1 Common Objects

1.1 Shared Variable

The usage of shared variables is common in concurrent datatypes. For example, some concurrent datatypes may temporarily store an integer value sent by a thread in its variable. However, CSP is more like a functional programming language and does not support mutable variables.

The behavior of a shared variable can be captured by a recursive function in CSP. 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**. Or 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, so that these concurrent datatypes can use multiple variables without conflicts. When other synchronization objects use the shared variable module, the process of the object needs to run in parallel with the process of the variable. This is implemented by the **runWith** function for all common objects. Such encapsulation reduces the complexity in the synchronization object implementation.

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

Figure 1: CSP implementation of global variable process module

The example CSP code in figure 2 is a simple example of a process using the shared variable. The first line in the examples creates a shared variable called **VarA**. The process **P** first reads the value of the variable, then discards it and sets the variable value to 1. The process **Q** is the combined process of **P** and variable process.

```
instance VarA = ModuleVariable({0,1,2},0)
P = VarA::getValue?a → VarA::setValue!a → STOP
Q = VarA::runWith(P)
```

Figure 2: CSP Example of a process using a shared variable

1.2 Semaphore

2 MenWoman

The MenWomen object is another synchronization object to be analyzed in this essay. The MenWomen object offers two operations, **ManSync** and **WomanSync**. For simplicity, a process calling **ManSync** is called a man process, and a process calling **WomanSync** is called a woman process. After the synchronization, a man process pairs with a woman process, and both processes obtain the identity of the other process.

2.1 Implementation

One way to implement the MenWoman object is to use a monitor and a shared variable indicating the stage of synchronization.

- 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 waked 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 3 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 4. Every function call begins with a Call event containing all parameters. And every function call ends with a Return event containing the return value.

2.2 Test

//This technique mostly applies to all objects.

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

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 6 is the definition of a testing process and a testing system for MenWomen object in CSP.

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

```
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 4: CSP implementation of the MenWomen

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

Figure 5: CSP implementation of the testing processes and system

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 6: CSP implementation of the testing processes and system

3 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

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

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.

3.1 Testing

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 non-deterministically 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

3.1.1 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

3.1.2 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.

3.1.3 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.