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 function represents a thread holding the value of variable in the function parameter. At any time, the variable thread is willing to answer a query for the variable value in channel **getChan**. Or the thread can receive an update on the variable value in channel **getChan**, after which the function recurses with the new variable value.

```
Var(value) = getValue!value → Var(value)
             □ setValue?value → Var(value)
```

In order for other objects to use the shared variable, the variable function needs to run in parallel with threads, and synchronizing on **getChan** and **setChan** events. It is convenient to implement this in a function **runWith**, to run a given process in parallel with the variable process.

For example, the CSP code below is a simple example of a process using the shared variable. The process first reads the value of the variable, then discards it and set the variable value to 1.

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

1.1.1 Semaphore

2 Testing Objects

2.1 ABC

In the ABC object, three threads are involved in each round of synchronization. On the higher level, three different threads call function **syncA**, **syncB**, **syncC** with their arguments respectively, and each function returns the arguments of two other function calls.

```
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
  }
}
```

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

- Initially **aClear** semaphore is up
- A thread calling **aSync** takes down the **aClear** semaphore, puts down its value, and raises **bClear** semaphore, and finally waits for the **aSignal** semaphore to raise.
- Another thread calling **bSync** takes down the **bClear** semaphore, put down its value, and raises **cClear** semaphore, and waits for **bSignal** semaphore to raise.
- One more thread calling **cSync** takes down the **cClear** semaphore, put

down its value, and raises **aSignal** semaphore, and waits for **cSignal** semaphore to raise.

- The first thread is able to continue. The thread takes a copy of the other two argument, raises the **bSignal** thread, and exits with the arguments pair.
- The second thread is able to continue and does the similar thing as the first thread. Before exiting, the second thread raises **cSignal**.
- The last thread is then able to continue. Before exiting, the last thread raises **aClear**, allowing a new round to start.

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 **down** the semaphore to continue. So in the semaphore implementation, it is essential to take a copy of the two other argument, before raising the semaphore for the next thread.

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.

2.1.1 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 thread t_1 calls **aSync** with a and returns (b, c), the second thread t_2 calls **bSync** with d and returns (e, f), and last thread t_3 calls **cSync** with g and returns (h, i). The spec process should then check each synchronization point that the return value of each functional call is the pair of arguments of 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
```

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 compilation of specification process took the longest time.

2.1.2 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 linearizes. 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

2.1.3 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.

2.1.4 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.