

SR

An *SR* (*Synchronised Resources*) program is a collection of *resources* and *globals*.

A *resource* is a template for resource instances from which resource instances can be dynamically created and destroyed.

A *global* is basically a single, unparameterised, automatically created instance of a resource.

A resource can be viewed as an abstract data object that consists of two parts:

- 1) a *specification* that specifies the interface of the resource, and
- 2) a *body* that contains the code that implements the behaviour of the abstract data object.

A global is a collection of objects shared by resources and other globals in the same address space (virtual machine). The major differences between a global and a resource are:

- 1) Globals allow variables to be shared among all resource instances whereas resources allow variables to be shared only by the procs within each resource instance.
- 2) Procs declared in the spec of a global can be referenced outside the global directly via the

global's name, whereas procs declared in the spec of a resource must be referenced outside the resource indirectly by means of a resource capability variable for that resource instance.

Resource Specifications and Bodies

The general form of a resource is:

```
resource resource_name
  imports
  constants, types, or operation declarations
body resource_name (parameters)
  imports
  declarations, statements, procs
  final code
end resource_name
```

The parameters and all parts of the spec and body are optional, as is the resource name following **end**.

The following code defines a **stack** resource:

```
resource Stack
  type results = enum (OK, OVERFLOW,
                      UNDERFLOW)
  op push (item:int) returns r:result
  op pop (res item:int) returns r:result
```

```

body Stack (size:int)
  var store [1:size]:int, top:int := 0
  proc push (item) returns r
    if top < size ->
      store[++top] := item
      r := OK
    [] top = size ->
      r := OVERFLOW
    fi
  end

  proc pop (item) returns r
    if top > 0 ->
      item := store[top--]
      r := OK
    [] top = 0 ->
      r := UNDERFLOW
    fi
  end
end Stack

```

The body of the **stack** resource contains a parameter **size**. When new instances of a stack resource are created this parameter is used to set the size of the local array and in the if tests.

Imports

When one resource wants to use another, it must import the other resource. For example, if a resource `stack_user` wishes to use the publically declared operations of the `stack` resource, we would have:

```
resource Stack_User ( )
  import Stack
  var x:Stack.result
  ...
end Stack_User
```

Creating and Destroying Resource Instances

Since several instances of a resource can be created we need some mechanism to distinguish between the different instances of a resource. This is provided by *resource capabilities*. A *resource capability* acts a pointer to a specific instance of a resource. The following code creates two instances of a `stack` resource:

```
resource Stack_User
  import Stack
  var x: Stack.result
  var s1, s2: cap Stack
  var y:int

  s1 := create Stack(10)
  s2 := create Stack(20)
```

```

...
s1.push (4); s1.push (37); s2.push (98)
if (x := s1.pop(y)) != OK -> ... fi
if (x := s2.pop(y)) != OK -> ... fi
...
end

```

There are two instance of the **stack** resource, differentiated by the two resource capability variables **s1** and **s2**. Each instance has its own copies of its local variables **store** and **top**; and the operations specified in the spec can be referenced outside **stack** via the resource capability variables, e.g. **s1.push (37)**.

The execution of an *SR* program begins with the implicit creation of one instance of the program's **main** resource. The initial code of the **main** resource can in turn create instances of other resources.

A resource instance can be destroyed by the **destroy** statement:

```
destroy resource_capability
```

When an *SR* program terminates, the initially created instance of the **main** resource is destroyed after executing its final code. This final code can in turn destroy other instances of resources.

Processes

SR uses the *process* as its unit of concurrent computation. A *process* is an independent thread of control that executes sequential code. The form of a process is:

```
process process_name (quantifier, quantifier, ...)  
    block  
end
```

The name of the process can optionally appear after the keyword `end`. The quantifiers have the same form as the quantifiers in `for-all` statements.

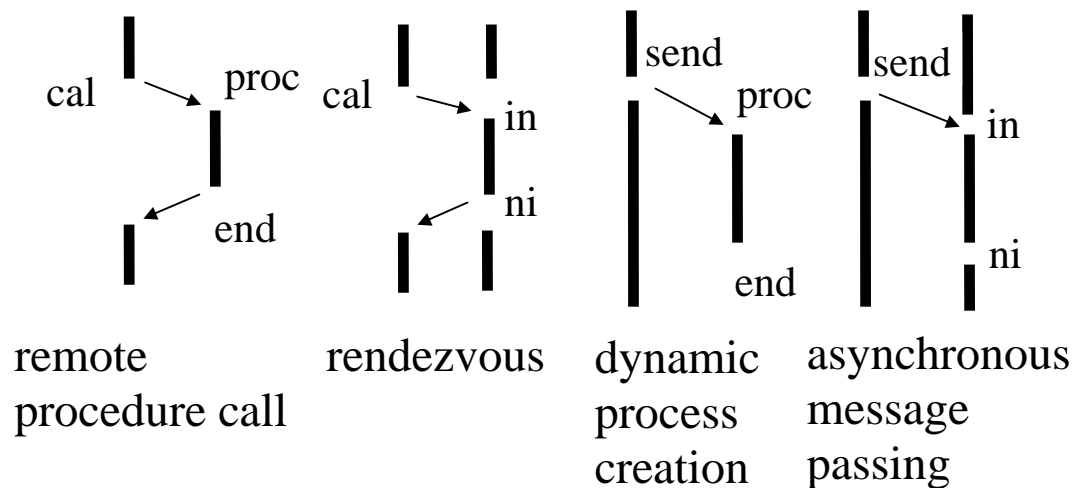
```
resource mult ( )  
    const N := 20  
    var a[N,N], b[N,N], c[N,N]: real  
  
    # read in some initial values for a and b  
    ...  
  
    # multiply a and b in parallel, place result in c  
    process multiply (i := 1 to N, j := 1 to N)  
        var inner_prod:real := 0.0  
        fa k := 1 to N ->  
            inner_prod += a[i,k] * b[k, j]  
        af  
        c[i,j] := inner_prod
```

```

end
final
  # output result in c
  fa i := 1 to N ->
    fa j:= 1 to N ->
      writes (c[i, j], ' ')
    af
  write
af
end
end mult

```

The process is actually a shorthand for *SR*'s general concurrency mechanisms which is implemented by operations, procs, calls, sends and ins.



So a process is an operation serviced by a proc which was invoked by a send.

SR, Java and (inevitably) “JR”

As we have seen, the standard Java concurrency model is limited. It provides threads, a primitive monitor-like mechanism, and remote method invocation (RMI). Though useful, they offer little flexibility in the design and implementation of concurrent programs. Recently an attempt has been made to extend Java with the concurrency model provided by the SR concurrent programming language. The result is a new language, a superset of Java, “JR”. JR adapts the following features from SR: dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, dynamic process creation, support for rendezvous, and asynchronous message passing. JR takes a novel object-oriented approach to synchronization. JR has been designed to integrate the SR concurrency model with Java in a manner that retains the “feel” of Java.