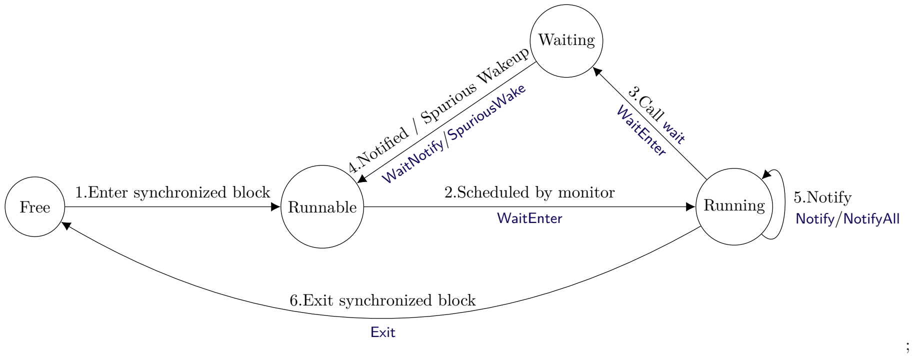

Figure 1: Time Line

while loop, and a generic while statement is implemented in continuation pass style. (Maybe leave this continuation pass style while loop in appendix?)

The monitor process has two states. When there is no running process, the monitor process 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 server process should respond to method calls from the running process, but not allow a process to obtain the monitor lock and proceed to its synchronized block. In either state, the monitor process can spuriously wake up a waiting process.

There is a corner case to highlight in the CSP implementation. In Java, if a running process calls notify method when there is no waiting processing, the function call does nothing. To make sure that the running process is not blocked in this case, when the running process calls notify method, the process send a notify to the server. Then if there are process waiting to be notified, the server process sends a waitNotify event to the waiting process.

3 MenWomen

The MenWomen problem is a classical synchronization problem. In this problem, some processes need to pair with some other processes, by exchanging their identities. Figure 7 is an interface of a concurrent datatype for the MenWomen problem. For simplicity, a process calling ManSync is called a man process, and a process calling WomanSync is called a woman process.

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

Figure 7: Interface

```
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
```

```
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 8: Scala implementation of the MenWomen process using a monitor

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 8 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 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 8 is a Scala implementation of the MenWomen process using a monitor by Gavin Lowe. The Scala code is further translated to a CSP code in Figure 9. Every function call begins with a Call event containing all parameters. And every function call ends with a Return event containing the return value.

```
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 9: CSP implementation of the MenWomen

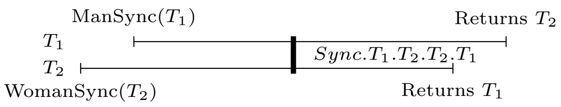

Figure 2: Time Line

3.2 Linearization Test

//This technique mostly applies to all objects.

A generic and scalable testing system is introduced to make the test cover as many cases as possible. The testing system includes an arbitrary number of processes, and each process calls any functions provided by the synchronization objects for any finite or infinite number of times. For the CSP implementation, each process non deterministically chooses a function and its parameters, or the process chooses to terminate. Figure 11 is the definition of a testing process and a testing system for MenWomen object in CSP.

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

Figure 10: CSP implementation of the testing processes and system

There are two properties one should check for the system. First, the traces of the system are valid. If a thread t_1 calls ManSync and returns the thread identity t_2, then t_2 must call WomanSync somewhere. This is done by checking that the system refines a linearization specification (introduced earlier?). Second, the system should not deadlock when matching is possible. This can be done by another specification process, which records the set of men processes and women processes to determine if matching is possible. However, it is easier to modify the linearization specification process and check that the system failure refines the specification process.

The implementation of the linearizer specification relies on the synchronization point. The synchronization point of a man process and a woman process is represented by a Sync event. The $Sync.t_1.a.t_2.b$ represents the synchronization of a man process t_1 with parameter a with a woman process t_2 with parameter b.

With the Sync event, it is simple to define the linearizer for a process. The process can choose to call ManSync, synchronize with a woman process, and return the identity of the woman process. Or, the process can choose to call WomanSync, synchronize with a man process, and return the identity of man process. Also, the process chooses to terminate when it finishes calling a function.

```
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
```

Figure 11: CSP implementation of the testing processes and system

4 ABC

In the ABC object, three threads are involved in each round of synchronization. For simplicity, a process calling syncA, syncB, and syncC is called a A process, B process, C process. In each round of synchronization, A A-process, B-process, and C-process synchronize, and each process returns the argument of two other processes.

For the above semaphore implementation of ABC object, In each round

- Initially semaphore aClear is raised.
- An A-process acquire 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 operates in turn with a slight change of semaphore and variable name.
- The A-process is able to continue after a C-process raises semaphore aSignal. The A-process reads the shared variable b and c, raises the semaphore bSignal, and returns. This also happens in turn for the B-process and the C-process.

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

Unlike monitor in Java and Scala, raising a semaphore immediately allows another thread waiting to acquire the semaphore to continue. So in the semaphore implementation, 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 the ABC object still works correctly when only three threads are involved, but fails the linearization test with four threads.

```
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 12: Scala implementation of the ABC using semaphores

4.1 Testing

For the MenWomen object, Using the standard linearization testing technique, the following Sync channel can be 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 spec process 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.

```
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
```

//Preparation of test case: I will describe There are two test cases. The first test case involves three threads. Each of the thread chooses a data nondeterministically and then calls one of aSync, bSync, cSync. The second test case involves four thread, which chooses a data non-deterministically and calls aSync. In both cases, the systems should be traced refined (is it this direction in words) by the specification process. In addition, both systems should never deadlock. Because in both system, there are always threads willing to communicate as aSync, bSync, cSync respectively.

When testing with both the correct and the faulty versions of ABC object, FDR finishes the first test case relatively quickly, but requires a long time to finish the second test. With logging message from FDR, it was found the compilation of specification process took the longest time. //TODO: Table

4.1.1 Speeding up model compilation

Consider the specification process. Let N be the number of threads in the system, M be the size of the set of all possible arguments. The specification process is the alphabetized parallel of N individual linearizers. In each linearizer, the process first chooses to perform one of aSync, bSync or cSync, chooses the argument of the functional call for Call event, then chooses the rest of arguments for Sync event, and finally performs one event before recursing into itself.

There are $O(3 * N^3 M^9)$ different transitions before the individual linearizer recurses into itself. However, according to the specification process, once the argument and return value of syncA is determined, all remaining arguments and return value are also determined. So only $O(3 * N^3 M^3)$ transitions are valid according to the specification.

```
  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)
```

With the above analysis, it is tempting to optimize the individual linearizer by using the information from the specification process. Instead of choosing all possible remaining arguments, the individual linearizer could choose arguments that are correct according to the specification process.

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

It is possible to further simplify the Sync channel, as now the arguments representing return value are redundant. This change does not reduce the number of transitions for an individual linearizer, but it may help FDR simulating the model faster.

```
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)
) ...
```

With the above optimizations, the testing finishes quickly for both test cases.

//TOADD: Table of compilation time

4.1.2 Explanation of the error case

With the traces of the counterexample from FDR, it is possible to see what goes wrong in the faulty version when there are four threads.

//TODO: Draw diagram using Scala code for this.

- Thread T_A, T_B, T_C call aSync, bSync or cSync respectively, and put down its argument in turn.

- Thread T_A raises bSignal without saving a copy of return value (b,c). The other two threads T_B, T_C are able to continue and exit.

- Thread T_D, T_B, T_C call aSync, bSync or cSync respectively, and put down its argument in turn.

- Thread T_A uses the wrong overwritten (b, c) as return value.

4.1.3 Conclusion

In the above section, we tested a semaphore based concurrent datatypes. With the testing result, we showed that it is important to be reminded that a thread waiting for the semaphore to raise can immediately continue and overwrite shared variables, after the semaphore is raised by another process.