Distributed Computing (1986) 1:133–149

DISTRIBUTED COMPUTING

© Springer-Verlag 1986

The evolution of the SR language

Gregory R. Andrews and Ronald A. Olsson
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA

Gregory R. Andrews *was born in Olympia, WA, in 1947. He received the B.S. degree in mathematics from Stanford University in 1969 and the Ph.D. degree in computer science from the University of Washington in 1974. From 1974–1979 he was an Assistant Professor of Computer Science at Cornell University. Since 1979 he has been an Associate Professor of Computer Science at the University of Arizona. During 1983–1984, he was a Visiting Associate Professor of Computer Science at the University of Washington. He has also consulted for the U.S. Army Electronics Command and Cray Laboratories. His research interests include concurrent programming languages and distributed operating systems; he is currently co-authoring (with Fred Schneider) a textbook on concurrent programming. Dr. Andrews is a member of the Association for Computing Machinery (ACM). From 1980–1983 he was Secretary-Treasurer of the ACM Special Interest Group on Operating Systems. He has also been on the Board of Editors of Information Processing Letters since 1979.*

Ronald A. Olsson *was born in Huntington, NY, in 1955. He received B.A. degrees in mathematics and computer science and the M.A. degree in mathematics from the State University of New York, College at Potsdam, in 1977. In 1979, he received the M.S. degree in computer science from Cornell University. He was a Lecturer of Computer Science at the State University of New York, College at Brockport, from 1979 to 1981. Since 1981 he has been a graduate student in computer science at the University of Arizona and will complete his Ph.D. in June 1986. His research interests include programming languages, operating systems, distributed systems, and systems software. Mr. Olsson is a student member of the Association for Computing Machinery.*

Reprinted Issue
without
Advertisements

Abstract. As a result of our experience, the SR distributed programming language has evolved. One change is that resources and processes are now dynamic rather than static. Another change is that operations and processes are now integrated in a novel way: all the mechanisms for process interaction – remote and local procedure call, rendezvous, dynamic process creation, and asynchronous message passing – are expressed in similar ways. This paper explains the rationale for these and other changes. We examine the fundamental issues faced by the designers of any distributed programming language and consider the ways in which these issues could be addressed. Special attention is given to the design objectives of expressiveness, simplicity, and efficiency.

Key words: Distributed programming – Programming languages – Operating systems – Synchronization

1 Introduction

We and several colleagues are implementing a distributed operating system called Saguaro (Andrews et al. 1985), which will control a network of Sun workstations. Given our past work on SR (Andrews 1981, 1982), it was a natural choice for the implementation language. Using the original version of the language (SR_0), we programmed prototypes of several Saguaro components. This substantiated the general appropriateness of the language, but also pointed out several deficiencies.

Offprint requests to: G.R. Andrews

This work is 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

Consequently, we reexamined the entire language and redesigned several parts. We believe SR is now an attractive language for implementing distributed systems such as Saguaro.

This paper describes SR, the revised version of SR_0. We do so by examining the main issues addressed in redesigning the language. We describe what has changed and why, and what has not changed and why not. The issues discussed are fundamental ones that must be addressed by the designers of any distributed programming language. Since language design is the art of trading off competing concerns, there is no single "right" answer for many of the issues. Some tradeoffs, however, are better than others. Our hope is to shed some light on how to make these tradeoffs.

The next section examines how distributed programs are structured and activated. Section 3 discusses issues related to interprocess communication and synchronization. Section 4 considers a few additional issues. Finally, Sect. 5 contains concluding remarks. Throughout, points and aspects of SR are illustrated by examples drawn from Saguaro and similar systems. A synopsis of the language is contained in the Appendix; the full language is described in (Andrews and Olsson 1985).

2 Program structure

Every programming language provides mechanisms for specifying solutions to problems from a specific domain. Although the kinds of mechanisms needed are dictated by the problem domain, there are usually many ways in which the mechanisms could be realized. Above all, a language should be *expressive:* It should be possible to solve relevant problems in a straightforward way. One way to make a language expressive is to include a plethora of mechanisms. However, if a language is to be attractive, it must be *simple* to understand and use, and *efficient* to compile and execute (Hoare 1973). These criteria suggest that a language contain a small number of efficiently implementable mechanisms. The tradeoff for the language designer is to meet all three objectives simultaneously.

Our problem domain is distributed programming. A *distributed program* contains multiple processes that execute concurrently and interact by exchanging messages. Consequently, such a program can be executed on a *network computer:* a collection of processing nodes connected by a communications network. Our specific concern is writing systems programs, such as operating systems, that control a network computer. For this problem domain, it is imperative that there is a close match between the language mechanisms and the underlying hardware architecture.

The structure of a distributed program can be viewed as a graph with nodes corresponding to one or more processes and arcs corresponding to communication channels. In this section we consider how the nodes and arcs are specified and how a program graph is generated. In Sect. 3 we examine the internal structure of the nodes and how they access the arcs.

2.1 Component structure

Resources and processes

The first issue in the design of any programming language is deciding what the fundamental component is to be; i.e., the main unit from which programs are constructed. Two possibilities exist in sequential programming languages: procedures or modules containing procedures. Two similar possibilities exist in distributed programming languages: processes or modules containing processes.

Processes are the fundamental component in many distributed programming languages (e.g., CSP (Hoare 1978), NIL (Parr and Strom 1983; Strom and Yemini 1983), and Concurrent C (Gehani and Roome 1985)). This choice is perfectly adequate for simple components such as text-processing filters. Also, more complex components can always be implemented by collections of interacting processes. However, we believe that a better choice for the fundamental component is a construct that contains processes; below we explain why.

The fundamental component in SR is the *resource,* just as in SR_0. Each resource has a specification part, which defines the *operations* provided by the resource, and an implementation part, which gives the implementation of the resource. A resource is implemented by one or more processes that execute on the same processor. Processes interact by means of operations, which are parameterized communication channels. Processes in the same resource may also share variables. We have retained this combination of mechanisms because we have found it to provide a very nice tradeoff between the three design objectives of expressive power, simplicity, and efficiency.

Although some components in distributed systems are filters, most, especially in system programs, are servers. In fact, *all* modules in Saguaro are servers that manage objects such as files and

service client requests to access these objects. Even Saguaro's command interpreters, which normally only initiate activity, also provide operations that are used to control command execution.

SR's resources provide a natural mechanism for programming server modules. For example, Saguaro contains several *file managers*. Among other things, these file managers handle requests to *open* files. Since opening a file is relatively time consuming, and separate *open* operations execute essentially independently, each *open* is handled by a separate process. This concurrency enhances the performance of file managers. To clients, however, a file manager is a single abstract object, and *open* is a single operation. Such an abstract object can be programmed in SR as a single resource containing multiple processes. In addition, the fact that a different process handles each *open* is transparent to a file manager's clients. Thus, if we ever change how *open* is implemented, we will not have to change resources that invoke *open*, and hence will not have to recompile them.

Once a file has been opened, access to it is managed by a *file server* module. Each different file is handled by a separate file server. Within each file server there is one process for each client that has opened the same file. This process services file-access requests from its client and keeps track of client-specific information such as the the next record to be accessed. The file-access processes within a file server share file descriptor information so that all see the current status of the file and the contents of buffered file blocks. A separate process is used for each client since this simplifies the implementation of file servers by partitioning client-specific data into distinct processes. Again, SR's resource construct provides a natural mechanism for programming file servers.

Most other Saguaro modules also contain multiple processes. Sometimes these processes are used to enhance concurrency, as in file managers. Sometimes they are used to manage independent tasks, as in file servers. Sometimes multiple processes are used to perform background tasks, such as writing blocks to disk. Because resources can contain multiple processes, each module can be implemented by a single resource. Moreover, a resource's clients see one abstract object and need not be concerned with how it is implemented.

Shared variables

It is tempting in a distributed programming language to prohibit shared variables since this mirrors the absence of shared memory in network computers. It also avoids the need to synchronize access to shared variables. However, the reality, especially in systems programs, is that shared variables facilitate efficient solutions to many problems. For example, processes in Saguaro's file servers share file descriptors and buffers as mentioned above. Also processes in disk server modules (resources) share disk buffers. In each case, however, only processes in the same server, executing on the same processor, share variables. Thus SR allows shared variables within a resource but not between resources. This encapsulates the use of such variables and makes access to them efficient.

If shared variables were not allowed within SR's resources, common data would either have to be managed by a separate caretaker process, or all clients that needed access to common data would have to be serviced by the same process. The first alternative is inefficient since it requires exchanging messages with the caretaker. The second alternative leads to a complex server process since it would have to multiplex activities of several clients. Ideally, each process should manage *one* activity. Sometimes shared variables are needed to make this both feasible and efficient.

To summarize, we have found SR's resource construct to be an excellent fundamental component. Resources are expressive since they allow most server modules to be realized directly by a single program component.[1] They have a simple structure that is similar to that of modular constructs in procedure-based languages such as Euclid (Lampson et al. 1977) and Modula-2 (Wirth 1982). Finally, resources are efficient since processes that share variables are guaranteed to execute on the same processor. The fundamental component in several other distributed programming languages (Distributed Processes (Brinch Hansen 1978), StarMod (Cook 1980), Argus (Liskov and Scheifler 1983), and EPL (Black et al. 1978, 1984) has a similar structure and hence similar attributes.

2.2 Resource activation

The next key issue is how the main program components – resources – are activated. Here, SR differs from SR_0. In SR_0, programs were *static:* the number of instances of resources and processes

1 In some applications, such as network communication packages, logical modules span multiple machines. In such situations, a module would have to be implemented using multiple SR resources, one per processor. The decomposition into resources would in this case reflect the programmer's decision as to what executes on each processor

within resources was fixed and specified at compile time. This allowed the number of components in a program to be specified directly in the source program and allowed resources and operations to be referenced directly by their names. Also, it was possible to ensure that SR_0 programs would not exceed available memory (barring stack overflow). Finally, the static nature of SR_0 made it possible to optimize interactions between resources that were loaded on the same processor.

Despite these apparent advantages of a static language, SR is now *dynamic:* all resources and processes are created explicitly by executing statements. We have made the change because a dynamic language is much more expressive. Also, this additional expressiveness could be realized with minimal impact on the language's simplicity and efficiency. In the remainder of this section we describe why resources are now dynamic; the rationale for dynamic processes is described in Sect. 3.1.

SR resources are now parameterized patterns, instances of which are created and possibly destroyed as a program executes. This yields several benefits relative to SR_0's static resources. First, multiple, similar instances of a resource can be created from the same source-program component. For example, the source program for Saguaro contains one resource for each kind of server, yet Saguaro itself contains multiple instances of each kind of server. Second, it is possible to bring up and test different configurations of a system without recompiling. Third, existing resource patterns can be reused in new programs, perhaps after changing the types of parameters; thus, resources form a toolkit for program construction. Finally, when resources are dynamic, it is possible for a program to grow and shrink during execution. In Saguaro, many servers are created and destroyed dynamically in response to changing levels of user activity, hardware failures, and the addition of new processors.

The additional expressiveness of dynamic resources is realizable with a minimal increase in the complexity of the language, requiring only the addition of **create** and **destroy** statements. Because resource instances are created dynamically, they must be referenced indirectly, which is supported in SR by *capabilities*. Fortunately, capabilities were already in SR_0 because it was important to allow communication paths between resources to vary dynamically, even though resources themselves were static. For example, access to operations must be dynamic in order to support Unix-like I/O redirection. In a sense, SR is now simpler because all inter-resource operations are invoked using capabilities. Previously an inter-resource operation was invoked either using a capability or the fixed names of the resource and operation.

Dynamic resources are implemented for essentially the same cost as static resources. Even though SR_0 was a static language, its run-time kernel contained a primitive for creating resources (Andrews 1982).

For each source program, code was generated to invoke this primitive once for each resource in the program; this initialization code was executed at the start of program execution. We used this approach because it was far simpler to implement than modifying the link-editor to pre-allocate storage for resources and initialize the kernel's resource-descriptor table. To create resources dynamically, the same kernel primitive is used; the only difference is that the kernel now needs to check whether the target processor contains a template for the named resource. Supporting dynamic resources also caused us to add a kernel primitive for destroying resources. Finally, when a resource operation is invoked, we now have to make sure that the resource still exists; this check was unnecessary in SR_0. The only real efficiency loss resulting from the change to dynamic resources is that it is no longer possible to optimize at compile-time interactions between resources that reside on the same processor. However, this has a minor impact on overall performance since we are able to make intraprocessor interactions almost as efficient as before, and these interactions are inherently far more efficient than those between resources on different processors.

2.3 Examples of SR components

In this section, we illustrate the actual structure of an SR program by outlining parts of resources for a few modules of a simplified version of the Saguaro file system. Files in Saguaro, like those in Unix, include ordinary data files, devices such as terminals and disks, and pipes. 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*. A client accesses a file by using a file descriptor, which contains capabilities for operations supported by the file. These file specifications, which are common to the different kinds of files, can be declared in SR using a **global** component.[2]

2 A **global** component serves a role analogous to a '.h' file in a C program

```
global File
    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 = ()
    type File_Desc = rec( read : cap Read; write : cap Write;
                          seek : cap Seek; close : cap Close)
    ...  # other global file declarations
end
```

The **optype** declarations define the types of parameters and return values for the various file operations. Parameters to operations are passed by value (the default), result (**res**), or value/result (**var**). The declaration of *File_Desc* defines a file descriptor to be a record that contains capabilities for the various types of file operations. Each field of *File_Desc* can be bound to any instance of an operation of the specified **optype**, as described in the next section. This allows the kind of file and how it is implemented to be hidden from a client.

The remainder of the Saguaro file system consists of resources that declare patterns for servers. A resource has an *interface part* and an *implementation part*. The general forms of these parts are:

```
spec identifier                # interface part
    import specifications
    operation, constant, and type declarations
resource identifier(formals) separate

body identifier                # implementation part
    declarations
    initial declarations; statements end
    processes
    final declarations; statements end
end
```

Normally, these parts are separately compiled. They may also be combined by omitting the keywords **separate** and **body** and the occurrence of the resource identifier following **body**. Many of the components within the interface and implementation parts are optional; also these components can be declared in any order.

The interface part (**spec**) of a resource names other components the resource uses; declares the operations, constants, and types provided by the resource; and specifies the types of resource parameters. All objects declared in the interface part are exported from the resource. Note that resource parameters are declared in the interface part since they need to be visible to other resources that create instances of the one being declared.[3]

3 Resource parameters are declared at the end of the interface part so that objects exported from the resource are immutable; e.g., an exported type cannot define an array whose size depends on the value of a parameter

The body of a resource contains the processes that implement the resource, declarations of objects shared by those processes, and (optional) initialization and finalization code. The shared objects can include constants, types, variables, and operations. None of the objects declared in the body are visible outside the resource.

Note that resources cannot be nested. Another virtue of being able to create resources dynamically is that nesting is not needed; if one resource is implemented using another, it can either create the instance it needs or be passed a capability to it as a parameter. Precluding nesting also makes programs easier to understand and makes the compiler and run-time support simpler.

To continue with our example, the Saguaro file system contains one kind of server for each kind of file. A client acquires access to a file by calling the *open* operation, which is implemented by file managers. *Open* returns a file descriptor that contains capabilities for operations on the opened file; a new instance of the appropriate kind of server is created the first time a file is opened. File managers also provide a *close* operation that is invoked by a server each time the file managed by that server is closed; the last time a file is closed, the file manager destroys the server. Finally, file managers provide file-directory operations. Thus, file managers have the following specification:

```
spec FileManager
    import File, FileServer, TerminalServer, ...
    op open(path_name[0:*] : char; ...) returns fd : File.File_Desc
    op close(...)
    ...      # directory operations
resource FileManager(table sizes) separate
```

Note that *FileManager* imports the global declarations in *File*, and uses the file descriptor type *File_Desc*. *FileManager* also imports the declarations exported by *FileServer*, *TerminalServer*, and other resources such as disk servers that implement file system modules.

Ordinary data files are serviced by instances of the *FileServer* resource. These instances provide a special open operation, *fsopen*, which is called by a *FileManager* each time a file handled by the *FileServer* instance is opened. *Fsopen* returns a file descriptor containing capabilities for the file-access operations provided by the server. *FileServer* resources thus have the following outline:

```
spec FileServer
    import File, FileManager, DiskServer
    op fsopen(...) returns fd : File.File_Desc
resource FileServer(fm : cap FileManager; disk : cap DiskServer; ...)
    local declarations
    processes to service file-access operations
end
```

Here, the specification and implementation part are combined; this is permissible if the specifications of imported components have been previously compiled. Also note that instances of *FileServer* have several parameters. When created, a *FileServer* is passed a capability for the *FileManager* that created it (so the server will know which *FileManager's close* operation to invoke), a capability for the *DiskServer* that manages the disk on which the file resides, and other arguments describing the file.

Other kinds of files are serviced by instances of resources similar to *FileServer*. For example, *TerminalServer* resources service interactive terminals. This resource has a specification similar to that of a *FileServer*. The differences are that a *TerminalServer's tsopen* operation is parameterized differently than *fsopen*, and instances of *TerminalServer* have different parameters than instances of *FileServer*.

2.4 Resource creation and binding

Each instance of a resource is located on one network node; different instances of the same resource pattern can be located on different nodes. An instance is created by executing

cap_var:= **create** resource_identifier(actuals)
on machine

Execution of **create** causes a new instance of the named resource to be created on the indicated machine; the default location is the machine on which **create** is executed.[4] A capability for the new instance is returned by **create** and assigned to *cap_var*, which must have type "**cap** resource_identifier". For example, an instance of *FileServer* can be created on the same machine as the creator by executing

fsc:= **create** *FileServer*(...)

where *fsc* is declared as

var *fsc*: **cap** *FileServer*

Arguments are passed by value to the new instance; these are typically used to pass capabilities for other resources and to indicate the sizes of arrays in the resource body, as was illustrated in the previous section. Execution of **create** terminates once the initialization code in the created resource terminates.[5]

The capability returned by **create** can be used to invoke the operations exported by the newly created resource instance or to destroy the instance (see below). For example, *fsc* above provides access to the *fsopen* operation exported by the created instance of *FileServer*. Copies of capabilities can also be passed to other resources. This provides the means by which instances of resources acquire channels to each other. Note that clients of a resource need not be aware where it is located; all they need is a capability for the resource.

In addition to capabilities for entire resources, SR supports capabilities for individual operations. The *File_Desc* type declared in *File* illustrates this. A capability for a specific type of operation can be bound to any instance of that operation. This is used within file managers, for example, to construct file descriptors that are returned to clients who open files.

The initial value of any capability variable is **null**; an error results if a process invokes an operation using such a capability. Capability variables can also be assigned the value **noop**; in this case, using the capability to invoke an operation has no effect. For example, the *seek* field of a file descriptor for a terminal server can be set to **noop** since terminal servers do not provide *seek*.

Since resources are dynamically created, there must be some way to get rid of instances when they are no longer needed. An instance is destroyed by executing

destroy *cap_var*

where *cap_var* contains a capability for the instance. The effect of executing **destroy** is to terminate all activity in the resource (if any) and then to execute the resource's finalization code. The **destroy** statement terminates when the finalization code completes. In Saguaro, we use **destroy** to delete instances of resources, such as file servers, once they have finished providing their service.

Each SR program is assumed to contain one resource named 'main'. Execution of the program begins with the creation of one instance of this resource. It creates other resource instances, which can in turn create yet more depending on the application. Naturally, resource patterns must have been previously loaded on each machine on which they might be created. (This is specified as input

4 The possible values for machine are contained in a predefined, implementation-dependent enumeration type

5 All statements, such as **create**, that control resources or invoke resource operations may terminate prematurely due to a processor or network failure. The way failures are handled in SR is described in Sect. 3.5

to the SR link-editor.) Note that our approach to program activation makes it possible for the programmer to control the order in which resources get instantiated. This is useful, for example, to ensure that the resources used by another resource all exist and have been properly initialized before any of their operations are invoked. We have found being able to control instantiation order to be yet another advantage of dynamic resource creation.

3 Communication and synchronization

Resources and operations are the fundamental mechanisms provided in SR. Resources are patterns for objects; operations are patterns for actions on objects. As noted, resources contain one or more processes that potentially share variables. We now turn attention to how these processes are created, how they invoke and service operations, how they synchronize access to shared variables, and how they handle processor and network failures.

3.1 Process structure and message-passing primitives

The number of processes per resource could be *static,* or at least bounded at compile time. In this case, processes would be implicitly created when the containing module was created. This approach is taken in CSP (Hoare 1978) and Ada (1983). Alternatively, the number of processes could be *dynamic,* with new processes created as needed. Several languages provide dynamic processes: DP (Brinch Hansen 1978), Mesa (Mitchell et al. 1979), StarMod (Cook 1980), and Argus (Liskov and Scheifler 1983). A hybrid approach is also possible. In this case, some processes, such as those performing background tasks, are created implicitly and others are created explicitly. We take the hybrid approach in SR, for reasons described below.

Processes in different resources exchange messages to interact. Message passing can be provided by **send/receive** primitives or by remote procedure call. With **send/receive** primitives, data values flow in one direction between a sending and a receiving process. The sender of a message either continues immediately after issuing the send as in PLITS (Feldman 1979), or delays until the message has been received as in CSP (Hoare 1978). In the former case, send is a *non-blocking* primitive, so message passing is asynchronous. In the latter case, send is a *blocking* primitive, so message passing is synchronous. Receive is invariably a blocking primitive, although a non-blocking variant is often provided.

With *remote procedure call,* data can flow in two directions: the process that calls an operation sends data to a process that services the call, then waits for the results of the call (if any) to be returned. Thus, remote procedure call provides a structured equivalent of the **send/receive** and **receive/send** sequences used by clients and servers. The server side of a remote procedure call can be provided by a process that is dynamically created to service the call, as in DP or Argus. Alternatively, an already existing process could engage in a *rendezvous* with the caller, as is done in Ada. StarMod supports both possibilities.

Different languages include different combinations of these choices for process structure and message passing. The question is: What combination is best? Theoretically, the question is moot since it is *possible* to solve any problem using any combination of these mechanisms. However, different combinations are better suited to solving some problems than others. The ideal combination is one that allows each problem to be solved in the most straightforward and efficient manner possible. More precisely, it should not be necessary to introduce superfluous processes or work queues, or to obfuscate the interface to a module merely to overcome the limitations imposed by a particular choice.

The question of which combination of choices is best has been addressed at length by Liskov et al. (1986). Their observations are:

1. **Send/receive** is viable with either static or dynamic process structure. However, the client/server relationship is the dominant one in most distributed programs. Since clients almost always require answers to their service requests, remote procedure call provides by far the most convenient, and also familiar, client interface.
2. The combination of rendezvous and static process structure is not sufficiently rich to be attractive by itself. In particular, there are two classes of problems that are difficult to solve with rendezvous: local delay and remote delay. *Local delay* occurs in a server when an object needed to service a request is not currently available. *Remote delay* occurs when a server, in processing a request, calls another server and encounters a delay. In both cases, it may be necessary for the server to honor other requests in order to remove the conditions that led to the delay. It is possible to cope with local delay as long as the rendezvous mechanism allows the choice of which operation to service next to be based on

attributes of pending operations, including their parameters. Rendezvous alone, however, is not capable of dealing effectively with remote delay since when such delay will occur cannot be predicted by the server.

For these reasons, Liskov et al. (1986) conclude that either rendezvous or static process structure has to be abandoned. This is a reasonable conclusion if only *one* combination of primitives is available. It is also a reasonable conclusion in Argus' application domain of transaction processing systems (Liskov and Herlihy 1983). However, rendezvous and static process structure are adequate, provided a language has a rich enough set of primitives. Moreover, we believe it is beneficial to have a variety of primitives, especially to program distributed operating systems such as Saguaro (Scott 1983). Our experience with SR_0 bears this out, as described below.

SR_0 has a static process structure. Operations are invoked by asynchronous **send** or by **call**; operations are serviced by a rendezvous mechanism called the **in** statement. The **in** statement allows the choice of which operation to service to be based on the attributes of pending invocations, including their parameters. Thus, local delay can be handled in SR_0. Moreover, **send** can be used to avoid remote delay: instead of calling an operation that could delay, **send** to it and later use a rendezvous to receive the reply. This technique converts potential remote delays into local delays since the server waits only to receive new messages. It also obviates the need to create a process to service the remote operation.[6]

In addition to making it possible to avoid remote delay, **send** also has other uses. For example, it can be used to program pipelines of filter processes. A filter is a data transformer that consumes streams of input values, performs some computation based on those values, and produces streams of results. Filters are most simply and efficiently programmed using asynchronous **send** and **receive**. This is because it is never necessary to delay when a message is produced: no assurance is required that the message has been received, and no return value is needed. Producer delay can be avoided when synchronous **send** or remote procedure call is used, but this requires programming extra buffer processes. We use **send** within Saguaro for implementing pipelines of filters and for situations, such as writing windows or writing files, where it is not necessary to delay the invoking process. One additional use of **send** is described later in Sect. 3.4.

To summarize, we have found SR_0's combination of static process structure, rendezvous, **call** and **send** to be adequate. In fact, it is ideal for many problems. We have, however, decided to change from a static to a hybrid process structure. Thus, we now allow the programmer to choose whether a new process is created to service an operation, or whether it is serviced by performing a rendezvous with a previously created process. Our reasons for revising SR_0 in this regard are essentially the same as the reasons we changed to dynamic resource creation: dynamic processes provide maximum flexibility in how a resource body is programmed, and they could be implemented at modest cost by making minor alterations to the language kernel. We have also added a few other mechanisms that meet these same criteria. Among these are an early reply mechanism and a concurrent invocation primitive. All mechanisms, however, are merely variations on a small set of basic concepts. This keeps the language relatively simple, as we shall now show.

3.2 Basic SR communication primitives

SR, like SR_0, provides two basic ways to invoke operations – **call** and **send** – and allows operations to be serviced by **in** statements. SR also allows operations to be serviced by **proc**'s, which combine aspects of procedures and processes. In the four possible combinations, these primitives provide local and remote procedure calls, dynamic processes, rendezvous, and asynchronous message passing.

The invocation statements have the forms:

call operation_denotation(actuals)
send operation_denotation(actuals)

An operation denotation names a specific operation in a specific instance of a resource that declares that operation. A capability is usually used. (A resource instance can denote its own operations directly by their names.) For example, if *cap_var* contains a copy of a resource capability, and *op_id* is the name of one of the operations exported by that resource, that operation can be called by executing

call *cap_var.op_id*(actuals)

A **call** invocation terminates when the operation has been serviced and results have been re-

6 Note that **send** must be asynchronous or else the sender would still encounter delay until the message is received. A message can be viewed as being a lightweight process that delivers itself

turned.[7] The use of the keyword **call** is optional in the **call** statement; it is always omitted in function calls, which return values. A **send** invocation terminates when the arguments have been delivered to the machine on which the resource declaring the denoted operation resides. This makes the semantics of **send** semi-synchronous rather than totally asynchronous (Bernstein 1984). We have chosen this semantics because the sender is assured that there was adequate buffer space for the message and that the servicing resource existed and was reachable at the time of the invocation. The programmer still has no assurance, however, that the message *will* be processed.

By default, an operation may only be called. This can also be specified explicitly by appending the *operation restriction* "{**call**}" to the declaration of the operation. To indicate that an operation is to be invoked only by **send,** the operation restriction "{**send**}" is used. A programmer can also specify that an operation can be invoked by either **call** or **send** by using the operation restriction "{**call, send**}".[8]

An operation is serviced either by a **proc** or by **in** statements. A **proc** is declared like a procedure and sometimes executed as a procedure but has the semantics of a process. The form of a **proc** is:

```
proc operation_identifier(formal_identifiers) returns result_identifier
    declarations
    statements
end
```

The operation identifier is the same as that of some operation declared in the module containing the **proc.** The formal and optional result identifiers are local to the **proc.** They are used to access the arguments of the invocation and to construct the result; these identifiers have the types declared in the **op** declaration. These will usually be the same identifiers as those in the corresponding **op** declaration, but need not be.

An instance of a **proc** is created whenever the associated operation is invoked. This instance executes as a process. If the **proc** was invoked by **send,** the instance executes concurrently with the invoker. If the **proc** was invoked by **call,** the caller waits for the **proc** to terminate or to execute an early reply (see Sect. 3.3). A **proc** can call itself. The effect is essentially the same as a recursive procedure call. Our compiler optimizes the common case of a **proc** that does not execute an early reply and that is called from within the module in which it is declared; this is implemented as a conventional procedure call since the semantics are the same.

The other way to service operations is by means of **in** statements, which are essentially the same as in SR_0. The form of an **in** statement is:

```
in operation_command □...
□ operation_command ni
```

Each operation command is like a **proc,** except it can also contain synchronization and scheduling expressions. Its general form is:

```
operation_identifier(formal_identifiers)
  returns result_identifier
  and Boolean_expression
  by arithmetic_expression⟶
    declarations
    statements
```

Within an **in** statement, an invocation is *selectable* if the associated Boolean expression is true or omitted (the Boolean expression can reference invocation arguments). An **in** statement delays until there is some selectable invocation. Then, one such invocation is selected and the corresponding block is executed. An **in** statement terminates when that block terminates; if the operation was called, the **call** also terminates at this time. Selectable invocations are serviced in order of arrival at the servicing resource. 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 expression following **by.**

A simple example will help clarify the four possibilities provided by **call, send, proc,** and **in**. Below is an outline of the body of the *FileManager* resource, whose specification was given in Sect. 2.3.

```
body FileManager
  op count_manager() {send}
  op local_open(...) returns fsc : cap FileServer
  shared variables
  proc open(pn, ...) returns fd      # externally visible open
    var fsc : cap FileServer
    ...
    # search path pn to find file location, size, etc.
    ...
    fsc := local_open(...)           # get capability for a FileServer
    fd := fsc.fsopen(...)            # open the FileServer; get back a file descriptor
  end
```

7 Invocations can terminate prematurely due to a failure, as discussed in Sect. 3.5

8 We have found the ability to be able to use both **call** and **send** to invoke the same operation to be useful in Saguaro. For example, in the window manager at times we want to ensure that a screen update is completed, but other times it is sufficient that it be done eventually. A similar situation occurs with writing blocks to disks: at times we want to ensure they have been written, at other times it is sufficient that they merely be buffered

```
    proc count_manager()
        local variables for reference counts for open files
            and capabilities for FileServer resources
        ...
        in local_open(...) returns fsc →
                # increment reference count for file; create FileServer if necessary
                # return capability for FileServer
        [] close(...) →
                # decrement reference count; destroy FileServer if necessary
        ni
        ...
    end
    initial
        initialize shared variables
        send count_manager()    # fork one instance of count_manager
    end
end
```

Recall that *FileManager* exports two operations: *open* and *close*. Since *open* is implemented by a **proc,** a new instance of this **proc** is created each time *open* is called. This allows opens to be processed concurrently, except when they access the *FileManager's* shared variables within *local_open*. In contrast, *close* is serviced by an **in** statement within **proc** *count_manager*. One instance of this **proc** is created when *FileManager* is initialized; it repeatedly services calls of *close* as well as *local_open*. Thus, invocations of *local_open* and *close* execute with mutual exclusion, which ensures that reference counts are accurate. Note that neither *local_open* nor *count_manager* are exported from *FileManager*. Also note the operation restriction "{**send**}" on the declaration of *count_manager*.

Resources often contain "worker" processes such as *count_manager* above. Often, only one instance of such a process is needed and it is not parameterized. In this case, the process can be declared as

process *worker*
  declarations
  statements
end

and the declaration "**op** *worker*()" and initialization statement "**send** *worker*()" can be omitted. One instance of such a process is implicitly created after other initialization statements have been executed. Note, however, that this is merely a useful abbreviation for the above pattern; processes can always be created explicitly when needed.

3.3 Additional communication primitives

SR includes several primitives in addition to the four basic ones (**call, send, proc,** and **in**). All open up the underlying implementation of the basic primitives to provide additional expressiveness with minimal impact on the simplicity and efficiency of the language. For example,

receive *operation*(*v1, v2, ..., vN*)

waits for the next invocation of *operation* and then assigns the arguments to variables *v1, ..., vN*. It is merely an abbreviation for a common special case of the **in** statement. Together with the **send** form of invocation, **receive** supports asynchronous message passing in a familiar way. Note that **receive** can also be used to service **call** invocations, assuming the serviced operation has the appropriate restriction, and that it does not have result parameters or a return value. In this case, the effect is the same as synchronous message passing.

A **proc** usually terminates by reaching the end of its statement list. Similarly, an **in** statement usually terminates by reaching the end of the selected block. The **return** statement provides a mechanism for forcing early termination. In particular, executing

return

causes the smallest enclosing **in** statement or **proc** to terminate; if the invocation being serviced was called, the corresponding **call** statement also terminates at this time.

The **reply** statement provides additional flexibility in servicing **call** invocations. Execution of

reply

causes the **call** invocation being serviced in the smallest enclosing **proc** or **in** statement to terminate early.[9] In contrast to **return,** however, the executing process continues after executing **reply.**

An important use of **reply** is to program *conversations* in which a client and server engage in a repeated exchange of messages. For example, in Saguaro a client accessing a file engages in a conversation with a file server process. In particular, the *fsopen* operation exported from *FileServer* is programmed as follows:

```
proc fsopen(...) returns fd
    op read  File.Read
    op write  File.Write
    op seek  File.Seek
    op close  File.Close
    ...         # other declarations and initialization
    fd.read := read; fd.write := write; fd.seek := seek; fd.close := close
    reply
    do true →
```

9 Executing reply has no effect on send invocations

```
        in read(...) → ...
        [] write(...) → ...
        [] seek(...) → ...
        [] close(...) → exit
        ni
    od
    ...        # finalization, including calling close in a FileManager
end
```

Here, a new process is created to service each invocation of *fsopen*. This process declares instances of the different file-access operations. Capabilities for these operations are assigned to the fields of *fd;* then **reply** is used to return the file descriptor to the *FileManager* that called *fsopen*. After replying, the process services the file-access operations until the file is closed. Note that the fact that **reply** is used is transparent to the caller of *fsopen*. Also note that here the parameterization of each file-access operation comes from a previously specified **optype** declaration. Finally, note that the operation declarations are local to *fsopen;* consequently, a new set of operations is created each time an instance of *fsopen* is called. This provides each file server client with its own communication channels.

It is also convenient in distributed programs to be able to invoke several operations concurrently since this can be much more efficient than invoking them sequentially. Often it is desirable to perform a *multicast* in which several operations are passed the same arguments; e.g., to update all copies of a replicated file. Other times it is desirable to pass different arguments to different operations; e.g., to read several different records from a distributed database. In both these cases, it is appropriate for the invoker to wait for all the operations to be serviced before proceeding. This is not always the case, however. For example, when reading from a replicated file, it may only be necessary to wait for one of the reads to complete. Or, if weighted voting is used with a replicated file (Gifford 1979), just some subset of the replicas need to be reached on each read and write.

SR provides a concurrent invocation statement that is flexible enough to support all the above possibilities.[10] This statement has the form

co concurrent command []...
[] concurrent command **oc**

Each concurrent command has the form

invocation statement [⟶post-processing block]

where the invocation statement is either a **call** statement or a function call, and the post-processing block is optional. A **co** statement is executed by first starting all invocations. Then, as each invocation terminates, its post-processing block if any is executed; post-processing blocks are executed atomically one at a time. Execution of **co** terminates when some post-processing block executes **exit,** or when all invocations and post-processing blocks have terminated. If a **co** statement terminates before all invocations have terminated, uncompleted invocations are not terminated prematurely, however. This is 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 command can also be preceded by a quantifier, which provides a compact notation for specifying multiple commands. 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 containing capabilities for each file resource. Reading a replicated file, terminating when one copy returns an OK status or all copies have been queried, can be programmed as

```
co (i:=1 to N) status:=file[i].read(arguments) ⟶
      if status=OK ⟶ exit fi
oc
```

Note that all invocations assign to the same *status* variable; this causes no problem here since return values and result arguments are copied atomically back into the invoking process as each invocation terminates. Invocations are started in parallel, but post-processing blocks are executed one at a time. We considered adding two additional synchronization primitives to SR. The first, **defer,** would be used within **in** to defer processing of an operation. This would allow invocations to be replied to in a different order than they were selected. The second primitive, **forward,** would allow an invocation being serviced to be partially processed and then passed to another operation. This would forward responsibility to reply to the original invocation. We did not add these primitives for three reasons. First, we have found very few situations that require either of them. The powerful selection mechanisms incorporated into **in** make **defer** rarely needed, and **forward** never

10 The statement is similar to the **coenter** statement in Argus (Liskov and Scheifler 1983)

has had many uses (Gentleman 1981). Second, both are somewhat complex primitives, especially **defer.** The arguments of a deferred operation must remain accessible outside the conventional scope of an **in** statement, and there is no general way to ensure that a deferred operation is ever replied to, or not replied to more than once. With **forward,** an operation can only be forwarded to another operation having compatible parameters. Third, both primitives can be simulated using **send** and either **receive** or **in**. While the same is true of **reply** and **co**, those primitives are both useful and simple; thus we have made the tradeoff to include them.

3.4 Synchronizing access to shared variables

Since processes in a resource can share variables, they need to be able to synchronize access to them. Consequently there must be some mechanism in the language to support this. There are several possible approaches.

The simplest approach – at least from the standpoint of requiring no additional language constructs – is to employ a distinct process for shared variable synchronization. Whenever another process requires synchronized access to the shared variables, it sends a request to this synchronizer process. The synchronizer then either performs the access itself, or grants access permission to the other process. Especially if a language contains a rendezvous mechanism with selection control as powerful as SR's, the synchronizer and the interface to it are easy to program. This was the approach taken in SR_0. Unfortunately, the overhead of exchanging messages with the synchronizer outweighs most of the performance gain that can result from the use of shared variables. Special cases could be recognized and optimized, but these cases are hard to detect, especially when processes can be created dynamically.

At the opposite extreme, shared variable synchronization could be provided implicitly by means of atomic data types and atomic actions upon them, as done in Argus (Liskov and Scheifler 1983). However, support for this approach requires a fairly substantial operating system. Hence, this approach is inappropriate for a language intended for writing operating systems.

A variety of intermediate-level solutions are possible: semaphores, conditional critical regions, and monitors (Andrews 1983). These all add additional mechanisms to the language. However, how to use these mechanisms is fairly well understood. Also, semaphores and monitors can be implemented very efficiently, which is important if the potential efficiency provided by shared variables is to be realized. StarMod provides semaphores as a basic data type (Cook 1980); Ada also provides semaphores by means of a pre-defined package. EPL provides monitors since it is based on Concurrent Euclid (Holt 1983).

The choice between semaphores and monitors is a difficult one. Monitors enable shared variables to be encapsulated with the operations on them and make mutual exclusion of these operations automatic. Semaphores require that mutual exclusion be explicitly programmed but provide more flexibility exactly because mutual exclusion is not automatic. Since we have found that one of the primary used of shared variables is for tables that are mostly read, often concurrently, we prefer using semaphores. Moreover, the encapsulation provided by monitors is of marginal benefit since the shared variables and processes that access them are already encapsulated by the resource construct. The final – in our case compelling – argument in favor of semphores is that the communications primitives in SR *already* support them, as we show below. Thus, we can provide semaphores without having to add another construct to the language.

Operations in SR can be serviced by more than one process. Together with **send** and **receive,** shared operations can be used to program semaphore-like objects. A semaphore can be simulated in SR by an operation declared as:

op *semaphore*( ) {**send**}

The semaphore **P** and **V** operations can then be programmed as:

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

Such a semaphore is initialized by executing the appropriate number of **V** operations. Our compiler optimizes operations used in this way so they are implemented just as semaphores would be.

SR also supports arrays of operations, and these can be used to support arrays of semphores. Note, however, that the basic primitives used to implement semaphores can also be used to implement objects more general than semaphores. For example, we use "data-containing" semaphores to implement buffer pools in Saguaro's device servers. A buffer is produced by sending its address to an operation queue; a buffer is consumed by receiving its address from an operation queue. A similar approach can also be used to implement the admin-

istrator/worker pattern of process interaction (Gentleman 1981) in which an administrator process receives requests and passes them on to one of a fixed number of workers. This flexibility results from having a combination of well-integrated communication primitives.

3.5 *Coping with failures*

Programs that execute on network computers have the potential to be more robust than programs that execute on single processors. Robustness is, in fact, one of the primary goals of the Saguaro system, as it is for many other distributed operating systems. Our concern in Saguaro is being able to provide service despite hardware failures such as processor, network, and device crashes. Consequently, SR must provide mechanisms for detecting and coping with such failures.

Hardware failures can be handled in a distributed programming language in one of three general ways. One approach is to provide high-level language mechanisms that either hide the occurrence of failures or are used to recover from them. Examples of this approach are atomic actions (Liskov and Scheifler 1983), fault-tolerant actions (Schlichting and Schneider 1983), and replicated procedure calls (Cooper 1984). Each of these techniques is useful, especially for applications programs. However, each requires an extensive runtime kernel that is in essence a special-purpose operating system. Thus, they are too high-level for systems programming.

At the opposite extreme, a language could merely provide a time-out mechanism by which a process can avoid indefinite delay while waiting for an invocation to complete or to arrive. This approach is supported in Ada by the **delay** statement. SR_0 also employed this approach, but at an even lower-level than Ada. In particular, in SR_0 one had to program an interval timer and then use it to provide a delay facility. The SR_0 approach is flexible and requires no language kernel support, but it is relatively clumsy.

In SR we take an intermediate approach. Although (approximate) failure detection ultimately relies on the use of timeout, what the programmer requires is some way to ascertain that a failure has occurred. The abstract concept is thus that a component has failed. The kernel is charged with detecting the failure. The programmer is charged with handling the failure once it has been detected. For example, if one resource has detected that another has failed, it may be appropriate to create a new instance of the failed resource. Note that it is not possible, however, for a resource to recover itself since if it has failed, it ceases to execute.

In addition to failures, numerous other kinds of exceptions could occur in programs. Consequently, languages (e.g., CLU (Liskov et al. 1981), Mesa (Mitchell et al. 1979), and Ada) sometimes provide exception handling mechanisms. However, we agree with Black (Black 1982) that such mechanisms are unnecessary. Most existing systems are written in languages, such as C, that do not include exception handling mechanisms. While this does not mean that this is the best approach, the only exceptions that *should* occur in a correct program are ones that result from the finite limitations of the hardware on which a program executes. Anything else should either be precluded by careful coding or guarded against by explicit error checks. Time may prove that we are wrong about exception-handling mechanisms, but at present we prefer to be cautious until we have more experience. Caution also has the beneficial side-effect of keeping SR much simpler than it would otherwise be.

SR now provides two basic mechanisms to support handling failures that result from crashes or hardware limitations. First, invocation and resource control statements (**call, send, create,** and **destroy**) return an implicit completion status. This status is stored in the capability used in the statement; it can be examined after the statement terminates. The possible status values are:

Success	statement terminated normally
Crash	statement cannot finish due to processor or network failure
NoSpace	insufficient space to create resource or proc, or to store invocation
Terminated	resource or operation server no longer exists
Undefined	initial value and value while statement is executing

The second mechanism is a predefined Boolean function, *failed*. The argument to *failed* is either a capability or a machine name. When the argument is a capability, *failed* returns *true* if the resource or operation pointed to by the capability resides on a machine that has crashed or is unreachable. When the argument is a machine, *failed* returns *true* if the machine has crashed or is unreachable. A process can invoke *failed* at any time to test the status of a resource or machine. In addition, a process can use *failed* in **in** statements to avoid permanent delay waiting for an invocation to arrive. This is done by including an exception command of the form

failed(argument)⟶statements

in an **in** statement. This command is selectable when the value of *failed* is *true*. For example, we use exception commands in file server processes (see Sect. 3.3) so that a file gets closed if the client who opened the file fails.

4 Additional issues

The distinguishing attributes of a distributed programming language are the way programs are structured and the ways in which processes communicate and synchronize. However, the data types and sequential statements in a language also affect its utility. In addition, it is necessary to be able to communicate with input/output devices.

In this section, we describe interesting aspects of data types, sequential statements, and device interfaces. In redesigning these parts of SR, we have followed the same criteria – expressiveness, simplicity, and efficiency – that guided the design of other parts of the language. Moreover, we have strived for uniformity between the sequential and concurrent components of the language.

4.1 Data types

Like most recent languages, SR contains a variety of scalar and structured types. Most are conventional, with capabilities being the one that is basically unique to SR. Another somewhat uncommon attribute of SR is that all declarations can contain expressions that reference previously declared objects including variables and parameters. This results from uniformly applying the single scope rule: an identifier exists and may be used from its point of declaration to the end of the block in which it is declared or the end of any block into which it is imported. Thus, the size of an array or the value of a constant can be calculated during execution. We have found this to be very useful, without being expensive to implement, since storage must be allocated dynamically anyway.

Because pointers are often needed in systems programming, we have now added them to SR. Their use is restricted, however, to ensure that possible misuse is confined, and more importantly, that a pointer used on one machine cannot point to a location on a different machine. In particular, pointers can only be used within a resource; they may not appear in resource specifications. (There is one exception to this, which is described below in Sect. 4.3).

Unlike many languages, SR does not provide a special set of mechanisms for programming "abstract data types". Resources and procs are the only encapsulation mechanisms. We have found this to be sufficient for our purposes, especially since our compiler supports file inclusion and we allow different kinds of declarations to be intermixed. Thus, it is possible to simulate an abstract data type either by programming it as a resource or by programming its representation and operations and including them in resources that need them. Our approach does have disadvantages, though. If an abstract type is programmed as a resource, it can be inefficient since the resource encapsulating the type may be on a different machine than its invoker.[11] On the other hand, if an abstract type is simulated by including a file containing its definition, the type is not secure since its representation is accessible.

SR also does not provide a generic resource facility because we have found very little use for such a mechanism. For those situations where two resources vary only with respect to the types they import or export, we have found it sufficient to copy and edit an existing resource template. Moreover, by not including generic resources, both the language and its compiler are much simpler.

We have retained SR_0's array mechanisms since we have found them to be very convenient and flexible. For example,

$a[2:5] := a[1:4]$

shifts a four-element slice of a to the right one position. Slices can also be used to perform character string operations such as substring extraction. As another example,

$b[1:*] :=$ 'Saguaro'

assigns a character string to part of b. Here, the star means that the upper bound of b is unspecified and is instead to be calculated, at run-time if necessary, from the number of elements on the right hand side. This same mechanism is used to declare that the size of a formal parameter is to be determined from the size of its actual. In addition to being expressive and simple, array operations are efficient to implement, except in esoteric cases such as extracting a non-contiguous slice from a two-dimensional array.

4.2 Statements

SR_0 included a multiple assignment statement, of which single assignment is simply a special case. We have retained this since it is both convenient

11 Our implementation optimizes calls of procs that do not cross machine boundaries, but these are still not as efficient as conventional procedure calls

and has the same semantics as SR's parameter passing. We have also added two special assignment statements that are used to increment and decrement variables. Although these statements are certainly not necessary, they do make the language easier to use. Again, as with **process** declarations and the **receive** statement, we have opted for providing an abbreviation for a commonly occurring special case.

In SR_0, the only control statements were **if** and **do** statements essentially identical to Dijkstra's guarded commands. SR_0 employed these because they are syntactically similar to the **in** statement and have a similar non-deterministic semantics. (Note that in Ada, the **select** statement is nondeterministic, but control statements are not; thus, what should be similar constructs are quite dissimilar). Although pure guarded commands are certainly adequate to express any compution, there are three situations in which they result in awkward program structure.

First, to prevent an SR_0 **if** statement from aborting, it is often necessary to include "**else** $\longrightarrow$ **skip**" as the last guarded command.[12] Almost all the people who have written SR_0 programs have quickly (and vociferously) tired of having to do this. Therefore, we have changed so that executing **if** has no effect if no guard is true. This makes SR's **if** similar to the **if** statement in other languages. Moreover, this change does not violate the spirit of guarded commands since the programmer still has to consider the case in which no guard is true.

The second deficiency of having only guarded commands for control constructs is that iterative loops can be more complex than necessary. For example, to iterate through a one-dimensional array, the SR_0 programmer has to declare a variable to serve as the loop index, initialize the variable, and write a **do** statement that expresses the termination condition and updates the loop index. We have found that programmers often forgot one or more of these steps (often the update of the loop index); also, they quickly became as annoyed as when coding "**else**→**skip**" in **if** statements. Consequently, we have added the for-all statement to handle commonly occurring simple loops. For example,

fa *i*:=1 **to** 10 $\longrightarrow$ *putint(a[i])* **af**

prints *a*[1], ..., *a*[10]. The quantifier used to express iteration in the for-all statement has the same form as the (optional) quantifier in the **co** and **in** statements; this is another example of using similar constructs for similar purposes.

Finally, it is awkward in SR_0 to program early exits from loops or to go to the next loop iteration before reaching the end of the loop body. For example, to exit a loop prematurely, it is necessary to use a flag variable, which must be declared, initialized, set, and tested. To avoid this, we have added **exit** and **next** statements to SR. An **exit** statement is used to force early termination of the smallest enclosing **do** or for-all statement. For a **do** statement, execution of **next** causes the **do** statement's guards to be re-evaluated. For a for-all statement, execution of **next** causes the next combination of bound variable values, if any, to be assigned; if no values remain, the statement terminates. Including **exit** and **next** makes the programmer's job easier and makes programs more readable, yet does not destroy the structure provided by the iterative statements. We also allow **exit** and **next** to be used within post-processing blocks in the **co** statement since they are useful (an example was given in Sect. 3.3) and also since postprocessing in **co** is iterative in nature. Once again we have a mechanism that is simple, widely applicable, and efficient to implement.

To summarize, we have retained SR_0's integration between the sequential and concurrent statements, but have added several constructs that make the programmer's task easier. Moreover, all the additions are both simple and efficient. We believe the resulting language strikes a reasonable middle ground between parsimony and excessiveness.

4.3 Real resources

A systems programming language must contain mechanisms to control physical devices such as disks, terminals, and interval timers. We have found SR_0's real resources, which are a variant of resources motivated by Modula's device modules (Wirth 1977), to be effective for dealing with such problems and have therefore retained them. A real resource is structurally identical to a regular resource and can use all the same language mechanisms. In addition, variables and operations in a real resource can be placed "**at**" addresses. Such variables are used to read and write control and status registers; such operations are used to service interrupts that occur through the specified interrupt location. Another feature is that real resources can employ the **byte** data type. Variables of type **byte** are compatible with any variable having the

12 **else**, like otherwise in Ada, is the negation of the disjunction of the other guards

same size; they are used, for example, to program memory allocators and I/O transfers.

We have added two mechanisms to real resources to enhance expressiveness and efficiency. First, reference parameters are allowed in declarations of operations exported from real resources. Such parameters allow the programmer to pass the address of an I/O buffer, for example, instead of the buffer itself. Second, the pointer type is allowed in objects exported from real resources. To ensure that memory addresses do not cross machine boundaries, real resources that employ these mechanisms may only be accessed by resources executing on the same machine. Finally, interrupt operations may be serviced by either **in** statements or procs. We do not restrict how interrupt operations are serviced, but an interrupt operation that is serviced by a **proc** should be call-only to avoid the overhead of creating a process to service the interrupt. Also, an **in** statement that services interrupt operations should not contain complex synchronization or scheduling, again to minimize interrupt-handling overhead.

To support initial program development and testing, a version of the SR compiler runs on top of Unix. In this environment, real resources cannot be used for input/output. Instead, the Unix compiler provides pre-defined functions that allow access to the Unix file system.

5 Conclusions

The main ways SR has evolved are that resources and processes are now created dynamically, and that operations can be invoked and serviced in a variety of ways. In particular, local and remote procedure call, rendezvous, asynchronous message-passing, and semaphores are all supported by combining a small number of primitives in a variety of ways. Fault-tolerant programming is also now supported. Finally, the types and sequential statements provided by the language have been expanded to simplify the programmer's task.

The criteria that guided the redesign of SR are expressiveness, simplicity, and efficiency. Expressiveness has been our major concern since a language is of little use if it is difficult to solve a problem. However, many of our decisions have been tempered by the concerns for simplicity and efficiency. Thus, we have chosen to provide a variety of mechanisms, but only those that are based on a few underlying concepts and that are efficient to implement. For example, we have provided a variety of mechanisms for invoking and servicing the fundamental concept of operations, yet these are all simple to implement since they are merely variations on ways to enqueue and dequeue messages. In fact, the run-time support for the new SR is essentially the same size as that of SR_0. The compiler is, however, larger since the language contains more constructs.

Acknowledgements. Nick Buchholz, Mike Coffin, Irv Elshoff, Roger Hayes, Kelvin Nilsen, Titus Purdin, and Rick Schlichting have all been involved in the evolution of SR by suggesting changes and serving as a sounding board for our latest (greatest) ideas. Fred Schneider provided numerous helpful comments on earlier drafts of this paper. Every writer should be so fortunate to have such a thorough editor.

References

Reference Manual for the Ada Programming Language. January 1983. ANSI/MIL-STD-1815A

Andrews GR (1981) Synchronizing Resources. ACM Trans Program Lang Syst 3:405–430

Andrews GR (1982) The distributed programming language SR-mechanisms, design and implementation. Software Pract Exper 12:719–754

Andrews GR, Schneider FB (1983) Concepts and notations for concurrent programming. ACM Comput Surv 15:3–43

Andrews GR, Schlichting RD, Buchholz NC, Hayes R, Purdin T (1985) The Saguaro distributed operating system. TR 85-9, Dept of Computer Science, the University of Arizona

Andrews GR, Olsson RA (1985) Report on the distributed programming language SR. TR 85-23, Dept. of Computer Science, The University of Arizona

Bernstein AJ (1984) The semantics of timeout. TR 84/065, Dept of Computer Science, SUNY at Stony Brook

Black AP (1982) Exception handling: The case against. TR 82-01-02, Dept of Computer Science, The University of Washington

Black AP, Hutchinson N, McCord BC, Raj RK (1984) EPL programmer's guide. Eden project, Dept of Computer Science, University of Washington

Brinch Hansen P (1978) Distributed processes: a concurrent programming construct. Comm ACM 21:934–941

Cook R (1980) *Mod – a language for distributed programming. IEEE Trans Software Eng SE 6:563–571

Cooper EC (1984) Replicated procedure call. Proc 3rd ACM Symp on Principles of Distributed Computing, Vancouver, BC, pp 220–232

Feldman JA (1979) High level programming for distributed computing. Comm ACM 22:353–368

Gehani NH, Roome WD (1985) Concurrent C. AT&T Bell Laboratories Report

Gentleman WM (1981) Message passing between sequential processes: the reply primitive and the administrator concept. Software Pract Exper 11:453–466

Gifford DK (1979) Weighted voting for replicated data. Proc 7th Symposium on Operating Systems Principles, Pacific Grove, CA, 150–162

Hoare CAR (1973) Hints on programming language design. SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston

Hoare CAR (1978) Communicating sequential processes. Comm ACM 21:666–677

Holt RC (1983) Concurrent Euclid, the Unix system, and Tunis. Addison-Wesley

Lampson BW, Horning JJ, London RL, Mitchell JG, Popek GJ (1977) Report on the programming language Euclid. SIGPLAN Notices 12:1–79

Liskov B et al. (1981) CLU Reference Manual, Lecture Notes in Computer Science 114, Springer-Verlag, Berlin

Liskov B, Scheifler R (1983) Guardians and actions: linguistic support for robust, distributed programs. ACM Trans on Prog Lang and Systems 5:381–404

Liskov B, Herlihy M (1983) Issues in process and communications structure for distributed programs. Proc Third Symposium on Relability in Distributed Software and Database Systems, Clearwater Beach, Florida, pp 123–132

Liskov B, Herlihy M, Gilbert L (1986) Limitations of remote procedure call and static process structure for distributed computing. Proc 13th ACM Symp on Principles of Programming Languages, St. Petersburg, Florida, pp 150–159

Mitchell JG, Maybury W, Sweet R (1979) Mesa language manual, version 5.0. Rep CSL-79-3, Xerox Palo Alto Research Center

Parr FN, Strom RE (1983) NIL: A high-level language for distributed systems programming. IBM Systems Journal 22:111–127

Schlichting RD, Schneider FB (1983) Fail-stop processors: an approach to designing fault-tolerant computing systems. ACM Trans Comput Syst 1:222–238

Scott ML (1983) Messages vs. remote procedures is a false dichotomy. SIGPLAN Notices 18:57–62

Strom RE, Yemini S (1983) NIL: An integrated language and system for distributed programming. Research Report RC 9949, IBM Research Division

Wirth N (1977) Modula: a language for modular multiprogramming. Software Pract Exper 7:3–35

Wirth N (1982) Programming in Modula-2. Springer, New York

Appendix – Synopsis of the SR Language

Components

global	**global** identifier constants or types **end**

resource specification	[**spec** identifier
	  [**import** component_identifiers]
	  [constants, types, or operations]]
	resource identifier([parameters]) [**separate**]

resource body	[**body** identifier]
	  [declarations]
	  [**initial** block **end**]
	  procs
	  [**final** block **end**]
	end

proc	**proc** identifier([formal_identifiers])
	  [**returns** result_identifier]
	  block
	end

block	[declarations] statements

Declarations

constant	**const** identifier = expression
type	**type** identifier = type_specification
operation type	**optype** identifier = ([parameters]) [**returns** result]
variable	**var** identifier [subscripts], ... : type [:= expression, ...]
operation	**op** identifier [subscripts] ([parameters]) [**returns** result] **op** identifier [subscripts] optype_identifier

Statements

sequential	**skip** variable, ... := expression, ... variable ++ variable — **if** boolean_expression → block [] ... **fi** **do** boolean_expression → block [] ... **od** **fa** quantifier → block **af** **exit** **next**
operation invocation	[**call**] operation([actuals]) **send** operation([actuals]) **co** [(quantifier)] call_invocation [→ block] // ... **oc**

operation service	**in** [(quantifier)] operation([formal_identifiers])
	  [& boolean_expression] [**by** expression] →
	  block [] ... **ni**
	receive operation ([variables])
	return
	reply

resource control	capability := **create** resource([actuals]) [**on** machine] **destroy** capability

Optional items are enclosed in brackets. Plurals and ellipses are used to indicate one or more occurrences of items.