0.1 Converting Scala Program to CSP Model

0.1.1 Shared Variable

//Background - CSP - Example?

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 a 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.

//CSP code here

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 process in synchronization with the variable process.

//CSP code here

0.1.2 Semaphore

0.2 Testing Objects

0.2.1 ABC

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

The ABC object may be implemented with semaphore. In each round

//feels like a lot of repetition

- 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.

0.2.2 Testing

Using the standard linearization testing technique, the following channel can be used to represent the synchronization of three threads. //CSP Code

The first test case involves three thread. Three threads non-deterministically choose a data and then call **aSync**, **bSync**, **cSync**.

The second test case involves four thread. In addition to three threads in the first threads, another thread calling **aSync** is added.

//CSP Code

When testing with both the correct and the faulty version of ABC object, FDR is able to finish first test case relatively quickly, but requires unacceptable long time for the second test. With further diagnosing the running time of testing, it was found compilation of system specification process took the longest time.

//TODO: analyze the size of graph