0.1 Converting Scala Program to CSP Model

0.1.1 Shared Variable

The usage of shared variables is common in concurrent datatypes. For example, in later chapters some concurrent datatypes use mutable integer variables to indicate the stages of communication. However, CSP (is more like a functional programming language and) does not support mutable variables. To model mutable variables, a server process is created for each variable. A server process is implemented by some recursive CSP functions and runs in parallel with the other part of system.

The server process keeps the value of the variable in its parameter, and is always willing to synchronize with a thread that wants to read the value of variable through **getChan**, or a thread that wants to update the value of variable through **setChan**.

The CSP implementation of mutable variable is organized in a module so that all other concurrent datatypes can define variables consistently.

//TODO: CSP code and a simple example of a process using the variable

0.1.2 Monitor

//TODO: Describing Java Monitor.

//I find the state graph useful for explaining the CSP implementation

Similarly, the implementation of monitor requires a server process, like the server process of mutable variable.

//TODO: Change to TikZ diagram

- Before a thread enters the **synchronized** block, the thread may need to wait other process to leave the block, or more concisely, changing from **Runnable** state to **Running** state. It must first synchronize with the server process on a **WaitEnter** event.
- Before the process leaves the synchronized block, it must synchronize on an **Exit** event, to switch to **Finished** state.
- The running process may wait until some variable to change. For this, the process must send a **Wait** event to the server, indicating changing from **Running** state to **Waiting** state. Later when resuming from **Waiting** state, the process first needs to switch to **Runnable** state with a **WaitNotify** event, then a **WaitEnter** event.
- Also inside the block, the running process may send a **Notify** or **NotifyAll** event to notify one or all process in waiting state.

The server process should behave differently when there is a **Running** process or not. When there is no process in **Running** state, the server process should only non-deterministically pick a process waiting to enter the synchronized block. When there is a process in **Running** state, the server process should only be willing to accept

There are a few corner cases to consider.

In Java, when a process calls notify, but there is no waiting process to wake up, the function call does nothing. However, if the server process sends a **Notify** event when no process is waiting, the server process will simply deadlock. Thus, it is essential to prevent the server process doing a **Notify** event in such case.

One way to achieve this is to maintain a list of waiting processes. When the server process needs to do a **Notify**, after the running process sends a **Notify** event, the server process non-deterministically picks a waiting process from the list and send a **WaitNotify** event. Similarly, for notifyAll, the server process non-deterministically sends **WaitNotify** to all waiting process. It is also possible to simplify the lists of waiting processes to the number of waiting processes, because when the server sends a **WaitNotify** event, only waiting processes will be synchronized.

The monitor may spuriously wake up some processes at some time. For this purpose, the server process may non-deterministically choose to wake up processes whenever possible. //TODO: Livelock.

Now that waiting processes may be waked up spuriously, it becomes helpful to use **wait** with a while loop like **while(cond) wait()**, to check if variable has actually been changed. While loop, again, is not in CSP. Also **cond** is a bit unusual, because when evaluating the condition, the process may need to read global variable. For this reason, a generic while loop is defined. //TODO: While loop

0.2 Testing

0.2.1 Linearization

0.2.2 Filter Channel

A filter channel is like a OneOne channel. Process sending data continues only when the data is taken by some receiver. Process receiving data only takes values from senders that satisfies some predicate, and has not been taken by anyone else.

//TODO: An example with sync diagram