The Implementation of the SR Concurrent Programming Language

TR95-2

Robert Gebala and Carole McNamee

Department of Computer Science
California State University, Sacramento
Sacramento, CA 95819-6021

916-278-6834
916-278-5949 (fax)
gebalar, mcnamee@dromio.ecs.csus.edu

October 19, 1995

Abstract

Programming languages supporting distributed computation have emerged. These languages exploit the resources that are made available by networked and/or multiprocessor systems. While compilers for these languages are increasingly available, few documents describe the details of their implementation. This report details the implementation of one such distributed language, SR, including both its compiler, and its extensive runtime support system that provides the underlying support for SR's concurrency mechanisms.

Keywords: Concurrent programming languages, compilers, runtime support systems

1 Introduction

This report provides a detailed description of the implementation of the SR concurrent programming language [Andrews, Olsson]. The SR implementation is composed of three main components: the compiler (**sr**), the linker (**srl**), and the run-time support (RTS) library [Morgenstern]. This report describes the overall organization of the SR compiler and RTS.

The motivation for this presentation follows from the authors' efforts to implement an interpreter for the SR language. [Gebala] The details summarized in this report have been extracted from the source code provided with the standard SR distribution from the University of Arizona. [SR] The authors would have found such a summary helpful in the above mentioned effort and provide it for others who might find it useful. As such, the authors credit the many contributors to the SR project, most notably Greg Andrews at the University of Arizona, Ron Olsson at the University of California Davis, and Gregg Townsend at the University of Arizona.

2 The SR Compiler

An SR program consists of one or more resources or globals. These resources and globals are templates for run-time instances executing within one or more virtual machines, which in turn, reside on one or more physical machines. As a convention, should an SR program contain more than one resource the last one is taken to be the main resource. **sr** performs two-passes over the SR source code to produce the C code for each resource or global. None of the generated C source contains the function main. After generating the target C code, the C compiler is invoked to produce object codes.

In order to produce an executable module, **srl** links all the object code generated during the previous phase, the SR run-time support library, and the C run-time libraries. The SR RTS provides the main function expected of all C programs. System initialization and creation of the main resource are all performed by the RTS library's main.

For example, if an SR program a.sr contains resource b, **sr** will create a C program b.c. The C compiler then produces the object code b.o, which gets passed to **srl** to generate the executable a.out as shown in Figure 1.

2.1 SR Compilation Process

The **sr** parser explicitly builds a syntax tree (or parse tree) as the scanner recognizes tokens from an SR source program. An SR parse tree is a tree representation of an SR program; that is, **sr** uses a binary parse tree to internally represent its input. From this internal representation, the semantic analyzer can, say, verify if it is valid to send to an operation.

A parse tree node structure, as described in the **sr** source header file structs.h, contains the following members:

- *e_opr* is the node type. This may be a declaration, an expression, an item list, a statement list, an identifier, and so on.
- *e_sig* is the node signature used to represent, for instance, the type of value returned by a statement or declaration.

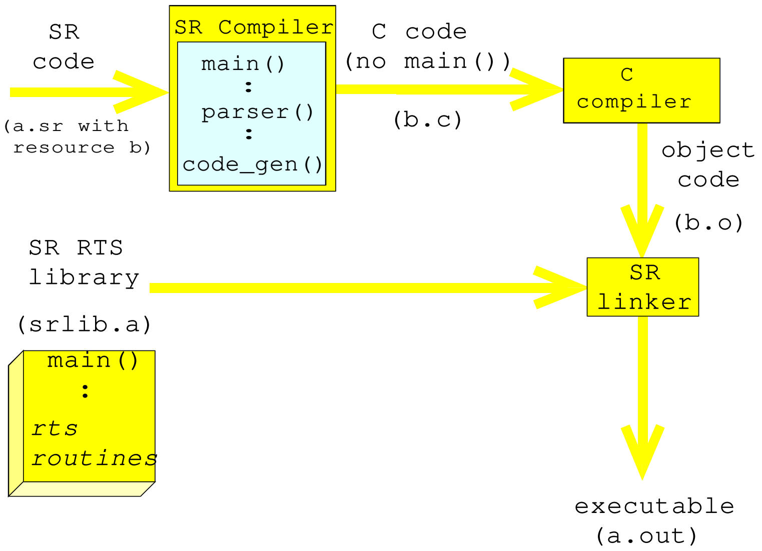

Figure 1: Generating an executable program from the SR compiler.

- *e_r* is the right child of this node.
- *e_locn* is the source line number that this node represents.
- *e_u* is a union containing information specific to a node type. This field may contain a pointer to the left child of this node, or a pointer to a integer/real/string value, or even a symbol table node. Symbol tables will be discussed in Section 2.2.

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

Figure 2: A very simple SR program, hello.sr.

The internal representation, or parse tree, for the program in Figure 2 is depicted in Figure 3. An SR component refers to either a resource or a global. In Figure 3, resource hello is represented by the right and left subtrees of the root node component. Resource hello contains no spec. A seq node is an organizational node representing a list of statements that will be executed sequentially. The (parse tree) representation of an SR statement is the right subtree of its seq node. A libcall node contains the name of the library routine and a parameter list for this routine. write is a predefined routine implemented in the RTS library.

C code is generated from the parse tree representation. This generated C code is stored in the Interfaces directory associated with every compilation. The code contains calls to the RTS, which includes modules for process management and interprocess communication. In addition, the RTS calls routines, through function pointers, which are declared in **sr**'s C code output. Two such functions are the initial and final routines associated with each resource.

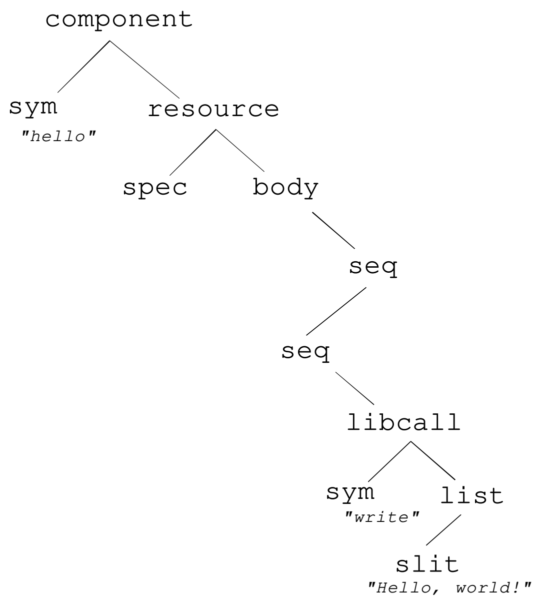

Figure 3: Parse tree for hello.sr.

2.2 Symbol and Attribute Table Management

The symbol table's structure is influenced in large part by the language's scoping rules. It serves as a central repository of information about program objects such as resource, global, operation, variables, constants, type definitions, etc. These data structures are described in detail in this section. Details are extracted from the **sr** source file structs.h.

SR 2.3 implements the symbol table as a tree of symbol nodes, each node of type *Symbol*. Identifiers are searched from the current block's symbols (leaf nodes) all the way up to its proper ancestors. Each symbol has at least one of the following attributes attached to it:

- *Var* represents user variables, temporaries, and constants
- *Signat* specifies a symbol's signature (or type)
- *Res* identifies resource and global
- *Op* maintains information on user-defined operations
- *Param* describes a parameter object
- *Proto* describes resource and operation prototyping
- *Recdata* provides information for symbols of type record
- *Class* provides a mechanism to identify operations belonging to the same class

Table 1: Structure of the Symbol node.

MemberName	*Information*
s_kind	symbol kind
s_depth	nesting depth
s_name	SR source name
s_gname	generated name
s_imp	ptr to symbol table of imported resource
s_next	forward link
s_prev	backward link
s_child	pointer to symbol node of an enclosed block
s_sig	pointer to signature
s_u	may be a pointer to: Var, Signat, Res, Op, Param, Proto, Recdata, Class

2.2.1 Symbol Structure

All program objects, such as variables, operations, blocks, parameters, and so on, are represented by a symbol node. Table 1 summarizes the symbol structure. Details are as follows:

- *s_kind* is the symbol kind. Table 2 tabulates the known symbol kinds and the use of each.
- *s_depth* identifies the scope's nesting depth. Predefined identifiers are at depth 0; a resource or global starts at depth 1.
- *s_name* is the name known by SR. Normally, it indicates the lexeme that is found in the source text. Some symbols have names which are assigned by **sr**.
- *s_gname* is the generated name, and is used when emitting the target C code.
- *s_imp* is a pointer to the symbol node of the resource from which the current symbol is imported.
- *s_next* and *s_prev* provide forward and backward links, respectively. Symbols on the same level and in the same block are maintained as a doubly-linked list.
- *s_child* is yet another pointer to a symbol node. This is used by K_IMP and K_BLOCK; the latter, to realize scoping rules.
- *s_sig* provides a pointer to the symbol's signature.
- *s_u* is a union which, depending on the symbol kind, may have different contents. Table 3 shows the possible information maintained in this union.

Table 2: The different kinds of symbols.

SymbolKind	*Purpose/Meaning*
K_BLOCK	scoping
K_VAR	name of variable or constant
K_FVAL	formal parameter, passed by value
K_FVAR	formal parameter, variable
K_FRES	formal parameter, result only
K_FREF	formal parameter, passed by reference
K_RES	resource
K_OP	user-defined operation
K_IMP	link to imported symbol
K_PREDEF	predefined function
K_FIELD	record or union field
K_TYPE	user-defined type
K_ELIT	enumeration literal

2.2.2 Var Structure

A *Var* structure is maintained for each variable. This includes named user variables, temporary variables generated by the compiler, record fields, and so on, as shown in Table 4. Details are as follows:

- *v_vty* defines the variable variety as listed in Table 5.
- *v_dcl* specifies how the object is declared as shown in Table 6.
- *v_sym* provides a pointer to the object's symbol node.
- *v_sig* provides a pointer to the variable's signature.
- *v_value* provides a pointer to a parse tree node. This field has a meaning only if the object is a compile-time constant.
- *v_seq* is a serial number used to generate names. This is used as the argument number or index in an operation's formal parameter list.
- *v_set* and *v_used* are flags used by the compiler to detect uninitialized and unreferenced variables.

Table 3: Information maintained in s_u field.

SymbolKind	*fieldname*	*Information*
K_RES	su_res	pointer to Res structure
K_OP	su_op	pointer to Op structure
K_VAR	su_var	pointer to Var structure
K_BLOCK	su_pb	pointer to parameter block name
K_PREDEF	su_pre	predefined function index
K_TYPE K_FIELD	su_lnk	link to another symbol node
K_ELIT K_FVAL K_FVAR K_FRES K_FREF	su_seq	enumeration literal or parameter number

Table 4: Structure of Var.

MemberName	*Information*
v_vty	variable variety
v_dcl	how declared
v_sym	symbol node pointer
v_sig	signature pointer
v_value	parse tree node
v_seq	serial number
v_set	initialized flag
v_used	referenced flag

2.2.3 Signat Structure

Each memory object's type is stored in the signature structure. The fields of *Signat* are shown in Table 7. Details are as follows:

- *g_type*, the entry type, is one of the following:
  1. T_VOID (no type applies)
  2. T_ANY (argument type for externals)
  3. T_NLIT (NULL/NOOP literals)
  4. T_BOOL
  5. T_CHAR
  6. T_INT
  7. T_ENUM
  8. T_REAL

Table 5: Variable varieties.

Variety	*Meaning*
V_GLOBAL	global variable
V_RPARAM	resource parameter
V_RVAR	resource variable
V_PARAM	normal procedure parameter
V_REFNCE	reference parameter
V_LOCAL	local variable
V_RMBR	record member/field

Table 6: Definition of the v_dcl field.

v_dcl	*Meaning*
O_VARDCL	declared as variable
O_CONDCL	declared constant
O_FLDDCL	declared as record member
O_PARDCL	declared as parameter

9. T_FILE
10. T_STRING
11. T_ARRAY
12. T_PTR
13. T_REC
14. T_VCAP (virtual machine capability)
15. T_RCAP (resource capability)
16. T_OCAP (operation capability)
17. T_GLB (global)
18. T_RES (resource)
19. T_OP (operation)

- *g_usig* is a pointer to the signature of the underlying type. For example, given an array A of record R, the *g_usig* of A is a pointer to the signature of R.
- *g_lb* and *g_ub* are pointers to expression nodes giving the lower and upper bounds, respectively, of array objects.
- *g_sym* is a pointer to a symbol node of a record member or argument list, or resource symbol.
- *g_rec* points to the record detail structure if the object is of record type.

Table 7: Structure of Signat.

MemberName	*Information*
g_type	signature type
g_usig	underlying type
g_lb	lower bound
g_ub	upper bound
g_sym	symbol node pointer
g_rec	record detail pointer

2.2.4 Res Structure

Resource and global details are maintained in a *Res* structure shown in Figure 8. The fields of this structure are as follows:

Table 8: Structure of Res.

MemberName	*Information*
r_opr	operator field
r_sym	symbol node pointer
r_sig	signature pointer
r_param	pointer to symbol node for parameters
r_abstract	abstract flag

- *r_opr* is an operator field, which is either O_RESOURCE or O_GLOBAL.
- *r_sym* is a pointer to the resource's or global's symbol node.
- *r_sig* is a pointer to the object's signature.
- *r_param* is a pointer to a symbol node for this object's parameters. This symbol node has type K_BLOCK, specifying a resource parameter block.
- *r_abstract* is a boolean field, indicating whether this is an abstract resource or global. An abstract resource has specifications but no body. It is intended to be extended by other resources. For more information on abstract resources, refer to [Andrews].

2.2.5 Op Structure

Information about an operation is stored in a structure called *Op*, whose structure is summarized in Table 9. This structure contains the following information:

- *o_sym* provides a pointer to the operation's symbol node.
- *o_asig* provides a pointer to its declared signature.

Table 9: Structure of Op.

MemberName	*Information*
o_sym	symbol node pointer
o_asig	pointer to declared signature
o_usig	pointer to underlying signature
o_seg	segment or location
o_impl	how implemented
o_class	class pointer
o_classmate	pointer to classmate
o_exported	export flag
o_dclsema	declared as semaphore?
o_nosema	nonzero if cannot implement as a semaphore
o_sepctx	needs separate context?

- *o_usig* provides a pointer to the operation's underlying signature.
- *o_seg* specifies where this operation resides. An operation may reside in one of the following segments:
  1. inside a proc (SG_PROC)
  2. declared at the resource level (SG_RESOURCE)
  3. imported from another resource (SG_IMPORT)
  4. imported from a global (SG_GLOBAL)
- *o_impl* indicates how the operation is implemented. From the perspective of the compiler, an operation is implemented or declared in one of five ways:
  1. declared as a proc (I_PROC)
  2. implemented by an input statement (I_IN)
  3. declared (or optimized) as a semaphore (I_SEM)
  4. declared as an operation capability (I_CAP)
  5. declared as external (I_EXTERNAL)
- *o_class* serves as a pointer to this operation's class if it is a member of one.
- *o_classmate* provides a link to another operation in the same class.
- *o_exported* indicates whether this operation is exported or not.
- *o_dclsema* specifies if the operation is indeed declared as a semaphore.
- *o_nosema* indicates if a semaphore implementation (or optimization) is not feasible.
- *o_sepctx* specifies if the operation requires a separate context. A proc requires a separate context if one of the following is true:

1. it uses the forward statement
2. it uses the reply statement
3. it invokes the predefined setpriority routine

2.2.6 Class Structure

Operations which are serviced by the same input statement belong to one class. Should an operation be serviced by more than one input statement, SR computes the transitive reflexive closure of the operations serviced by the same input statements to define a single class.

For example, in Figure 4, assume that operations a, b, and c are declared in the same resource. All three operations belong to the same class. Note that an operation may belong to exactly one class.

```
⋮
in a() →
      ⋮
[] b() →
      ⋮
ni
⋮
in b() →
      ⋮
[] c() →
      ⋮
ni
```

Figure 4: An SR program fragment with two input statements.

Table 10: Structure of Class.

MemberName	*Information*
c_num	class number
c_members	class cardinality
c_first	pointer to Op structure
c_seg	class segment

The *Class* structure provides a mechanism for the SR compiler to keep track of which operations belong to the same class. The information maintained by this structure is summarized in Table 10. Details are as follows:

- *c_num* is the class number. A class number of 0 indicates global.
- *c_members* defines the number of members in this class.

- *c_first* is a pointer to an *Op* structure. This represents the first member of this class. From the *Op* structure, other members can be tracked down via the *o_classmate* field.
- *c_seg* indicates this class' segment. This may be SG_GLOBAL, SG_RESOURCE, or SG_PROC.

2.2.7 Proto Structure

Operation and resource prototypes are represented internally by the SR compiler using the *Proto* structure. This structure is shown in Table 11, and is composed of the following fields:

Table 11: Structure of Proto.

MemberName	*Information*
p_rstr	restriction
p_param	pointer to parameters
p_def	typedef name

- *p_rstr* is generally a restriction operator field but may also be O_RESOURCE (for resource prototyping) or O_FINAL (for generating final code). As a restriction field, the values may be O_RNONE (no restriction, wherein call and send are allowed), O_RSEND (send only for operations declared as process), O_RCALL (call only for operations declared as procedure).
- *p_param* is a pointer to a *Param* structure described below. The first item in a parameter list, as it is implemented, is always the return value. This first *Param* may be a dummy structure for procs which do not return a result.
- *p_def* is the type definition name of the parameter structure.

2.2.8 Param Structure

Each entry in a parameter list is represented by a *Param* structure as shown in Table 12.

Table 12: Structure of Param.

MemberName	*Information*
m_name	parameter name
m_seq	position number
m_sig	pointer to signature
m_pass	how passed
m_next	pointer to next parameter

The fields comprising this structure are:

- *m_name* is the name of this parameter, if there is one.

- *m_seq* is a number identifying the ordinal position of this parameter in the parameter list. The return value is 0; the first parameter 1, and so on.
- *m_sig* is a pointer to this parameter's signature.
- *m_pass* is an operator field, specifying how this parameter is passed.
- *m_next* is a pointer to the next parameter in the list.

2.2.9 Recdata Structure

To implement record data types, SR 2.3 maintains a *Recdata* structure as shown in Table 13. Details are as follows:

- *k_size* is the size (in bytes) of the record structure.
- *k_tdef* is the name of the generated type definition.
- *k_init* is the record initializer name.

Table 13: Structure of Recdata.

MemberName	*Information*
k_size	record size
k_tdef	name of generated typedef
k_init	name of initializer

2.3 An Example

Having examined all the possible attributes that may be attached by the SR compiler to symbols in the source program, we now look at how these tables are organized for the program shown in Figure 5.

```
resource b()
    var z1 := 1234
    var z2 : int

    process p (i := 1 to 2)
        fa k := 1 to 4 →
            write("in p",i,k)
        af
        write(z1)
    end
    process q
        z2 := 87654
        write("in q")
        z2 := z1+z2
        write(z2)
    end

    z1 := 1234
    z2 := 567
end
```

Figure 5: A short SR program with two processes p and q.

The symbol and attribute tables are shown in Figure 6. Fields which are not significant are not shown.

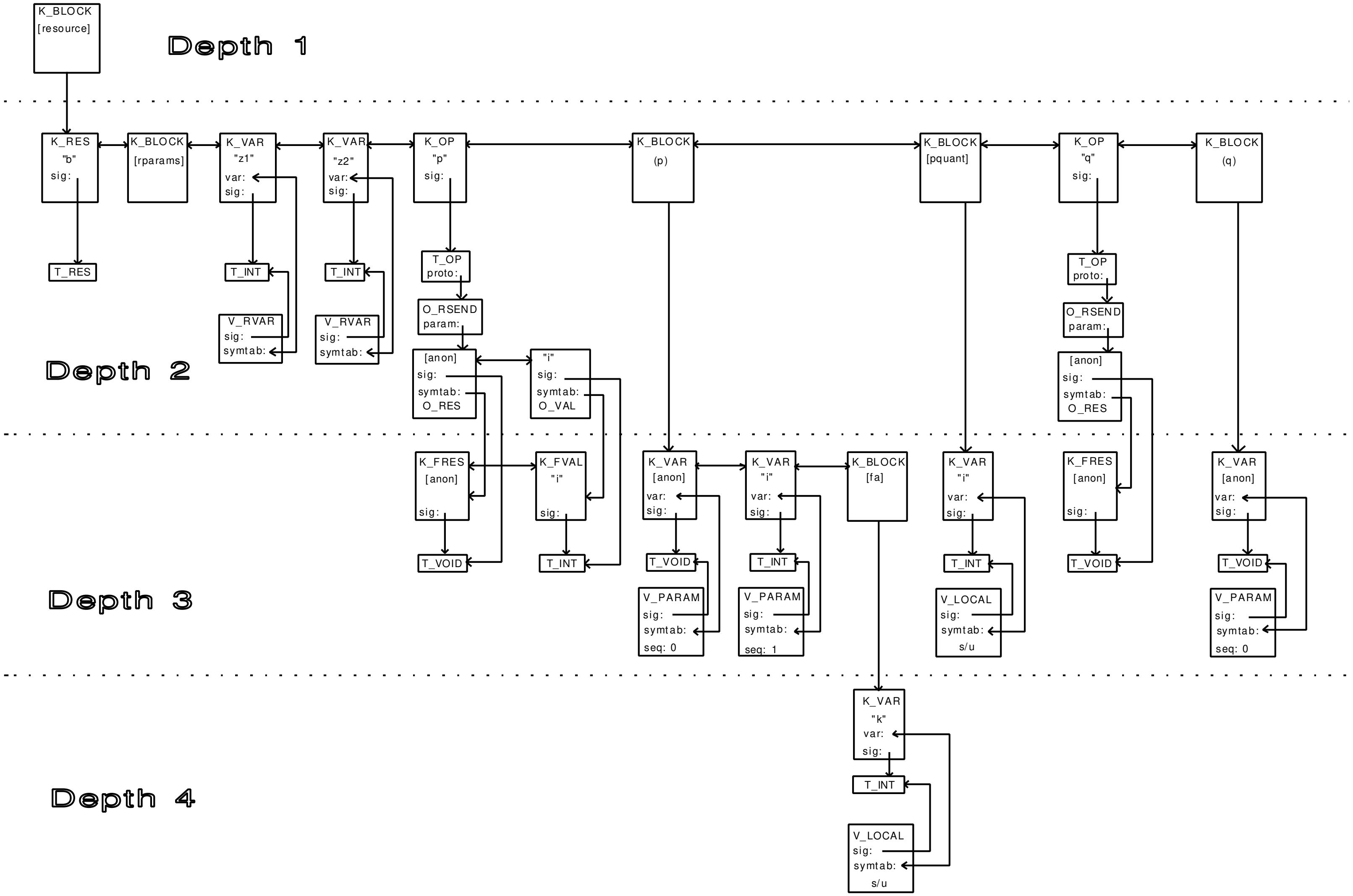

Figure 6: Symbol and attribute tables for resource b.

Each symbol node is identified by a name. Symbol names which are found in the program text are denoted by enclosing double quotes. Some symbols, such as [resource], [rparams], [pquant], [anon], [fa], (p), (q), are generated by **sr**.

In the Figure 6, nesting depths are separated by dotted lines. Directed vertical lines from one symbol node to another symbol node represent the *s_child* field. Horizontal lines from one symbol to another represent the *s_next* and *s_prev* fields of the symbol node. In addition, the diagram shows exactly how the structures are maintained; that is, how the symbol tree is represented. If the symbol node marked [resource] is the root, its children are all the symbol nodes at depth 2, and so on. Also shown in the diagram are the attributes associated with each symbol.

The first K_BLOCK named [resource] is the root of the symbol tree for the program in Figure 5. The nesting depth at this level is 1. At depth 2, the symbol tree shows that the resource is named b and receives no parameters. It it did have parameters, the *s_child* field of the K_BLOCK node labeled [rparams] will point to a symbol node at depth 3. The following two K_VARs correspond to the resource variables z1 and z2. Both variables are declared as integers as indicated by their signatures. The next symbol corresponds to the implicit declaration of operation p. Operation p is restricted to send, as its prototype shows (O_RSEND). This operation does not return any result (O_RES is [anon] with T_VOID signature), but accepts one integer value parameter (O_VAL with T_INT signature).

Operation p's body introduces a new scope as indicated by the symbol type K_BLOCK named (p). The formal parameters of operation p are represented at depth 3 as parameter variables. The K_BLOCK labeled [fa] introduces yet another scope for the for-all statement. The variable k is local to the fa block.

The next K_BLOCK at depth 2 corresponds to an implicit quantifier to create multiple instances of process p.

Operation q, like p, is restricted to send. This operation neither returns a result nor receives a parameter. The K_BLOCK for q introduces one additional symbol, the dummy [anon], corresponding to the unnamed parameter 0, the return value.

The parse tree for process q (lines 10-15) is shown in Figure 7. A process declaration is a short-hand for an op serviced by a proc. In addition, such operations are restricted to a send; that is, a call to that process will generate an error at compile-time.

An op parse node contains the operation name (left-subtree) and prototype (right-subtree). A prototype node has the information about formal parameters of and restrictions on the operation. The left-subtree of the list node under proto contains the name and type of the result. If q receives parameters, they would appear on the right subtree of the same list node.

The proc node contains a list of the argument names and the block containing the IR of the code to service operation q. The target code for process q, internally represented by the block subtree under proc, can be generated by performing a preorder traversal of the tree rooted at block.

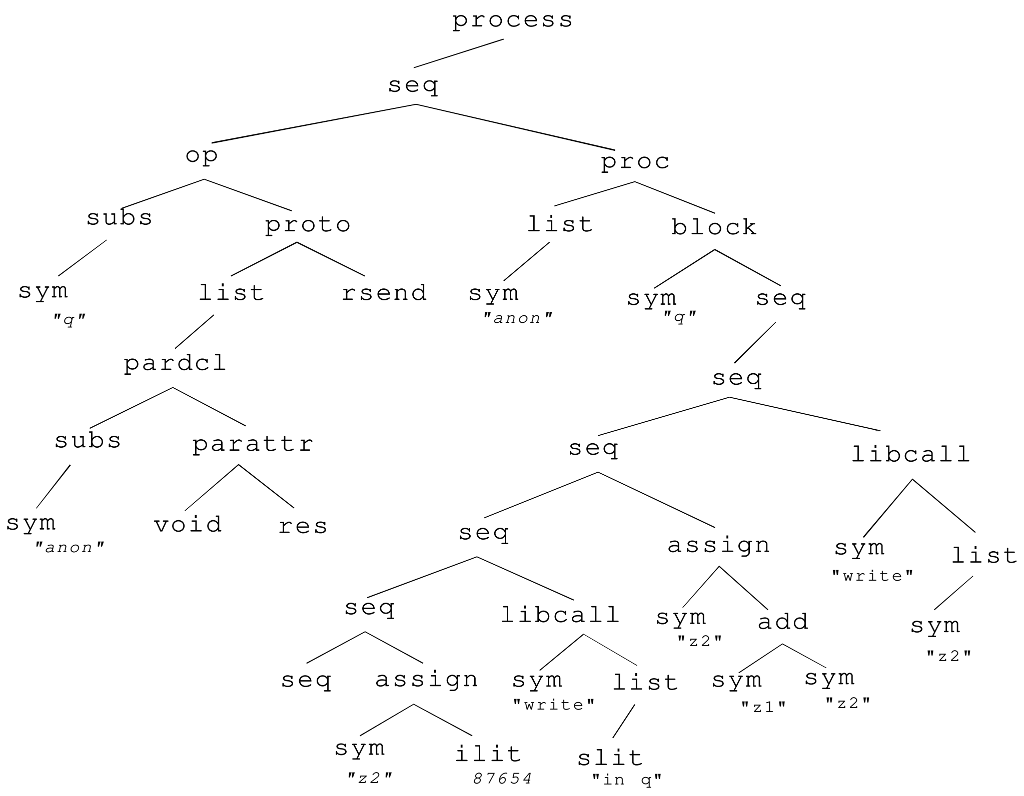

Figure 7: Parse tree for process q.

3 Run-Time Support System

The SR RTS may very well be considered the soul of the SR implementation: SR programs run under the control of the RTS. Section 3.1 shows how the RTS manages run-time memory; Section 3.2 explains how the main resource is created and destroyed, and how operations are represented at run-time; and Section 3.3 discusses how SR processes are managed.

3.1 Memory Management

Certain memory regions, such as those which contain resource variables, are owned by a specific resource. The RTS records the ownership of these memory blocks by using memory headers. This header contain the following information:

- *res* contains the address of the resource instance which owns the allocated region.
- *mnext, mlast* are forward and backward pointers, respectively, in a global memory list.
- *rnext, rlast* are forward and backward pointers, respectively, in a resource memory list.

Some data structures of a given type are grouped by the RTS into *pools*. Memory objects in this category are not owned by any resource [Morgenstern]. A pool is represented internally using the structure below.

- *name*, the pool name.
- *lockname*, name of pool's lock. This is only used for debugging.
- *free*, a list of free pool items.
- *inuse*, a list of in-use pool items.
- *blk*, a pointer to the next allocated pool block.
- *el_size*, the size of a pool element.
- *el_max*, the maximum number of items allowed.
- *el_cur*, the current number of items allocated.
- *init*, the function to call on new elements.
- *reinit*, the function to call on used elements.
- *pmutex*, the lock which protects access to the pool.

To illustrate, consider a pool of operations shown in Figure 8. The pool block contains three items, where the first one is reserved for the block link. The two pool items, labeled 1 and 2, are both in-use.

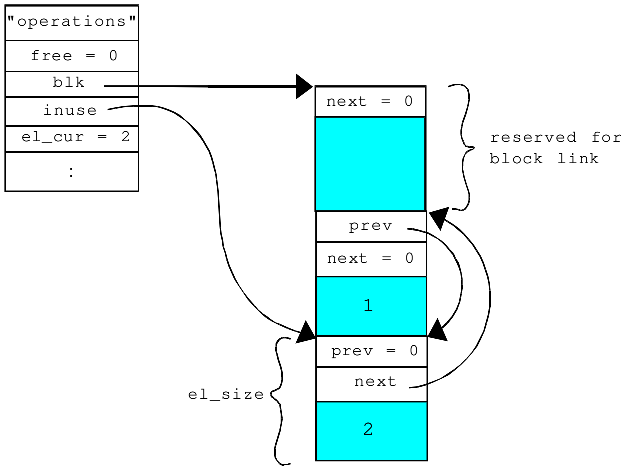

Figure 8: An operation pool with two in-use items.

As specified in pool.c, when a request is made to add an element into the operations pool, the RTS returns the first element in the free list[1], if there is one. If no pool item is free and the maximum number of allowed elements has not been reached, a new block of pool items is allocated from which a free list is built. Figure 9 shows the new pool. In this example, we assume that the RTS allocates three pool elements per block at a time, including the item reserved for the block link.

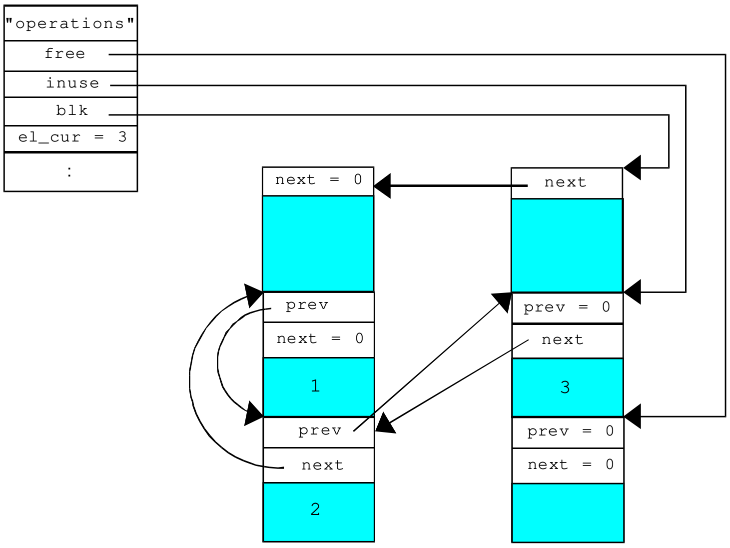

Figure 9: The operation pool after a request to add an item.

[1]Freed elements are kept in the pool for future reuse.

3.2 Resource and Operation Management

As mentioned in Chapter 2, in some cases a resource may be used in place of a global. In fact, SR 2.3 implements an instance of a global as a resource instance with special properties:

- a global instance is created automatically the first time it is referenced.
- only one global instance per global declaration may exist in one virtual machine (VM).
- a global may only be destroyed if its VM is destroyed explicitly.

On program startup, the RTS creates one virtual machine on the local physical machine. Once the virtual machine is initialized, an instance of the program's main resource is created [Andrews]. A process (with process type INIT_PR, and subsequently named *body*) is created to execute the initialization code for that resource. The initialization routine can be found from the *resource pattern*[2].

A resource pattern is a triple consisting of:

- resource name.
- address of initialization routine. By convention, the routine name is the resource name prepended with R_.
- address of finalization routine. By convention, the routine name is the resource name prepended with F_.

The RTS maintains a table of active resource instances per VM. This table is implemented as a linked-list [Andrews]. A resource table entry, called Rinst, is the RTS representation of a resource instance. Each Rinst is allocated from a pool of *active resources*. Table 14 depicts the contents of this structure, defined in res.h.

Details are as follows:

- *rpatid*, the resource pattern identifier.
- *seqn*, an identifying sequence number for a given resource instance. This is used by RTS to verify that a resource capability refers to an existing resource instance.
- *is_global*, a boolean flag which distinguishes resource instance from global instance.
- *rv_base*, the base address of the resource variable area. A create request block (CRB) is used whenever an instance of a resource needs to be created, either locally (on the same physical machine) or remotely (on another physical machine).
- *crb_addr*, the address of the CRB.
- *rmutex*, the descriptor lock.
- *procs*, a list of active processes owned by this resource instance.
- *meml*, a list of memory blocks owned by this resource instance.

[2] The resource pattern is declared in the generated C code for the given resource.

Table 14: Resource instance representation.

MemberName	*Information*
rpatid	resource pattern id
seqn	sequence number
is_global	global flag
rv_base	resource variable base address
crb_addr	CRB address
rmutex	descriptor lock
procs	active process list
memlist	memory block list
rcp	resource capability address
oper_list	operations list
status	status flag
next	link to next resource instance

- *rcp*, the address of the resource capability.
- *rc_size*, the number of bytes used to represent the first component of a resource capability, Rcap.
- *oper_list*, a list of operations owned by this resource.
- *status*, a status flag. This may be one of the following: initial, final, or reply.
- *next*, the link to the next resource instance (or the free list).

3.2.1 Resource and Operation Capability

A resource capability has two components: a structure called Rcap and a vector of operation capabilities (Ocap). Figure 10 shows how a resource capability is maintained.

Rcap consists of the following:

- *vm*, the identifier of the VM where the resource is instantiated. The identifier of the main virtual machine (MAIN_VM) is 1.
- *seqn*, a sequence number assigned when the resource instance is created; that is, when a request is made to add a resource instance entry into the resource table (the pool of active resources).
- *res*, a pointer to the resource instance Rinst.

Ocap contains basically the same information that Rcap maintains, except that instead of having a pointer to Rinst, the third component is a pointer to an operation table entry, Oper.

Opers are allocated from the pool of operations. This structure contains the following fields:

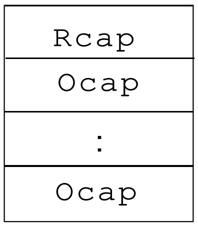

Figure 10: Representation of a resource capability.

- *res*, a pointer to the resource instance which owns this operation.
- *seqn*, operation sequence number. This should be the same sequence number as its Ocap's.
- *pending*, indicates the number of pending invocations.
- *type*, the operation type. Some of the operation types are PROC_OP (operation implemented by a proc), INPUT_OP (operation implemented by an input statement), PROC_SEP_OP (a proc operation which requires a separate context), and SEMA_OP (operations implemented by, or optimized as, a semaphore).
- *u*, a union which may be one of the following:
  1. *code*, address of the code which implements the operation. Typically used for operations of type PROC_OP.
  2. *clap*, input operation class. Used for INPUT_OP operations.
  3. *sema*, a semaphore used for SEMA_OP operations.
- *next*, a pointer to the next Oper, if one exists.
- *omutex*, a lock used by MultiSR.

3.2.2 Main Resource Creation

Before the main resource is created, the environment in which the SR program runs needs to be initialized. Described below is the process of initializing the environment; that is, setting up the virtual machine, and the creation of the main resource. [Morgenstern] provides a description of the SR RTS interface.

On startup, what the RTS mainly does is to initialize the process management system as discussed in 3.3. This is achieved by invoking the routine sr_init_proc and passing the address of the startup code. In a nutshell, sr_init_proc will build a context for the SR process to execute the startup code, in addition to setting up the job servers.

Each job server executes the code as specified in function vm_jobserver in process.c. The first thing it does is to create an idle process by calling make_idle_proc. This builds a context for an idle process, and enqueues this newly created process into sr_idle_list. This idle process may be moved to the sr_ready_list if there is nothing else to run from the ready queue. In such an event, a call to sr_reschedule is made to add the idle process to the ready queue, and a subsequent call to sr_scheduler is performed to request a context switch. The startup process will then be dispatched.

startup is responsible for initializing the RTS and creating the main resource (by calling sr_create_resource). The following are initialized: event list, RTS memory, capability for main resource and global, different pools, I/O descriptors, and random seed.

sr_create_resource, implemented in res.c, initializes the creation request block that is passed to it, and returns a resource capability pointer for the created resource. In addition, it also performs the following[3]:

1. Allocate a new resource instance descriptor.
2. Make the newly created resource the owner of CRB.
3. Spawn (via sr_spawn) a process to execute the initialization of this resource. The address of the initialization routine is found from the resource pattern[4].
4. Call sr_activate to add the newly spawned process into the ready list. This eventually causes the main resource initializer to be executed.

The resource initializer is responsible for allocating space for the resource variables. It calls sr_alloc_rv to reserve a specified number of bytes of memory whose owner is the current resource. Proc and input operations are also created in the resource initialization routine. Moreover, the resource body is executed by this process. As such, this process is named 'body' with process type INIT_PR. After executing the code specified in the resource body, a call to sr_finished_init is made. This function does not return, and it kills the calling process.

Figures 11 and 12 shows part of the data structures maintained by the RTS to record resource and operation instances for the example in Section 2.3. Recall that an Rinst is allocated from a pool of active resources, and each Oper actually reside in the operations pool. These pools, and the memory list, are not explicitly shown in these figures.

3.2.3 Resource Destruction

When there are no more processes to run, a dummy process named 'final' is created. This process has type FINAL_PR and its sole responsibility is to execute the final code for the resource. Like the initial code, the final code may be located from the resource pattern.

The last statement in the final code is always a call to sr_finished_final. This tells the RTS that the final code has completed. A destroying process is activated to execute sr_stop, and the calling process ('final') is killed.

3.3 Process Management

The SR RTS provides a light-thread process management package. A process is dispatched to execute a resource's initialization and finalization code. In addition, operations which requires

[3] A call to sr_create_res is made to accomplish the enumerated items.

[4] It must be noted that spawned processes are not immediately dispatched by the scheduler. This only means that a new context has been created.

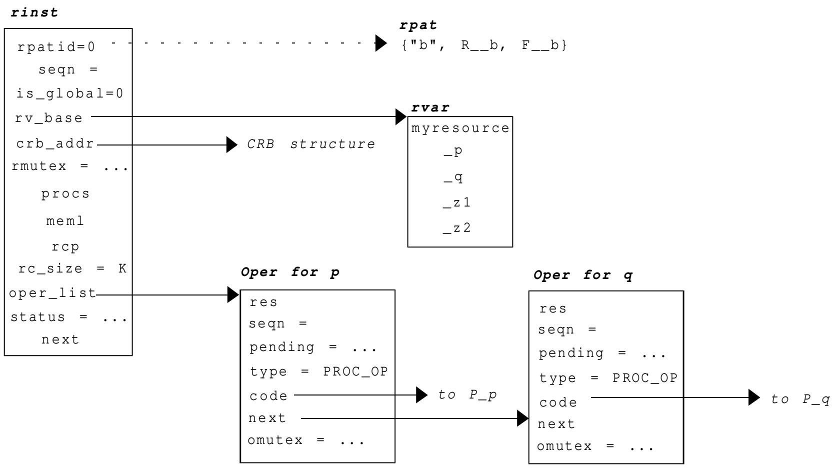

Figure 11: RTS structures for the SR program in Section 2.3.

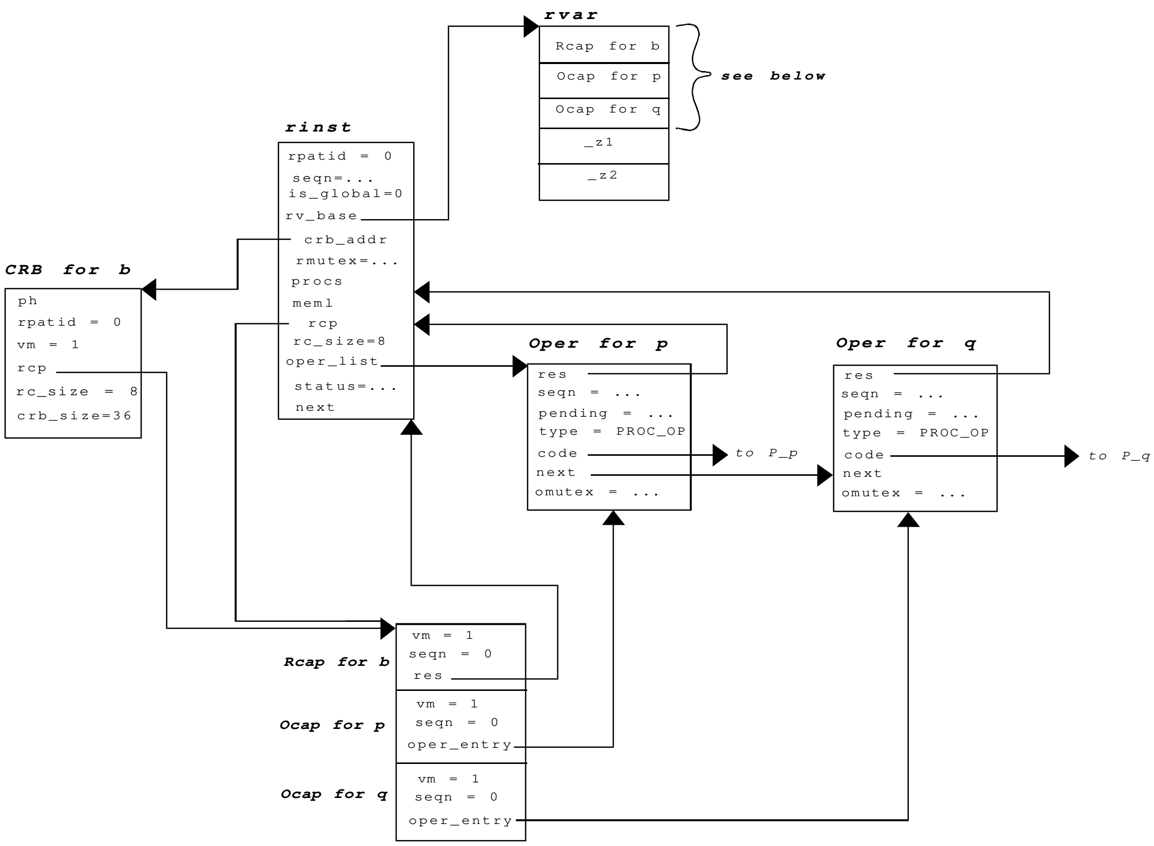

Figure 12: Extended RTS structures for the SR program in Section 2.3.

a separate context, such as those serviced by procs, run as SR processes. process.c contains the RTS routines to comtrol SR processes.

3.3.1 Initializing the Process Management System

As mentioned in Section 3.2 the process management system is initialized by the RTS via a call to sr_init_proc, passing the address of the startup code to it. sr_init_proc does the following:

1. Initialize the semaphore pool.
2. Set the following queues to empty: sr_ready_list, sr_io_list, and sr_idle_list.
3. Set up a free list of process descriptors.
4. Create an SR process context for the startup code and add it to the ready queue. It should be the first process in the ready queue. The startup process has INITIAL_PR type with READY status.
5. Create the vm job servers[5], i.e., build a context for each one. The first job server to get to the ready queue will execute the startup code.

3.3.2 Representation of an SR Process

Each SR process is represented by a structure called proc_st, maintained by the process management subsystem of RTS. Some of this structure's members are listed below.

- *ptype*, the process type. A FREE_PR process is a free process which resides in the free list process pool. A process with type INITIAL_PR is reserved for the process which executes the resource's initial code, while a FINAL_PR process executes the final code of a resource. IDLE_PR processes are created by make_idle_proc. Processes of this type sit in the idle list until no other process is in the ready list.
- *itype*, the invocation type if process type is PROC_PR. This may be one of the following: CALL_IN (call/input), SEND_IN (send/in), COCALL_IN (concurrent-call/in), COSEND_IN (concurrent-send/in), and REM_COCALL_IN (remote concurrent-call/in).
- *pname*, the process name known to the RTS. When executing user code, this may be the proc name, 'body' (resource initial code or body), or 'final' (resource final code).
- *priority*, the process execution priority. This tells the scheduler where to insert the process in the ready list.
- *stack*, the process stack containing its context.
- *status*, the process status. A process may be in one of the following states: ACTIVE, DOING_IO, READY, BLOCKED, BLOCKED_DOING_IO, INFANT (initial state of spawned processes), FREE (for FREE_PR type processes), TO_BE_FREED, and DISCARDED.
- *blocked_on*, a pointer to a list of processes to which a process is currently blocked on.

[5]A VM job server grabs the first process in the ready list and becomes it.

- *should_die*, a boolean flag indicating whether the process should be destroyed. This is set if another process attempts to kill a process which may be executing in another job server.

- *waiting_killer*, a pointer to a semaphore. This semaphore is used if an active process requests to kill another active process running in another job server. The killing process waits for the other process to set this semaphore. The process being killed will set this semaphore when it is ready to die.

- *res*, a pointer to the resource which owns this process.

3.3.3 Process Creation

SR processes are created thru a call to sr_spawn. Process creation involves allocating a new descriptor from the pool of processes. Once a process descriptor has been allocated, this descriptor, a representation of an SR process, is added to the list of processes for the resource. Stack space is also allocated for this process. The stack is set up so that the process may be properly activated later. This newly created process has an INFANT status.

Normally, a newly spawned process is added to the ready list. Its status is set to READY and a call is made to sr_add_readyq. This routine maintains the ready list in non-increasing process priority.

3.3.4 Process Destruction

Destroying a process, achieved by sr_kill, is a non-trivial task. The possibility of having multiple job servers implies that multiple processes may be active at any given time. Consequently, the process to be killed may be executing in another job server. Enumerated below is the algorithm used to destroy a process:

1. Remove the process from process list for its owning resource.

2. Remove the process from the appropriate scheduler list. The status of the process to be destroyed dictates the steps involved to achieve this end. We look at the different cases:

    - *Case 1*: status is INFANT. Simply set the status to TO_BE_FREED and free the process.
    - *Case 2*: status is READY. Dequeue the process from sr_ready_list. Set process status to TO_BE_FREED, and free the process.
    - *Case 3*: status is either ACTIVE or DOING_IO. If the process to be destroyed is the current one, set its status to TO_BE_FREED and call sr_scheduler. The process will be freed in that routine. However, if the process to be killed in not the current one, that is, executing in another job server, the following steps are taken: (a) a semaphore wait is created and initialized to 0. waiting_killer of the process to be killed is made to point to this newly created semaphore; (b) set should_die flag of process to be killed to TRUE; (c) wait for the other process to die by doing a P(wait). The process to be killed will do a V on this semaphore the next time it is dispatched (i.e., in sr_scheduler); and (d) delete the semaphore wait.

- *Case 4*: status is BLOCKED or BLOCKED_DOING_IO. Here, the process is removed from its blocked_on queue, the status is set to TO_BE_FREED, and then the process is freed.

References

[Andrews] Andrews, G. R., and Olsson, R. A. *The SR Programming Language: Concurrency in Practice.* Redwood City, CA: Benjamin/Cummings; 1993.

[Gebala] Gebala, R., SRI: An Interpreter for the SR Concurrent Programming Language. Master's Project, Department of Computer Science, California State University, Sacramento, Fall 1995.

[Morgenstern] Morgenstern, A. and Thomas, V. The SR Run-Time System Interface. Department of Computer Science, The University of Arizona, Aug. 1992.

[Olsson] Olsson, R. A., Andrews, G. R., Coffin, M. H., and Townsend, G. M. SR: A Language for Parallel and Distributed Programming. TR 92-09, Department of Computer Science, The University of Arizona, Mar. 1992.

[SR] SR Version 2.3, Department of Computer Science, The University of Arizona.