Parallelism, Distribution, and Synchronization in SR

Ronald A. Olsson

Division of Computer Science
University of California, Davis
Davis, CA 95616 U.S.A.

olsson@ucdavis.edu

Abstract

This paper introduces the newest version of the SR concurrent programming language, and illustrates how it provides support for different execution environments, ranging from shared-memory multiprocessors to distributed systems. SR uses a few well-integrated mechanisms for concurrency to provide flexible, yet efficient, support for concurrent programming.

1. Introduction

The SR concurrent programming language has been around, in one form or another, for over ten years. The earliest version, now called SR_0, contained mechanisms for asynchronous message passing and rendezvous [1, 2]. Its form of rendezvous, unique at the time, provided a means by which the process servicing a rendezvous could choose which invocation to service based on the values of invocation parameters. Experience using SR_0 substantiated the general appropriateness of the language, but also pointed out several deficiencies. That experience lead us to redesign the language [3]. The result (SR version 1) [4, 5] provided additional mechanisms for remote procedure call, dynamic process creation, and semaphores, as well as a means for specifying distribution of program modules.

Experience using version 1 has lead to further evolution of the language. Version 2 of SR retains much of version 1's structure. However, it also enhances the mechanisms that support sharing of objects. This sharing is especially important in shared-memory environments, for which earlier versions of SR were not not really intended. (It is also important for supporting libraries, e.g., mathematical and windowing libraries.)

SR supports many 'features' useful for concurrent programming. However, our goals have always been to keep the language simple and easy to use, while at the same time to provide an efficient implementation. We achieve these goals by integrating common notions, both sequential and concurrent, into a few powerful mechanisms. We implement these mechanisms as part of a complete language to determine their feasibility and cost, to gain hands-on experience, and to provide a tool that can be used for research and teaching.

This paper introduces version 2 of SR, henceforth referred to as simply SR. It illustrates how a single language can provide support for different execution environments, ranging from shared-memory multiprocessors to distributed systems. This paper focuses on the highlights of the language; details can be found in [6].

The rest of this paper is organized as follows. Section 2 gives an overview of the SR model of computation. Section 3 shows—in part by means of examples—how synchronization, parallelism, and distribution are supported in SR. Finally, Section 4 contains some concluding remarks, including a brief discussion of some current research related to SR.

2. SR Model of Computation

An SR program can execute within multiple address spaces, which can be located on multiple physical machines. Processes within a single address space can also share objects. Thus, SR supports programming in distributed environments as well as in shared-memory environments.

The SR model of computation allows a program to be split into one or more address spaces called *virtual machines.* Each virtual machine defines an address space on one physical machine. Virtual machines are created dynamically; they are referenced indirectly through *capability variables.* Virtual machines contain instances of two related kinds of modular components: *globals* and *resources.*

Each of these components contains two parts: a specification (aka a spec) and an implementation (aka a body). An import mechanism is used to make available in one component objects declared in the spec of another. In these two ways, globals and resources are similar to modules in Modula-2 [7] but they are created differently. Instances of resources are created dynamically, by an

CH2961-1/91/0000/0407$01.00 © 1991 IEEE

explicit create statement. These instances, and the services they provide, are referenced indirectly through *resource capability variables.* Instances of globals are also created dynamically. However, they are created implicitly as needed—specifically, when an instance of an importing resource or global is itself created and an instance of that global does not already exist on the same virtual machine. Furthermore, each virtual machine can contain only a single instance of a global. Globals, and the services they provide, can be referenced directly through their names.

The spec of a global or resource can contain declarations of types, constants, and operations; a global's spec can additionally contain declarations of variables. An operation defines a service that must be provided somewhere in the program. It can be considered a generalization of a procedure: it has a name, and can take parameters and return a result. An operation declared in a resource's spec must be serviced in that resource's body. Similarly, an operation declared in a global's spec *can* be serviced in the global's body; it can also be serviced within an importing resource or global.

The body of a global or a resource can contain declarations of additional objects; these objects are visible only within the body, not to any importer. Bodies also contain code that, among other things, services operations. The code is split into units called *processes* and *procs*. Processes are created implicitly when the enclosing global or resource is created. Instances of procs are created when they are invoked; they too execute as independent processes. All processes created within a global or a resource execute on the same virtual machine on which the enclosing global or resource was created. Processes and procs can declare additional variables and operations; they must contain the code that services invocations of any locally declared operations.

Figure 1 summarizes SR's model of computation. The ellipses indicate that repetition is allowed. In its simplest form, a program consists of a single virtual machine executing on one physical machine, possibly a shared-memory multiprocessor. A program can also consist of multiple virtual machines executing on multiple physical machines. In this paper, how data and processor(s) are shared within a virtual machine is called *parallelism;* how virtual machines are placed on physical machines is called *distribution.*

Processes on the same or different virtual machines can communicate through operation invocation. Operations may be invoked: directly through the operation's declared name; through a resource capability variable; or indirectly through an *operation capability variable.* These capability variables are strongly typed and may point to operations with structurally equivalent signatures. They may also be passed as parameters to operations

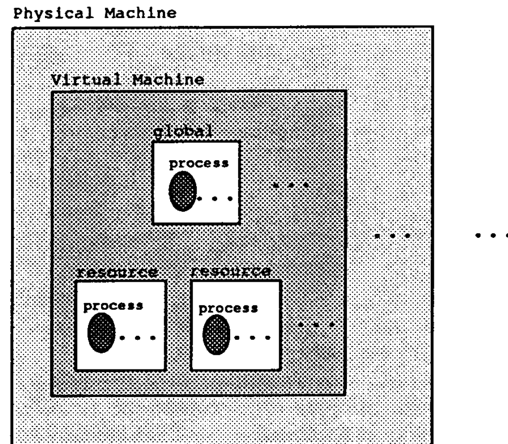

Figure 1. SR model of computation

during invocation or to resources during resource creation, allowing processes in different resource instances, on possibly different virtual machines, to communicate.

Communication between processes is independent of their virtual machine locations. For example, message passing between processes in the same resource instance has the same syntax and semantics as message passing between processes on different virtual machines.

3. Language Support

This section describes how synchronization, parallelism, and distribution are supported in SR. Examples are used to illustrate the key points.

3.1. Support for Synchronization

SR is rich in the functionality it provides for concurrent programming: dynamic process creation, semaphores, message passing, remote procedure call, and rendezvous. However, these are all provided through a single mechanism: the operation.

Operations can be invoked in two ways, synchronously (`call`) or asynchronously (`send`), and can be serviced in one of two ways, by procs or by input statements (`in`). This yields the following four combinations:

Invoke	*Service*	*Effect*
`call`	`proc`	(possibly remote) procedure call
`call`	`in`	rendezvous
`send`	`proc`	dynamic process creation
`send`	`in`	asynchronous message passing

One virtue of this approach is that it allows the

declaration of an operation to be separated from the code that services it (i.e., proc or input statements). This allows resource and global specifications to be written and used without concern as to how an operation is serviced.

SR provides abbreviations of the above basic mechanisms to simplify the most common usages such as background process creation, semaphores, and simple asynchronous message passing. Briefly: a `process` is an abbreviation for a proc and an implicit send to it when the enclosing resource or global is created; a `sem` declaration is an abbreviation for an operation declaration, a `P` is an abbreviation for an input statement, and a `V` is an abbreviation for a send; and a `receive` statement is an abbreviation for a simple form of input statement.

SR also provides three statements—forward, return, and reply—that provide additional flexibility in servicing invocations. Only reply is used in an example later in this paper. A process executing a reply statement causes the invocation being serviced to complete; result parameters and return values are immediately passed back to the caller. The process that executes a reply statement then continues execution with the statement following the reply.

To illustrate operations, consider the following scenario. A collection of printers is managed by a manager process. User processes make requests to output to particular printers. The manager assigns requests for service for each printer in a shortest-job-next (SJN) fashion according to the number of lines to be printed. The program in Figure 2 outlines this scenario. When `mult_print` is created, `N` instances of the background process `user` and one instance of `manager` are created. Users request printing using a synchronous call; when done printing, they release the printer using an asynchronous send. The manager services one invocation of `request` or `release` on each iteration of its loop. The `st` (such-that) clause on the operation guard specifies that a request by a user for printer `id` can be accepted only if the printer is free; the `by` specifies that if multiple invocations are pending for a particular printer, the invocation with the minimum number of lines should be serviced first. Note how the such-that and by clauses reference the invocation parameters, `id` and `lines`.

3.2. Support for Parallelism

SR provides support for parallelism on several levels. First, processes within a resource instance can share variables. They can coordinate access to shared variables through shared semaphores or other operations declared within the resource. Second, processes that execute in possibly different resource instances but on the same virtual machine can share variables and operations declared in the spec of globals.

```
resource mult_print()
  op request(id, lines: int)
  op release(id)
  const N := ... # number of users
  const P := ... # number of printers
  process user(i := 1 to N)
    do true ->
        ...
      call request(id, lines)
      # use printer number id
      send release(id)
        ...
    od
  end
  process manager
    var free[1:P] := ([P] true)
    do true ->
      in request(id,lines) st free[id]
                           by lines ->
         free[id] := false
       [] release(id) ->
         free[id] := true
      ni
    od
  end
end
```

Figure 2. Multiple printer manager and users

Consider, for example, a program that is to be written for execution on a shared-memory multiprocessor. It might be written as a single resource program, with processes sharing variables and operations declared at the resource level. For a program of any complexity, though, splitting the program into multiple resources is desirable. This kind of structure is possible, too. Resources can be created on a single virtual machine, with shared variables and operations declared in one or more globals.

As a concrete example, consider barrier synchronization, a common pattern of synchronization used in parallel numerical algorithms. It is typically used in iterative algorithms, such as techniques for solving partial differential equations, that require all tasks to complete one iteration before they can begin the next iteration. A simple way to code such algorithms is to employ worker processes and one coordinator process. The workers and coordinator communicate to ensure the necessary barrier synchronization.

If the workers and coordinator are in the same resource, a barrier can be written as shown in Figure 3. Each worker first performs some action—typically the action involves accessing part of an array determined by the process's subscript `i`. Then, each worker signals the

```
resource barrier()
  const N := 20  # number of processes
  sem continue[N] := ([N] 0)
  sem start := 0
  # typically, declare data array too.
  process worker(i := 1 to N)
    do true ->
      # code to implement task i
      # i.e., perform one iteration.
        ...
      V(start)
      P(continue[i])
    od
  end
  process coordinator
    do true ->
      fa w := 1 to N -> P(start)  af
      fa w := 1 to N -> V(continue[w])
      af
    od
  end
end
```

Figure 3. Barrier synchronization within one resource

coordinator that it has finished its iteration and waits for the coordinator to inform it that all its siblings have also completed their iterations. The coordinator consists of two for-all loops. The first loop waits for completion signals from each worker. The second loop signals each worker that it can continue.

If the workers and coordinator are in different resources, they can be coded as shown in Figure 4. As shown, the shared variables and semaphores are declared in a global. Three kinds of resources are employed: one for workers, one for the coordinator, and one for the main resource. Each imports the global. The main resource simply creates N instances of the worker resource and an instance of the coordinator resource. Processes in those instances interact via semaphores declared in the global; they do so directly by name. Likewise, they can access shared variables (e.g., the data array) declared in the global.

As the example in Figure 4 illustrates, a global allows objects to be shared by all processes executing in the same address space. The same technique can be used to code a work queue shared among many processes in different resource instances. Specifically, the work queue can be declared as an operation in a global's spec. A process adds an item to the queue by sending to that operation; a process removes an item from the queue by receiving from that operation. This kind of work queue is useful in a number of applications, e.g., an adaptive

```
global barrier
  const N := 20  # number of processes
  sem continue[N] := ([N] 0)
  sem start := 0
  # typically, declare data array too.
end
resource worker(i: int)
  import barrier
  process w
    do true ->
      # code to implement task i
      # i.e., perform one iteration.
        ...
      V(start)
      P(continue[i])
    od
  end
end
resource coordinator()
  import barrier
  process c
    do true ->
      fa w := 1 to N -> P(start)  af
      fa w := 1 to N -> V(continue[w])
      af
    od
  end
end
resource main()
  import barrier, worker, coordinator
  fa i := 1 to N -> create worker(i)
  af
  create coordinator()
end
```

Figure 4. Barrier synchronization with a global

quadrature algorithm. (See [6] for details.)

3.3. Support for Distribution

As suggested in Section 2, virtual machines are the unit for program distribution. Virtual machines can be created (or destroyed) dynamically as needed in response to program execution. Instances of resources and globals can then be created on virtual machines. Processes in different virtual machines communicate with other processes by invoking operations.

To illustrate, consider the problem of *conversational continuity*. A client process interacts with a server process and wishes to carry out a private conversation with it, i.e., send further requests for work to it. As a specific instance of this kind of problem, consider another line printer server, different from the one in Figure 2.

Here, a client connects to the server and then sends it lines to be output. The code outline in Figure 5 illustrates this technique. The main resource creates an instance of the server resource and N instances of client resources. It passes to each client instance a capability for the server so that the client can invoke the server's `print` operation.

This kind of interaction can be accomplished in general by having the server process reply to client invocations, passing back capabilities for its local operations. Here, a client process invokes the server's `print` operation, passing it n, the number of lines that the client will later send it. In response to that invocation, the server assigns a capability for its local operation, `put`, and returns that to its client by executing a reply. The reply allows both the client and server to continue execution. The client sends n messages to its server; the server receives n messages from its client and outputs the information contained in each message. The use of a local operation here ensures that only the client that initiated printing can send messages to the server.

```
resource main()
  import client, server
  var s: cap server
  s := create server()
  # create N clients
  fa i := 1 to N ->
    create client(s)
  af
end
resource client
  import server
body client(s: cap server)
  process (i := 1 to ...)
    var x[20]: string[10]
     ...
    var c: cap(x: string[10])
    c := s.print(20)
    fa i := 1 to 20 -> send c(x[i])  af
     ...
  end
end
resource server
  op print(n: int)
          returns c: cap(x:string[10])
body server()
  process printer
    do true ->
      in print(n) returns c ->
        op put(x:string[10])
        c := put
        # pass back capability for put
        reply
        fa i := 1 to n ->
          var x: string[10]
          receive put(x)
          # output x on the printer
           ...
        af
      ni
    od
  end
end
```

Figure 5. Conversational continuity

A communication structure similar to the one described above is found in several other concurrent algorithms, e.g., sorting using a pipeline of processes and finding primes using a sieve of processes.

The program shown in Figure 5 executes on only a single virtual machine, and therefore also on a single physical machine. However, it can be easily modified so that, for example, each client executes on a different virtual machine. Only `main`'s loop needs to be changed; the new loop is:

```
# create N clients
fa i := 1 to N ->
  var vmcap: cap vm
  vmcap := create vm()
  create client(s) on vmcap
af
```

Each iteration of the above loop creates a new virtual machine (by creating a new instance of `vm`) and then creates a client on that virtual machine. The above loop can be further modified so that each virtual machine is on a different physical machine. For example, the assignment statement that creates virtual machines can be changed to the following:

```
vmcap := create vm() on i
```

The value of `i` is taken to be a physical machine number; its use is installation dependent but can be made to be relatively portable.

Two kinds of transparency with respect to operation invocations are illustrated by the above example. First, an operation is invoked in the same way regardless of how a program is distributed. The invocations by the client of the operations in the server remain the same regardless of whether the client and server are located on the same virtual or physical machine. Second, an operation is invoked in the same way regardless of how the operation is serviced. For example, the body of the server resource could be changed so that invocations of `print` were serviced by a proc, thus creating a new process to service each invocation. Only the body would change; the spec and the way `print` is invoked would remain the same.

4. Concluding Remarks

In many ways, the mechanisms that SR provides for parallelism, distribution, and synchronization are a superset of those found in other languages, such as Ada [8], Concurrent C [9], Argus [10], and occam [11]. SR achieves this flexibility by having just a few well-integrated mechanisms, which can be used alone or freely in combination with others.

One interesting question is whether such generalization is inherently more costly. For example, since SR operations subsume rendezvous, local procedure call, remote procedure call, process creation, semaphores, etc., are they therefore expensive to use? Our implementation currently recognizes some commonly occurring patterns and generates lower-cost code than would be required in the worst, most general case. The version 1 SR compiler, for example, optimizes certain message passing scenarios to use low-cost semaphores, and certain remote procedure call scenarios to use conventional procedure call. The results are that the cost of synchronization in SR is competitive with those reported for other languages [12].

One current effort involves identifying further optimization of synchronization mechanisms, including those that cross resource boundaries. Our overall approach applies source-level transformations to concurrent programs, replacing costly synchronization mechanisms with less costly ones [13]. The techniques involve the application of dataflow analysis and an extension of interprocedural analysis and inter-module analysis to concurrent programs. An interesting aspect of this work is the use of attribute grammars to perform such analysis [14]. These techniques are also applicable to other concurrent programming languages, e.g., Ada, Concurrent C, Argus, and occam.

Our current implementation of SR (version 1.1) works on a variety of UNIX-based systems and is in the public domain. We expect the implementation of version 2 to be completed by Fall 1991. For information on how to obtain SR, contact the author or the SR project (by electronic mail to sr-project@cs.arizona.edu).

Acknowledgements

Greg Andrews created SR_0 and has been the driving force behind SR's evolution. SR version 2 has come about through the ideas and efforts of many people, most notably Greg Andrews, Dave Bakken, Mike Coffin, Gregg Townsend, and the author. Carole McNamee and Caren Asimow provided very useful comments on earlier drafts of this paper.

References

[1] G.R. Andrews, Synchronizing resources. *ACM Trans. on Prog. Languages and Systems 3*, 4 (Oct. 1981), 405-430.

[2] G.R. Andrews, The distributed programming language SR—mechanisms, design and implementation. *Software—Practice and Experience*, 12 (8) (Aug. 1982), 719-754.

[3] G.R. Andrews and R.A. Olsson, The evolution of the SR language. *Distributed Computing* 1, 3 (July 1986), 133-149.

[4] R.A. Olsson, *Issues in Distributed Programming Languages: The Evolution of SR*. TR 86-21 (Ph.D. Dissertation), Dept. of Computer Science, The University of Arizona, August 1986.

[5] G.R. Andrews, R.A. Olsson, M. Coffin, I.J.P. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, An overview of the SR language and implementation. *ACM Trans. on Prog. Lang. and Systems*, 10 (1) (Jan. 1988), 51-86.

[6] G.R. Andrews and R.A. Olsson, *Concurrent Programming in SR*. Benjamin/Cummings Publishing Company, to appear 1991.

[7] N. Wirth, *Programming in Modula-2*. Springer-Verlag, New York, 1982.

[8] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A, 1983.

[9] N. Gehani, *The Concurrent C Programming Language*. Silicon Press, Summit, New Jersey, 1989.

[10] B. Liskov and R. Scheifler, Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. on Prog. Lang. and Systems 5*, 3 (July 1983), 381-404.

[11] A. Burns, *Programming in occam 2*. Addison-Wesley, 1988.

[12] M.S. Atkins and R.A. Olsson, Performance of multitasking and synchronization mechanisms in the programming language SR. *SOFTWARE—Practice and Experience*, 18 (9) (Sept. 1988), 879-895.

[13] C.M. McNamee and R.A. Olsson. Transformations for optimizing interprocess communication and synchronization mechanisms. submitted for publication.

[14] C.M. McNamee and R.A. Olsson. An attribute grammar approach to compiler optimization of intra-module interprocess communication. submitted for publication.