Concurrent MPL

July 4, 2017

1 Concurrent MPL constructs

Concurrent MPL constructs are the constructs using which concurrent MPL programs are written. In this chapter, various concurrent MPL constructs are discussed.

2 Introduction

Concurrent MPL constructs are plug, id, hput-hcase, get-put, split-fork and close-halt. A few noticeable observations about concurrent MPL constructs are below:

- MPL constructs with the exception of run contruct perform an action on a channel. In MPL concurrency is modeled using *message passing* between processes and channels are the conduit through which the messages are exchanged. So, it is only logical that most constructs deal with channel.
- Most concurrent MPL constructs come in pair, i.e get-put, hput-hcase, split-fork and halt-close. id, run and plug are standalone constructs. The constructs of the pair are dual to each other. This is intuitive because in message passing view of concurrency for someone to respond to an action, someone else should drive that action on the opposite end of the channel and viveversa. Thus one of the constructs of in a pair is a driver construct and the other is a reaction construct. For example, for a process to receive a value on a channel, some other process should have put a value on the channel.

A brief description of the constructs are as below:

run	calls a process
id	equates two channels
plug	connects two processes by a channel
get	gets a value on a channel
put	puts a value on a channel
hput	puts a handle on a channel
hcase	cases on the handles obtained on channel
split	splits a channel into two channels
fork	forks two new processes
close	closes a channel
halt	closes a channel. Usually the last channel is halted.

Table 1: Machine Transitions for the SAMPL

In the next section, the concurrent MPL constructs are described in details.

3 get-put

These are the simplest MPL constructs. get receives a value on a channel and put puts a value on a channel.

- On an output channel, get results in **Get** protocol and put results in **Put** protocol.
- On an input channel, get and put constructs result in **Put** and **Get** protocols respectively.

Thus, get/put constructs result in different protocols depending on the polarity of the channel over which they act. This is done in order to infer the same protocol for a channel from the two processes attached at the two ends of the channel.

3.1 get-put Example

Table 1 shows an example of an MPL program that uses get-put construct. The process connectivity diagram of an MPL program represents how different processes of the program are connected via channels. Figure 2 represents the process connectivity diagram for the example program in Table 1.

In the program there are three processes, **p1**, **p2** and **p3** apart from the main process. These processes have been represented by three circles named as $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ respectively in the process connectivity diagram.

Channel **ch2** is plugged between processes **p2** and **p3** and channel **ch1** is plugged between processes **p3** and **p1**. These channels are drawn as solid lines joining $\mathbf{P_2}$-$\mathbf{P_3}$ and $\mathbf{P_3}$-$\mathbf{P_1}$ respectively in the process connectivity diagram. **ch2** acts as an output channel for process **p2** and an input channel for process **p3**. The (+) sign on **ch2** at process $\mathbf{P_2}$'s end and (-) sign on $\mathbf{P_3}$'s end represents the polarity of channel with respect to the two processes it is plugged between.

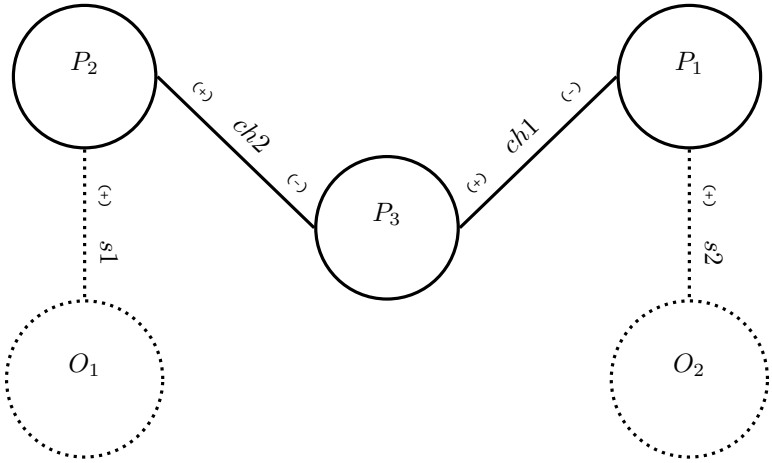

Figure 1: get-put Example

```
protocol IntTerm (A) => P =
    GetInt    :: Get (A|P) => P
    PutInt    :: Put (A|P) => P
    Close     :: TopBot    => P


proc p2 :: |  => IntTerm (Int),Put(Int|TopBot) =
    |  => s1,ch2 -> do
        hput GetInt on s1
        get x on s1
        put x on ch2
        hput Close on s1
        close s1
        halt ch2


proc p3 :: |  Put (Int|TopBot) => Put (Int|TopBot) =
    | ch2 => ch1 -> do
       get x on ch2
       put x on ch1
       close ch2
       halt ch1

proc p1 :: | Put (Int|TopBot) => IntTerm (Int) =
     | ch1 => s2 -> do
        get x on ch1
        hput PutInt on s2
        put x on s2
        hput Close on s2
        close s2
        halt ch1

run => intTerm1,intTerm2 -> do
    plug
      p2 ( | => intTerm1,ch2)
      p3 ( | ch2 => ch1)
      p1 ( | ch1 => intTerm2 )
```

Table 2: Example : get-put constructs