An Overview of the SR Language and Implementation

GREGORY R. ANDREWS, RONALD A. OLSSON, MICHAEL COFFIN, IRVING ELSHOFF, KELVIN NILSEN, TITUS PURDIN, and GREGG TOWNSEND

The University of Arizona

SR is a language for programming distributed systems ranging from operating systems to application programs. On the basis of our experience with the initial version, the language has evolved considerably. In this paper we describe the current version of SR and give an overview of its implementation. The main language constructs are still resources and operations. Resources encapsulate processes and variables that they share; operations provide the primary mechanism for process interaction. One way in which SR has changed is that both resources and processes are now created dynamically. Another change is that inheritance is supported. A third change is that the mechanisms for operation invocation—**call** and **send**—and operation implementation—**proc** and **in**—have been extended and integrated. Consequently, all of local and remote procedure call, rendezvous, dynamic process creation, asynchronous message passing, multicast, and semaphores are supported. We have found this flexibility to be very useful for distributed programming. Moreover, by basing SR on a small number of well-integrated concepts, the language has proved easy to learn and use, and it has a reasonably efficient implementation.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; *network operating systems*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; *modules*, *packages*; D.3.4 [**Programming Languages**]: Processors—*code generation*; *compilers*; *optimization*; *run-time environments*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *multiprocessing/multiprogramming*; *synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*message sending*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*

General Terms: Languages

Additional Key Words and Phrases: Distributed programming languages

1. INTRODUCTION

During the past two years we have redesigned and reimplemented the SR (Synchronizing Resources) programming language. Like its predecessor, SR_0 [2, 3], SR remains a language for writing distributed programs. Also, the main

This research was supported by NSF under grant DCR-8402090 and by the Air Force Office of Scientific Research under grant AFOSR-84-0072. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notices thereon.
Authors' addresses: Gregory R. Andrews, Michael Coffin, Irving Elshoff, Kelvin Nilsen, and Gregg Townsend, Department of Computer Science, The University of Arizona, Tucson, AZ 85721; Ronald A. Olsson, Division of Computer Science, The University of California at Davis, Davis, CA 95616; Titus Purdin, Department of Computer Science, Colorado State University, Fort Collins, CO 80523.

language constructs—resources and operations—are conceptually the same. However, on the basis of our experience using SR_0 to write numerous programs, including prototypes of the Saguaro operating system [6], we have modified the language in several ways. In essence, SR is to SR_0 what Modula-2 is to Modula [38, 39]: a second-generation language that incorporates refinements based on experience with its predecessor.

The redesign of SR has been guided by three major concerns: expressiveness, ease of use, and efficiency. By expressiveness we mean that it should be possible to solve distributed programming problems in the most straightforward possible way. This argues for having a flexible set of language mechanisms, for both writing individual modules and combining modules to form a program. Distributed programs are generally much more complex than sequential programs. Sequential programs usually have a hierarchical structure; distributed programs often have a weblike structure in which components interact more as equals than as master and slave. Sequential programs usually contain a fixed number of components since they execute on a fixed hardware configuration; distributed programs often need to grow and shrink dynamically in response to changing levels of user activity and changing hardware configurations. Sequential programs have a single thread of control; distributed programs have multiple threads of control. Thus a distributed programming language necessarily contains more mechanisms than a sequential programming language.

One way to make a language expressive is to provide a plethora of distinct mechanisms. However, this conflicts with our second concern, ease of use. As Hoare has so aptly observed, if programs are to be reliable, the language they are written in must be simple to understand and use [21]. The way we have resolved this tension between expressiveness and simplicity is that SR provides a variety of mechanisms, but they are based on only a few underlying concepts. Moreover, these concepts are generalizations of those that have been found useful in sequential programming, and they are integrated with the sequential components of SR so that similar things are expressed in similar ways. The main components of SR programs are parameterized resources, which generalize modules such as those in Modula-2. Resources interact by means of operations, which generalize procedures. Operations are invoked by means of synchronous **call** or asynchronous **send**. Operations are implemented by procedure-like **procs** or by **in** statements. In different combinations, these mechanisms support local and remote procedure call, dynamic process creation, rendezvous, message passing, and semaphores—*all* of which we have found to be useful. The concurrent and sequential components of SR are integrated in numerous additional ways in an effort to make the language easy to learn and understand and hence easy to use.

A further consequence of basing SR on a small number of underlying concepts is good performance. SR provides a greater variety of communication and synchronization mechanisms than any other language, yet each is as efficient as its counterpart in other languages. We have also designed the language and implemented the compiler and run-time support in concert, revising the language when a construct was found to have an implementation cost that outweighed its utility. In addition, some of the expressiveness within the language has been realized by "opening up" the implementation. For example, the various

mechanisms for invoking and servicing operations are all variations on ways to enqueue and dequeue messages.

A detailed discussion of how these concerns have influenced the evolution of SR is given in [4]. This paper presents an overview of the language and its implementation. We summarize the main language mechanisms,[1] give examples of their use, describe the most interesting aspects of the implementation, and give performance figures. We also discuss the relation between SR and other approaches to programming distributed systems.

2. LANGUAGE OVERVIEW

An SR program is composed from three kinds of separately compiled *components*: resource specifications, resource bodies, and globals. Resources are the main building block; they are the unit of abstraction and encapsulation. Globals contain declarations of constants and types shared by resources; they have the form

```
global identifier
   declarations of constants and types
end identifier
```

A resource is a parameterized pattern, instances of which are created dynamically. Resources define operations and are implemented by one or more processes that execute in the same address space. Processes interact by means of operations; processes in the same resource may also share variables. SR provides a variety of mechanisms for implementing and invoking operations. These mechanisms can be used in various combinations to program resources that implement objects ranging from sequential stacks and queues, through monitors, to complex servers. Moreover, inheritance is supported so resources can implement classes of objects. The remainder of this section is a summary of the most interesting aspects of resources and operations.

2.1 Resources

A resource has a *specification* and a *body*. The specification identifies other components the resource uses and declares operations, constants, and types exported by the resource. The basic form of a resource specification is

```
resource identifier
   import identifier list
   declarations of operations, constants, and types
end identifier
```

All objects declared in the specification are exported from the resource; they may be used in other resources that import this resource.

The body of a resource contains the processes that implement the resource, declarations of objects shared by those processes, and initialization and

[1] The complete language is described in [5].

finalization code. The general form of a body is

```
body identifier(formal parameters)
    import identifier list
    declarations of shared objects
    processes
    initial block end
    final block end
end identifier
```

The identifier on a body is the same as that on a previously compiled specification; a body inherits all objects declared in or imported into that specification. As indicated, the body of a resource is parameterized; actual parameters are assigned during resource creation (see Section 2.1.2). The shared objects may be constants, types, variables, and additional, local operations; none of these objects is visible outside the body. A *block* is a sequence of declarations and statements.

Most of the pieces in the specification and body of a resource are optional and may occur in any order. Also, an object can be referenced any time after it has been declared. This permits the values of constants to depend on previously declared objects. Statements and declarations can also be intermixed. This permits sizes of arrays to depend on input values.

The general ordering rule in SR is "declare before use"; that is, an object must be declared before being referenced. Thus a global component or resource specification must be declared before any resource that imports it. Note that the body of a resource may also contain an imports phrase. This permits a body to employ components in addition to those imported by the resource's specification and thus facilitates keeping interfaces precise. This also permits two resource bodies to reference each other's specification.

The specification and body of a resource may be combined when it is not necessary to compile them separately. This is illustrated by the following example, which declares a resource that implements a queue of integers:[2]

```
resource Queue
    op insert(item : int)
    op remove() returns item : int
body Queue(size : int)
    var store[0:size-1] : int
    var front := 0, rear := 0, count := 0
    proc insert(item)
        if count<size → store[rear] := item; rear := (rear+1)%size; count++
        [] else →     # take actions appropriate for overflow
        fi
    end
    proc remove() returns item
        if count>0 → item := store[front]; front := (front+1)%size; count--
        [] else →     # take actions appropriate for underflow
        fi
    end
end Queue
```

[2] Semicolons are optional in SR; our convention is to use them as separators for declarations or statements that appear on the same line. Identifiers following **end** are also optional; our convention is to include them only on lengthy objects. One-line comments begin with '#' and terminate with the end of the line on which the comment begins.

The specification of *Queue* exports two operations. The body employs the common array-based representation; it is parameterized by the size of the array that is to be used when an instance of *Queue* is created. Note that *Queue* is a "sequential" data type since the operations on an instance should not be executed concurrently; a synchronized queue that may be shared by processes is given later. Also note that the structure of this resource is very similar to the structure one would find in languages such as Modula-2, Euclid [26], and Ada [1]. The actual forms of declarations and statements are somewhat different, however, for reasons mostly having to do with our goal of integrating the sequential and concurrent mechanisms of SR.

2.1.1 *Inheritance and Resource Families.* A resource can import globals and other resources and thus can reference objects exported by those components. An additional mechanism, the **extend** phrase, is provided to allow one resource to inherit all objects declared in the specification of one or more other resources. This facilitates construction of families of closely related resources, as illustrated below and in a larger example in Section 3. The extension mechanism also facilitates subdividing the specification of a resource into pieces that are imported by different resources; this is also illustrated in Section 3.

One way to program a resource that implements a queue of integers was shown above. However, there are numerous ways to represent a queue, for example, using a linked list rather than an array. A family of queue resources could be programmed as follows. First, a resource specification is given for the representation-independent operations on a queue:

```
resource Queue
   op insert(item : int)
   op remove() returns item : int
end
```

Then, other resources that extend *Queue* are programmed for each different representation. For example, the array representation could be reprogrammed as

```
resource ArrayQueue
   extend Queue
body ArrayQueue(size : int)
   body as programmed above
end ArrayQueue
```

A resource that extends another can also declare additional operations if that is appropriate.

Above, *Queue* is an *abstract resource* that has no body. It is used merely to specify an interface. By contrast, *ArrayQueue* is a *concrete resource* having both a specification and body.

2.1.2 *Resource Creation.* Instances of concrete resources are created dynamically by means of the **create** statement. Execution of

variable := **create** *resource_identifier*(arguments)

causes a new instance of the named resource to be created. Arguments are passed by value to the new instance and then the resource's initialization code, if any, is executed (as a process). Execution of **create** terminates when the initialization code terminates. A *resource capability* is returned by **create** and assigned to

variable. This capability can be used to invoke the operations exported by the resource, can be copied and passed to other resources, and can be used to destroy the instance. For example, given

```
var qcap : cap ArrayQueue
```

which declares a capability for *ArrayQueue*, execution of

```
qcap := create ArrayQueue(20)
```

creates a 20-element *ArrayQueue* and returns a capability for it. Subsequently, an *item* can be inserted into the queue by executing

```
qcap.insert(item)
```

or removed by executing

```
item := qcap.remove()
```

where *item* is of type **int**. Essentially *qcap* is a record, the fields of which are capabilities for the two operations exported by *ArrayQueue*. If it becomes appropriate to destroy this instance of *ArrayQueue*,

```
destroy qcap
```

can be executed. The **destroy** statement terminates after the resource's finalization code (if any) terminates and space allocated to the resource has been freed.

By default, execution of **create** places a new resource in the same virtual machine (address space) as that of the resource on which **create** is executed. It is also possible to cause an instance to be placed in a different virtual machine by appending "**on** *vmcap*" to the **create** statement, where *vmcap* is a capability for a virtual machine. New virtual machines are generated by creating instances of the *vm* pseudoresource in a manner analogous to creating a resource. A virtual machine can be placed on a specific physical machine (network node) by appending "**on** machine_id" to the **create** statement.

In addition to capabilities for concrete resources, which are generated by create statements, SR provides capabilities for abstract resources and for individual operations. For example, a capability for the abstract resource *Queue* could be declared and then assigned to from a capability that points to an instance of a concrete resource that extends *Queue*. Thus capabilities for abstract resources can be used to mask completely the concrete resource that implements an abstract interface.

Capabilities for individual operations provide a mechanism similar to procedural parameters. They are used to support finer grained control over the communication paths between resources. (Examples are given later.) Finally, capability variables and individual fields within capabilities for resources can be set to the special values **null** (error) or **noop** (no effect); **null** is the initial value of each capability variable.

2.2 Operations and Communication Primitives

Resources are patterns for objects; operations are patterns for actions on objects. Operations are declared in **op** declarations, as illustrated in the previous examples. (Arrays of operations are also supported.) Such declarations can appear in resource specifications, within resources, or within processes. Operations are

Table I

Invocation	*Service*	*Effect*
call	**proc**	procedure call (possibly remote or recursive)
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

invoked by **call** or **send** statements; they are serviced by procedures (**proc**) or input (**in**) statements. In the four possible combinations, these primitives have the effects given in Table I. In addition, semaphores can be simulated using **send** and **in** with parameterless operations. This section shows how these effects are achieved.

2.2.1 *Invocation Statements.* The invocation statements have the forms

```
call operation(arguments)
send operation(arguments)
```

where operation is a field of a resource capability, an operation capability, or the name of an operation declared in the scope of the invocation statement. The keyword **call** is optional; it is omitted when operations are invoked in expressions. Arguments are passed by value (**val**, the default), result (**res**), or value/result (**var**). Arguments can also be passed by reference (**ref**) within a virtual machine (address space).

A **call** statement terminates when the operation has been serviced and results have been returned (or when a failure has been detected, as discussed in Section 2.2.4). A **send** statement terminates when the arguments have been stored on the machine on which the resource that services the operation resides. Thus, **call** is *synchronous*, whereas **send** is *semisynchronous* [9].[3]

By default, an operation may be invoked by either **call** or **send**. It is possible to restrict invocation to just one of these by appending the *operation restriction* "{**call**}" or "{**send**}" to the operation's declaration. In keeping with our desire not to impose restrictions on usage, however, we do not a priori preclude invoking an operation by **send**, even if it has result parameters or a return value. Occasionally we have found it useful to **send** to a function, for example, to update a graphics display.

2.2.2 *Servicing Operations.* An operation is serviced either by a **proc** or by one or more **in** statements. A **proc** is a generalization of a procedure: It is declared like a procedure and may be called like a procedure, but has the semantics of a process. Its general form is

```
proc operation_identifier(formal_identifiers) returns result_identifier
    block
end operation_identifier
```

[3] We have chosen semisynchronous rather than asynchronous semantics for **send** for two reasons. First, this is the semantics that is invariably implemented when the sender and receiver of an operation execute on the same machine. Second, semisynchronous semantics provides the sender with assurance that adequate buffer space for the invocation exists and that the servicing resource exists and was reachable at the time the operation was invoked. Of course, the programmer still has no assurance that the invocation *will* be serviced.

where the operation identifier is the same as that in an **op** declaration in the resource containing the **proc**. Whenever that operation is invoked, an instance of the **proc** is created. This instance executes as a process and uses the formal and result identifiers (if any) to access the arguments of the invocation and to construct the result.[4] If the operation was called, the caller waits for the instance to terminate (or reply); the effect is thus like a procedure call and is in fact implemented like a procedure call for calls within the same virtual machine. However, if the operation was invoked by **send**, the sender and instance of the **proc** execute concurrently; the effect in this case is like forking a process.

The other way to service operations is to employ **in** statements, which have the general form

```
in operation_command [] ... [] operation_command ni
```

Each operation command is structurally like a **proc** except that it may also contain a synchronization and scheduling expression:

```
operation_identifier(formal_identifiers) returns result_identifier
   and synchronization_expression by scheduling_expression → block
```

An **in** statement delays the executing process until some invocation is selectable; then the corresponding block is executed. An invocation is *selectable* if the Boolean-valued synchronization expression in the corresponding operation command is true; the synchronization expression is optional and is implicitly true if omitted. In general, the oldest selectable invocation is serviced. This can be overridden by the use of **by**, which causes selectable invocations of the associated operation to be serviced in ascending order of the arithmetic scheduling expression following **by**.

Recall that **proc** supports procedure call and process forking, depending on whether the operation serviced by a **proc** is invoked by **call** or **send**. Input statements support rendezvous or message receipt, again depending on whether an operation serviced by **in** is invoked by **call** or **send**. Thus input statements combine aspects of both Ada's **select** statement [1] and CSP's guarded input statement [25]. They are even more powerful, however, since the synchronization expression may reference formal parameters, and thus selection can be based on parameter values. Input statements may also contain scheduling expressions that reference formal parameters. These mechanisms greatly simplify solving many synchronization problems. Yet, as we shall see in Section 4, they can be implemented quite efficiently.

A simple example will help clarify these mechanisms and their use. Following is a resource that implements a bounded buffer of integers:

```
resource BoundedBuffer
   op insert(item : int)
   op remove() returns item : int
```

[4] The types of the formals and result are specified in the **op** declaration; they are not repeated in the **proc** declaration.

```
body BoundedBuffer(size : int)
   op bb()
   initial send bb() end    # see text for a simpler way to program this
   proc bb()
      var store[0:size-1] : int
      var front := 0, rear := 0, count := 0
      do true →
         in insert(item) and count<size →
               store[rear] := item; rear := (rear+1)%size; count++
         [] remove() returns item and count>0 →
               item := store[front]; front := (front+1)%size; count--
         ni
      od
   end
end BoundedBuffer
```

The interface part of *BoundedBuffer* is identical to that of *Queue*, which is appropriate since a bounded buffer is just a synchronized queue; in fact, *BoundedBuffer* could have been programmed by extending *Queue* rather than by repeating the specification. The implementation of *BoundedBuffer* contains one **proc**, *bb*. Initially one instance of *bb* is activated (using **send**); that instance executes as a process that repeatedly services invocations of *insert* and *remove*. Invocations of *insert* can be selected as long as there is room in the buffer; invocations of *remove* can be selected as long as the buffer is not empty. Note that *insert* and *remove* are serviced by a single process and thus execute with mutual exclusion. Also note that the implementation is not visible to resources that use instances of *BoundedBuffer*, the resource body could equally well have used a monitor-like implementation in which *insert* and *remove* are each serviced by a **proc** and semaphores are used to synchronize them (see below for how semaphores can be simulated).

Resources often contain "worker" processes such as *bb* above. SR provides a **process** declaration to simplify programming such processes. For example, the above resource could be coded more compactly by deleting the declaration of the *bb* operation, deleting the initialization code, and replacing the line.

proc *bb*()

by

process *bb*

Process declarations are thus an abbreviation for the specific pattern that was employed in *BoundedBuffer*.

Another useful abbreviation is provided by the **receive** statement. In particular,

receive *operation*(*v1*, ..., *vN*)

is an abbreviation for an **in** statement that waits for an invocation of *operation* and then assigns the values of the formal parameters to variables $v1, \ldots, vN$. Together with the **send** form of invocation, **receive** supports asynchronous message passing in a familiar way. Synchronous message passing is supported by

receive together with the **call** form of invocation. Semaphores can also be simulated using **send** and **receive**. For example, given the declaration

```
op s()
```

the following statements simulate the semaphore **P** and **V** operations:

```
receive s()     # P operation
send s()        # V operation
```

This simulation can be used within a resource to synchronize access to shared resource variables. In fact, operations declared and used in this way are implemented just as if they were semaphores.

2.2.3 *Additional Communication Primitives.* A few additional primitives are provided to support process interaction. All are useful, simple, and efficient.

The **return** and **reply** statements provide flexibility in servicing invocations. Execution of **return** causes the smallest enclosing **in** statement or **proc** to terminate early. If the invocation being serviced was called, the corresponding **call** statement also terminates, and results are returned to the caller. Execution of **reply** causes the **call** invocation being serviced in the smallest enclosing **in** statement or **proc** to terminate.[5] In contrast to **return**, however, the process executing **reply** continues with the next statement. An important use of **reply** is to allow a **proc** to transmit return values to its caller yet continue to exist and execute after replying. This facilitates programming *conversations*, as will be shown in Section 3.2.

The final communication primitive is the **co** statement, which supports concurrent invocations. The form of a **co** statement is

```
co concurrent_invocation → post_processing
// ...
// concurrent_invocation → post_processing
oc
```

A concurrent invocation is a **call** or **send** statement, or an assignment that contains only a single invocation of a user-defined function. The postprocessing blocks are optional. Execution of **co** first starts all invocations. Then, as each invocation terminates, the corresponding postprocessing block is executed (if there is one); postprocessing blocks are executed one at a time. Execution of **co** terminates when all invocations and postprocessing blocks have terminated or when some postprocessing block executes **exit**.

If a **co** statement terminates before all its invocations have terminated, uncompleted invocations are not terminated prematurely because such an invocation could be being serviced, in which case terminating it could put the server process in an unpredictable state. Also, it is sometimes useful to have uncompleted invocations get serviced even after **co** terminates; for example, all reachable copies of a replicated database should be updated even if the updater terminates after only a majority of the copies have been updated.

A concurrent invocation in **co** can also be preceded by a *quantifier*, which implicitly declares a *bound variable* and specifies a range of values for that

[5] Execution of **reply** has no effect for operations that are invoked by **send**.

variable.[6] Within **co**, a quantifier provides a compact notation for specifying multiple invocation/postprocessing pairs. For example, a replicated file might be updated by executing

co (*i* := 1 to N) **call** *file*[*i*].*update*(values) **oc**

where *file* is an array of capabilities containing one entry for each file resource. Similarly, reading a replicated file, terminating when one copy has been read, might be programmed as

co (*i* := 1 to N) **call** *file*[*i*].*read*(arguments) → **exit** **oc**

In both cases, the quantifier's bound variable *i* is accessible in both the invocation statement and postprocessing block. Thus the last example could be modified to record which of the *file*[*i*] was the first to respond by saving the value of *i* in a variable global to **co** before executing **exit**.

2.2.4 *Failure Handling.* To program distributed systems such as Saguaro, a language must contain mechanisms for dealing with hardware failures such as processor and network crashes. Also, exceptions can occur, such as memory overflow, invoking an operation in a destroyed resource, or passing a pointer outside a virtual machine. SR provides two mechanisms for failure handling.[7] First, *handlers* can be appended to invocation and resource control statements, and to **proc** declarations. An invocation handler, as in

call *qcap.insert*(*item*) [block handling overflow exception]

is executed if the invoked operation fails in a way that is detected by the run-time system, or if the invoked operation raises an exception by executing

abort(cause)

The different kinds of causes are declared in a predefined enumeration type; user-defined exceptions are not supported, at least at present. Within a handler, a predefined function can be called to determine the cause of failure.

If an invocation (or resource control) handler is executed, execution of the enclosing **proc** continues as dictated by the code in the handler. If an invocation fails and is not handled, or if a **proc** encounters a local exception, the handler attached to the **proc** is executed. The **proc** terminates when the handler terminates. If a failure is not handled by a **proc**, the **proc** aborts and the failure and cause are passed back up the (dynamic) call chain if the **proc** was called. Note that at any point in a **proc**, there is exactly one handler that could be entered.

Handlers enable a **proc** to avoid waiting forever if an invocation fails. Another form of permanent delay can also occur in a distributed program: A server could block at an input statement waiting for an invocation that will never arrive because a client **proc**, resource, or processor has failed. SR provides a second mechanism, the **when** statement, to enable a server to monitor such failures.

[6] Quantifiers can also be used within **in** statements to facilitate servicing elements of an array of operations, and they are the basis for one of the iterative statements discussed in the next section.

[7] These mechanisms have not yet been implemented, and hence we have no experience with their use. They appear to be useful and easily implemented, but only time will tell.

Execution of

b := **when** *failed*(*source*) **send** *operation*(arguments)

requests that the run-time support monitor *source*, which may be a physical machine, virtual machine, resource instance, or process, depending on the value of the argument.[8] If *source* fails, an invocation of *operation* is sent with the indicated arguments (which are evaluated when the **when** statement is executed, not when the failure occurs). Thus an asynchronous failure is turned into an asynchronous invocation that can be serviced by a **proc** or **in** statement just like any other invocation.

The **when** statement returns a value of type **binding**, which serves to identify the binding between the *source* being monitored and the *operation* to be invoked. Monitoring can be changed by assigning a new value to a binding variable; it can be canceled by setting a binding variable to **null**. Monitoring is implicitly canceled when the lifetime of a binding variable ends.

2.3 Types, Declarations, and Sequential Statements

SR provides a variety of data types and sequential statements similar to those found in other languages. However, their form and many of their details are different. Largely this is to facilitate the integration of the sequential and concurrent components of SR. It also results from our desire to make it easy to program commonly occurring algorithmic patterns.

In addition to the usual kinds of basic and structured types, SR provides capabilities and what is called an **optype** (operation type). An **optype** defines a pattern for an operation; that is, the types of the parameters and return value. Such a type is used when the same operation pattern is used more than once, for example, when several resources implement the same type of operation such as file read. What makes capabilities and operation types especially useful is that type checking is based on structural equivalence; hence, an operation capability can be bound to any operation that has the specified pattern.

SR's **if** and **do** statements are based on Dijkstra's guarded commands [21] so their structure is similar to that of the **in** statement. However, if no Boolean expression is true, **if** has no effect (unlike Dijkstra's **if**, which aborts in this situation). We also allow the last guard of **if**, **do**, and **in** to be "**else**", which is interpreted as the conjunction of the negations of the other boolean expressions.

SR also provides a novel iterative statement, **fa** (for all), that employs quantifiers introduced earlier with the **co** statement. The form of **fa** is

fa quantifier , ..., quantifier → block **af**

where quantifiers have the general form

bound_variable := initial_value direction final_value **st** boolean_expression

For example, the single statement

```
fa i := lb(a) to ub(a)-1,
    j := i+1 to ub(a) st a[i]>a[j] →
        a[i], a[j] :=: a[j], a[i]    # swap statement
af
```

[8] A predefined function is provided so that a server can determine the identity of an otherwise anonymous client.

sorts array a into ascending order. This statement employs the builtin array lower-bound ('lb') and upper-bound ('ub') functions. Note how the range of values for the second bound variable j depends on i. Also note the use of **st** (such that), which is used to limit execution of the body of **fa** to those values of i and j for which $a[i] > a[j]$.

The two final sequential statements are **exit** and **next**. The **exit** statement is used to force early termination of the smallest enclosing iterative or **co** statement. The **next** statement is used to force return to the loop control of the smallest enclosing iterative statement; it can also be used within a postprocessing block in a **co** statement to force **co** to wait for another invocation to terminate. Note how these statements, like quantifiers, have consistent uses in support of both sequential and concurrent programming.

2.4 Implementation-Specific Mechanisms

SR can be used to write programs that execute on top of an existing operating system or stand alone on a "bare" network of processors. Our current implementation is built on top of UNIX; SR programs run as UNIX processes on one or more interconnected machines. Work is under way on a stand-alone implementation. Any implementation must provide mechanisms for interacting with input/output (I/O) devices. This section is a brief summary of the I/O mechanisms provided in our current implementation and those that will be provided in the stand-alone implementation.

In the UNIX implementation, I/O is supported by the **file** data type and several operations on that type. The operations permit files to be created, opened, closed, removed, read, and written. Three forms of read and write are supported: formatted, character string, and line at a time. Predefined file literals provide access to the standard input, output, and error files commonly employed by UNIX programs. Since devices such as terminals are integrated into the UNIX file system, it is easy for an SR program to interact with multiple terminals: The various terminal "files" are simply opened and accessed like normal files. Routines are also provided to access the arguments on the command line that triggers execution of an SR program. It is also possible for an SR program to access any C subroutine, including system calls such as those implementing window packages. Work is under way to implement an "execute" facility that allows a sequential program, such as a C command, to be executed as an SR process within an envelope resource.

In a stand-alone implementation, there is no underlying operating system to, depending on one's perspective, provide convenient functions or get in one's way. Thus an SR programmer needs to be able to program device controllers and memory allocators. Mechanisms are provided to access bytes, bind variables to memory addresses, bind operations to interrupt vector locations, and service interrupts.

3. EXAMPLES

In this section we present two larger examples.[9] The first, a decentralized solution to the classic dining philosophers problem, illustrates a complete program

[9] Additional examples are given in [30].

containing multiple resources. The solution employs many of the SR communication primitives and illustrates the use of the I/O primitives. The second example outlines parts of the Saguaro file system [6]. It illustrates the use of several additional mechanisms.

3.1 Decentralized Dining Philosophers

In the dining philosophers problem, N philosophers (typically five) sit around a circular table set with N forks, one between each pair of philosophers. Each philosopher alternately thinks and eats. Before eating, a philosopher must acquire the two closest forks.

This problem can be solved in three basic ways. In all cases, philosophers are represented by processes; the approaches differ in how forks are managed. The first, centralized approach is to have a single servant process that manages all N forks. The second, decentralized approach is to distribute the forks among N servant processes, with each servant managing one fork. The third approach is similar to the second, but employs one servant per philosopher instead of one servant per fork. In this case, each philosopher interacts with its own personal servant; that servant interacts with its two neighboring servants. Each fork is either held by one of the two servants that might need it or is in transit between them. A hungry philosopher may eat when its servant holds two forks (and presumably "spoon feeds" the philosopher).

Each approach can readily be programmed in SR (see [30] for details). Following is a solution that employs the third approach since that one is the most intricate and also illustrates the largest number of SR mechanisms. The specific algorithm that the servants employ is adapted from [14]. It has the desirable properties of being fair and deadlock free. The basic solution strategy also has application to other, realistic problems such as file replication and database consistency.

Our solution employs three concrete resources—*Servant*, *Philosopher*, and *Main*—and two abstract resources—*PhilosopherOps* and *ServantOps*. The abstract resources define the servant operations invoked by philosophers and other servants, respectively:

```
resource PhilosopherOps
  op getforks() {call}
  op relforks()
end

resource ServantOps
  op needL() {send}
  op needR() {send}
  op passL() {send}
  op passR() {send}
  op links(l, r : cap ServantOps)      # links to neighbors
  op forks(haveL, dirtyL, haveR, dirtyR : bool)    # initial fork values
end
```

The interface to a *Servant* is the union of these two interfaces:

```
resource Servant
  extend PhilosopherOps, ServantOps
end
```

Some of the *Servant* operations are restricted to be invoked only as indicated; it does not matter how the others are invoked.

Instances of *Philosopher* are created by the *Main* resource, as shown later. Each instance is passed a capability *myservant* for the *PhilosopherOps* exported by its personal servant, an identity *id* used for output, and a value *t* indicating the number of times the philosopher is to eat and think before it dies:

```
resource Philosopher
  import PhilosopherOps
body Philosopher(myservant : cap PhilosopherOps; id, t : int)
  process phil
    fa i := 1 to t →
      myservant.getforks()
      write("Philosopher", id, "is eating")    # eat
      myservant.relforks()
      write("Philosopher", id, "is thinking")    # think
    af
  end
end Philosopher
```

Instances of *Servant* service invocations of *getforks* and *relforks* from their associated instance of *Philosopher*. Servants communicate with neighboring instances using the *needL*, *needR*, *passL*, and *passR* operations. A philosopher is permitted to eat when its servant has acquired two forks. A servant may already have both forks when *getforks* is called, or it may need to request one or both from the appropriate neighbor servant. Two variables are used to record the status of each fork: *haveL* (*haveR*) and *dirtyL* (*dirtyR*). Starvation is avoided by having servants give up forks that are dirty; a fork becomes dirty when it is used by a philosopher.

```
body Servant()
  var l, r : cap ServantOps
  var haveL, dirtyL, haveR, dirtyR : bool
  op hungry()    # hungry and eat are local operations
  op eat()
  proc getforks()
    send hungry()    # let server know philosopher is hungry
    receive eat()    # wait for permission to eat
  end
  process server
    receive links(l,r)
    receive forks(haveL,dirtyL,haveR,dirtyR)
    do true →
      in hungry() →
        # ask for forks I do not have; ask right neighbor for its left fork,
        #   and left neighbor for its right fork
          if ~haveR → send r.needL() fi
          if ~haveL → send l.needR() fi
        # wait until I have both forks
          do ~(haveL & haveR) →
            in passR() → haveR := true; dirtyR := false
            [] passL() → haveL := true; dirtyL := false
            [] needR() & dirtyR → haveR := false; dirtyR := false
                                  send r.passL(); send r.needL()
```

```
            [] needL() & dirtyL → haveL := false; dirtyL := false
                                  send l.passR(); send l.needR()
            ni
          od
        # let my Philosopher eat; wait for forks to be released
          send eat(); dirtyL := true; dirtyR := true; receive relforks()
      [] needR() →
        # right neighbor needs left fork, which is my right fork
          haveR := false; dirtyR := false; send r.passL()
      [] needL() →
        # left neighbor needs right fork, which is my left fork
          haveL := false; dirtyL := false; send l.passR()
      ni
    od
  end server
end Servant
```

Notice the various combinations of invocation and service statements that are employed. For example, *getforks* is implemented by a **proc** and hides the fact that getting forks requires sending a *hungry* message and receiving an *eat* message. Other operations, including *relforks*, are implemented by **in** statements. Also, invocations of *needL* and *needR* are serviced by two different **in** statements, reflecting the two states in which a servant might give up a fork to a neighbor. Finally, note that **send** is used to invoke the need and pass operations of a neighbor; **call** cannot be used for this because deadlock could result if two neighboring servers invoked each other's operations at the same time.[10]

The final resource, *Main*, initializes execution of the program. By appropriate directive to the SR linker, one instance of *Main* is implicitly created when execution of the program begins. It prompts for input about the number of philosophers n and the number of times t each philosopher is to execute. *Main* then creates n instances of *Philosopher* and n instances of *Servant* and passes them capabilities so they can communicate with each other. *Main* also sends each *Servant* the initial values for its local variables.

```
resource Main
  import Philosopher, PhilosopherOps, ServantOps, Servant
body Main()
  initial
    var n, t : int
    put("how many Philosophers? (at least 2) "); read(n);
    put("how many sessions per Philosopher? (at least 1) "); read(t);
    var s[1:n] : cap Servant
    var si[1:n] : cap ServantOps
    var pi[1:n] : cap PhilosopherOps
    # create the Servants and Philosophers
      fa i := 1 to n →
        s[i] := create Servant(i)
        si[i] := ServantOps from s[i]
        pi[i] := PhilosopherOps from s[i]
        create Philosopher(pi[i],i,t)   # returned capabilities are not needed
      af
```

[10] The body of *Servant* could be programmed differently to use procedures to implement the need and pass operations. However, these procedures would then have to use semaphores to synchronize access to the shared variables.

```
    # give each Servant its links to neighboring Servants
      send s[1].links(si[n],si[2])
      fa i := 2 to n-1 → send s[i].links(si[i-1],si[i+1]) af
      send s[n].links(si[n-1],si[1])
    # initialize each Servant's forks; must be asymmetric to avoid deadlock
      send s[1].forks(true,false,true,false)
      fa i := 2 to n-1 → send s[i].forks(false,false,true,false) af
      send s[n].forks(false,false,false,false)
  end
end Main
```

Note that the sizes of the arrays depend on input value *n*. Also note that capabilities can be passed to *Philosophers* as resource parameters. However, separate operations are required to pass each servant capabilities for its neighbors since a resource has to be created before *Main* has a capability for it.

3.2 Components of the Saguaro File System

Our final example outlines a few components of the Saguaro file system [6].[11] This example illustrates several additional features of SR including a global component, operation types, operation capabilities, and operations declared local to **procs**. The components are given in the order in which they would be compiled.

Files in Saguaro, like those in UNIX, include ordinary data files, devices, and a generalization of pipes called channels. Different kinds of files have different representations and are serviced by different resources. However, all files are streams of bytes and are accessed using the same operations: *read*, *write*, *seek*, and *close*. The patterns for these operations as well as several file system constants are declared in a **global** component:

```
global FileDefs
  optype read(res buf[0:*] : char; count : int) returns actual_count : int
  optype write(buf[0:*] : char; count : int) returns actual_count : int
  optype seek(kind : int; offset : int)
  optype close()
  const EOF := -1
  ...       # declarations of other global constants
end
```

Files are managed by *DirectoryMgr* resources. The client-visible subset of operations on directories is declared in an abstract resource:

```
resource DirectoryOps
  import FileDefs
  type file_desc = rec( index : int; read : cap FileDefs.read;
                        write : cap FileDefs.write; seek : cap FileDefs.seek;
                        close : cap FileDefs.close)    {private}
  op open(path_name[0:*] : char; ...) returns fd : file_desc
  ...    # other operations to remove files, list directories, etc.
end
```

A client calls the *open* operation to acquire access to a file. If successful, *open* returns a file descriptor, which contains a table index used within the file system and capabilities for the various types of file operations. The type restriction "{**private**}" appended to the declaration of *file_desc* ensures that resources that

[11] The complete system, which consists of over 4000 lines of SR source, is described in [32].

implement the directory operations are the only ones that can assign to fields of the record.

DirectoryMgr extends *DirectoryOps* with additional operations used within the implementation of the file system:

```
resource DirectoryMgr
  extend DirectoryOps
  import FileDefs, DiskServer
  op fclose(...)
  ...     # additional operations on directories
end
```

There is one instance of this resource for each physical file system in Saguaro. Each client has two *DirectoryOps* capabilities; one is bound to the instance of *DirectoryMgr* on which the root file resides, the other to the instance on which the client's current working directory resides.

File operations are implemented by *Terminal*, *Channel*, and *FileServer* resources, depending on the kind of file. Here we consider only the case of *FileServer* resources. When an ordinary file is opened, the appropriate *DirectoryMgr* determines if there is an existing *FileServer* for the file. If so, the *DirectoryMgr* allocates that *FileServer*; if not, the *DirectoryMgr* creates a new *FileServer*. Thus each instance of *FileServer* services all clients who have opened the same file; this allows file-specific information, such as buffers, to be shared. A client accesses a file using the file descriptor returned by *open*. When a client is finished with a file, it invokes the *close* operation in the associated *FileServer*. That *FileServer* then informs its *DirectoryMgr* that the user has finished by invoking the *DirectoryMgr*'s *fclose* operation. If no other clients have the file open, the *DirectoryMgr* then destroys the *FileServer*.

When created, a *FileServer* is passed capabilities for the directory manager that created it and the disk server that services the disk on which the file resides. *FileServer* exports one operation, *fopen*, which is called by a *DirectoryMgr* each time the file is opened.

```
resource FileServer
  import FileDefs, DirectoryOps, DirectoryMgr, DiskServer
  op fopen(...) returns fd : file_desc
body FileServer(fclose : cap DirectoryMgr.fclose; disk : cap DiskServer; ...)
  op lock()      # used as a semaphore for mutual exclusion
  ...     # declarations of other shared objects, such as buffers
  ...     # initialization, including one send lock() to initialize lock semaphore
  proc fopen(...) returns fd
    op read  FileDefs.read    # local operations to access file
    op write  FileDefs.write
    op seek  FileDefs.seek
    op close  FileDefs.close
    var rwptr := 0            # read/write offset in file
    ...       # other declarations and initialization.
    fd := file_desc(0,read,write,seek,close)    # record construction
    reply    # return capabilities in fd and then continue
    do true →    # service file operations until client invokes close.
      in read(...) → ...
      [] write(...) → ...
      [] seek(...) → ...
```

```
      [] close(...) → exit
      ni
    od
    ...          # clean up tables
    fclose(...)  # call fclose in parent DirectoryMgr
  end fopen
end FileServer
```

Since *fopen* is implemented by a **proc**, a new process is created to service each invocation. This process declares private instances of the file-access operations and returns capabilities for them to the *DirectoryMgr* that called *fopen*. That manager in turn passes the capabilities to the client who called *open* (some of the capabilities are set to **null** if the client did not request or does not have permission to perform all operations). After executing **reply**, the *fopen* process engages in a *conversation* with that client, servicing the file-access operations until the client closes the file. Each client who opens a file thus has its own server process, which manages client-specific data. The server processes use the *lock* operation as a semaphore to protect critical sections that access shared buffers and tables.

The final component of our example is the body of the *DirectoryMgr* resource, which must be compiled after the body of *FileServer* since it creates instances of *FileServer*.

```
body DirectoryMgr(...)
  import FileServer, Terminal, Channel
  op lock()
  ...          # local tables and other shared declarations
  ...          # initialize lock and shared variables
  proc open(pn, ...) returns fd
    var fsc : cap FileServer
    ...  # search path pn to find file location, size, etc.
    receive lock()  # protect critical section
    if FileServer does not exist →
      fsc := create FileServer(fclose, ...)
      ...  # allocate table entry and store fsc in it
    fi
    ...  # increment reference count in file server table
    send lock()
    fd := fsc.fopen()  # create service process
    ...  # check access rights and nullify fields in fd as needed
  end open
  proc fclose(...)
    receive lock()
    ...  # decrement reference count in file server table
    ...  # if count is 0, destroy the FileServer
    send lock()
  end fclose
  # declarations of other operations
end DirectoryMgr
```

Above, *open* is implemented by a **proc** to permit time-consuming, noncritical portions of different opens, such as path-name searching, to proceed concurrently. Consequently, critical sections need to be protected using the *lock* operation as a

semaphore. An alternative design would be to service all critical *DirectoryMgr* operations by an **in** statement in one process.

4. IMPLEMENTATION OVERVIEW

The current SR implementation is built on top of UNIX and consists of three major components: the compiler, linker, and run-time support. Below we describe how SR programs are built and executed using these components and then how the major language features are implemented. We conclude by giving the status of the current implementation and some measurements of its size and performance. The same basic philosophy that guided the design of the language has guided its implementation: common uses of language features should have a simple, efficient implementation.

The compiler has a traditional internal structure: a lexical analyzer, recursive-descent parser, and code generator. The lexical analyzer and parser employ common techniques. The code generator emits C source code, which is passed through the C compiler to produce machine code (MC).[12] The SR compiler supports separate compilation of entire resources, resource specs, resource bodies, and globals.

The linker provides the means by which the user constructs a program from previously compiled resources. The input to the linker is a list of resources and a list of physical machines on which the program is to execute. The linker parses and verifies the legality of its input (e.g., it checks to make sure that the resources have been compiled in an acceptable order) and then uses the standard UNIX linker to create a load module. The input to the linker also designates one of the physical machines as the program's "main" physical machine and one of the resources as the program's "main" resource. A virtual machine is created on the main physical machine when the program begins. One instance of the main resource is then created, and begins execution, within that virtual machine. Each virtual machine (VM) executes as a single UNIX process in which concurrency is simulated by the run-time support. VMs exchange messages using UNIX sockets.

The run-time support (RTS) provides the environment in which the MC executes. The RTS provides primitives for resource creation and destruction, operation invocation and servicing, and memory allocation; it also supports the implementation-specific language mechanisms described in Section 2.4. Internally, the RTS contains a *nugget*: a small collection of indivisible process management and semaphore primitives. The RTS hides the details of the network from the MC; that is, the number of machines and their topology is transparent to the MC. When the RTS receives a request for a service provided on another machine—for example, create a resource or invoke an operation—it simply forwards the request to the destination machine. Upon arrival at that machine, the local RTS processes the request just as though it had been generated locally. Results from such requests are transmitted back in a similar fashion.

[12] Originally the Amsterdam Compiler Kit (ACK) [37] was used for code generation. We switched to generating C so that people who want to use our compiler would not have to purchase ACK.

4.1 Resource Creation and Destruction

On each VM, the RTS maintains a table of active resource instances. A resource capability consists of (1) a VM identity, a pointer into the resource instance table, and a sequence number and (2) an operation capability for each of the operations declared in the resource's specification (operation capabilities are described below in Section 4.2). The sequence number for a resource is assigned when the instance is created; it is stored in the resource instance table. The RTS uses sequence numbers to determine whether a resource capability refers to a resource instance that still exists, that is, whether the referenced resource instance has been destroyed.

The MC for the **create** statement builds a creation block that contains the identity of the resource to be created, the VM on which it is to be created, and the values of any parameters. This block is passed to the RTS, which transmits it to the designated VM. When the creation block arrives at the designated VM, the (local) RTS allocates a table entry for the instance and fills in the first part of the resource capability accordingly. The RTS then creates a process to execute the resource's initialization code.

The MC for every resource includes initialization code even if there is no user-specified initialization code. The key functions of such code are to allocate memory for resource variables (the size of which may depend on the parameters in the resource heading), to initialize resource variables that have initialization expressions as part of their declaration, and to create operations declared in the resource spec or outer level of the body. To accomplish operation creation, the MC interacts with the RTS. For each operation that is being created, the RTS allocates and initializes an entry in its operation table (see Section 4.2); if the operation is in the resource's specification, the RTS also fills in the appropriate field in the resource capability that will be returned from **create**. The initialization process executes this implicit initialization code, then any user-specified initialization code, and finally additional implicit initialization code to create any background processes in the resource.

To destroy a resource instance, the MC passes the RTS a capability for the instance. If the resource contains finalization code, the RTS creates a process to execute that code. When that process terminates, or if there was no finalization code, the RTS uses the resource instance table to locate processes, operations, and memory that belong to the resource instance. The RTS then kills the processes, frees the entries in the resource and operation tables, and frees the resource's memory. The sequence number in each freed entry is incremented so that future references to a resource that has been destroyed or to one of its operations can be detected as being invalid.

When an SR program begins execution, first the nugget and then the RTS initialize themselves. Then an instance of the main resource is created much in the same way that any other resource instance is created.

4.2 Operations

The RTS also maintains an operation table on each VM. This table contains an entry for each operation that is serviced on that VM and is currently active. The entry indicates whether the operation is serviced by a **proc** or by input

statements. For an operation serviced by a **proc**, the entry contains the address of the code for the **proc**. For an operation serviced by input statements, the entry points to its list of pending invocations. An operation capability consists of a VM identity, an index into the operation table, and a sequence number. The sequence number serves a purpose analogous to the sequence number in a resource capability: it enables the RTS to determine whether an invocation refers to an operation that still exists. (An operation exists until its defining resource is destroyed or its defining block terminates.)

4.2.1. *Invocation Statements.* To invoke an operation, the MC first builds an invocation block, which consists of header information and actual parameter values. The MC fills in the header with the kind of invocation (**call**, **send**, concurrent **call**, or concurrent **send**) and the capability for the operation being invoked. Then, the MC passes the invocation block to the RTS. If necessary, the RTS transmits the invocation block to the VM on which the operation is located (recall that capabilities contain VM identities). The RTS then uses the index in the operation capability to locate the entry in the operation table, and thus determine how the operation is serviced. For an operation serviced by a **proc**, the RTS creates a process and passes it the invocation block.[13] For an operation serviced by input statements, the RTS places the invocation block onto the list of invocations for the operation; then it determines if any process is waiting for the invocation, and, if so, awakens such a process. In either case, for a call invocation the RTS blocks the calling process; when the operation has been serviced, that process is awakened and retrieves any results from the invocation block.

The implementation of **co** statements builds on the implementation of **call** and **send** statements. First, the MC informs the RTS when it begins executing a **co** statement. The RTS then allocates a structure in which it maintains the number of outstanding call invocations (i.e., those that have been started but have not yet completed) and a list of call invocations that have completed but have not been returned to the MC. Second, the MC performs all the invocations without blocking. For each call invocation the MC places an *arm number*—the index of the concurrent command within the **co** statement—in the invocation block. Third, since send invocations complete immediately, the MC executes the postprocessing block (if any) corresponding to each send invocation. The MC then repeatedly calls an RTS primitive to wait until call invocations complete. For each completed call invocation, the MC executes the postprocessing block (if any) corresponding to the invocation; specifically, it uses the arm number in the invocation block as an index into a jump table of postprocessing blocks. When all invocations have completed, or when one of the postprocessing blocks executes **exit**, the MC informs the RTS that the **co** statement has terminated. The RTS then discards any remaining completed **call** invocations and arranges to discard any call invocations for this **co** statement that might complete in the future. The infrequent situation in which a postprocessing block itself contains a **co** statement is handled by a slight generalization of the above implementation.

4.2.2. *The Input Statement.* The input statement is the most complicated statement in the language and has the most complicated implementation. In its

[13] In some cases, the RTS can avoid creating a process; see Section 4.2.3 for details.

most general form, a single input statement can service one of several operations and can use synchronization and scheduling expressions to select the invocation it wants. Moreover, an operation can be serviced by input statements in more than one process, which thus compete to service invocations. However, as we shall see, the implementation of simple, commonly occurring cases is quite efficient.

Classes are fundamental to the implementation of input statements. They are used to identify and control conflicts between processes that are trying to service the same invocations. Classes have a static aspect and a dynamic aspect. A static class of operations is an equivalence class of the transitive closure of the relation "serviced by the same input statement." At compile time, the compiler groups operations into static classes on the basis of their appearance in input statements. At run time, actual membership in the (dynamic) classes depends on which operations in the static class are extant. For example, an operation declared local to a process joins its dynamic class when the process is created and leaves its dynamic class when the process completes execution. The RTS represents each dynamic class by a *class structure*, which contains a list of pending invocations of operations in the class, a flag indicating whether or not some process has access to the class, and a list of processes that are waiting to access the class. Each operation table entry points to its operation's class structure.

At most one process at a time is allowed to access the list of pending invocations of operations in a given class structure. That is, for a given class, at most one process at a time can be selecting an invocation to service or appending a new invocation. Processes are given access to both pending and new invocations in a class structure in first-come, first-served order. Thus, a process waiting to access the invocations will eventually obtain access as long as all functions in synchronization and scheduling expressions in input statements eventually terminate.

The RTS and nugget together provide seven primitives that the MC uses for input statements. These primitives are tailored to support common cases of input statements and have straightforward and efficient implementations. They are

access(*class*). Acquire exclusive access to *class*, which is established as the current class structure for the executing process. That process is blocked if another process already has access to *class*. The RTS will release when this process blocks access trying to get an invocation or when this process executes *remove* (see below).

get_invocation( ). Return a pointer to the invocation block the executing process should examine next. This invocation is on the invocation list in the current class structure of the executing process; successive calls of this primitive return successive invocations. If there is no such invocation, the RTS releases access to the executing process's current class structure and blocks that process.

get_named_inv(*op_cap*). Get the next invocation of operation *op_cap* (an operation capability) in the executing process's current class; a pointer to the invocation block is returned.

get_named_inv_nb(*op_cap*). Get an invocation of *op_cap*. This primitive is identical to *get_named_inv* except that it does not block the executing process if no invocation is found; instead, it returns a null pointer in that case. It is used when the input statement contains a scheduling expression.

remove(invocation). Remove the invocation block pointed at by *invocation* from the invocation list of the executing process's current class. The RTS also releases access to the executing process's current class structure.

input_done(invocation). Inform the RTS that the MC has finished executing the command body in an input statement and is therefore finished with the invocation block pointed at by *invocation*. If that invocation was called, the RTS passes the invocation block back to the invoking process and awakens that process.

receive(class). Get and then remove the next invocation in *class*. This primitive is a combination of *access(class)*, *invocation* := *get_invocation*( ), and *remove(invocation)*; hence, it returns a pointer to an invocation block. It is used for simple input statements and for **receive** statements.

The ways in which these primitives are used by the MC is illustrated below by four examples. More complicated input statements are implemented using appropriate combinations of the primitives.

Consider the simple input statement

in $q(x) \rightarrow$ ... ni

This statement delays the executing process until there is some invocation of q, then services the oldest such invocation. (Note that **receive** statements expand into this form of input statement.) For this statement, if q is in a class by itself, the MC executes *invocation* := *receive*(q's class). If q is not in a class by itself, the MC executes *access*(q's class), *invocation* := *get_named_inv*(q), and *remove(invocation)*. In either case, the MC then executes the command body associated with q, with parameter x bound to the value for x in the invocation block, and finally executes *input_done(invocation)*.

Second, consider

in $q(x) \rightarrow$ ... [] $r(y,z) \rightarrow$ ... ni

This statement services the first pending invocation of either q or r. Note that q and r are in the same class because they appear in the same input statement. Here, the MC first uses *access*(q's class) and then *invocation* := *get_invocation*( ) to look at each pending invocation in the class to determine whether it is an invocation of q or r (there might be other operations in the class). If the MC finds an invocation of q or r, it calls *remove(invocation)*, then executes the corresponding command body with the parameter values from the selected invocation block, and finally executes *input_done(invocation)*. If the MC finds no pending invocation of q or r, the executing process blocks in *get_invocation* until an invocation in the class arrives. When such an invocation arrives, the RTS awakens the process, which then repeats the above steps.

As the third example, consider an input statement with a synchronization expression:

in $q(x)$ and $x > 3 \rightarrow$ ... ni

This statement services the first pending invocation of q for which parameter x is greater than three. The MC first uses *access*(q's class) to obtain exclusive access to q's class. The MC then uses *invocation* := *get_invocation*( ) or *invocation* := *get_named_inv*(q) to obtain invocations of q one at a time. The first primitive is used if q is in a class by itself; otherwise, the second is used. For

each such invocation, the MC evaluates the synchronization expression using the value of the parameter in the invocation block. If the synchronization expression is true, the MC notifies the RTS of its success by calling *remove*(*invocation*), executes the command body associated with *q*, and calls *input_done*(*invocation*). If the synchronization expression is false, the MC repeats the above steps to obtain the next invocation.

Finally, consider an input statement with a scheduling expression:

in $q(x)$ by $x \rightarrow$ ... ni

This statement services the (oldest) pending invocation of *q* that has the smallest value of parameter *x*. In this case, the MC uses the same steps as in the previous example to obtain the first invocation of *q*. It then evaluates the scheduling expression using the value of the parameter in the invocation block; this value and a pointer, *psave*, to the invocation block are saved. The MC then obtains the remaining invocations by repeatedly calling *invocation* := *get_named_inv_nb*(*q*). For each of these invocations, the MC evaluates the scheduling expression and compares it with the saved value, updating the saved value and pointer if the new value is smaller. When there are no more invocations (i.e., when *get_named_inv_nb* returns a null pointer), *psave* points to the invocation with the smallest scheduling expression. The MC acquires that invocation by calling *remove*(*psave*), then executes the command body associated with *q*, and finally calls *input_done*(*psave*).

Note that synchronization and scheduling expressions are evaluated by the MC, not the RTS. We do this for two reasons. First, these expressions can reference objects such as local variables for which the RTS would need to establish addressing if it were to execute the code that evaluates the expression. Second, these expressions can contain invocations; it would greatly complicate the RTS to handle such invocations in a way that does not cause the RTS to block itself. A consequence of this approach to evaluating synchronization and scheduling expressions is that the overhead of evaluating such expressions is paid for only by processes that use them.

4.2.3 *Optimizations.* Two kinds of optimizations are applied to certain uses of operations. First, for a call invocation of a **proc** that is in the same resource as the caller and that does not contain a **reply** statement, the compiler generates conventional procedure-call code instead of going through the RTS, which would create a process.[14] The compiler generates code that builds an invocation block on the calling process's stack and passes the block's address to the called **proc**. Thus the code in the **proc** is independent of whether it is executed by the calling process or as a separate process. A similar optimization is performed for a call invocation of a **proc** that is located on the same VM as the caller and that does not contain a **reply** statement. In this case, however, the RTS must be entered since the compiler cannot determine whether an operation in another resource is located on the same VM as its caller (recall that program linking follows and is independent of compilation).[15] Also, the invoking process must create an

[14] A **proc** that executes **reply** executes concurrently with its caller after replying; hence, the **proc** must execute as a process in this case.

[15] In fact, the compiler might not even know whether an operation is implemented as a **proc** because it might not yet have compiled the body of the resource containing the invoked operation.

invocation block since it is possible that the invoking process might be in a resource that is destroyed before the invoked **proc** completes.

The second optimization is that certain operations are implemented directly by the nugget's semaphores rather than by the general mechanisms described above. The main criteria that an operation must satisfy to be classified as a semaphore operation are that the operation: (1) is invoked only using **send**, (2) has no parameters or return value, (3) is serviced by input (or **receive**) statements in which it is the only operation and in which there are no synchronization or scheduling expressions, and (4) is declared at the resource level. Note that these criteria are relatively simple to check. Furthermore, they capture typical uses of operations that provide intraresource synchronization such as controlling access to shared variables.

4.2.4 *Failure Handling Mechanisms.* The address of a process's **proc** handler is recorded by the RTS when the process is created. If the **proc** does not contain an explicit handler, the RTS instead records the address of a special **abort** handler. The address of a process's current invocation handler is maintained by the MC in a location known to the RTS. If an invocation statement does not contain an explicit handler, the MC instead sets the invocation handler to what is recorded as the process's **proc** handler. Thus, from the RTS's view, there is always one handler associated with each process's **proc** and invocation failures. When the RTS detects an exception or is informed by the MC that an exception has been raised (i.e., an **abort** statement was executed), it transfers control to the appropriate handler's code. The special **abort** handler will cause the RTS to abort the current process and to pass the failure up the call chain if the **proc** was called; the above actions are then applied recursively. Aborting a process also causes a failure to be passed to the callers of any invocations currently being serviced by **in** statements or of any invocations pending for operations declared within the **proc**.

When a **when** statement is executed, the MC creates an invocation block for the specified operation and arguments. It then passes to the RTS the block's address and the identity of the object to monitor, that is, the argument to *failed*. The RTS records this information and initiates the monitoring of the object. Monitoring of physical and virtual machines is accomplished by means of "heartbeat" messages that each machine periodically broadcasts to the others. When a process or resource instance is to be monitored, the local RTS sends a message to the remote VM informing that VM's RTS to monitor the object; the remote RTS will send a "failed" message to the local RTS should the object fail or terminate. (The local and remote RTS could of course be the same.) The local RTS also periodically checks the states of the virtual and physical machines on which the process or resource instance resides to ensure they have not failed or terminated. Should the RTS detect that any object it is monitoring has failed, it performs its usual actions to invoke the user-specified operation.

4.3 Status, Plans, and Statistics

An initial implementation of a large subset of SR became operational under 4.3BSD Vax UNIX in November 1985. Since then, the language has been further

refined and the implementation has been modified, extended, and ported to Sun workstations and an Encore Multimax. Currently the full language has been implemented except for failure handlers and a few minor features. The current implementation also includes facilities to invoke C functions as operations, thereby gaining access to underlying UNIX system calls. The UNIX versions of the implementation will be completed in early 1988, at which time they will be made available to interested groups. Work is under way on a version of the implementation that will allow SR programs to run stand-alone.

Our implementation was first used in graduate classes in concurrent programming in which students wrote moderate-sized (500–5000 lines) distributed programs including card games, automatic teller machines, simple airline reservation systems, and prototypes of a command interpreter and a file system for a distributed operating system. Subsequently, SR has been used to program experiments relating program structure and process interaction patterns [7], a highly parallel interpreter for Prolog, and Saguaro's file system [32].

Almost all of the SR compiler, linker, and RTS is written in machine-independent C. The only exception is that the RTS nugget contains some assembly language code for process management (e.g., stack setup and context switches). A few *awk* and *sed* tools are used to simplify maintenance of the compiler, and *lex* is used to generate the lexical analyzers for the compiler and linker. The compiler consists of approximately 17,000 lines of C source code. The linker is about 1100 lines of C. The RTS and nugget together contain about 4200 lines of C, including code for the I/O and network interfaces to UNIX; in addition, the nugget contains about 100 lines of assembly code.

The SR compiler processes 3200 lines per CPU minute on a Vax 8650. To give some comparison, the C compiler processes about 22,000 lines per CPU minute. Thus the SR compiler is about seven times as slow. This is not surprising since SR is a higher level language than C and the SR compiler generates C code that has to be processed by the C preprocessor, C compiler, and Vax assembler. When machine-code generation is turned off, the SR compiler processes 17,600 lines per CPU minute. Stated differently, 18 percent of compilation time is spent in the SR compiler itself; 82 percent is spent processing the generated C code.

At run time, the Vax RTS (including the nugget) requires about 24K for text and 5K for static data. Entries for resource instances, operations, and processes are dynamically allocated. Included in each load module are an additional 15K of text and 6K of data for the UNIX I/O and network library routines.

The time to process an invocation depends on whether it is generated by a **call** or a **send** and whether it is serviced by a **proc** or an **in**. We obtained timing data for six simple SR programs: one for each of the four combinations of invocation and service, one that optimizes calling a **proc**, and one that uses semaphores. (All operations in the test programs are parameterless.) For comparison purposes, we also obtained timing data for a simple C program. Each test program generates and services 100,000 invocations. The times, in microseconds, to process single invocations are listed in Table II. These times were obtained using the *time* command on a Vax 8600. They are averages of ten executions of each test program and include loop overhead, and so actual invocation processing time is somewhat less than shown.

Table II

Program	*Description*	*μsec/inv*	*Relative to C*
C	C equivalent of *A*	5	1
A	**call** to **proc**; procedure call	8	2
A1	**call** to **proc**; new process	511	102
A2	**send** to **proc**; new process	470	94
B	**call** to **receive**; 2 processes	390	78
B1	**send** to **receive**; 1 process	160	32
B2	**send** to **receive**; semaphore	13	3

The "Relative to C" column shows the ratio of the time per invocation for the given program to that for the C program *C*; for example, *A* is about two times slower than *C*.

Each SR test program consists of a single resource. However, except for *A*, the above times would be the same if the invoker and server were in different resources provided they were in the same VM. A synopsis of the different test programs follows.

C: C test program that simply invokes an empty procedure 100,000 times.

A: *A* is the SR equivalent of *C*: it calls an empty **proc** in the same resource, which the compiler optimizes as described in Section 4.2.3. The overhead relative to *C* results from the need to support the general case in which the **proc** might also be invoked from outside the resource.

*A*1: *A*1 generates **call** invocations of an operation serviced by a **proc** that executes a **reply**. Thus a new process is created to service each invocation and a context switch is required.

*A*2: *A*2 generates **send** invocations to an operation serviced by a **proc**. It is like *A*1, but there is no context switch overhead.

B: *B* contains two processes. One generates **call** invocations; the other consumes them using **receive**. The processes must alternate execution for each invocation. Context switching is the dominant cost in this program.

*B*1: *B*1 contains a single process that send messages to itself. This shows the cost of using **send** and simple **in** statements with no context switches.

*B*2: *B*2 is like *B*1 except the operation is implemented as a semaphore.

The overhead in the four most costly SR tests results from the allocation of invocation blocks, procedure calls to enter the RTS and within the RTS, maintenance of RTS structures, and in some cases process creation and context switches. The major cost for the semaphore test program is the procedure calls required to get to the nugget's **P** and **V** primitives.

The above measurements give some idea of the absolute and relative costs of different combinations of invocation and service. (A more thorough discussion of SR's performance appears in [8].) The comparisons of the SR programs with the C program are somewhat unfair, however, because SR is quite a different language. For example, SR has mechanisms, such as **send** and **in**, that have no counterparts in C; SR also has dynamic resources and processes, which require a more complicated RTS. On the other hand, it is desirable that SR programs such as *A* that use only C-like mechanisms should not be too much slower than

their C counterparts, and this is the case. Work is currently under way to improve both the MC and the RTS to further speed up the implementation.

5. DISCUSSION

SR lies in between recent languages in which a distributed program appears to execute on one virtual machine (e.g., Ada, Linda [23], and NIL [31, 36]) and more conventional languages in which a distributed system is built from distinct programs, one per machine. In SR, the invoker of an operation need not be concerned with where that operation is serviced, but mechanisms are provided to enable the programmer to exert some control over a program's execution environment. For example, the programmer can control where a resource is created and can determine whether a resource or machine has failed. In this respect, SR is similar to the V kernel [15]. However, SR and the V kernel take quite different approaches to constructing distributed programs. SR is a strongly typed language with an integrated set of mechanisms for sequential and distributed programming; the V kernel is a typeless collection of message-passing primitives that are invoked from a sequential language such as C. The V kernel has been designed with efficiency being the most important criteria; SR has been designed to balance expressiveness, ease of use, and efficiency.

The most important aspects of SR are discussed and related to other approaches to distributed programming in the remainder of this section.

5.1 Integration of Language Constructs

There is a large similarity between the sequential and concurrent mechanisms in SR. For example, the **if**, **do**, and **in** statements have similar appearances and the same underlying nondeterministic semantics. The **fa**, **co**, and **in** statements all use quantifiers to specify repetition. Finally, the **exit** and **next** statements are interpreted uniformly within iterative statements (**do** and **fa**) and the **co** statement.

CSP [25] has a similar integration of mechanisms. By way of contrast, Ada [1] provides distinct mechanisms for sequential and concurrent programming. In Ada, tasks are used for concurrent modules, but packages are used for sequential modules; also, **select** is used for selecting alternative entries, but **if–then–else** is used for selecting alternative conditions. As a specific example, consider how one would program a *Queue* package and *BoundedBuffer* task in Ada, and compare them to our *Queue* and *BoundedBuffer* resources. In Ada, the differences are marked. In SR, the differences are minimal and in fact the interfaces to the two resources are identical. A similar difference between the sequential and concurrent components of a language results whenever an existing sequential language is extended with concurrency constructs (e.g., Concurrent C [22]).

The SR mechanisms for communication and synchronization are also well integrated. Operations support all of local and remote procedure call, rendezvous, dynamic process creation, asynchronous message passing, multicast, and semaphores. In addition, there is just one way—capabilities—in which an operation is named in an invocation. Moreover, capabilities are used for virtual machines, entire resources, and individual operations—another example of similar mechanisms for similar concepts—and are first-class objects in the language. Such

integration and flexibility is not achieved in languages like Ada and EPL [11], where many mechanisms, each having special rules and restrictions, are used to achieve the same effects that are realizable with just a few SR mechanisms.

The integration of the various language mechanisms plus the almost total lack of restrictions make SR easy to learn and use. Students who have used SR for term projects were able to learn the language and design and code their projects in about two weeks. Although these projects were of modest size, they supported multiuser interactions and used most of the language features that would be used in "real" concurrent programs. These features—resource creation/destruction, operations, capabilities, and invocations—caused the students few conceptual difficulties.

5.2 Resources

The structure of the **resource** construct is similar to that of modular constructs in procedure-based languages, such as Euclid [26] and Modula-2, and other distributed programming languages, such as Distributed Processes [13], StarMod [19], Argus [27], and EPL. Like Modula-2 and Ada, SR allows the specification of a resource to be compiled separately from its body. This permits the interface to a resource to be separated from its implementation. It also permits construction of programs, such as the file system in Section 3.2, in which resources invoke each other's operations.

SR goes beyond the above languages in two ways. First, a resource body can be parameterized. This permits instances to have different internal characteristics and external communication connections. In this respect, SR is more like LYNX [35]. Second, SR includes an inheritance mechanism, the **extend** phrase, that permits an interface to be split into multiple parts and supports multiple implementations of the same abstract interface. In this respect, SR is more like Mesa [29] and Emerald [12]. As with other aspects of the language, we have tried to provide integrated mechanisms that support functionality that has been found to be useful.

Resources provide the only data-abstraction mechanism in SR. They are used to program sequential "abstract data types" such as *Queue* as well as concurrent data types such as *BoundedBuffer*. Having just one abstraction mechanism makes the language smaller and hence easier to learn than if two separate mechanisms were provided, one for sequential types and one for concurrent types. There is a disadvantage, though: The implementation of sequential types is not as efficient as it might be since a resource that implements a type might be located on a different machine than its clients. We are able to perform some optimizations when a resource and its clients are located on the same (virtual) machine, but not as many as would be possible if "sequential" resources were distinguished as such in the language and were forced to be located on the same machine as their clients. A second potential shortcoming of resources is that they are not polymorphic: They may not have types as parameters. We have not, however, found many situations in which a generic resource facility would justify its large implementation cost.

We do not allow resources to be nested, primarily because nesting is not needed. If one resource needs the services provided by another, it can either create an instance of the needed resource or be passed a capability for it.

Precluding nesting also simplifies the implementation. One disadvantage, though, is that different resources cannot share variables, although pointers can be passed between resources on the same virtual machine. We also do not allow processes to be nested for essentially the same reasons. In contrast, Ada allows arbitrary nesting of tasks, packages, and subprograms. This makes the implementation of Ada much more complicated and costly and makes many programs more difficult to understand [18].

A resource can contain initialization and finalization code. Initialization code gives the programmer a way to control the order in which initialization is done; for example, the programmer can ensure that resource variables are initialized before processes are created. Initialization code is executed as a process so that it can use any of the language mechanisms (another instance of our aversion to imposing restrictions). For example, initialization code can service operations, create other resources, or do whatever else might be required.

Finalization code provides a means by which a resource can "clean up" before it disappears. For example, if a resource has obtained a lock for a file, it can record that it owns the lock; its finalization code can then release that lock if the resource is ever destroyed. Finalization code is executed as a process, again so it can use any of the language mechanisms. Our approach is similar to that in NIL. A different approach is used in Ada. When an Ada task is aborted, it does not get control—it is just destroyed.[16] Thus, in the above example, there is no way the task itself can release the lock; such a release can only be done by another task that is monitoring the task that was aborted.

SR supports multiple active processes within each resource instance; a separate, potentially concurrent thread of control is associated with each **proc** invocation. This is similar to the approaches taken in Ada, EPL, Linda, and NIL. A different approach is taken in DP and LYNX, where threads execute as coroutines. We prefer our approach since an SR process corresponds to the usual conceptual notion of a process. Also, this approach admits a multiprocessor-based implementation in which processes in the same resource might truly execute concurrently. Finally, this approach accommodates immediate processing of operations that service interrupts. A drawback of having concurrent threads is that processes must synchronize access to shared variables to avoid race conditions. However, how to do so is now well understood, and SR's operations can be used to simulate semaphores in an efficient way.

5.3 Operations

Operations in SR can be invoked either synchronously (**call**) or asynchronously (**send**). Many other languages (e.g., Ada and CSP) provide only synchronous message passing. While this is very useful, especially for programming client/server interactions, asynchronous message passing is also useful. First, it can be used to avoid *remote delay* in which a server, in processing a request, invokes an operation in another server that might delay [28]. In particular, **send** can be used to invoke the remote operation whenever it is necessary for the first server to honor other requests in order to remove the conditions that led to remote

[16] Task destruction may not be immediate; for example, a task is allowed to complete servicing a rendezvous before the task is destroyed.

delay. This was shown in the *server* process in the *Servant* resource in Section 3.1. In a language that provides only synchronous message passing, extra processes must be employed to avoid remote delay; this often complicates problem solutions. Asynchronous message passing is also useful whenever it is not necessary to delay the invoker of an operation. For example, it can be used to program pipelines of filter processes, where it is most natural for the producer to continue after sending a message to the consumer.

The **co** statement provides additional flexibility in invoking operations. It allows the invoker to call several operations at the same time and to continue when an appropriate combination of replies has been received. In addition, the postprocessing block associated with each concurrent invocation allows the programmer to handle the reply from each invocation in a manner appropriate to that invocation. **co** *can* be simulated using **send** and **in**. However, such a simulation results in a much more complex program. It also requires changing the interface between the invoking and servicing processes since parameters and results have to be sent as separate messages. In addition to being useful, **co** is relatively simple to implement since its implementation can use the basic invoke and reply primitives in the RTS. Thus **co** illustrates how opening up the implementation provides additional, useful flexibility. Note that the **co** statement is similar to Argus's **coenter** statement and to the V kernel's multicast mechanisms [16].

All operations are invoked using capabilities.[17] In addition to capabilities for entire resources, capabilities for individual operations are provided. This makes some programming jobs easier since it overcomes the limitations of Eden's capabilities, which can only be bound to entire modules [10]. For example, a command server in Saguaro is passed a record of operation capabilities. One field of the record is for standard input and another is for standard output. These fields can be bound to operations in different resources; for example, the capability for standard input might be bound to a read operation in a file server, while the capability for standard output might be bound to a write operation in the terminal driver.

Operations can be declared within a process (a *local* operation) or at the resource level (a *resource* operation). Local operations support the programming of conversations, as shown in Section 3.2. Resource operations provide the most commonly used form. Of importance is that resource operations, like resource variables, may be shared; that is, they can be serviced by **in** statements in more than one process. Shared resource operations are almost a necessity, given that multiple instances of a **proc** can service the same resource operation. They are also useful since they can be used to implement conventional semaphores, "data-containing" semaphores, and server work queues. A data-containing semaphore is a semaphore that contains data as well as a synchronization signal. As an example, we use such semaphores to implement buffer pools in Saguaro. A buffer is produced by sending its address to a shared operation; a buffer is consumed by receiving its address from the shared operation. A shared operation can also be used to permit multiple servers to service the same work queue. Clients request

[17] An operation can also be invoked using just its name if that name is declared in the current scope. Such a name is treated as a capability constant for the named operation.

service by invoking a shared operation. Server processes (in the same resource) wait for invocations of the shared operation; which server actually receives and services a particular invocation is transparent to the clients. In addition to being useful, shared resource operations can be implemented almost as efficiently as nonshared operations; the only additional requirement is grouping operations into classes, each of which has a lock (as discussed in Section 4.2.2).

5.4 Issues Related to Program Distribution

Many distributed programs have a hierarchical structure in which resources provide operations that are invoked only by higher level resources. This is not always the case, however. In some programs, such as the file system example in Section 3.2, two resources interact as equals, with each resource both providing operations used by the other and using operations provided by the other. An additional interaction pattern that has been found to be useful is the *upcall* [17] in which data flows from a server back to a client. All these interaction patterns are supported in SR since a resource may contain more than one process that is servicing invocations and capabilities can be used to pass operations between resources. (A set of experiments using SR to program different upcall program structures is reported in [7].) Ada supports such interaction patterns, although each of its servers is limited to be a single task.

In distributed programs it is important to be able to specify the machine on which the different parts of a program are to execute; for programming a distributed operating system, it is essential. For example, this allows device drivers to be placed on the appropriate machine and provides a basic tool for load sharing. This is one of the lessons learned from Eden [10]. The Eden implementors found it valuable to be able to specify the machine on which an object executes, even though their overall philosophy is to provide an environment in which objects are location independent. SR supports programmer control over placement since the location for a resource can be specified when the resource is created; Argus provides similar support. By contrast, Ada provides no support for placement of tasks.

Related to controlling where a resource is placed is recognizing that there is an inherent difference in efficiency between invoking an operation that is local and one that is remote. Our implementation optimizes calls within a virtual machine as much as possible. Also, the language allows resources that are placed in the same virtual machine to use pointers and reference parameters. This necessitates run-time enforcement and can lead to exceptions, but makes many programs much more efficient than they would be if we insisted that all parameters be copied and prohibited the use of pointers outside a resource.

A distributed programming language must also provide support for detecting and handling machine or network crashes and for dealing with local exceptions, which are inevitable even in the most carefully designed program. SR provides two failure handling mechanisms: handlers and **when** statements.[18] Handlers are used by clients on the invoking side of operations; **when** statements are used by servers of operations. The differences between these two mechanisms reflects

[18] The design of the **when** statement is based on ideas in [34].

that, in general, a client communicates with one server at a time but a server has many potential clients. In contrast, LYNX provides a single exception handling mechanism that uniformly handles failures of either the receiving or sending side of a link; this is possible because only one thread of control can be bound to each end of a link. SR's failure handling mechanisms are higher level than a simple timeout mechanism, such as that used to detect invoker failure in Ada **select/accept** statements, and lower level than mechanisms like atomic actions [27], fault-tolerant actions [33], and replicated procedure call [20]. We feel that our approach is appropriate for the intended application domain of SR. Timeout intervals are used to implement failure detection, but the programmer need not be concerned with such low-level details. We expect that the SR mechanisms will be more efficient than higher level failure-handling mechanisms, and hence they are more appropriate for a systems programming language. In fact, the SR mechanisms can be used to implement high-level mechanisms such as atomic actions.

The final requirements for a language that is used to write distributed operating systems are the abilities to execute user programs and to accommodate a changing hardware configuration. The only language we know of that comes close to meeting these requirements at present is LYNX, in which it is possible for a process to be compiled after and then connect to an already executing program. Although resources and communication links can be created and destroyed dynamically in SR, the machine configuration and collection of resources that comprise a program are static input to the SR linker. To overcome this limitation, we are currently working on the design of two mechanisms. The first is an "execute" facility to load and start execution of an external program, which would interact with the host SR program by being linked to a set of SR library routines. The second mechanism is a generalization of operations that would support group communication somewhat analogous to that provided by V [16].

ACKNOWLEDGMENTS

Nick Buchholz, Roger Hayes, Steve Manweiler, and Rick Schlichting provided valuable feedback on SR_0 and served as a sounding board for new proposals. Hayes, Manweiler, and Stella Atkins wrote large applications that pushed the implementation to its limits. Atkins also helped obtain performance measurements and provided detailed comments on earlier versions of this paper. The students in CSc 552 and CSc 652 forced us to get the compiler finished and made extensive use of the language. Finally, the referees provided useful advice on the content and presentation of this paper; Michael Scott's comments were especially perceptive and helpful.

REFERENCES

1. ADA. Reference manual for the Ada programming language. ANSI/MIL-STD-1815A, American National Standards Institute, New York, Jan. 1983.
2. ANDREWS, G. R. Synchronizing resources. *ACM Trans. Program. Lang. Syst., 3*, 4 (Oct. 1981), 405–430.
3. ANDREWS, G. R. The distributed programming language SR—Mechanisms, design and imple mentation. *Softw. Pract. Exper. 12*, 8 (Aug. 1982), 719–754.

4. ANDREWS, G. R., AND OLSSON, R. A. The evolution of the SR language. *Distrib. Comput. 1*, 3 (July 1986), 133–149.
5. ANDREWS, G. R., AND OLSSON, R. A. Revised report on the SR programming language. TR 87-27, Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz., Nov. 1987.
6. ANDREWS, G. R., SCHLICHTING, R. D., HAYES, R., AND PURDIN, T. The design of the Saguaro distributed operating system. *IEEE Trans Softw. Eng. SE-13*, 1 (Jan. 1987), 104–118.
7. ATKINS, S. Experiments in SR with different upcall program structures. Tech. Rep. Dept. of Computer Science, Simon Fraser Univ., Burnaby, B.C., Canada, Apr. 1987.
8. ATKINS, S., AND OLSSON, R. A. Performance of multitasking and synchronisation mechanisms. CSE-87-10, Div. of Computer Science, Univ. of California at Davis, July 1987.
9. BERNSTEIN, A. J. Predicate transfer and timeout in message passing systems. *Inf. Process. Lett. 24*, 1 (Jan. 1987), 43–52.
10. BLACK, A. P. Supporting distributed applications: Experience with Eden. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4, 1985). ACM, New York, pp. 181–193.
11. BLACK, A. P., HUTCHINSON, N., MCCORD, B. C., AND RAJ, R. K. EPL programmer's guide. Eden Project, Dept. of Computer Science, Univ. of Washington, Seattle, Wash., June 1984.
12. BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. Distribution and abstract types in Emerald. *IEEE Trans. Softw. Eng. SE-13*, 1 (Jan. 1987), 65–76.
13. BRINCH HANSEN, P. Distributed processes: A concurrent programming construct. *Commun. ACM 21*, 11 (Nov. 1978), 934–941.
14. CHANDY, K. M., AND MISRA, J. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst. 6*, 4 (Oct. 1984), 632–646.
15. CHERITON, D. R. The V kernel: A software base for distributed systems. *IEEE Software 1*, 2 (Apr. 1984), 19–42.
16. CHERITON, D. R., AND ZWAENEPOEL, W. Distributed process groups in the V kernel. *ACM Trans. Comput. Syst. 3*, 2 (May 1985), 77–107.
17. CLARK, D. D. The structuring of systems using upcalls. In *Proceedings of the 10th Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec. 1–4, 1985). ACM, New York, pp. 171–180.
18. CLARKE, L. A., WILEDEN, J. C., AND WOLF, A. L. Nesting in Ada programs is for the birds. In *Proceedings of the ACM SIGPLAN Symposium on Ada Progamming Languages* (Boston, Mass., Dec. 9–11, 1980). ACM, New York, pp. 139–145.
19. COOK R. *Mod—A language for distributed programming. *IEEE Trans. Softw. Eng. SE-6*, 6 (Nov. 1980), 563–571.
20. COOPER, E. C. Replication procedure call. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27–29, 1984). ACM, New York, pp. 220–232.
21. DIJKSTRA, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.
22. GEHANI, N. H., AND ROOME, W. D. Concurrent C. *Softw. Pract. Exp. 16*, 9 (Sept. 1986), 821–844.
23. GELERNTER, D. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan 1985), 80–112.
24. HOARE, C. A. R. The emperor's old clothes. *Commun. ACM 24*, 2 (Feb. 1981), 75–83.
25. HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.
26. LAMPSON, B. W., HORNING, J. J., LONDON, R. L., MITCHELL, J. G., AND POPEK, G. J. Report on the programming language Euclid. *SIGPLAN Not. 12*, 2 (Feb. 1977), 1–79.
27. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
28. LISKOV, B., HERLIHY, M., AND GILBERT, L. Limitations of remote procedure call and static process structure for distributed computing. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Fla. Jan. 13–15, 1986), pp. 150–159.
29. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. Mesa language manual, version 5.0. Rep. CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif, Apr. 1979.
30. OLSSON, R. A. Issues in distributed programming languages: The evolution of SR. TR 86-21 (Ph.D. dissertation), Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz., Aug. 1986.

31. PARR, F. N., AND STROM, R. E. NIL: A high-level language for distributed systems programming. *IBM Syst. J. 22*, 1/2 (1983), 111–127.
32. PURDIN, T. Enhancing file availability in distributed systems (the Saguaro file system). TR 87-26 (Ph.D. dissertation), Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz., Oct. 1987.
33. SCHLICHTING, R. D., AND SCHNEIDER, F. B. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst. 1*, 3 (Aug. 1983), 222–238.
34. SCHLICHTING, R. D., CRISTIAN, F., AND PURDIN, T. Mechanisms for failure handling in distributed programming languages. TR 87-13, Dept. of Computer Science, Univ. of Arizona, Tucson, Ariz., June 1987.
35. SCOTT, M. L. Language support for loosely coupled distributed programs. *IEEE Trans. Softw. Eng. SE-13*, 1 (Jan. 1987), 88–103.
36. STROM, R. E., AND YEMINI, S. NIL: An integrated language and system for distributed programming. Res. Rep. RC 9949, IBM Research Division, Yorktown Heights, N.Y., Apr. 1983.
37. TANENBAUM, A. S., VAN STAVEREN, H., KEIZER, E. G., AND STEVENSON, J. W. A practical tool kit for making portable compilers. *Commun. ACM 26*, 9 (Sept. 1983), 654–660.
38. WIRTH, N. Modula: A language for modular multiprogramming. *Softw. Pract. Exper. 7*, (1977), 3–35.
39. WIRTH, N. *Programming in Modula-2*. Springer Publ., New York, 1982.

Received June 1986; revised July 1987; final revision accepted September 1987