Chapter 8. DDF Domain

Authors: *Soonhoi Ha*

8.1 Programming Stars in the DDF Domain

A DDF star, as distinct from an SDF star, has at least one porthole, either an input or an output, that receives or sends a variable number of particles. Such portholes are called *dynamic*. Consequently, for a DDF star, how many particles to read or write is determined at run time, in the `go` method. Consider an example, the `LastOfN` star:

```
defstar {
      name {LastOfN}
      domain {DDF}
      desc {
Outputs the last token of N input tokens,
where N is the value of the control input.
      }
      input {
            name {input}
            type {anytype}
            num {0}
      }
      input {
            name {control}
            type {int}
      }
      output {
            name {output}
            type {anytype}
      }
      private {
            int readyToGo;
      }
      constructor {
            input.inheritTypeFrom(output);
      }
      setup {
            waitFor(control);
            readyToGo = FALSE;
      }
      go {
            if (!readyToGo) {
                  control.receiveData();
                  waitFor(input, int (control%0));
                  readyToGo = TRUE;
            } else {
                  int num = int (control%0);
                  for (int i = num; i > 0; i--) input.receiveData();
```

```
                output%0 = input%0;
                output.sendData();
                waitFor(control);
                readyToGo = FALSE;
            }
        }
    }
```

The `LastOfN` star discards the first *N-1* particles from the `input` porthole and routes the last one to the `output` porthole . The value *N* is read from the `control` input. Since the control data varies, the number of particles to read from the `input` porthole is variable, as expected for a DDF star. We can specify that the `input` porthole is *dynamic* by setting the `num` field of the `input` declaration to be 0 using the preprocessor format:

```
num {0}
```

The firing rule of the star is controlled by the `waitFor` method of the `DDFStar` class (actually, it is defined in the base class, `DynDFStar`). The `waitFor` method takes a porthole as an argument, and an optional integer as a second argument. It indicates that the star should fire when amount of data specified by the integer (default is 1) is available on the specified port. In the above example, the `setup` method specifies that the star should first wait for a `control` input. When a `control` input arrives, the `go` method reads the control value, and uses `waitFor` to specify that the star should fire next when the specified number of inputs have arrived at `input`. The private member `readyToGo` is used to keep track of which input we are waiting for. The line

```
for (int i = num; i > 0; i--) input.receiveData();
```

causes the appropriate number of inputs (given by `num`) to be consumed.

The next example is a DDF star with a dynamic output porthole: a `DownCounter` star.

```
defstar {
    name {DownCounter}
    domain {DDF}
    desc { Count down from the input value to zero. }
    input {
        name {input}
        type {int}
    }
    output {
        name {output}
        type {int}
        num {0}
    }
    go {
        input.receiveData();
        for (int i = int (input%0) - 1 ; i >= 0; i--) {
            output%0 << i ;
            output.sendData();
        }
    }
```

```
}
```

The `DownCounter` star has a dynamic `output` porthole that will generate the down-counter sequence of integer data starting from the value read through the `input` porthole. The code in the `go` method is self-explanatory.

It is possible, if a bit strange, for a star to alternate between SDF-like behavior and DDF-like behavior. To assert that its next firing should be under SDF rules, the star calls. The following example shows a star that uses the same input for control and data. An integer input specifies the number of particles that will be consumed on the next firing. After these particles have been consumed, the star reverts to SDF behavior to collect the next control input. In the following, `readyToGo` and `num` are private integers.

```
setup {
      clearWaitPort();
      readyToGo = FALSE;
}
go {
      int i;
      if (!readyToGo) {
            // get input token from Geodesic
            input.receiveData();
            num = int(input%0);
            waitFor(input, num);
            readyToGo = TRUE;
      } else {
            for (i = 0; i < num; i++) {
                  input.receiveData();
                  output%0 << int(input%0);
                  output.sendData();
            }
            readyToGo = FALSE;
            clearWaitPort();
      }
}
```

Because of the `clearWaitPort()` in the setup method, the star begins as an SDF star. It consumes one data, stores its value in `num`, and issues a `waitFor` command. This changes its behavior to DDF and specifies the number of input tokens that are required. On the next firing, it will read `num` input tokens and copy them to the output, and then it will revert to SDF behavior.