Chapter 11. SR domain

Authors: *Stephen Edwards*

Other Contributors: *Christopher Hylands*

11.1 Introduction

Synchronous Reactive (SR) is a statically scheduled simulation domain in Ptolemy designed for concurrent, control-dominated systems. Simple stars for the SR domain are easy to write, but more complex ones that take full advantage of the domain are more subtle. Stars can be written in either C++ or Itcl.

11.2 Communication in SR

Time in SR is divided into discrete instants. In each instant, the communication channels in SR contain a valued event, have no event, or are "undefined," corresponding to when the system could not decide whether there was an event or not. These channels are not buffered, unlike Ptolemy's dataflow domains, and do not hold their values between instants.

Stars in the SR domain have input and output ports, much like they do in other domains. However, primarily because absent events are different from undefined ones, the interface to these ports are unique.

Because SR domain ports are unbuffered, output ports can be read just like input ports. It is often convenient to do this when checking to see whether the value on an output port is already correct and does not need to be changed.

Input/Output Porthole Interface

`int SRPortHole::known()`
: Return `TRUE` when the value in the port is is known.

`int SRPortHole::present()`
: Return `TRUE` when the value in the port is present.

`int SRPortHole::absent()`
: Return `TRUE` when the value in the port is absent.

`Particle & InSRPort::get()`
: Return the particle in the port. This should only be called when `present()` returns `TRUE`.

Output Porthole Interface

`Particle & OutSRPort::emit()`
: Force the value on the output port to be present and return a reference to the output particle.

`void OutSRPort::makeAbsent()`
Force the value on the output port to be absent.

11.3 Strict and non-strict SR stars

Broadly, there are two types of stars in the SR domain: strict and non-strict. If any input to a strict star is unknown, then all of its outputs are unknown. A two-input adder, for example, behaves like this--it cannot say anything about its output until the values of both inputs are known. A non-strict star, by contrast, can produce some outputs before all of its inputs are known. A two-input multiplexer is an example: when the selection input is known, it can ignore the unselected input.

Non-strict stars are the key to avoiding deadlocked situations when there are cyclic connections in the system. If all the stars in a cycle are strict, they each need all of their inputs before producing an output--all will be left waiting. The deadlock can be broken by introducing a non-strict star into the cycle that can produce an output without having all inputs from other stars in the cycle

A number of methods set attributes of SR stars. These should be called in the `setup()` method of a star as appropriate. By default, none of these attributes is assumed to hold.

`SRStar::reactive()`
Indicate the star is reactive--it needs at least one present input to produce a present output.

`Star::noInternalState()`
Indicate the star has no internal state--its behavior in an instant is a function only of the inputs in that instant, and not on history.

By default, a star in the SR domain is strict. Here is (abbreviated) `ptlang` source for a two-input adder:

```
defstar {
      name { Add }
      domain { SR }
     input {
          name { input1 }
          type { int }
      }
     input {
          name { input2 }
          type { int }
      }
     output {
          name { output }
          type { int }
      }
     setup {
          reactive();
```

```
            noInternalState();
        }
        go {
            if ( input1.present() && input2.present() ) {
                output.emit() <<
                    int(input1.get()) + int(input2.get());
            } else {
              Error::abortRun(*this,
                 "One input present, the other absent");
            }
        }
    }
```

Non-strict stars inherit from the `SRNonStrictStar` class. Here is abbreviated source for a non-strict two-input multiplexer:

```
    defstar {
         name { Mux }
         domain { SR }
         derivedFrom { SRNonStrictStar }
        input {
            name { trueInput }
            type { int }
        }
        input {
            name { falseInput }
            type { int }
        }
        input {
            name { select }
            type { int }
        }
        output {
            name { output }
            type { int }
        }
        setup {
            noInternalState();
            reactive();
        }
        go {
            if ( !output.known() && select.known() ) {
                if ( select.present() ) {
                    if ( int(select.get()) ) {
                       // Select is true--
                       // copy the true input if it's known
                       if ( trueInput.known() ) {
                            if ( trueInput.present() ) {
                                output.emit() <<
                                 int(trueInput.get());
                            } else {
```

```
                        // true input is absent:
                        // make the output absent
                        output.makeAbsent();
                    }
                }
            } else {
                // Select is false--
                //copy the false input if it’s known
                if ( falseInput.known() ) {
                    if ( falseInput.present() ) {
                        output.emit() <<
                           int(trueInput.get());
                    } else {
                        // false input is absent:
                        // make the output absent
                        output.makeAbsent();
                    }
                }
            }
        } else {
            // Select is absent:
            // make the output absent
            output.makeAbsent();
        }
    }
  }
}
```