Programming in SR

Peter Lewerin

May 26, 2008

Abstract

An introduction to the SR programming language. It covers modular programming using resources and concurrent programming concepts such as semaphores, message passing, dynamic processes/remote procedure calls, and rendezvous. The various statement categories of SR are described in detail.

Contents

1 Introduction

Welcome to the SR (SYNCHRONIZING RESOURCES) programming language. SR is a high-level language for concurrent (parallel) programming. At first glance, SR resembles languages like Modula-2 or Pascal, but programmers familiar with C++ or Java shouldn't have much trouble getting to grips with it.

In this section, I'll demonstrate a few basic features of SR to make the following sections easier to understand. The language will be described in more detail in section 4. The sections in between describe the declaration and use of `resources` (modules), and process synchronization, respectively. Section 5 lists reserved words, separator symbols, operators, and functions.

To start off, program 1 is a simple example of an SR program that simply prints a greeting to the screen.

Program 1 The Hello, World! program in SR.

```
resource Hello ()
    write("Hello, world!")
end Hello
```

It is compiled and executed like this (with `>` being the prompt character):

```
> sr -o hello hello.sr
> hello
Hello, world!
```

Of the three lines in program 1, lines 1 and 3 simply tell the compiler where the program starts and stops. The middle line,

```
write("Hello, world!")
```

is a *statement*. *statements*

SR has the usual array of statements for *declarations*, *definitions*, and program *flow control*, as well as *operators* with which to build expression statements, including calls to library functions (such as the call to `write` in this example). In addition, it also has statements that handle *synchronization* of concurrently executing processes.

Several statements can be written on the same line, but in that case they must be separated by semicolons:

```
var n : int ; read(n)
```

Apart from that, the parser is open-minded about layout. E.g., this:

```
if a > b -> c := a / 2 fi
```

is equivalent to this:

```
if a > b ->
    c := a / 2
fi
```

The basic input and output functions of SR are `read` and `write`. They accept a variable number of arguments, reading or writing them as fields separated by white-space.

`read`
`write`

Comments are bits of text that aren't meant to be read by the compiler but to make the thinking behind the program clearer to a human reader.

comments

```
/* this is a multi-line comment,
   which can go on
   for several lines before ending */

# a single-line comment, running up to the end of the line
```

SR is a *statically typed* language, with simple data types like *integers*, floating-point numbers (known as *reals*), *characters*, and *booleans*, and structured types like *arrays*, *strings*, and *records*. Every *constant* and *variable* is assigned a type during compilation, and each *operation* (see below) has an immutable type signature that specifies the types of its parameters and its return value, if any.

types

The type system of SR is described in more detail in section 4.3.

```
const GREETING := "Hi there"      # a constant declaration
var number : int                  # a variable declaration
op addnums(int; int) returns int  # an operation declaration
```

An operation is a generalization of a function call. In a sequential program, function calls basically just make the execution thread jump to another part of the program for a while before jumping back. In a concurrent program, it is rarely that simple: the corresponding action might involve spawning a new execution thread to do the function's work, or an execution thread might go to sleep waiting for another thread to request that the function be executed. Hence a slightly different terminology: instead of calling a function, an operation is *invoked*, and instead of executing the function, the operation is *serviced*. The various forms of invoking and servicing is described in the latter part of section 3.

operations

2 Resources

The simplistic program in the previous section had only one resource structure. SR programs can have several resources that encapsulate different parts of the program or support code reuse as library modules[1].

When a program starts up, one of the resources (the last one defined) is *enlivened*, meaning that an instance is created and the resource's internal code is executed. That code can create instances of other resources, and those can create more instances, and so on. Once created, an instance stays around until it is explicitly destroyed or until the program ends[2].

2.1 A resource use example

Let's write a resource that provides math constants (e.g., `PI` with the value of π)[3] and functions (such as `square(x)`, which calculates $f(x) = x^2$), and have another resource use it.

To begin with, the resource must have a *specification part* (or *interface*) that tells other resources what they get:

```
const PI := 3.14159265
op square(real) returns real
```

The resource must also have an *implementation part* (or *body*) containing the internal mechanisms that make the resource work:

```
proc square(x) returns r
    r := x * x
end
```

Putting it together, we get program 2:

In order to use the `Math` resource, `UseMath` has 1) *imported* `Math`, and 2) acquired a *capability* (basically, a handle for a resource instance, see page 19) which is necessary for invoking the operations of another resource.

2.2 Global resources

Global resources are singleton[4] resources: an instance of each defined global is implicitly created when the program starts. They differ from regular resources in that

- they allow variable declarations in the interface
- operations declared by a global can be implemented in the importing resource instead (unless they are implemented by `proc`, see section 3.3)

[1] Resources in SR are, to a degree, similar to classes in many object-oriented programming languages.

[2] I.e., it has *indefinite extent*.

[3] Of course (to borrow a phrase from John von Neumann), to define π in a finite computer program is to be in a state of sin.

[4] Meaning that there can be only one.

Program 2 A utility resource for math, and another resource using it.

```
resource Math
    # specification part
    const PI := 3.14159265
    op square(real) returns real
body Math ()
    # implementation part
    proc square(x) returns r
        r := x * x
    end
end

resource UseMath ()
    import Math
    write("The value of PI is (approximately)", Math.PI)
    var mc : cap Math := create Math()
    write("The square of 4 is", mc.square(4))
end
```

◆ It isn't strictly necessary to refer to the constant `PI` by `Math.PI`: simply `PI` will suffice.
◆ The `create` statement enlivens a resource and returns a capability handle for it.

- a global's operations can be invoked without using a capability handle

Constants, semaphores, and variables can be shared among resources using a global.

2.3 Resource parameters

The data content of a resource instance can be set at creation by passing data to it as arguments (see program 4 and section 4.2).

3 Synchronization

In SR, it is possible to have statements executed concurrently, by putting them inside *process* definitions. Program 5 is about as likely to write the words in the order "world", "Hello" as in the correct order. Clearly, the two processes need to be *synchronized* so that we can be sure P_2 doesn't execute its output statement until P_1 is done. When using SR it is important to remember that *execution order is indeterminate* in many cases. The ability to execute code concurrently goes hand in hand with a flexibility in scheduling that must be taken into consideration by the programmer.

3.1 Semaphores

A *semaphore* is a special kind of variable that is used to synchronize processes by acting as a supply counter. By protocol, processes will check semaphores to see if a

Program 3 `Math` rewritten as a global resource.

```
global Math
    const PI := 3.14159265
    op square (real) returns real
body Math
    proc square (x) returns r
        r := x * x
    end
end

resource UseMath ()
    import Math
    write("The value of PI is (approximately)", Math.PI)
    write("The square of 4 is", Math.square(4))
end
```

Program 4 Passing data to a resource.

```
resource Pet
    op says() returns string[4]
body Pet (sound : string[4])
    proc says() returns s
        s := sound
    end
end

resource Owner ()
    import Pet
    var dog : cap Pet := create Pet("woof")
    var cat : cap Pet := create Pet("meow")
    write("My dog says", dog.says())
    write("My cat says", cat.says())
end
```

Program 5 Two processes write one word each concurrently.

```
resource ParallelHello ()
    process P1
        nap(int(random(500, 1000)))
        write("Hello")
    end
    process P2
        nap(int(random(500, 1000)))
        write("world")
    end
end ParallelHello
```

◆ The `nap(int(random(500, 1000)))` statement suspends the process for some time t such that $0.5 \leq t < 1.0$ seconds.

certain resource[5] is available before trying to use it. The value of the semaphore is the number of resources of the relevant kind that are currently available for use.

- If a semaphore is *set* (the value is greater than 0) a process is allowed to claim one resource. The value of the semaphore is immediately *reset* (decreased by one) to reflect that one less resource is available.
- When a process is ready to release a resource, it *sets* the semaphore (increasing the value by one) to reflect that the resource is again available for use by other processes.
- If a process wants to use a resource and the semaphore is *unset* (the value is 0, indicating that none are available), the process *blocks* and does not resume execution until the semaphore again becomes set.

In SR the following statements[6] manipulate semaphores:

- `P ( ⟨semaphore⟩ )`: blocks the current process until ⟨*semaphore*⟩ is set, then immediately resets it in one atomic (indivisible) action.
- `V ( ⟨semaphore⟩ )` : sets ⟨*semaphore*⟩.

Program 6 demonstrates using a semaphore to synchronize processes. In this case, the process P_1 starts, naps, writes "Hello", and then sets the semaphore `done`. P_2 most likely starts before P_1 stops napping, but is forced to wait for `done` to be set. When `done` is set, P_2 *unblocks*, writes "world" and exits.

Program 6 Using a semaphore to synchronize processes.

```
resource ParallelHello ()
    sem done := 0
    process P1
        nap(int(random(500, 1000)))
        write("Hello")
        V(done)
    end
    process P2
        P(done)
        write("world")
    end
end ParallelHello
```

◆ A semaphore initialized to 0 is known as a *blocking semaphore*.

[5] In this section, "resource" refers to program or system resources such as shared memory, devices, etc. rather than the source code structure `resource`.

[6] V stands for *verhoog* (Dutch for "increase") and P for *probeer te verlagen* ("try-and-decrease"). To aid memory, the P can be read as a symbol for a hand grabbing something, while V is an open hand releasing that something.

In a similar situation, a number of processes first perform a setup procedure, and then start processing data:

```
resource SetupAndGo ()
    process p (i := 1 to 10)
        # ...setup procedure...
        # ...data processing...
    end
end
```

Some of the processes might start processing data before all processes have finished setting up. If we don't want that, we need to synchronize the processes, as in program 7. Here, the `coordinator` process acts as an airline clerk that collects boarding cards

Program 7 Barrier synchronization.

```
resource SetupAndGo ()
    const N := 10
    sem done := 0
    sem continue := 0

    process p (i := 1 to N)
        # ...setup procedure...
        V(done)
        P(continue)
        # ...data processing...
    end

    process coordinator
        fa i := 1 to N ->
            P(done)
        af
        fa i := 1 to N ->
            V(continue)
        af
    end
end
```

◆The `fa···af` statement executes its body once for every value in the series $1 \ldots N$, setting the counter variable `i` to that value before every iteration (see page 24).

from all passengers before letting them board the plane one by one.

When a process uses a shared resource (such as a global variable), it is sometimes necessary to update the resource by a sequence of non-atomic actions. Such a sequence is known as a *critical section*. If more than one process is executing in the critical section concurrently, the resource might be left in an inconsistent state. To avoid this, processes must be coded in such a way that as soon as one process enters a critical section, other processes are unable to enter theirs. This is known as *mutual exclusion*, and one way to enforce it is to use semaphores.

Program 8 A program using mutual exclusion to avoid a race condition.

```
resource Mutex ()
    sem mutex := 1
    var n: int := 0
    process p(j := 1 to 2)
        var a : int
        fa i := 1 to 10 ->
            P(mutex)
            # critical section begins
            a := n
            nap(int(random(100, 500)))
            n := a + 1
            # critical section ends
            V(mutex)
        af
    end
    final
        write("n = ", n)
    end
end
```

◆The `final`···`end` section encloses a set of statements that are executed when the resource is destroyed. It is similar to a class destructor in C++.

In program 8, the semaphore `mutex` is used to let a process exclude others from the critical section. By executing `P(mutex)`, the process claims access to the critical section, and by executing `V(mutex)` the process releases it. The other process is blocked at the `P(mutex)` statement until the semaphore is set again.

Shared memory restriction

Semaphores can only be used to synchronize processes that share memory.

3.2 Message passing

Manipulation of a semaphore is a bit like passing a message from one process to another. If you add the ability to carry data, you have *message passing*, which is another way to synchronize processes. A *message* consists of the name of an operation and a (possibly empty) list of arguments. The message is *passed* by invoking the operation and *handled* by servicing it.

There are two different modes of message passing: 1) *synchronous* message passing where the process that passes the message blocks until the message has been handled, and 2) *asynchronous* message passing where the process continues to execute after passing the message. In either mode, the process that handles the message blocks until the message is passed.

When a message is passed *asynchronously*, the acts of passing and handling the message do not need to be completed in order or even happen in the same time span.

asynchronous message passing

Invoked by	Serviced by	Mode
`send`	`receive`	asynchronous
`call`	`receive`	synchronous

Table 1: Different modes of message passing.

There is a potential problem with this: if the message involves data sharing between the invoking and serving processes, there is in effect a race condition regarding those data. If a new message is passed before the previous message has been fully handled, the message data will be inconsistent.

To avoid this problem, the message data should instead be passed as arguments to the operation (as in program 9), in which case the data will be kept in a buffer, or "mailbox", until the message is handled.

Program 9 A program using asynchronous message passing with data buffering.

```
resource AsynchHello ()
    op out(string[5])
    process P1
        send out("hello")
        send out("world")
    end
    process P2
        var s : string[5]
        fa i := 1 to 2 ->
            receive out(s) ; write(s)
        af
    end
end
```

When a message is passed synchronously, the process that acts first must wait until the other process does its part. In program 10, there is no need for a mailbox, because P_1 can't pass any further messages until the previous one has been handled.

synchronous message passing

Return value restriction

An operation that returns a value may not be serviced by `receive`.

3.3 Service by proc

Another way to service an operation is with a `proc`, in which case a new process is started in response to the operation being invoked. The body of the `proc`, containing local declarations and statements, is executed within that process.

Again, there are two different modes depending on whether the invoking process blocks or not: 1) *dynamic process creation*, where the invoking process continues to run independently of the servicing process, and 2) *(remote) procedure call*, where the invoking process blocks until the servicing process is done.

Program 10 A program using synchronous message passing.

```
resource SynchHello ()
    op out()
    var s : string[5]
    process P1
        s := "hello" ; call out()
        s := "world" ; call out()
    end
    process P2
        fa i := 1 to 2 ->
            receive out() ; write(s)
        af
    end
end
```

◆The keyword `call` isn't necessary; the message by itself constitutes an implied call.

Invoked by	Serviced by	Mode
`send`	`proc`	dynamic process creation
`call`	`proc`	(remote) procedure call

Table 2: Different modes of service by `proc`.

Invoking an operation serviced by a `proc` asynchronously (using `send`) is known as *dynamic process creation*. Every time the operation is invoked in this way, a new process starts and continues running until it is interrupted or runs out of statements to execute (see program 11 for an example). dynamic process creation

Invoking an operation serviced by a `proc` synchronously (with `call`) is a *remote procedure call* if the `call` and `proc` statements are in different resources (see programs 2 and 12), or just a procedure call if made within the same resource. remote procedure call

Array declarator restriction

An operation whose declarator is an array declarator (see section 4.1) may not be serviced by a `proc`.

3.4 Rendezvous

The third way to service an operation is within a clause in an `in` ('input') statement. Synchronously invoking an operation to be serviced by `in` is called (extended) *rendezvous*.

In program 13, P_2 executes an `in` statement and blocks, waiting for one of its clauses to be activated. P_1 executes `call` and blocks. This activates one of the clauses, unblocking P_2. When the clause is done, both processes are left in an unblocked state.

Program 11 A program that uses dynamic process creation to create five processes, then kills them as soon as they report in.

```
resource DynamicHello ()
    op out(int; string[5])
    op done()
    op quit()
    proc out(id, s)
        write("Process", id, "says", s)
        send done()
        receive quit()
        write("Process", id, "says goodbye")
    end
    fa i := 1 to 5 -> send out(i, "hello") af
    fa i := 1 to 5 ->
        receive done()
        send quit()
    af
end
```

Program 12 A program using remote procedure call.

```
resource Printer
    op out(string[5])
body Printer ()
    proc out(s)
        write(s)
    end
end
resource RPCHello ()
    import Printer
    var pc : cap Printer := create Printer()
    pc.out("hello")
    # or, equivalently:
    call pc.out("hello")
end
```

Program 13 A program using rendezvous.

```
resource RendezvousHello ()
    op out(string[5]; int)
    op line()
    process P1
        out("hello", 0)
        out("hello", 2)
        line()
        stop
    end
    process P2
        do true ->
            in out(s, n) st n > 0 ->
                fa i := 1 to n ->
                    write(s)
                af
            [] out(s, n) st n = 0 ->
                write()
            [] line() ->
                write("-----")
            ni
        od
    end
end
```

- The `stop` statement shuts down all processes and quits the program.
- `st` = such that.

Like a `proc` server, the `in` statement allows returns values:

```
...
[] foo(a) returns b ->
    # do stuff and set b
...
```

Single message restriction

In order to avoid race conditions, the `in` statement will only handle one message at a time.

4 The language in detail

In this section, I sometimes use the following conventions when explaining how to write SR code:

1. Source code fragments are written in `typewriter style`
2. Metasyntactical placeholders are written as ⟨*string*⟩
3. Optional elements are enclosed in [square brackets]

Example:

`put(`[⟨*`file handle`*⟩`, `]⟨*`string`*⟩`)`

is supposed to show that one can write source code that begins with the word `put` and an opening parenthesis, an optional expression that amounts to a file handle together with a comma, and then any sort of expression that can be understood as a string, and finally a closing parenthesis.

4.1 Declarators and identifiers

A *declarator* is used to name a constant, semaphore, operation, or variable in a declaration. An *identifier* is used to reference any such object in a statement.

A *simple declarator* is just a *word*[7] that isn't reserved (see section 5.1):

```
a  Trinity  catch22  go_west  NUM
```

By convention, names beginning with uppercase letters are used for 1) constants (the whole name is uppercase) and 2) resources (mixes upper and lower case).

An *array declarator* is a simple declarator followed by a comma-separated *bounds list* within brackets[8]:

```
a[lower:upper]
a[upper]
a[low1:upp1, low2:upp2]
a[b1, b2]
```

[7] That is, an unbroken sequence of letters, digits, and the underscore (_) character that starts with a letter.

[8] It is (mostly) possible to use variable to specify a bounds value.

Each bounds value can be specified as a pair of integers $i : j$. The number of elements will be $1 + j - i$ and the elements are numbered $i \ldots j$. Alternatively, the bounds value can be specified by a single integer N, which is the equivalent to writing $1 : N$.

A bounds value for an array declarator used in a formal parameter can be left unspecified, to allow differently sized arrays to be passed in[9].

```
var a[10], b[200] : int
op foo([*]int)
proc foo(x)
    fa i := 1 to ub(x) -> write(x[i]) af
end
foo(a) ; foo(b)
```

4.2 Parameters

Parameters are used to manage data transfer between caller and callee. *Formal parameters* (*formals* or just *parameters*) are used for declarations, and *actual parameters* (*actuals* or *arguments*) for calls.

Except in some built-in functions, SR does not support variable-length parameter lists or default parameter values.

Formal parameters are written in a semi-colon-separated list. Each parameter has the form

[⟨*mode*⟩]⟨*declarator-list*⟩ : ⟨*type*⟩

where

- The ⟨*mode*⟩ can be specified as `val`, `var`, `res`, or `ref`, or omitted (in which case `val` is assumed)
- A ⟨*declarator-list*⟩ is a comma-separated list of declarators
- In an operation declaration, only the type(s) of the parameters need to be given. E.g., instead of `a, b : int; c : real`, the operation's parameters can be declared as `int; int; real`

The mode of a parameter determines how the corresponding argument is evaluated during delivery. The modes are:

Call by value (`val`): the argument is evaluated and the value is copied into the parameter.

Call by reference (`ref`): the argument is not evaluated. Instead, the parameter becomes an alias for the argument. All changes to the argument also change the parameter, and vice versa.

[9] The built-in functions `lb(array)` and `ub(array)` return the lower and upper bounds, respectively, of the indicated arrays.

Call by copy-restore (`var`): the argument (which must be assignable) is evaluated and the value is copied into the parameter. Upon return, the value of the parameter is copied back into the argument. If the argument's value is changed while the operation is executing, the parameter is unaffected, and the argument's new value will be overwritten when the operation returns.

Call by result (`res`): the argument must be assignable. The parameter's value is undefined. Upon return, the value of the parameter is copied back into the argument.

A list of actual parameters is a list of expressions separated by commas. The following requirements apply:

- the list of actual parameters must have the same number of items as the list of formal parameters
- each argument must have a type compatible with the type of the corresponding parameter
- each argument must have a form consistent with the passing mode of the corresponding parameter (e.g., no constants where an assignable argument is required)

4.3 Types

SR uses the following types:

◇ *Integer* – a whole number in the range $-2^{31} \ldots +2^{31}-1$ (or the capacity of a machine word): `int`

```
var n : int := 57
```

◇ *Real* – a floating point number: `real`

```
var f : real := 1.086e4
```

(This value is the same as 10860, or 1.086×10^4)

◇ *Boolean* – one of the values `true` or `false` `bool`

```
var q : bool := true
```

◇ *Character* – any printable character within single quotes, and some unprintable characters as a \⟨*character*⟩ code within single quotes. Mostly the same as in C++/Java. `char`

```
var c : char := 'w'
var c : char := '\n'
```

◇ *String* – an abstract text type that supports concatenation. A string constant is a sequence of character constants within double quotes. `string`

```
var s : string[30]
s := "O Captain!" || " my Captain!"
write(s)
```

prints “O Captain! my Captain!”

◇ *Array* – An array is a single- or multidimensional collection of elements of the same type. An array variable can be declared with either an array declarator: *array*

```
var v[10] : int
```

or a simple declarator:

```
var v : [10] int
```

The elements of the array can be initialized by batch:

```
var w[10] : int := ([10]0)
```

resulting in

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

or by list:

```
var x[5] : int := (9, 8, 7, 6, 5)
```

resulting in

$$\begin{bmatrix} 9 & 8 & 7 & 6 & 5 \end{bmatrix}$$

or by a list/batch combination:

```
var y[5] : int := (9, 8, [3] 0)
```

resulting in

$$\begin{bmatrix} 9 & 8 & 0 & 0 & 0 \end{bmatrix}$$

Multidimensional arrays are initialized recursively:

```
var z[5, 3] : char := ([5]([3]'!'))
```

resulting in

$$\begin{bmatrix} ! & ! & ! \\ ! & ! & ! \\ ! & ! & ! \\ ! & ! & ! \\ ! & ! & ! \end{bmatrix}$$

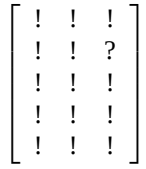
Element access:

```
write(x[4])
```

prints 6;

```
z[2, 3] := '?'
```

results in



◇ *Enumeration* – An *enumerated type* is an ordered type with identifiers for values. `enum`

```
type fruits = enum(apple, orange, banana)
```

creates an enumerated type called `fruits`. Its domain consists of the values `apple`, `orange`, `banana` and all operations that are defined for ordered types [10].

The values can be compared based on enumeration order (`apple` $<$ `orange` $<$ `banana`):

```
var a : fruits := orange
if a > apple -> write("Sure is") fi
```

◇ *Record* – A *record type* is an unordered, aggregate type with each element (field) having a name and a type. Each field can be read or assigned to, and a record object can be assigned to a record variable of the same or equivalent type (implicitly copying the values of the individual fields). `rec`

```
type point = rec(x : int ; y : int)
var pt1, pt2 : point
pt1.x := 8; pt1.y := -5
pt2 := pt1
```

◇ *Pointer* – The data domain of a pointer to type T consists of 1) all memory addresses that are valid for objects of type T, and 2) the special pointer value `null`, which is invariably an invalid address. `ptr`

```
var ip : ptr int
```

declares `ip` to be a pointer to `int` objects. The value of `ip` is indeterminate and unsafe to use at this point.

The address of an object, or the value of another pointer, can be assigned to a pointer if the types are compatible.

```
ip := @a
```

If `a` is an `int` object, this assigns the address of `a` to `ip`.

[10] Namely: increment / decrement and equality / comparison operators; the standard functions max, min, pred, succ, low, high; most type conversions.

```
ip := null
```

The value `null` (see above) can always be assigned to a pointer.

To indicate that we want to deal with the object that the pointer points to rather than the pointer itself, we attach a ^ (caret) operator to the pointer name. If the pointer `p` holds the address of an object, `p^` is an alias of that object. If `p` holds the value `null`, evaluating the expression `p^` will cause a "Segmentation fault" run-time error.

```
ip^:= 27
```

This assignment does not change `ip`; instead the object pointed to by `ip` (`a`) gets the assignment.

```
b := ip^
```

assigns the value pointed to by `ip` (the value of `a`) to `b`.

Pointers are useful for keeping tabs on dynamically allocated objects:

```
ip := new(int)
```

allocates a new `int` object and stores its address in `ip`.

```
free(ip)
```

releases the allocated object pointed to by `ip`. The address stored in `ip` doesn't change, but is now unsafe to use.

◇ *Capability* – A *capability* is a reference handle to an instance of a resource. `cap`

```
var c : cap Foo
```

declares `c` to be a capability of `Foo`.

At this point, `c` doesn't actually refer to any instance.

```
c := create Foo()
```

now, `c` refers to an enlivened instance and can be used to access operations in `Foo`'s interface.

```
var x : cap Foo := c
```

makes `x` and `c` refer to the same instance.

```
destroy c
```

removes the instance; both `x` and `c` are now unusable.

◇ *File* – A *file handle* is a reference handle to a file in the operating system's file system. `file`

```
var f : file
```

declares `f` to be a file handle.

Function	...reads...
`get([⟨h⟩, ]⟨variable⟩)`	a single string
`read([⟨h⟩, ]⟨vars⟩)`	multiple values
`scanf([⟨h⟩, ]⟨format⟩, ⟨vars⟩)`	values extracted from other text

Table 3: Input functions

A file handle must refer to an opened file to be usable.

`f := open(⟨pathname⟩, ⟨mode⟩)`

⟨*pathname*⟩ is a string; ⟨*mode*⟩ is either `READ`, `WRITE`, or `READWRITE`. Sets `f` to `null` if the file couldn't be opened.

`close(f)`

closes the file that the file handle refers to.

`remove(⟨pathname⟩)`

erases the file ⟨*pathname*⟩; returns `true` for success or `false` for failure.

`seek(⟨handle⟩, ⟨mode⟩, ⟨offset⟩)`

sets the read or write position in the file referred to by ⟨*handle*⟩. ⟨*offset*⟩ is an integer value, and ⟨*mode*⟩ is one of the following:

Mode	Sets file position to
`ABSOLUTE`	⟨*offset*⟩
`RELATIVE`	current position + ⟨*offset*⟩
`EXTEND`	end of file + ⟨*offset*⟩

`where(⟨handle⟩)`

returns the current read / write position in the file referred to by ⟨*handle*⟩.

Input functions (`get`, `read`, `scanf`): if a handle ⟨*h*⟩ is provided, input is read from the file that it refers to, otherwise input is read from `stdin` (*standard input*, normally the keyboard). The input is stored in one or more variables that are listed as parameters. See table 3.

`sscanf(⟨buffer⟩, ⟨format⟩, ⟨variables⟩)`

as `scanf`, but reads its values from the string ⟨*buffer*⟩.

Output functions (`put`, `write`, `writes`, `printf`): if a ⟨*handle*⟩ is provided, output is printed to the file that it refers to, otherwise it is printed to `stdout` (*standard output*, normally the screen). See table 4.

`sprintf(⟨buffer⟩, ⟨format⟩, ⟨values⟩)`

as `printf`, but stores the resulting string in ⟨*buffer*⟩.

The functions `scanf`, `sscanf`, `printf`, and `sprintf` work mostly like their C++ equivalents.

Function	...prints...
`put(`[⟨*h*⟩`, `]⟨*string*⟩`)`	a single string
`write(`[⟨*h*⟩`, `]⟨*values*⟩`)`	multiple values + newline
`write(`[⟨*h*⟩`, `]⟨*values*⟩`)`	multiple values
`printf(`[⟨*h*⟩`, `]⟨*format*⟩`, `⟨*values*⟩`)`	values inserted into other text

Table 4: Output functions

Type casts

In most cases, it is possible to cast a value to another type by using the expression $T(V)$ where T is the new type and V is the value.

```
int(3.7)
```

⇒ the integer 3

```
string(25)
```

⇒ the string ”25”

There is one kind of automatic type coercion: an integer value can be used where a real value is expected, and it will automatically be coerced to real. See also the relevant section of Andrews and Olsson [1993].

4.4 Top-level structures

The top-level structures of an SR program are *resources* and *globals*.

The basic form of a `resource` is `resource`

```
resource ⟨identifier⟩ ⟨parameters⟩
    ⟨statements⟩
end[⟨identifier⟩]
```

where ⟨*identifier*⟩ is the name of the resource.

An importable `resource` (note that the parameter list changes place):

```
resource ⟨identifier⟩
    ⟨specification-part⟩
body ⟨identifier⟩ ⟨parameters⟩
    ⟨implementation-part⟩
end [⟨identifier⟩]
```

A `final` block can be specified within a `resource`. The statements of the block will be executed when a `resource` instance is destroyed. `final`

```
final
    ⟨statements⟩
end
```

A `global` is similar to a `resource` (see section 2.2): `global`

```
global ⟨identifier⟩
    ⟨statements⟩
end [⟨identifier⟩]
```

or

```
global ⟨identifier⟩
    ⟨specification-part⟩
body ⟨identifier⟩
    ⟨implementation-part⟩
end [⟨identifier⟩]
```

4.5 Interface statements

Interface statements are statements that are used in the specification part of a resource.

Importing a `resource` (or `global`) makes the declarations in that `resource`'s specification part visible to the importing `resource`. `import`

```
import ⟨identifiers⟩
```

where ⟨*identifiers*⟩ can be the name of a single `resource`, or a comma-separated list of `resource` names.

A `resource` can also make declarations of its own:

Constant `const`

```
const ⟨declarator⟩ [: type] := ⟨expression⟩
```

Semaphore `sem`

```
sem ⟨declarator⟩ := ⟨expression⟩
```

Type (see section 4.3 for examples of type expressions) `type`

```
type ⟨declarator⟩ : ⟨expression⟩
```

Operation (see section 4.2 for notes on operation declaration parameters) `op`

```
op ⟨declarator⟩ ( ⟨parameters⟩ ) [returns [⟨declarator⟩:] ⟨type⟩]
op ⟨declarator⟩ : ⟨optype⟩
```

Operation-type `optype`

```
optype ⟨declarator⟩ ( ⟨parameters⟩ ) [returns ⟨type⟩]
```

4.6 Control statements

SR's *alternation* (or *selection*) statement has the form of a list of clauses (separated by `[]` keywords) between the keywords `if` and `fi`. Each clause has the form: `if ··· fi`

⟨*condition*⟩ `->` ⟨*statements*⟩

The simplest kind of alternation decides whether to execute a statement or not (based on the truth value of the condition):

```
if a > 10 ->
    write("Overload!")
fi
```

A fall-back, or default, execution branch means that the alternation will always execute *something* regardless of whether any conditions are fulfilled.

```
if a > 10 ->
    write("Overload!")
[] else ->
    write("Load OK!")
fi
```

The keyword `else` denotes a special condition that is true only if none of the other conditions are.

```
# potential bug alert!
if a > 20 ->
    write("Severe overload!")
[] a > 10 ->
    write("Overload!")
[] else ->
    write("Load OK!")
fi
```

SR does not quite guarantee that the conditions are tested in the order they are written in the source code. When `a` $>$ 20, both of the first two conditions are true and either one of the clauses might be chosen for execution.

Always make sure that only one condition can be true at any time:

```
if a > 20 ->
    write("Severe overload!")
[] a > 10 and a <= 20 ->
    write("Overload!")
[] else ->
    write("Load OK!")
fi
```

SR's *indefinite iteration* statement has the form of a list of clauses (separated by `[]` keywords) between the keywords `do` and `od`. Each clause has the form: `do ··· od`

⟨*condition*⟩ `->` ⟨*statements*⟩

The statement will select one of its clauses, execute the statements in that clause, and then re-select and execute until no conditions can be fulfilled.

```
a := check_load()
do a > 20 ->
    write("Severe overload!")
    a := check_load()
[] a > 10 and a <= 20 ->
    write("Overload!")
    a := check_load()
[] a > 0 and a <= 10 ->
    # do nothing
    a := check_load()
od
```

An intentionally endless loop can be written as
`do true -> ... od`
which allows us to rewrite the previous example like this:

```
do true ->
    a := check_load()
    if a > 20 ->
        write("Severe overload!")
    [] a > 10 and a <= 20 ->
        write("Overload!")
    [] a > 0 and a <= 10 ->
     # do nothing
    [] else ->
        exit
    fi
od
```

SR's *definite iteration* statement has a list of quantifiers (separated by commas) followed by a body of statements, all between the keywords `fa` (for all) and `af`. A quantifier has the form `fa ··· af`

⟨*variable*⟩ `:=` ⟨*begin*⟩ `to` ⟨*end*⟩

where ⟨*variable*⟩ is an implicitly declared variable whose extent is the `fa` statement.

The statement binds the quantifier variable to the value ⟨*begin*⟩ and executes the body. It then rebinds ⟨*variable*⟩ to ⟨*variable*⟩+1 and executes the body again, repeating until the new value of ⟨*variable*⟩ > ⟨*end*⟩.

In this example, the body is executed ten times, each time with the variable `i` bound to one of the values in the series $\{1, 2, \ldots, 10\}$:

```
fa i := 1 to 10 ->
    write("I must not tell lies")
af
```

The keyword `downto` can be used instead of `to`, making the quantifier expression count down instead of up:

```
fa i := 1 to 3, j := 3 downto 1 ->
    write(i, j)
af
```

As stated above, ⟨*variable*⟩ is set to ⟨*variable*⟩+1 (or ⟨*variable*⟩−1 if `downto` is used) before the next round of iteration. It is possible to change this using the keyword `by`:

```
i := 10 downto 1 by -2
```

The values that the quantifier variable will be bound to can be further restricted using an inclusion test[11]:

```
fa i := 1 to 20 by 2 st i >= 5 ->
    write(i)
af
```

produces the same output as

```
fa i := 1 to 20 by 2 ->
    if i >= 5 ->
        write(i)
    fi
af
```

but note that the latter variant will still execute its body ten times.

The simplest statements are `skip`, `exit`, `next`, and `stop`. A `skip` statement causes the current process to give up its time-slice. A `stop` statement stops all processes, destroys any live resources, and quits the whole program. `skip` `stop`

The `stop` statement can optionally take an integer argument, e.g. `stop(1)`. The argument's value is returned to the operating system as an application result code. Also, if an argument is given for `stop`, the final blocks of live resources will *not* be executed.

Within an iteration statement, `next` abandons the current round and begins a new one, while `exit` leaves the loop, see figure 1. `next` `exit`

[11] `st` stands for *such that*.

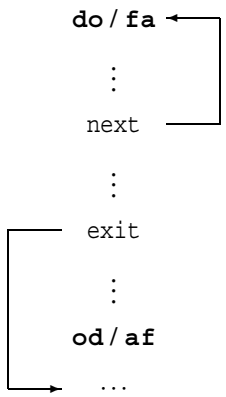


Figure 1: The `next` and `exit` statements

4.7 Interaction statements

Interaction statements are used to synchronize processes.

`V(`⟨*semaphore*⟩`)` sets a semaphore. `V`

`P(`⟨*semaphore*⟩`)` does a blocking reset of a semaphore. `P`

(See section 3.1)

`call` ⟨*operation*⟩ `(` ⟨*arguments*⟩ `)` invokes ⟨*operation*⟩ and blocks the current process. `call`

`send` ⟨*operation*⟩ `(` ⟨*arguments*⟩ `)` invokes ⟨*operation*⟩ without blocking. `send`

`receive` ⟨*operation*⟩ ⟨*parameters*⟩ blocks the current process until the corresponding message is passed, then binds the parameters (which must be declared variables) to the message's arguments. If the invoking process was blocked, it becomes unblocked. (See section 3.2) `receive`

`proc` ⟨*operation*⟩ ⟨*parameters*⟩ ⟨*body*⟩ defines a potential process to be started when ⟨*operation*⟩ is invoked. (See section 3.3) `proc`

`in` ⟨*clause-list*⟩ `ni` blocks the current process until a message that matches one of the clauses is passed (See section 3.4) `in··ni`

This code:

```
var a, b, c : int
in foo(x, y, z) ->
    a := x ; b := y ; c := z
ni
```

is equivalent to

```
var a, b, c : int
receive foo(a, b, c)
```

copying the values of the message arguments to the variables `a`, `b`, and `c` and unblocking the invoking process.

```
in foo(a, b) ->
    # do this
[] bar(x) ->
    # do that
[] else ->
    write("No message to handle, exiting in.")
ni
```

This is a *non-blocking* `in` statement. If none of the specified messages have been passed, the statements in the `else`-clause are executed and the process leaves the `in` statement.

```
in foo(a, b) st a > 20 ->
    # do this
[] foo(a, b) st a > 10 and a <= 20 ->
    # do that
ni
```

In this case, a *guard* (`st` ⟨*expression*⟩) is used to disambiguate events where the same operation being invoked, but the process' state is different. If *no* guard matches, the `in` statement will stay blocked, causing a deadlock where neither process can continue. The example could (almost) be rewritten as

```
in foo(a, b) ->
    if a > 20 ->
        # do this
    [] a > 10 and a <= 20 ->
        # do that
    fi
ni
```

except that in this case, the message *will* be handled (by executing the `if` statement) even if none of the clauses match.

Finally, a called process can unblock its caller using `return` or `reply`. The difference between them is that `return` ends the process that executes it, and `reply` doesn't. `return` `reply`

4.8 Definition statements

Variables `var`

`var` ⟨*declarator-list*⟩ `:` ⟨*type*⟩

`var` ⟨*declarator*⟩ `:` ⟨*type*⟩`[ :=` ⟨*initial-value*⟩`]`

Procedures: procedures can only be called from within the same `resource`. `procedure`

```
procedure ⟨id⟩ ( ⟨parameters⟩ ) [returns ⟨result⟩ : ⟨type⟩]
    ⟨body⟩
end [⟨id⟩]
```

where

- ⟨*id*⟩ is a simple declarator (see section 4.1) that names the procedure
- ⟨*parameters*⟩ is a list of formal parameters
- ⟨*result*⟩ is a simple declarator for a local-scope variable that is used inside a function to store the function's result value
- ⟨*type*⟩ is the result type of the function
- ⟨*body*⟩ is the statements that are executed when the procedure is called

Mutual recursion restriction

Procedures defined using `procedure` can't be mutually recursive: use `proc` servers instead.

Proc servers `proc`

```
proc ⟨op⟩ ( ⟨name-list⟩ ) [returns ⟨result⟩]
    ⟨body⟩
end [⟨op⟩]
```

where

- ⟨*op*⟩ is a simple declarator (see section 4.1) that names the operation the `proc` will serve
- ⟨*name-list*⟩ is a comma-separated list of parameter names (the parameter *types* are already specified in the operation declaration).
- ⟨*result*⟩ is a simple declarator for a local-scope variable that is used inside a function to store the function's result value
- ⟨*body*⟩ is the statements that are executed when the proc is called

Processes process

```
process ⟨id⟩
    ⟨body⟩
end [⟨id⟩]

process ⟨id⟩ ( ⟨quantifier-list⟩ )
    ⟨body⟩
end [⟨id⟩]
```

where

- ⟨*id*⟩ is a simple declarator (see section 4.1) that names the process
- ⟨*quantifier-list*⟩ is a comma-separated list of quantifiers (used in the definition of a process family).
- ⟨*body*⟩ is the statements that are executed when the process is started

Processes are started when their defining resource or global is created. A process definition acts as an abbreviation for a set of `op` / `proc` / `send` statements:

```
process P1
    # ...
end
```

is an abbreviation for

```
op P1()
proc P1()
    # ...
end
send P1()
```

Similarly,

```
process P2 (i := 1 to N)
    # ...
end
```

is an abbreviation for

```
op P2(int)
proc P2(i)
    # ...
end
fa i := 1 to N -> send P2(i) af
```

5 Reference

5.1 Reserved words

```
->          false       real
//          fi          rec
[]          file        receive
P           final       ref
V           forward     reply
af          global      res
and         high        resource
any         if          return
begin       import      returns
body        in          sem
bool        int         send
by          low         separate
call        mod         skip
cap         new         st
char        next        stderr
chars       ni          stdin
co          noop        stdout
const       not         stop
create      null        string
destroy     oc          to
do          od          true
downto      on          type
else        op          union
end         optype      val
enum        or          var
exit        proc        vm
extend      procedure   xor
external    process
fa          ptr
```

5.2 List separator symbols

Kind of list	Separator
formal parameter list	;
clause-list	[]
actual parameter list	,
array bounds list	,
declarator list	,
resource import list	,
quantifier list	,

5.3 Operators

Operator	Meaning	Type	Precedence
`<<:=`	Shift left by	Assignment	0
`>>:=`	Shift right by	Assignment	0
`||:=`	Join with	Assignment	0
`&:=`	Bitwise and with	Assignment	0
`|:=`	Bitwise or with	Assignment	0
`**:=`	Raise to the power of	Assignment	0
`%:=`	Remainder by	Assignment	0
`/:=`	Divide by	Assignment	0
`*:=`	Multiply by	Assignment	0
`+:=`	Increase by	Assignment	0
`-:=`	Decrease by	Assignment	0
`:=:`	Swap values	Assignment	0
`:=`	Assign	Assignment	0
`or`	Logical or	Disjunction	1
`xor`	Logical exclusive or	Disjunction	1
`|`	Bitwise or	Disjunction	1
`and`	Logical and	Conjunction	2
`&`	Bitwise and	Conjunction	2
`>`	Greater than	Comparison	3
`>=`	Greater than or equal to	Comparison	3
`<`	Less than or equal to	Comparison	3
`<=`	Less than or equal to	Comparison	3
`!=`	Not equal to	Equality	3
`~=`	Not equal to	Equality	3
`=`	Equal to	Equality	3
`<<`	Left shift	Bitwise shift	4
`>>`	Right shift	Bitwise shift	4
`+`	Addition	Addition	5
`-`	Subtraction	Addition	5
`||`	Concatenation	Addition	5
`*`	Multiplication	Multiplication	6
`/`	Division	Multiplication	6
`%`	Remainder	Multiplication	6
`mod`	Modulus	Multiplication	6
`**`	Exponentiation	Exponentiation	7
`not`	Logical not	(Prefix)	8
`~`	Bitwise invert	(Prefix)	8
`+`	Multiply by $+1$	(Prefix)	8
`-`	Multiply by -1	(Prefix)	8
`++`	Preincrement	(Prefix)	8
`--`	Predecrement	(Prefix)	8
`@`	Address of	(Prefix)	8
`?`	Number of invocations	(Prefix)	8
`++`	Postincrement	(Postfix)	9
`--`	Postdecrement	(Postfix)	9
`^`	Pointer dereference	(Postfix)	9

5.4 Standard functions

Function	Returns
numargs()	The number of command line arguments
getarg(0,x)	(Sets x to the name of the command)
getarg(n,x)	(Sets x to the nth command line argument)
abs(x)	The absolute value of x
min($x_1,\ldots,x_n$)	The smallest member of the list
max($x_1,\ldots,x_n$)	The largest member of the list
pred(x)	The predecessor of x
succ(x)	The successor of x
low(T)	The smallest representable value in type T
high(T)	The largest representable value in type T
lb(*array*)	The lower bound of the *array*
ub(*array*)	The upper bound of the *array*
length(s)	The number of characters in the string s
maxlength(s)	The capacity of string s
new(T)	The address of a new object of type T
free(*ptr*)	(Frees an object allocated with new)
sqrt(x)	The square root of x
log(x)	The (natural) logarithm of x
log(x,b)	The b-logarithm of x
exp(x)	The value of e raised to the power of x
exp(x,b)	The value of b raised to the power of x
ceil(x), floor(x), round(x)	The value of x, rounded to whole
sin(r), cos(r), tan(r)	The sine, cosine, or tangent of an angle r
asin(x), acos(x), atan(x)	The arc sine, cosine, or tangent of x
random()	A pseudorandom number $0 \leq n < 1$
random(x)	A pseudorandom number $0 \leq n < x$
random(a,b)	A pseudorandom number $a \leq n < b$
seed(x)	(Seeds the random number generator)
age()	The 'age' (in ms) of the virtual machine
nap(x)	(Blocks the executing process for x ms)
myresource()	The capability for the enclosing resource

5.5 Acknowledgements

Birgitta Lindström of the Computer Science and Languages department, University of Skövde, has provided invaluable guidance and feedback during the completion of this document.

5.6 Bibliography

References

Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*, chapter Appendix C: Operators and Predefined Functions. Benjamin-Cummings Publishing Co., Inc, 1993.

Ronald Olsson et al. A Language for Parallel and Distributed Programming. Technical Report TR92-09, Department of Computer Science, University of Arizona, 1992.

Stephen J. Hartley. An Operating Systems Laboratory Based on the SR (Synchronizing Resources) Programming Language. *Computer Science Education*, 3(3), 1992.

Jonas Mellin. A short introduction to SR: Compared and contrasted with C++. Technical report, Department of Computer Science, University of Skövde, 1997.

Links

Stephen J. Hartley provides excellent SR resources at
`http://www.mcs.drexel.edu/~shartley/OSusingSR/`

Appendix C of “The SR Programming Language: Concurrency in Practice” can be found online at
`http://www.cs.arizona.edu/mpd/language.html`