Carole Mattern McNamee
June 1991
Computer Science

Compiler Optimization of Interprocess Communication and Synchronization Mechanisms

Abstract

The development of high-speed parallel processors and networking systems has created a need for programming languages that efficiently utilize these parallel and distributed systems. Many concurrent programming languages provide explicit high-level interprocess communication and synchronization mechanisms, including remote procedure call, synchronous rendezvous, and asynchronous message passing. The implementations of these high-level mechanisms are complex and resultant programs are often costly in terms of execution time. This dissertation provides, what the author believes to be, the first comprehensive effort to optimize high-level interprocess communication and synchronization mechanisms at the compiler level.

We describe source-level transformations for mechanisms including asynchronous message passing, rendezvous, and remote procedure call. We present new transformations for client/server programs using rendezvous and/or asynchronous message passing. Our optimizing transformations can be applied to many commonly occurring communication and synchronization patterns and to programs written in concurrent programming languages such as Ada, Concurrent C, and SR. Results of applying the transformations to SR programs indicate reductions in execution time of up to 95% in some cases.

In addition, we describe the automatic application of these optimizing transformations, which requires the extension of dataflow techniques to concurrent languages. We present new techniques for handling shared variables and the extension of interprocedural analysis to interprocess analysis. The use of attribute grammars to perform dataflow analysis extends previous uses of attribute grammars for dataflow analysis. To show the

feasibility of this approach, we describe an actual automated implementation of our optimizing transformations for the SR language. A Prolog-based dataflow analyzer and pattern matcher identifies optimizable code and applies the transformations at the parse tree level.

Compiler Optimization

of

Interprocess Communication and Synchronization Mechanisms

Carole Mattern McNamee

B.S. (Simmons College) 1969

M.S. (Stanford University) 1973

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor Of Philosophy

in

Computer Science

in the

Graduate Division

of the

University Of California, Davis

Approved By:

____

Ronald A. Olsson, Chair

____

Robert M. Keller

____

Mahadevan Ganapathi

Committee in Charge

1991

Acknowledgements

There are many people to whom I owe a debt of gratitude for their guidance and support throughout this dissertation process. First and foremost is my major professor, Ronald Olsson. His contributions to this project and to my growth as a researcher are immeasurable. I am deeply grateful for all the time and energy that he has spent working with me. He is an outstanding mentor and I feel privileged to have had the opportunity to work with him.

I also thank the other members of my committee, Professor Robert Keller and Professor Mahadevan Ganapathi, who provided many helpful comments on the dissertation. Professor Keller was especially kind in sharing his Prolog expertise with me during the implementation phase of this project.

I am grateful to Caren Asimow for finding errors in this dissertation when I thought it couldn't be done and for her assistance with the implementation. I would also like to thank fellow graduate student Debbie Frincke for her willingness to be a sounding board on many occasions throughout my graduate career. There are two additional people to whom I owe a special thanks, Professor Mary Jane Lee and Professor Ronald Ernst from California State University, Sacramento. They encouraged me to return to school to get a Ph.D. and have been supportive throughout the process.

Finally, I would like to thank my family, friends, and fellow graduate students for their patience and support, especially: my husband, Mark, for his constant encouragement and for assuming more duties at home than we ever imagined would be necessary, my father for his constant interest in my progress, and my three children, Beth, Mark, and Katie, who have thrived despite a mother who is often "there, but not there".

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The development of high-speed parallel processors and networking systems has created a need for programming languages that efficiently utilize these parallel and distributed systems. Languages that provide *implicit parallelism* and languages that provide *explicit* control of parallelism have evolved. An implicitly parallel language is typically, both syntactically and semantically, an existing sequential language. However, the implementation of such a language identifies activities that can be executed in parallel and generates parallel target code. In contrast, an explicitly parallel language includes language features that provide explicit control over processes and their interactions. Two fundamental models for explicit interprocess communication and synchronization have emerged: those based upon shared memory and those based upon message passing [AS83]. The shared memory model uses shared variables with exclusive access provided by mechanisms such as semaphores, monitors, and conditional critical regions. The message passing model uses mechanisms such as asynchronous message passing, rendezvous, and remote procedure call. The message passing model is more general; its interprocess communication and synchronization mechanisms may be used in both distributed and shared memory multiprocessor environments.

Many languages providing explicit control over processes and their ability to interact have emerged. The implementations of these languages typically require both a compiler and a *run-time support system.* The compiler translates interprocess communication requests into calls to primitives in the underlying run-time support system. The run-time support system provides specific primitives for process creation and termination, and for the management of interprocess communication [AOC+88, MRS89].

High-level mechanisms for interprocess communication and synchronization have evolved and their semantics are complex. These mechanisms include *asynchronous message passing*, *rendezvous*, and *remote procedure call* (i.e., *RPC)*. The fundamental form for message passing is the send/receive pair. The most common form for asynchronous message passing is a non-blocking send coupled with a blocking receive [AS83]. Asynchronous message passing has been implemented in Plits [Fel79], SR [AOC+88], and Concurrent C [GR89]. A blocking send coupled with a blocking receive provides the basis for synchronous message passing or rendezvous. Rendezvous has been implemented in Ada [Ada83], Concurrent C, occam [Bur88], and SR. More complex variations of these message passing mechanisms include selective receives where guards are used to restrict receipt of messages. An additional mechanism, remote procedure call — a synchronous call to an inactive body of code — has emerged as a particularly useful mechanism for client/server interactions. Argus [LS83], DP [Bri78], and SR provide remote procedure call mechanisms.

Programmers find the abstractions provided by these mechanisms convenient because they hide details such as whether the communication and/or synchronization is between processes residing in the same address space or between processes residing on different machines. Unfortunately, these same abstractions also hide such details from compilers for concurrent languages. Consequently, compilers can make few assumptions concerning these details, and the execution of programs employing these mechanisms is often unnecessarily costly.

The optimization of these primitives is desirable in both shared memory and message passing environments. In the most general case, any semantic-preserving optimization that enables a program to run faster is considered desirable. In more specific situations, responsiveness may be of critical importance. For example, air-traffic controller systems require the quickest possible response time. In addition, real-time systems often running under hard real-time constraints provide an example where optimization may be not only desirable, but necessary for a program to meet those constraints. For example, a system controlling an assembly line must complete processing one object on the assembly line before the next one arrives.

The optimization of interprocess communication and synchronization primitives is difficult for three reasons. First, the compiler can make few assumptions concerning the run-time machine environment in which a program executes. Second, the interactions between the compiler generated code and the run-time support system are complex. Finally, many semantic subtleties associated with high-level forms of interprocess communication can affect the applicability of optimizations.

While significant results have been achieved in the compiler optimization of sequential languages [ASU86], only minimal progress has been made in identifying and implementing optimizations of parallel languages [BLW87, MP90, RS88]. New techniques are required for the optimization of these new language features providing explicit control over processes and their interactions.

This dissertation describes an approach to the optimization of high-level interprocess communication and synchronization mechanisms present in explicitly parallel languages. A repertoire of source-level transformations that replace high-level mechanisms with less costly low-level mechanisms has been developed [MO90b]. These transformations can be applied to many commonly occurring communication and synchronization patterns and to programs written in concurrent programming languages such as Ada, Concurrent C, occam, and SR. Results of hand-applying the transformations to SR programs indicate reductions in execution time of up to 95% in some cases. To demonstrate the feasibility of this approach to optimization, an implementation of our optimizations for the concurrent programming language SR is described. Given an SR source program, a Prolog-based transformer identifies optimizable code and applies the transformations at the parse tree level. Dataflow analysis, as well as interprocess and intermodule analysis, provides the information necessary to ensure appropriate application of specific optimizations.

1.1 Overview of Approach

The approach to optimization of interprocess communication and synchronization mechanisms described in this dissertation entails two related but fundamentally different tasks. The first task involves the development of source-level transformations that improve the performance of programs using high-level forms of interprocess communication and synchronization including asynchronous message passing, rendezvous, and remote procedure call. The second task involves the development of techniques that enable automatic application of the transformations as part of compilation.

Many of the optimizing transformations require that the communicating processes reside in the same address space. The transformations typically reduce the need for context switching, simplify the specific form of interprocess communication, and/or reduce the complexity of the given form of communication. One, perhaps counter-intuitive, transformation actually increases the number of processes as well as the number of context switches to improve overall program performance. Many of these optimizations have been hand-simulated at the source-code level and timed on several classical client/server problems. Generally, the execution times for the optimized programs were significantly lower than those for equivalent unoptimized programs.

Optimizations of both asynchronous message passing and remote procedure call have been proposed and implemented for SR [AOC+88]. However, the asynchronous message passing optimization is limited to a small subset of the potentially optimizable cases, and the optimization of remote procedure call is performed at compile time for a few cases, and at run-time — within the run-time support system — for the remaining cases. The techniques described in this dissertation significantly extend the compile-time application of these two optimizations.

In addition, new optimizing transformations for rendezvous are presented. The transformations for rendezvous extend earlier efforts of Habermann and Nassi [HN80], Roberts et al. [REMC81], and Schauer [Sch82] by introducing a general technique for the transformation of guarded synchronization. The technique is based on a new application of the baton passing method, a method originally developed by Andrews [And89] for deriving concurrent programs. All of these transformations replace rendezvous by less expensive mechanisms, namely, procedure calls to inactive procedures that employ semaphores to preserve the original order of execution. This transformation technique is extended to include the optimization of high-level asynchronous message passing, as well.

Hikita and Ishihata [HI85] describe high-level program transformations removing the procedure calls altogether; however, the synchronization mechanisms in the transformed programs are not low-level enough to make implementation easy, particularly in the case of guarded synchronization. Other approaches to optimizing interprocess communication have focused on run-time support system enhancements [Hil82, JA82, Ves89]. Both Hilfinger and Vestal propose using coroutines to implement tasks that employ specific forms of rendezvous, namely monitors identified by programmer-supplied compiler directives and some additional compiler analysis. Our new program transformations for rendezvous and asynchronous message passing handle a wider range of synchronization problems because mutual exclusion is transformed explicitly, making no assumptions about the need for concurrency.

The source-level program transformations described in this dissertation require no programmer-supplied compiler directives. In addition, these transformations could be applied as an initial step, thus providing additional opportunities for the application of Hilfinger's and Vestal's optimizations. Stevenson [Ste80] proposes another approach to run-time support system enhancement with the use of an intermediate language to facilitate the development of several different implementations of Ada rendezvous.

Given this repertoire of transformations, the second major task involves the development of techniques that allow the optimizing transformations to be applied automatically as part of the compilation process [MO90a]. The techniques include the application of dataflow analysis and the extension of interprocedural analysis — *interprocess analysis* — to concurrent programming languages and, more specifically, to the optimization of interprocess communication and synchronization. The goal is to identify patterns of interprocess interaction. An interesting aspect of these techniques is the use of attribute grammars to perform the dataflow and interprocess analysis. This use of attribute grammars extends the *method of attributes* described by Babich and Jazayeri [BJ78] to include concurrent language features and both interprocess and intermodule analysis.

1.2 Overview of Implementation

To show the feasibility of the approach described above, an actual implementation was developed. SR is used as the base language for the optimization effort. It provides many forms of interprocess communication including asynchronous message passing, and synchronous calls to active bodies of code (rendezvous) and to inactive bodies of code (remote procedure call). By contrast, Ada provides only rendezvous, and Concurrent C provides asynchronous message passing and rendezvous. SR also provides a module and process structure that is in many ways a superset of those found in other languages. However, most of the optimizations we have developed for SR are applicable to other concurrent languages.

A subset of the optimizing transformations has been successfully incorporated into a compilation tool. The overall scheme of the tool is shown in Figure 1.1. As shown, the *transformer* takes an entire program as input and produces a transformed program with some embedded compiler directives. The transformer constructs an attributed parse tree from the source program; performs the necessary analysis using this parse tree; determines the optimizable operations; depending upon the type of optimization being performed, either inserts compiler directives or replaces whole subtrees with subtrees reflecting optimized code; and finally, reconstructs a source program from the modified parse tree. Prolog is used as the implementation language for the transformer because a significant amount of tree pattern matching was anticipated, and in fact occurs.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a brief overview of SR and relates its interprocess communication and synchronization mechanisms to those in Ada and Concurrent C. Issues that affect the cost of interprocess communication are discussed. Chapter 3 presents a repertoire of source-level optimizing transformations and a model for the generalization of transformations for rendezvous with guarded synchronization. The results of hand-applying these transformations are discussed throughout the chapter. Chapter 4 motivates the analysis necessary for the automatic application of the optimizing transformations. Chapter 5 describes the techniques used to implement the transformations, including dataflow analysis, intermodule analysis, and interprocess analysis, as well as the effects of shared variables on the analysis. Chapter 6

Figure 1.1: Scheme for applying optimizing transformations.

describes the optimizer's Prolog implementation and its performance. Chapter 7 provides a discussion of the results, future directions for this work, and relevance of this approach to other languages. Chapter 8 contains a summary and concluding remarks.

1.4 Summary of Contributions

This dissertation provides, what the author believes to be, the first comprehensive effort to optimize high-level interprocess communication and synchronization mechanisms at the compiler level. We describe source-level transformations for mechanisms including asynchronous message passing, rendezvous, and remote procedure call. This dissertation presents new transformations for rendezvous and asynchronous message passing. The application of these source-level transformations necessitates the extension of dataflow techniques to concurrent languages. We present new techniques for handling shared variables and the extension of interprocedural analysis to interprocess analysis. The use of attribute grammars to perform dataflow analysis extends previous uses for attribute grammars. Perhaps the most difficult task has been the analysis required to determine when process interactions are between processes residing in the same shared address space. For languages allowing dynamic process creation, as SR does, this problem is not solvable in the general case; however, it is possible to identify specific situations where address space is shared between processes. This dissertation describes these situations and the analysis required to identify them. For languages with a more static process and module structure, the identification of interprocess interactions that occur between processes sharing the same address space is a significantly easier task.

Chapter 2

Language Background

SR provides all three forms of high-level asynchronous and synchronous interprocess communication: message passing, rendezvous, and remote procedure call. It is, therefore, a logical choice for the study of the optimization of these mechanisms. This chapter introduces SR's model of computation — including its virtual machine, module, and process structure — and describes the syntax and semantics of its interprocess communication mechanisms [AOC+88].[1] We then describe interprocess communication mechanisms in both Ada [Ada83] and Concurrent C [GR89] and the implications of subtle semantic differences between these mechanisms and those in SR. Finally, several issues affecting the run-time performance of these mechanisms are discussed.

2.1 SR

2.1.1 SR Model of Computation

The SR model of computation allows a program to be split into one or more address spaces called *virtual machines*. Each virtual machine defines an address space on one physical machine. Virtual machines can be created dynamically using a create statement; they are referenced indirectly through *capability* variables. For example, in the following create statement the capability variable, `vm1`, is assigned a capability for a newly created virtual machine.

[1] SR version 1.1 is described. Details and more examples can be found in [AOC+88, AO89]. SR version 2 [AO, Ols91], currently under development, is mentioned in Chapter 7.

```
vm1 := create vm()
```

Virtual machines contain instances of *resources*, SR's module construct.

A resource can contain two parts: an optional specification (i.e., a spec) and an implementation (i.e., a body). An import mechanism is used to make objects declared in the spec of one resource available in another. Resources can contain optional initialization and finalization processes, other processes, shared variables, and shared operations — SR's mechanism for interprocess communication.

Instances of a resource are created dynamically by means of create statements. A create statement assigns a capability for referencing the newly created resource instance to a capability variable. An optional clause on the create statement indicates on which virtual machine the resource instance is to be created; the virtual machine is specified via a virtual machine capability. The default is the virtual machine on which the create statement is executed. (Resource instances cannot migrate.) Consider the statements below.

```
r1 := create r() on vm1
r2 := create r()
```

An instance of resource `r` is created on the virtual machine for which `vm1` is a capability and `r1` is assigned a capability for this instance. Similarly, capability variable `r2` is assigned a capability for an instance of resource `r`; however, this instance of `r` is created on the same virtual machine on which the create statement executes.

Processes are created dynamically as well, and contain code invoking and servicing shared operations. Additionally, processes contain traditional sequential code and can declare, invoke, and service local operations.

Operations may be invoked directly through the operation's declared name, indirectly through a variable declared as a capability for the operation, or through an operation's declared name qualified by a resource capability variable. Capability variables for operations are strongly typed and may point to operations with structurally equivalent signatures. They may be global, belonging to a resource, or local, belonging to a process. They may also be passed as parameters to operations during invocation or to resources during resource creation, thus allowing processes in different resource instances on possibly different virtual machines to communicate.

Figure 2.1: SR's model of computation.

Figure 2.1 summarizes SR's model of computation. The ellipses indicate that repetition is allowed. In its simplest form, a program consists of a single virtual machine executing on one physical machine, possibly a shared-memory multiprocessor. A program can also consist of multiple virtual machines executing on multiple physical machines.

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

2.1.1.1 Virtual Machine and Resource Creation — An Example

A typical pattern for virtual machine and resource creation is shown in Figure 2.2. Two virtual machines are created and capabilities for them are assigned to virtual machine capability variables `vm1` and `vm2`. An instance of resource `r` and an instance of resource `s` are then created on each of the virtual machines. Resource capability variables `r1`, `r2`, `s1`, and `s2` are assigned capabilities for these resources. Instances of resource `s` are passed capabilities for instances of resource `r`. The instance of resource `s` on `vm1` is passed the capability for the instance of resource `r` also located on `vm1`. Similarly, the instance of resource `s` on `vm2` is passed the capability for the instance of resource `r` also located on `vm2`. Thus, interprocess communication between processes in an instance of resource `r` and processes in an instance of resource `s` will be within a single virtual machine.

2.1.2 Sequential SR

This section briefly describes only those SR sequential constructs used in the examples in this dissertation. Primitives for sequential programming in SR are similar to those in many procedural languages. They include, among others, typical assignment statements, a skip statement that is semantically equivalent to an empty statement, and several flow of control constructs. Control primitives include an if statement and two iteration primitives: the do statement and the for-all statement. The many-armed if statement combines aspects of both an if statement and a case statement. It evaluates the guards for each arm until one evaluates to `true` and then executes the block of code associated with that guard. The order

```
resource main()
  import r, s
  initial
    var vm1, vm2:  cap vm
    var r1, r2:  cap r
    var s1, s2:  cap s
    vm1 :=  create vm()
    vm2 :=  create vm()
    r1  :=  create r()  on vm1
    r2  :=  create r()  on vm2
    s1  :=  create s(r1)  on vm1
    s2  :=  create s(r2)  on vm2
  end
end
```

Figure 2.2: SR program fragment illustrating virtual machine and resource creation.

in which the guards are evaluated is nondeterministic. If no guards evaluate to `true` and no else arm is associated with the statement, program execution continues with the program statement immediately following the if statement. The do statement is the mechanism for indefinite iteration. It can have more than one arm and it terminates when the guards in all arms evaluate to `false`. The for-all statement (`fa`) is used for definite iteration.

2.1.3 SR's Interprocess Communication and Synchronization Mechanisms

SR's interprocess communication mechanisms are operation oriented, i.e., messages are sent to "operations" rather than to specific processes. An operation is invoked either by an asynchronous send statement or by a synchronous call statement. An operation is serviced either by an input statement, executing within an active body of code, or by a proc, an inactive body of code for which a new process is created. Table 2.1 summarizes the effects of the various combinations of SR's interprocess communication primitives.

The remainder of this section presents the syntax and semantics of SR's high-level interprocess communication mechanisms.

Invocation	*Service*	*Effect*
call	**proc**	procedure call
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

Table 2.1: Summary of SR's interprocess communication mechanisms.

2.1.3.1 SR Operation Invocation

SR operations can be invoked by either non-blocking send statements or blocking call statements. Asynchronous message passing uses a non-blocking send to an operation. For example,

send `foo(x,y)`

is an asynchronous invocation of the operation `foo`, which has two parameters. Synchronous interactions use a blocking call to an operation, e.g.,

call `goo(a,b,c)`

The keyword `send` or `call` may be omitted from an invocation, in which case, the invocation will be synchronous.

SR also has a statement that provides a mechanism for concurrent invocation of operations. While this dissertation does not directly address the optimization of this construct, some discussion appears in Chapter 6.

2.1.3.2 SR Operation Implementation

SR operations are implemented either by input statements or by procs, similar to procedures. An operation implemented by an input statement may be implemented by more than one input statement; however, an operation implemented by a proc is implemented by only one proc. The following sections provide details of the input statement and the proc.

2.1.3.2.1 The SR Input Statement — Asynchronous Message Passing and Rendezvous

The SR input statement is generally blocking (however, see the discussion at the end of this section) and implements both asynchronous message passing and rendezvous. An input statement implements one or more operations and services invocations of the operations generated by asynchronous sends or synchronous calls. For example,

in `foo(r,s)` `->` *body* **ni**

services invocations of the operation `foo` such as those shown in the previous section. The identifiers `r` and `s` are formal parameters local to the input statement. The actual parameter values received are not accessible outside the body of the `in` statement unless they are, for example, assigned to variables declared outside the scope of the input statement, as in

in `foo(r,s)` `->` `i:=r; j:=s` **ni.**

SR's receive statement is a convenient notation for an input statement when the body of the input statement contains only assignments of parameter values to variables declared outside the input statement. For example, the above input statement can be written more concisely as

receive `foo(i,j).`

A process may service one of several operations by using an input statement with multiple arms. This provides a single thread of control and, thus, a form of mutual exclusion as only one message may be selected by one arm of the input statement at a time. In the example below, the input statement services one invocation of `read`, `write`, or `seek` each time it is executed.

```
in  read(...)   -> ...
[]  write(...)  -> ...
[]  seek(...)   -> ...
ni
```

Operations serviced in the same input statement belong to the same operation class. Messages to an operation are placed on the class queue associated with the operation. The input statement services messages from the class queue on a first-come-first-served (FCFS) basis. Thus, pending requests for `read`, `write`, and `seek` operations in the above example are placed on the same queue and serviced in FCFS order.

SR provides two mechanisms to alter the FCFS servicing of operation invocations: synchronization expressions and scheduling expressions. Synchronization expressions (i.e., guards) may be used to determine which pending invocations are eligible for servicing. An operation invocation is eligible for servicing if there exists an arm in an input statement that can service the operation whose synchronization expression is true. Because synchronization expressions may reference operation parameters, they must be evaluated for each invocation of the operation. An invocation is serviced by an arm of an input statement when the synchronization expression applied to the specific invocation evaluates to true and when all invocations that are queued before that invocation that are eligible to be serviced have been serviced. In the example below, one of the two operations, `deposit` and `fetch`, is serviced subject to values of the synchronization expressions.

```
in  deposit(item) & count < size  ->  ...
[]  fetch(item) & count > 0       ->  ...
ni
```

SR's scheduling expressions are used to impose ordering on the servicing of pending invocations. Invocations of operations are serviced in the non-decreasing order of the value of the scheduling expression. The scheduling expression may be a function of the operation's actual invocation parameters. For example, the input statement below services pending invocations for `disk_read` in "closest to the current position" order.

```
in  read_disk(cyl,...) by abs(cyl-curpos)  ->  ...  ni
```

In addition to the forms described above, the input statement may combine synchronization and scheduling expressions, thus providing additional control [AOC+88].

One additional feature of the input statement is the optional else arm. If present, the else arm is selected when there are no pending messages to other operations serviced by the input statement or none of the pending messages satisfy the synchronization expressions associated with the operations. Thus, the input statement can be non-blocking.

2.1.3.2.2 The SR proc — Remote Procedure Call

On the invoking side, remote procedure call uses a synchronous invocation as described in Section 2.1.3.1. On the servicing side, invocations are serviced by a process created specifically to invoke a passive procedure, called a proc. The proc is referred to as the implementation of an operation. The following is a proc implementing the operation `foo`.

```
proc foo(x,y)
  ...
end
```

When a proc is invoked, a process is created within the resource instance in which the operation resides. This process then invokes the proc as though it were a simple procedure. When the proc finishes executing, the process created to call it is destroyed. Obviously, when the invoking process and the proc reside within the same shared address space, the creation of an intermediary process to call the proc is grossly inefficient. As mentioned in Section 1.1, SR's compiler currently detects this situation on a limited basis using ad hoc techniques. The compiler then generates code for a simple procedure call, rather than a remote procedure call. Remaining optimizable cases are detected during program execution by the run-time support system, which then performs a simple procedure call in place of the standard RPC. This optimization is further discussed in Section 3.3.

It is important to note that remote procedure call provides no implicit mutual exclusion. Any number of processes may call the proc and mutually exclusive access to global variables referenced inside the body of the proc is not guaranteed.

2.1.3.3 Additional SR Mechanisms Related to Parallelism

2.1.3.3.1 Semaphores

SR provides semaphores with P and V commands. However, these semaphores are simply syntactic abbreviations for parameterless operations and simple receive and send statements, e.g., `P(x)` is

```
receive x()
```

and `V(x)` is

```
send x().
```

However, these operations are, perhaps surprisingly, not necessarily semaphores. Currently, SR's compiler optimizes such semaphores, using ad hoc techniques, only when the P's and V's occur within a single resource and are not invoked through capability variables. The P's and V's are optimized by using semaphores provided in the kernel of the run-time support system. In this dissertation, semaphores will only be used to represent true semaphore operations, i.e., those implemented as semaphores. Additional semaphore optimizations are discussed in Section 3.2.

2.1.3.3.2 Return and Reply Statements

SR return and reply statements are used by operation implementations to signal completion of an invocation and to pass back results to invoking processes that are blocked. The difference between these two statements is that the return statement terminates servicing of the operation invocation, while the reply statement continues the servicing of the operation. For example, in Figure 2.3(a) the process `foo` blocks waiting for its invocation of `goo` to complete. When the return statement is executed, proc `goo` terminates and process `foo` is notified that the invocation of `goo` is complete so that `foo` may continue its execution. In Figure 2.3(b), process `foo` blocks waiting for completion of its invocation of `goo` as in the previous example. However, when the process executing proc `goo` executes its reply statement, process `foo` is notified that the invocation of `goo` is complete, but continues execution after the reply statement, conceptually in parallel with `foo`.

The presence of a reply statement affects the applicability of some optimizing transformations. This issue is addressed in Section 4.1.1.

2.1.3.4 *Readers/writers* Example

To further introduce SR's interprocess communication mechanisms, consider an SR solution to the familiar *readers/writers* problem [CHP71]. This problem is chosen because it is representative of a large class of client/server synchronization problems. It is also used in subsequent chapters to illustrate the new optimizing transformations for rendezvous that are presented in this dissertation.

```
...

process foo
  call goo()
  ...
end

proc goo()
  ...
  return
  ...
end
```

(a): Code fragment using `return`

```
...

process foo
  call goo()
  ...
end

proc goo()
  ...
  reply
  ...
end
```

(b): Code fragment using `reply`

Figure 2.3: Use of `return` and `reply` in SR programs.

Figure 2.4 outlines a single resource program for readers/writers. One instance of the `server` process, `noreaders` instances of the `reader` process, and `nowriters` instances of the `writer` process are created when the program begins execution. The server uses an input statement (`in`) to handle requests and releases from the readers and writers. The variables `nr` and `nw` record the number of readers and writers, respectively, currently accessing the database. The guards on the request operations contain synchronization expressions that restrict the order, as dictated by the problem, in which processes access the database.

The operations `read_request`, etc. are invoked synchronously by the readers and writers. Asynchronous invocation of the release operations is also appropriate and can be specified by, e.g.,

```
send read_release().
```

2.2 Ada's Interprocess Communication and Synchronization

The Ada programming language provides a rendezvous mechanism similar to SR's; it does not provide asynchronous message passing or an RPC mechanism [Ada83]. Ada's rendezvous mechanism is invoked by a blocking call to an Ada "entry" (similar to an SR

```
resource rw()
    op read_request();        op write_request()
    op read_release();        op write_release()
       ...

    process server
        var nr:=0, nw:=0

        do true  ->
            in read_request() & nw=0 ->
               nr++
            []read_release() ->
               nr--
            []write_request() & (nr=0 and nw=0) ->
               nw++
            []write_release() ->
               nw--
            ni
        od
    end server

    process reader(n:=1 to noreaders)
               ...
           read_request()
               ...
           read_release()
               ...
    end reader

    process writer(n:=1 to nowriters)
               ...
           write_request()
               ...
           write_release()
               ...
    end writer
end rw
```

Figure 2.4: Conventional SR solution to *readers/writers*.

"operation"). The entries are serviced by Ada accept statements. A select statement is provided to enable selection of one of a number of entries and is analogous to the SR input statement with multiple arms. Guards are provided in the form of a when clause; however, unlike SR, Ada guards cannot access invocation parameters. The Ada equivalent of the SR input statement in Figure 2.4 is presented in Fig 2.5.

```
select
   when nw=0 =>
       accept read_request;
       nr++;
or
   when true =>
       accept read_release;
       nr--;
or
   when nr=0 and nw=0 =>
       accept write_request;
       nw++;
or
   when true =>
        accept write_release;
        nw--;
end select;
```

Figure 2.5: Ada select statement for the *readers/writers* problem.

One significant difference between SR's and Ada's rendezvous mechanisms is the order in which pending invocations are serviced. Ada places invocations for entries on entry-specific queues and services each entry queue in FCFS order. When invocations are pending for several entries whose guards are true, selection of an entry is nondeterministic, not FCFS as in SR. To illustrate this point, assume in the readers/writers example of Figure 2.5 that both `nr` and `nw` are zero and that invocations for both `read_request` and `write_request` are pending. Ada's nondeterministic selection semantics permit the servicing of either a `read_request` or a `write_request`. In contrast, SR's selection semantics specify that among pending invocations for `read_request` and `write_request` the invocation that was received first must be serviced first. The implications of FCFS versus nondeterministic inter-entry selection semantics are addressed in detail in [OM].

2.3 Concurrent C's Interprocess Communication and Synchronization

Concurrent C provides both a rendezvous mechanism similar to SR's and Ada's, and an asynchronous message passing mechanism [GR89]. As in SR and Ada, rendezvous is invoked by a call to a "transaction" (similar to SR's "operation" and Ada's "entry"). Transactions are serviced by accept statements similar to Ada's accept and SR's input statement. A select statement similar to Ada's is also provided. Ada-like guards are used to control selection of specific transactions. As in Ada, pending invocations are placed on transaction-specific queues. When more than one transaction-specific guard is true and those transactions have pending invocations, the selection semantics is nondeterministic. Similar to both SR and Ada, invocations for the same transaction are serviced in FCFS order. This FCFS servicing order can be controlled through the use of synchronization and scheduling expressions similar to those in SR. These synchronization and scheduling expressions can reference invocation parameters. The Concurrent C equivalent of the SR input statement in Figure 2.4 is given in Figure 2.6.

```
select
  (nw=0):
      accept read_request()
         nr++;
or
      accept read_release()
         nr--;
or
  (nr=0 and nw=0):
      accept write_request()
         nw++;
or
      accept write_release()
         nw--;
```

Figure 2.6: Concurrent C select statement for the *readers/writers* problem.

Concurrent C's asynchronous message passing mechanism is similar to SR's, although slightly more restrictive. Transactions are declared to be `async`. Invocations of `async` transactions are implicitly non-blocking and are serviced just like synchronous transactions. In SR, operations may be invoked either synchronously or asynchronously unless specifically restricted in the operation's declaration.

2.4 Implications of Semantic Differences between SR, Ada, and Concurrent C

Several differences between SR, Ada, and Concurrent C have been described. These include the input statement selection semantics and whether or not guards that constrain the servicing of operation invocations may access actual invocation parameters. The implications of these differences, although subtle, are significant when devising semantic-preserving transformations and when considering the applicability of the optimizations developed for SR to Ada and Concurrent C. For example, ensuring SR's FCFS servicing of invocations in some of the optimizing transformations is, in general, more difficult and more costly than ensuring nondeterministic servicing used in both Ada and Concurrent C. Section 3.4.2.2 discusses this issue in greater detail. The inability of Ada guards to reference invocation parameters presents additional problems when applying optimizing transformations for rendezvous. See Section 3.4.1 for more details.

Another factor affecting the development of specific transformations and the measurement of their performance is the availability of efficiently implemented semaphore operations. As described in Section 2.1.3.3, SR provides such a facility. Ada also provides efficiently implemented semaphores as a special package. Concurrent C can simulate semaphore operations [GR89]. However, each semaphore requires a manager process to execute P and V transactions. The cost of invoking the P and V transactions plus the incumbent context switching at least partially negates the benefits of using the semaphores. Efficiently implemented semaphores could certainly be obtained in Concurrent C. For example, if Concurrent C's run-time support uses semaphores, then compiler directives could be used to tell the compiler to generate code that directly employs those semaphores.

2.5 Hidden Costs of Interprocess Communication

When attempting to identify optimizing transformations for interprocess communication mechanisms, preservation of the semantics of the mechanisms at the language level is important. However, it is also necessary to examine the costs of interprocess communication associated with the underlying run-time support system, i.e., hidden costs. These costs can be significant and can affect the performance of the optimizing transformations. Context switching is one such cost. A *forced* context switch is a context switch necessary to complete a communication or synchronization operation. The number of forced context switches required for a given interprocess interaction to complete depends upon: whether the communication is synchronous or asynchronous; and whether an operation is invoked before or after a process attempts to service the operation.

In the case of asynchronous message passing, if an operation is invoked before a process attempts to service the operation, a single context switch is required to complete the communication. If a process attempts to service an operation before it is invoked, then two context switches are required to complete the communication: one to the invoking process, and another back to the servicing process.[2]

Rendezvous requires more forced context switches to occur because of the blocking nature of the operation invocation. If an operation is invoked before any attempt to service the operation, two context switches are required to complete the synchronization: one to the servicing process, and another back to the invoking process to signal completion of the call. On the other hand, if a process attempts to service an operation before it is invoked, three context switches are required to complete the synchronization: one from the blocked servicing process to the invoking process, a second from the invoker back to the servicing process, and, finally, a third from the servicing process back to the invoking process to signal completion of the call. [HN80]

The RPC mechanism also has some hidden costs associated with it, namely, process creation and destruction. Recall from Section 2.1.3.2.2 that RPC requires the creation of

[2]This number of forced context switches analysis assumes a uniprocessor environment. In a multiprocessor or distributed environment, the number of forced context switches has a slightly different characterization, but they still exist.

a process on the machine on which the procedure is located. This process then performs a simple procedure call. Once the procedure has completed, the executing process is destroyed and the process invoking the procedure is notified of its completion.

Other hidden costs are buried in the implementation of the input statement selection mechanism. These costs are, in many cases, implementation specific but must be considered when attempting to optimize the mechanisms that use them. For example, when a scheduling expression is used to order the servicing of messages to an operation, all pending messages to the operation are examined and the scheduling expression is evaluated for each message in order to find the message yielding the minimum value. SR's current run-time support system maintains the queues of pending invocations in FCFS order. Maintaining the pending invocation queues in "value of the scheduling expression" order could provide a significant reduction in the execution times of programs using such expressions. Additional discussion related to the optimization of scheduling expressions appears in Chapter 7.

Chapter 3

Optimizing Transformations

This chapter presents new and existing source-level optimizing transformations for SR's asynchronous message passing, remote procedure call, and synchronous rendezvous. These transformations significantly reduce the cost of interprocess communication and synchronization. In some cases, high-level mechanisms are replaced by less costly, low-level forms of communication. In other cases, the transformations reduce context switching and thus reduce cost. Many of these transformations are also applicable to Ada and Concurrent C.

Because SR provides an efficiently implemented semaphore operation, it is possible to effect at the source level many of the transformations of SR's high-level interprocess communication mechanisms. Also because the new transformations for rendezvous and asynchronous message passing mechanisms are particularly complex, assessing their potential performance was important. We illustrate these transformations via SR programs and we describe the results of extensive performance testing.

The performance of the new optimizing transformations is discussed throughout this chapter. Thus, we begin with a description of the tools and techniques used to measure their performance. This chapter then presents existing transformations for asynchronous message passing and remote procedure call, followed by a description of existing and new transformations for rendezvous and asynchronous message passing. Finally, we discuss the performance of the new transformations.

3.1 Measurement Tools

The following timing techniques apply primarily to the assessment of the transformations for rendezvous. Many of the optimizing transformations for rendezvous were applied by hand to SR programs with the resultant programs using the efficiently implemented semaphores. The execution times for the translated programs were compared to the execution times for the unoptimized programs. We wanted to measure the execution time associated with the implementation of high-level language features independent of the underlying costs of sending messages between processors. Therefore, for timing purposes the optimizations were applied to single resource programs only (although they can be applied, in many cases, to multiple resource programs as well).

The terms "run-time performance" and "cost" are used throughout this dissertation to mean the user execution time on a lightly loaded system. "Reduction in execution time" and "run-time reduction" both refer to the difference between the user execution times for the transformed and original programs as a percentage of the original program's user execution time. All user execution times have been adjusted to eliminate constant costs not associated with the cost of interprocess communication and synchronization (see Section 3.4.3).

The UNIX *time* command[1] was used to determine the user times for both optimized and unoptimized versions of several classical problems including *readers/writers* and *bounded-buffer*. Timing was done on a Sun 3/160, a Sun SPARCstation, an Encore Multimax, a Sequent Symmetry, and several VAXes including a 750 and an 8600. Programs run on the Encore Multimax and the Sequent Symmetry used only one processor. All systems were lightly-loaded when timing tests were run.

The run-time performance statistics presented in this chapter reflect run-time reductions for the *readers/writers* problem with from zero to one hundred readers, zero to one hundred writers, and 10,000 requests and releases per reader/writer when run on a Sun 3/160. Tests were run multiple times and with varying amounts of computation and

[1]Results using the *time* command and results using *getrusage*, a UNIX system call, were compared and found to be nearly identical.

explicitly induced context switching between reader/writer requests. The computation and induced context switches were used to simulate a random ordering of the reader/writer requests for service, i.e., `read_request`, `read_release`, `write_request`, and `write_release`. The numbers provided to illustrate the performance of our transformations are the range of all of the results over all configurations. They are also representative of the results obtained when the transformations were applied to other problems and run on the other systems. Section 3.4.3 contains more complete results and additional interpretation.

3.2 Asynchronous Message Passing Optimization

Source-level transformation of message passing operations involves the reduction of parameterless send/receive pairs to low-level semaphore operations. An operation is *semaphore optimizable* when all processes servicing and invoking the operation reside in the same shared address space. Additionally, for all invocations of the operation through a capability variable, all operations that are possibly invoked through that same capability variable must also be semaphore optimizable. This is because the invocation cannot selectively perform a semaphore operation without significant modification of the run-time support system.

Figure 3.1 presents a simple example of a semaphore optimizable operation.[2] Operation `a` is parameterless, invoked only by send statements, and implemented by a receive statement and, therefore, is semaphore optimizable.[3] Figure 3.2 illustrates the semaphore optimization of an operation, `done`, invoked through a capability variable. `cap_done` is a capability parameter (capability parameters and capability variables are similar) whose possible values are capabilities for the local operations `done` in each of the `client` processes.

The reduction of send/receive pairs to semaphore operations results in a significant reduction in user execution time. This is because the standard send/receive implementation

[2]Recall from Section 2.1.3.3 that SR semaphores are not necessarily implemented as such. The semaphores used in this optimization and others in this chapter are real semaphores.

[3]The presentation of of the semaphore optimization in Figure 3.1 is conceptual; the exact implementation syntax differs as seen in Section 5.4

```
resource simple()
  op a()
     ...
  process p(i:=1 to 10)
    ...
    send a()
    ...
    receive a()
    ...
  end

end simple
```

(a): Original code fragment

```
resource simple()
  sem a()
    ...
  process p(i:=1 to 10)
    ...
    V(a)
    ...
    P(a)
    ...
  end

end simple
```

(b): Transformed code fragment

Figure 3.1: Transformation of semaphore optimizable operations.

```
resource simple()
  op request(cap_done: cap (),...)
    ...
  process server
    ...
    do true ->
      receive request(cap_done,...)
      ...
      send cap_done()
    od
    ...
  end

  process client(i:= 1 to 10)
    op done()
    ...
    send request(done,...)
    ...
    receive done()
    ...
  end
end simple
```

(a): Original code

```
resource simple()
  op request(cap_done: cap(),...)
    ...
  process server
    ...
    do true ->
      receive request(cap_done,...)
      ...
      V(cap_done)
    od
    ...
  end

  process client(i:= 1 to 10)
    sem done()
    ...
    send request(done,...)
    ...
    P(done)
    ...
  end
end simple
```

(b): Transformed code

Figure 3.2: Transformation of semaphore optimizable operations applied to program using capability parameters.

uses generalized run-time support system routines, which are complex. To invoke the operation, memory is allocated for an "invocation block"; the invocation block is then placed on a pending queue for the operation and if processes are waiting to service the operation they are awakened [4]. To service the invocation, the run-time system removes it from the pending queue and once serviced, deallocates the memory assigned to it. In contrast, the implementation of a semaphore operation is much simpler. An invocation or V semaphore operation simply increments a counter associated with the semaphore or awakens a process if one is blocked waiting for the semaphore. Servicing a semaphore simply involves checking the semaphore's counter and then either blocking, if the semaphore is unavailable, or decrementing the counter. Atkins and Olsson report timing results for semaphore operations in SR [AO88].

The compiler for SR version 1.1 recognizes some semaphore optimizable operations via ad hoc techniques, i.e., not through source-level transformations. The operations that can be optimized are limited: they must be local to a resource, they may not be invoked through capability variables, and they cannot be passed through capability parameters. This dissertation formalizes the application of this optimization and extends its application across resource boundaries and to operations invoked through capability variables and parameters.

Additional optimizations of asynchronous message passing are addressed in Section 3.4.2.3 and Chapter 7.

3.3 Remote Procedure Call Optimization

Optimization of remote procedure call involves the identification of call invocations that invoke operations serviced in the same shared address space, i.e., that can use a local procedure call rather than the general invocation mechanism. The SR compiler currently uses ad hoc techniques to detect optimizable instances of remote procedure call when both the invoking process and the implementing proc reside within the same resource instance

[4] All processes waiting to service the operation are awakened. In some cases, it is possible to awaken only one. See the discussion of run-time support system optimizations in Chapter 7.

```
resource simple
  op g(x: int)
    ...
  process p
    ...
    call g(100)
    ...

  end

  proc g(x)
   ...
  end
end simple
```

(a): Original code

```
resource simple
  op g(x: int)
    ...
  process p
    ...
    call g(100) LOCAL
    ...

  end

  proc g(x)
    ...
  end
end simple
```

(b): Transformed code

Figure 3.3: Remote procedure call transformation applied to a simple program.

and the invocation is direct, i.e., not through a capability variable or parameter. Additional optimizable invocations are detected at run-time by the SR run-time support system, which then performs a local call in place of the standard remote call. This dissertation describes new techniques that permit compiler optimization of remote procedure call mechanisms across resource boundaries and despite the use of capability variables and parameters, thus, in many cases, relieving the run-time support system of this task. Figure 3.3 illustrates source-level transformations for this optimization. A compiler directive (LOCAL) is inserted in the program by the transformation system, directing the compiler to generate code for a simple procedure call.

Note that this optimization is occurrence specific, i.e., it is possible for some invocations of operation g to be LOCAL and thus optimizable and for other invocations to be remote and not optimizable. This is in contrast to the semaphore optimization where all invocations must be LOCAL for the operation to be optimizable.

Atkins and Olsson report timing results for remote procedure calls currently optimized in SR [AO88].

3.4 Rendezvous Optimization

3.4.1 Roberts' Transformations

A series of program transformations is presented in [REMC81] based upon Habermann and Nassi's optimizations of Ada rendezvous [HN80]. All of the transformations in [REMC81] are applied to client/server models with centralized servers, i.e., one server task accepts entries from any number of client tasks. The transformations replace server select/accept statements and their bodies by passive procedures containing the bodies. Client tasks remain unchanged; however, their calls to the Ada entries reduce to simple procedure calls when clients and server procedures share the same address space. Low-level semaphore operations are used to control execution of the procedures and ensure the semantics of Ada's select and accept statements. An SR version of this simple client/server transformation appears in Figure 3.4.

The new transformations presented in this dissertation focus on a client/server model presented in Example 4 in [REMC81]. This model provides mutually exclusive servicing of client requests for one of a number of operations via a guarded select statement. Roberts et al. present a sequence of transformations that ultimately results in a program with a server task controlling semaphores used to control entry into the procedures used to implement the accept statements. Because the server procedures must ensure that their guards are satisfied as well as obtain mutually exclusive access to their code, the procedures must wait on two semaphores. In addition, the value of other operations' guards may be altered by the servicing of an operation, necessitating the use of a joint semaphore operation for the two semaphores. The joint semaphore operation, JOINTP(x,y), simply waits until a P operation can be performed on each of its semaphore arguments, x and y, before allowing the server procedure to continue processing. An SR version of this client/server model and the transformed server appears in Figure 3.5. An Ada-like select statement with a when clause as well as a JOINTP have been introduced to preserve the original semantics of Roberts et al.'s transformation.

It is significant that the transformation outlined in Figure 3.5 does not completely eliminate the server task as the transformation in the simple client/server model does.

```
process server
   do true ->
        in oper1() -> <body1> ni
        in oper2() -> <body2> ni
        ...
        in opern() -> <bodyn> ni
    od
end server

process client
        ...
        call operi()
        ...
end client
```

(a): Simple client/server model

```
# initially
    sem1 := 1
    sem2 := 0
    ...
    semn := 0

proc oper1()
        P(sem1)
          <body1>
        V(sem2)
end oper1

...

proc opern()
        P(semn)
          <bodyn>
        V(sem1)
end opern

process client
        ...
        call operi()
        ...
end client
```

(b): Transformed server

Figure 3.4: SR version of simple client/server model and its transformed server.

```
process server
   do true ->
      select
         when cond1 ->
            in oper1(...)  ->
                    <body1>
            ni
         [] cond2 ->
            in oper2(...)  ->
                    <body2>
            ni
         ...
         [] condn ->
            in opern(...)  ->
                    <bodyn>
            ni
      end select
    od
end server
```

(a): Server in Roberts' Example 4

```
# initially
    mutex := 1
    statement_sem := 1
    op_sem1 := 0
    ...
    op_semn := 0

process server
    do true ->
        P(statement_sem)
        c1 := cond1
        c2 := cond2
        ...
        cn := condn
        if (~c1 and ~c2 and ... ~cn) ->
                raise error
        fi
        if c1 ->   V(op_sem1)    fi
        if c2 ->   V(op_sem2)    fi
        ...
        if cn ->   V(op_semn)    fi
    od
end server

proc oper1(...)
    JOINTP(op_sem1, mutex)
      <gobble other released op_sems>
      <body1>
    V(statement_sem)
    V(mutex)
end oper1

...

proc opern(...)
    JOINTP(op_semn, mutex)
      <gobble other released op_sems>
      <bodyn>
    V(statement_sem)
    V(mutex)
end opern
```

(b): Optimized server

Figure 3.5: Pseudo-SR version of client/server model in Roberts' Example 4 and its transformation.

Ada semantics specifies that all of the select statement guards be evaluated before any attempt at rendezvous is made and that a `PROGRAM_ERROR` must be raised if no guard is true [Ada83, REMC81]. As a result, the approach used for the simple client/server model is not semantic preserving in this case; specifically, there is no task to raise the `PROGRAM_ERROR`.[5]

Because guards in SR and Concurrent C can reference invocation parameters, new invocations of an operation can satisfy previously unsatisfied guards. As a result, processes simply block, rather than "raise an error", when the guards for all the arms of an input statement or a select statement evaluate to false. This subtle semantic difference allows Roberts' transformations to be applied automatically only to programs written in SR or Concurrent C.

3.4.2 New Transformations

The following transformations significantly extend those described above and are demonstrated using the familiar *readers/writers* problem, an instance of the client/server model presented in Figure 3.5. A conventional SR solution to this problem was presented in Figure 2.4.

3.4.2.1 Transformation Tools — Passing the Baton

Several of the optimizing transformations of programs using complex forms of interprocess communication, including guarded or selective synchronization, involve the application of *baton passing*, a technique originally developed by Andrews [And89, And91] as part of a method for deriving correct concurrent programs. Figure 3.6 illustrates a semaphore solution to the *readers/writers* problem obtained using this technique [And89, And91].

The `mutex` semaphore is used to provide mutually exclusive access to shared variables, and the `read_signal` and `write_signal` semaphores are used to awaken delayed readers and writers, respectively. The name "baton passing" reflects the fact that when a

[5] It can be argued that, at some level of system specification, the presence of the server task is not required. Both the original version and the transformed version that eliminates the task will satisfy the specification even though the two programs may not be semantically equivalent.

```
initially:      mutex:=1;
                nr:=0; nw:=0;
                dr:=0; dw:=0;
                read_signal:=0;
                write_signal:=0;

Reader process: P(mutex)                     Writer process: P(mutex)
                if nw = 0 ->                                 if nr = 0 and nw = 0 ->
                   skip                                         skip
                [] nw > 0 ->                                 [] nr > 0 or nw > 0 ->
                   dr++                                         dw++
                   V(mutex)                                     V(mutex)
                   P(read_signal)                               P(write_signal)
                fi                                           fi
                nr++                                         nw++
                SIGNAL                                       SIGNAL
                  read  whatever                               write whatever
                P(mutex)                                     P(mutex)
                nr--                                         nw--
                SIGNAL                                       SIGNAL
where SIGNAL is

             if nw = 0 and dr > 0 ->
                dr--
                V(read_signal)
             [] nr=0 and nw=0 and dw>0  ->
                dw--
                V(write_signal)
             [] else ->
                V(mutex)
             fi
```

Figure 3.6: Semaphore solution to *readers/writers* using the baton passing technique.

delayed reader or writer is signaled the `mutex` semaphore is not released; instead mutually exclusive access (i.e., the "baton") transfers directly to the signaled reader or writer. After completing its critical section, the signaled process must signal a still delayed process, if any is eligible to execute, or release exclusion.

In our transformations, the baton passing technique is used for a purpose entirely different from program derivation. Specifically, it is used to ensure the integrity of guarded synchronization when optimizing transformations are applied.

3.4.2.2 Rendezvous Transformations

As described in Chapter 2, SR and Concurrent C provide more powerful rendezvous selection mechanisms than Ada. Recall that synchronization expressions in SR and Concurrent C may contain references to actual operation parameters and must be evaluated for each message that the receiving statement attempts to service. The evaluation of the synchronization expressions is a function of the message selection process. In contrast, Ada select guards are evaluated before any messages are considered for servicing. As a result, they may not contain references to operation parameters in specific messages. Ada guards are not considered part of the message selection process but instead part of the code for the task itself, making it difficult to eliminate the server task [REMC81]. The synchronization expressions in SR and Concurrent C are a component of the communication mechanism itself. Therefore, their evaluation may be moved to separate procedures, thereby allowing processes to be eliminated from programs, making this more powerful mechanism unexpectedly easier than Ada's mechanism to optimize.

3.4.2.2.1 Process Elimination/Simplification of Form of Communication

Extending the technique of replacing rendezvous by a call to an inactive procedure, a server process with a loop containing an SR input statement with synchronization expressions can be replaced by passive procedures using several variants of the baton passing mechanism described in Section 3.4.2.1. These transformations significantly increase the number of communications and the complexity of the code but the communications are very low-level (semaphores) and thus much less costly. Figure 3.7 presents an SR solution to the *readers/writers* problem using the baton passing technique. Because the server process is eliminated, the joint semaphore operation present in Roberts' transformation is also eliminated. A dramatic reduction in execution time, $91 - 99\%$, is achieved.

Unfortunately, the signaling mechanism at the end of the procedures introduces a degree of nondeterminism (due to the nondeterministic semantics of SR's if statement). The nondeterministic servicing of delayed processes violates SR's FCFS servicing of invocations to operations serviced by the same input statement but is perfectly acceptable in Ada and Concurrent C where the semantics of the select statement specifies nondeterminism. The transformed procedures guarantee FCFS servicing upon entry assuming that the entry semaphore has a FCFS servicing policy, which it does. However, processes that enter a procedure and find that they are blocked waiting for a signal are not guaranteed to be awakened in the necessary FCFS order due to the nondeterminism inherent in the multi-armed if statement. To adhere to SR's FCFS servicing policy, delayed readers and writers must be signaled in the order in which they are delayed (to the extent possible given the synchronization constraints of the *readers/writers* problem). One approach to solving this problem is to maintain a list of blocked processes so that delayed processes may be awakened and allowed to continue when conditions permit. Mutually exclusive access to the list is required along with routines to insert into the list and delete from the list in FCFS order as constrained by the guards in the signaling mechanism. Since the SR input and send statements already perform these activities, it is expedient to use them to maintain such a list [Ols90]. This appears to be a step backward and, in fact, it is, but the input statement used in the signaling mechanism with only two operations is simpler than the original input statement in Figure 2.4, which services requests for four different operations. In addition, the input statement services not all requests, but only delayed requests. For the *readers/writers* problem, the difference between the total number of requests and the number of delayed requests is especially significant when the number of readers is much greater than the number of writers. Because the input statement is a higher-level communication mechanism than the semaphores used in the transformation in Figure 3.7, there is a performance penalty, but the run-time reduction is still significant, ranging from $51 - 97\%$. A transformation with FCFS servicing of delayed requests appears in Figure 3.8.

```
# initially
    mutex := 1
    read_signal := 0
    write_signal := 0

 resource rw()
       ...
     proc read_request()
         P(mutex)
         if nw > 0 ->
            dr++
            V(mutex)
            P(read_signal)
         fi
         nr++
         SIGNAL
     end read_request
       ...
     proc read_release()
         P(mutex)
         nr--
         SIGNAL
     end read_release
       ...
     process reader(n:=1  to noreaders)
                ...
             read_request()
                ...
             read_release()
                ...
     end reader
                ...
 end rw

where SIGNAL is:
         if nw=0 & dr>0 ->
             dr--
             V(read_signal)
          [] nr=0 & nw=0 & dw>0 ->
             dw--
             V(write_signal)
          []  else ->
             V(mutex)
         fi
```

Figure 3.7: *Readers/writers* with rendezvous replaced by calls to inactive procedures. Procedures `write_request` and `write_release` are similar to those for the read operations.

```
# initially
    mutex := 1
    read_signal := 0
    write_signal := 0

resource rw()
            ...
    op delay_read_signal()
    op delay_write_signal()
            ...
    proc read_request()
        P(mutex)
        if nw > 0 ->
            dr++
            V(mutex)
            send delay_read_signal()
            P(read_signal)
        fi
        nr++
        SIGNAL
    end read_request

    proc read_release()
        P(mutex)
        nr--
        SIGNAL
    end read_release
        ...
    process reader(i:= 1 to noreaders)
                ...
            read_request()
                ...
            read_release()
                ...
    end reader
                ...
end rw

where SIGNAL is:
        in delay_read_signal()  & nw=0  ->
            dr--
            V(read_signal)
        [] delay_write_signal()  & nr=0 & nw=0  ->
            dw--
            V(write_signal)
        [] else ->
            V(mutex)
        ni
```

Figure 3.8: Transformed code that preserves the FCFS semantics of SR. Procedures `write_request` and `write_release` are similar to those for the read operations.

3.4.2.2.2 Simplification of Communication Form

A somewhat counter-intuitive approach — i.e., increasing the number of processes but simplifying the communication mechanism — also produces a significant improvement in performance. Replacing each arm of an input statement with an active process and using semaphores to ensure mutual exclusion gives a run-time reduction of $72 - 94\%$ if nondeterminism is acceptable, and $64 - 87\%$ if FCFS semantics is required.

This transformation appears in Figure 3.9. Calls to operations are transformed into V/P pairs. Because the calling processes access the semaphores directly, all tasks must share the same memory. If the operations are guarded, then a baton passing mechanism or another synchronization technique is required to ensure correctness and fairness.

Interestingly, the increase in the number of processes results in only a relatively small increase in the total number of forced context switches. (Recall from Section 2.5 that a forced context switch is a context switch necessary to complete a communication or synchronization operation.) This is because the number of context switches needed to perform each interprocess synchronization does not increase. In the original code in Figure 2.4, readers and writers were forced to synchronize with the same server process. In the transformed code, readers and writers synchronize with different processes, but do so the same number of times. The small increase in the total number of forced context switches that was observed reflects the fact that pending sequences of `read_request` and `read_release` operations are serviced by the same process in the original program and by separate processes in the transformed program, necessitating the additional context switching. Obviously, the performance of this transformation is dependent upon the relative cost of rendezvous versus the cost of context switching. Context switching has been shown to be substantially less expensive than rendezvous in both SR and Ada [AO88, CDV$^+$86]. As in the case above, the signaling mechanism of Figure 3.8 may be used to preserve FCFS semantics.

3.4.2.2.3 Simplification of Communication Complexity

Yet another approach to optimizing interprocess communication mechanisms involves simple reductions in the complexity of a mechanism. For example, eliminating syn-

```
# initially
    mutex := 1
    read_signal := 0
    read_request := 0
    read_release := 0
    oktoread := 0
    read_release_ok := 0

resource rw()
                ...
    process read_begin
        do true ->
            P(read_request)
            P(mutex)
            if nw > 0 ->
                dr++
                V(mutex)
                P(read_signal)
            fi
            nr++
            V(oktoread)
            SIGNAL
        od
    end read_begin

    process read_end
        do true ->
            P(read_release)
            P(mutex)
            nr--
            V(read_release_ok)
            SIGNAL
        od
    end read_end
        ...
    process reader(i:=1 to noreaders)
                ...
            V(read_request);   P(oktoread)
                ...
            V(read_release);   P(read_release_ok)
                ...
    end reader
                ...
end rw
```

Figure 3.9: Transformation of operations into active processes using semaphores to control progress of processes. SIGNAL is as it is in Figures 3.7 and 3.8.

chronization expressions and/or reducing the number of arms in an input statement provides some improvement in performance. It is possible to do so using problem-specific information not apparent from the syntactic form of the code. Figure 3.10 gives a transformation of the *readers/writers* problem that replaces the input statements of Figure 2.4 by several less complex input statements. The input statements have fewer arms and all operation requests for servicing are not placed on the same pending invocation queues; thus they run more efficiently. Run-time reductions of $0-82\%$ were obtained on the Sun 3/160; slight performance losses, however, were observed on some machines for the case with many readers and no writers. When the number of readers greatly exceeds the number of writers the number of queued requests for service is small, as is the number of blocked processes. This results in efficient servicing of invocations despite the number of arms.

The above transformation is independent of execution environment. It can be applied to programs running in either distributed or multiprocessor environments. A variety of other transformations similar to the transformation in Figure 3.10 is currently under investigation. For example, it is possible to nest the input statements thereby giving enough context to eliminate the synchronization expressions. However, most of the transformations of this type suffer from the fact that they require problem-specific semantic information to apply the transformation, making automatic transformation unreasonable. The use of programmer-supplied directives and/or assertions would facilitate the application of these transformations. This issue is revisited in Chapter 7.

3.4.2.3 Asynchronous Message Passing Transformations

3.4.2.3.1 Simplification of Communication Form

A process containing an input statement with multiple arms that service asynchronous sends can be replaced by multiple processes, one for each arm, with semaphores controlling entry and mutual exclusion to improve performance. This transformation is similar to the multiple process transformation of synchronous communication primitives. To illustrate this approach, consider again the original *readers/writers* problem from Figure 2.4, but with synchronous calls to `read_release` and `write_release` replaced by asynchronous sends. The readers/writers solution in Figure 3.11 illustrates the transformation of asyn-

```
resource rw()
       ...
    op read_request();         op write_request()
    op read_release();         op write_release()
       ...
    process server
        do nr = 0 & nw = 0 ->
            in read_request() -> nr++
            [] write_request() -> nw++
            ni
        [] nr>0 ->
            in read_request() -> nr++
            [] read_release() -> nr--
            ni
        [] nw > 0 ->
            in write_release() -> ni
            nw--
        od
    end server

    process reader(n:=1 to noreaders)
              ...
            read_request()
              ...
            read_release()
              ...
    end reader

    process writer(n:=1 to nowriters)
              ...
            write_request()
              ...
            write_release()
              ...
    end writer
end rw
```

Figure 3.10: Transformation into multiple input statements, eliminating the need for synchronization expressions.

chronous primitives in conjunction with the transformation of synchronous calls to inactive procedures. Asynchronous sends to `read_release` and `write_release` are transformed to semaphore communications with active processes, providing a $91 - 97\%$ reduction in execution time. Again, true FCFS semantics can be reproduced but with some performance degradation. A $44 - 86\%$ reduction in execution time is obtained.

3.4.3 Performance and Interpretation of Baton Passing Transformations

The performance of the transformations presented in Sections 3.4.2.3 and 3.4.2.2 is summarized below. The transformations were applied to the *bounded-buffer* problem with results similar to those obtained for the *readers/writers* problem.

Recall from Section 3.1 that the *readers/writers* problem and its transformed variants were run on six different systems varying the number of readers, the number of writers, and the amount of work and context switching done by each reader/writer process between requests to read and release. The specific work done between reader/writer requests for service varied from nothing, to a random number of explicitly induced context switches, to a random number of numerical computations along with a random number of explicit induced context switches. Base cases reflecting constant costs unrelated to the costs of the interprocess communication and synchronization of the reader/writer processes were timed and their results were subtracted from the times for the conventional and transformed programs before computing run-time reduction percentages. The range of run-time reduction percentages is shown in Table 3.1.

The slightly smaller percentages observed on the SPARC are due to the use of Sun's lightweight process library package in that particular implementation of SR. The handling of processes by Sun's general package is less efficient than that provided by the specialized routines in SR's run-time support on the other systems.

The run-time reductions vary considerably depending upon the number of readers versus writers. The effect of varying the number of readers and writers can be seen in Figure 3.12 where the total number of reader/writer processes was held constant at one hundred but the number of readers and writers each varied from zero to one hundred. The results shown were obtained on a Sun 3/160 but are again representative of results obtained

```
# initially
    mutex := 1
    read_signal := 0
    read_release := 0
    ...

resource rw()
       ...
    proc read_request()
        P(mutex)
        if nw > 0   ->
            dr++
            V(mutex)
            P(read_signal)
        fi
        nr++
        SIGNAL
    end read_request
       ...
    process read_end
        do true ->
            P(read_release)
            P(mutex)
            nr--
            SIGNAL
        od
    end read_end
       ...
    process reader(i:=1 to noreaders)
               ...
            read_request()
               ...
            V(read_release)
               ...
    end reader
       ...
end rw
```

Figure 3.11: Transformations of synchronous and asynchronous communication primitives.

Table 3.1: Run-time reductions for optimizing transformations applied to the *readers/writers* problem.

on the other systems. The differences are largely due to the increased costs associated with high-level synchronization mechanisms. Specifically, when the number of writers is much greater than the number of readers and many requests for service are queued and their processes blocked, the costs associated with servicing requests from these queues is significant.

Another factor affecting the results is the somewhat naive approach to determining the base case where the number of unblocked processes, i.e., those that are ready to run, is always equal to the number of readers plus the number of writers. With the presence of interprocess synchronization in the actual reader/writer solution, the number of unblocked processes varies depending upon the state of other reader/writer processes. Surprisingly, run-time reductions exceeding 100 percent were observed for some of our optimizations when they were timed with large numbers of writer processes, i.e., the transformed program ran in less time than the base case. This anomalous behavior was observed only on the VAXes, the Encore Multimax, and the Sequent Symmetry. It can be explained as follows. In some cases, when write permission is granted to a writer process, all other processes are blocked and our timing tests perform a significant number of context switches from a writer process to itself, rather than to other unblocked processes. Given unoptimized code, a context switch should take constant time. That is, a context switch from one process to itself should take the same amount of time as a context from one process to another process. However, after additional testing and consultation with architecture experts, it was determined that the effect of data caching present in these machines speeds up the context switch from a process to itself, thus explaining the seemingly impossible results. In general, the presence of instruction and data caches as well as instruction pipelining in modern architectures make concise timing of programs impossible. (Weicker reports similar findings in his overview of common benchmarks [Wei90].) However, the magnitude of our run-time reductions on a variety of architectures is much greater than any possible variation due to a specific architecture.

The performance of the optimizing transformations when applied to the *bounded-buffer* problem is slightly less dramatic than for the *readers/writers* problem. Figure 3.13 illustrates the performance of the transformations when the number of producers and con-

Figure 3.12: Effect of varying the number of readers/writers on the performance of the optimizing transformations.

sumers varies from one to fifty and the total number of producers and consumers is either fifty or fifty-one. The buffer size is held constant at one hundred[6] and the number of items passed into and out of the buffer is held constant at 30,000.

The *bounded-buffer* problem requires more than the simple synchronization present in the *readers/writers* problem: the operations use parameters to deposit data into and fetch data out of the buffer. While the performance of the transformation with inactive procedures and baton passing should still be effective, the presence of the operation parameters makes the transformation to active processes less desirable than it is for the *readers/writers* problem. Examination of Figure 3.13 confirms this. The transformation to inactive procedures for synchronous communication (Baton Passing in Figure 3.13) gives good performance. The peak performance observed when the numbers of producers and consumers are both twenty-five is due to the fact that with nearly equivalent supply and demand for the buffer, processes are delayed less often and the transformed programs execute more quickly. This behavior is also observed in the transformation to both inactive procedures, used for a synchronous fetch, and active processes, used for an asynchronous deposit (Asynch release in Figure 3.13). The performance of the transformations using active processes only (Multiple process and Multiple process(FCFS)) is, as expected, not as good as it is when applied to programs doing synchronization only. However, its performance in a true multiprocessor environment may yield better results. The lack of symmetry in the performance of the transformations using both inactive procedures for synchronous communication and active processes for asynchronous communication (Asynch release and Asynch release(FCFS)) is a result of a performance degradation in the original program caused by unconstrained asynchronous requests made by the producers. Unconstrained asynchronous requests can result in unreasonable memory requirements and long pending invocation queues, i.e., a flow-control problem [GR89]. The performance of the transformation that eliminates synchronization expressions is similar to that observed for the *readers/writers* problem.

---

[6] Similar run-times were obtained for programs with buffer sizes significantly less than the total number of items passed into and out of the buffer.

Figure 3.13: Effect of varying the number of producers/consumers on the performance of the optimizing transformations.

3.4.4 General Form for Rendezvous Transformations

The source-level transformations for rendezvous and asynchronous message passing presented in this section have been applied to SR programs. Although the examples presented use only a single resource, they can be applied to multiple resource programs as well. In the case of the client/server model, clients may reside in different resources, and assuming that the multiple resources reside in the same shared address space, the performance of the transformations would be the same as for those applied to single resource programs. Additionally, although these transformations are intended for use in a shared memory environment, they are also semantically valid in a distributed environment when all of the inactive procedures or active processes used to implement the server are located (as is reasonable) on one processor. The calls to the procedures from the client tasks simply become remote procedure calls. Whether or not these transformations to inactive procedures provide any reduction in execution time in a distributed environment depends upon the relative cost of rendezvous and remote procedure call. Certainly no significant reduction in overall execution time will be achieved as the dominant cost is associated with the communication between machines rather than with the calling and servicing mechanisms at either end of the communication.

As stated earlier, these transformations are applicable to SR, to Concurrent C, and in some cases, to Ada. Ada's semantics for guarded selection prohibits elimination of the server process, making these transformations inappropriate as an automatic compiler optimization. However, they are still appropriate, in some cases, as high-level source-to-source transformations for Ada. Concurrent C, being similar to SR, is also a candidate for automatic optimization. Ada and Concurrent C both have a nondeterministic selection policy making their transformations more efficient than those implementing SR's FCFS selection policy.

Guarded operations with parameters may be transformed as long as their synchronization expressions are independent of operation parameters. This restriction is relevant for SR and Concurrent C, but not Ada where guards may not reference operation parameters.

Automation of these transformations is possible for the "baton passing" transformations and the general form for the rendezvous transformation is given in Figure 3.14. An n-operation input statement embedded within an infinite loop is transformed into n procedures, each requesting the semaphore mutex and each checking the conditions associated with their operation in the original server. The delaying mechanism when the operation's condition cannot be met is straightforward as is the signaling mechanism present at the end of each procedure. The general form for asynchronous message passing transformation is similar. Whether these transformations are applicable and how they are applied can be determined solely by syntactic analysis of the program being optimized.

Unfortunately, the optimizing transformations valid in both distributed and shared memory environments (i.e., similar to the one shown in Figure 3.10) are not easily automated as they require problem-specific semantic information.

3.5 Summary

A repertoire of source-level transformations is described. The performance of these transformations provides clear motivation for the development of techniques that permit their automatic application. These techniques are discussed in the following chapters.

```
process server
    do true ->
        in oper1(...) & cond1 ->
               <body1>
        [] oper2(...) & cond2 ->
               <body2>
        ...
        [] opern(...) & condn ->
               <bodyn>
        ni
    od
end server


process clienti
    ...
    operj(...)
    ...
end clienti
```

(a): Generalized server and client

```
# initially
    mutex:=1
    signal1:=0
      ...
    signaln:=0
    d1:=0
      ...
    dn:=0


proc oper1(...)
    P(mutex)
    # delay awaiting cond1
    if not(cond1) ->
        d1++
        V(mutex)
        P(signal1)
    fi
    <body1>
    # signal delayed processes
    if cond1 & d1>0 ->
        d1--
        V(signal1)
    [] cond2 & d2>0 ->
        d2--
        V(signal2)
    []  ...

    [] condn & dn>0 ->
        dn--
        V(signaln)
    [] else ->
        V(mutex)
    fi
end oper1

       ...
```

(b): Optimized server

Figure 3.14: Generalized transformation using the baton passing technique.

Chapter 4

Transformation Requirements Analysis

Chapter 3 presents source-level transformations that improve the performance of high-level language mechanisms for interprocess communication and synchronization. This chapter motivates and formally characterizes the analysis necessary to determine, statically and automatically, the applicability of those optimizing transformations. First, we motivate our analysis requirements by means of example SR programs that utilize a variety of language features. Then, we characterize the transformation requirements more formally using set theory. These formal characterizations provide the basis for the automated application of the transformations as seen in Chapter 5.

For all but trivial programming languages, static determination of most run-time program behavior is an undecidable problem. However, in many cases, "conservative" or "safe" estimates of run-time behavior can be used to perform the analysis necessary to enable program optimization.[ASU86] Two fundamental kinds of program behavior are involved in our analysis: *how* operations are serviced and invoked, and *where* operations are serviced and invoked. In general, the ability to determine how and where operations are serviced and invoked depends upon the features available in the language being analyzed. For example, in languages with a static process structure and static communication channels, static program analysis can provide good approximations to this information. In languages with a dynamic process structure and dynamic communication channels, such as SR, static program analysis is, in some cases, not possible.

Static analysis of SR programs can approximate *how* operations are serviced and invoked. SR operations are created dynamically only as part of resource or process creation and all dynamically created instances of an operation are simply instances of the same operation pattern. That is, SR programs provide a template for operations, and their characterization — including name, type, how they are serviced, and a conservative approximation to how they are potentially invoked — can be determined statically.[1]

In contrast, SR programs can create instances of virtual machines and resources and their associated processes at run time; and thus, static analysis of SR programs cannot determine *where* operations are serviced and invoked, nor can it provide even a conservative approximation to this information. Fortunately, the application of our optimizing transformations can use a less demanding requirement. It does not matter on *which* virtual machine invoking and servicing processes reside, only that they reside on the *same* machine. SR programs do provide a template for virtual machine and resource creation that makes it possible to identify patterns guaranteeing that resource instances and their associated processes will execute within the same virtual machine, although we cannot identify what virtual machine that might be.

We illustrate these issues through the presentation of actual SR programs that include concurrent and distributed features. We first address the issue of how operations are serviced and invoked by examining the analysis necessary for the optimization of single virtual machine programs. These programs, by definition, execute in a single address space so it is not necessary to deal with the harder "where" issue. Next, we present programs that potentially execute on multiple virtual machines. These programs are significantly more difficult to analyze and we discuss the issues involved. Finally, this chapter formalizes, using set theory, the constraints that must be satisfied for correct application of the various optimizations described in Chapter 3, including the semaphore optimization, the remote procedure call optimization, and the client/server transformations using both rendezvous and asynchronous message passing.

[1] A similar observation concerning static analysis of concurrent programs has been made by McDowell [McD89].

4.1 Single Virtual Machine Programs — Determining How Operations are Serviced and Invoked

In SR programs executing solely within a single virtual machine, all processes execute in the same shared memory environment and thus all interprocess interactions are, by definition, between processes executing within the same address space. The optimizability of these programs depends primarily on "how" operations are serviced and invoked.

How operations are serviced can be detected through straight-forward syntactic analysis. However, for programs using capability variables, how operations are invoked requires dataflow analysis to determine the set of values that the variables may hold when used to invoke an operation. Additionally, programs passing operation capabilities across process boundaries through capability parameters require an extension of interprocedural analysis to interprocess analysis. Programs passing operation capabilities across resource boundaries through resource capability parameters require interresource (i.e., intermodule) analysis. The remainder of this section presents example programs executing within a single virtual machine; they illustrate the problems involved in determining how operations are serviced and invoked.

4.1.1 How Operations are Serviced

The SR language requires that all operations be implemented directly through their declared names. As a result, static syntax analysis is sufficient for determining how operations are serviced. For example, in the program in Figure 4.1, operations `a` and `b` are both parameterless and implemented by simple receive statements and, thus, both appear to be semaphore optimizable (assuming they are also invoked by only send statements). Operation `b`, however, is also implemented by a multiple-arm input statement along with operation `c` making it ineligible for semaphore optimization. A single pass through the source program can provide lists of parameterless operations, as well as those operations implemented only with simple receive statements (or the receive statement's input statement equivalent).

Similarly, operations potentially eligible for the remote procedure call optimization

```
resource simple
body simple()
    op a()
    op b()
    op c(x: int)
    ...
    process p1
        receive a()
        receive b()
        in b() ->
            ...
        [] c(x) ->
            ...
        ni
    end p1
end simple
```

Figure 4.1: Program with operations implemented by receive and input statements.

can be determined statically. In Figure 4.2, both operations a and b are implemented by procs. Thus, it appears that any call statement invoking these operations can be optimized to a local procedure call. However, recall from Section 2.1.3.3.2 the semantics of the reply statement. When a reply statement within a proc is executed by the servicing process, the process invoking the proc is notified that its invocation is complete and the process servicing the proc then continues to execute in parallel with the invoking process. Therefore, the occurrence of a reply statement within proc b makes any call invocation of that operation ineligible for optimization as a local procedure call. As a result, only the call invocation to operation a may be optimized to a simple procedure call. Simple syntactic analysis can provide a list of operations implemented by procs containing no reply statements. This list contains those operations whose remote procedure call invocations are potentially optimizable to local procedure calls.

As yet another example, Figure 4.3 illustrates a potentially optimizable client/server program using rendezvous. The criteria for this optimization include some complex syntactic constraints, as well as a determination of how and where operations are invoked and implemented. The syntactic constraints are as follows. Optimization of operations using

```
resource simple
body simple()
    op a()
    op b(x: int)
    ...
    proc a()
          ...
    end a

    proc b(x)
          ...
        reply
          ...
    end b
end simple
```

Figure 4.2: Program with operations implemented by procs.

rendezvous in a client/server model requires that the operations be implemented within a process containing an input statement executing within an infinite loop. The infinite loop may be preceded by zero or more program statements. The infinite loop may not contain an exit mechanism (an exit statement in SR). Because the reachability of the infinite loop must be ensured, we restrict our infinite loops to those not embedded within another program statement. Similarly, we must guarantee the reachability of the input statement within the infinite loop, thus the input statement cannot be embedded within another statement. It can, however, be preceded or followed by zero or more program statements. These restrictions are reflected in Figure 4.3 where the do statement is preceded by an unspecified list of statements and the input statement appears directly within the do statement between two unspecified lists of statements.

Additional constraints apply more specifically to the input statement itself. Recall from Chapter 3 that the input statement may be multi-armed and that the arms may use synchronization expressions. For this particular optimization, however, these synchronization expressions may not reference operation invocation parameters. This restriction is necessary because the client/server optimizing transformations presented in Chapter 3

```
resource simple
body simple()
    op a()
    op b(x: int)
    ...
    process server
        statements
        do true ->
            statements
            in a()  and  exp1 -> ...
            [] b(x)  and  exp2 -> ...
            ni
            statements
        od
    end server

    process client(i:= 1  to n)
            call a()
            ...
            call b(i)
    end client
end simple
```

Figure 4.3: Program with operations implemented by an input statement within an infinite server loop.

provide FCFS servicing of pending invocations for any given operation. When synchronization expressions reference specific invocation parameters, pending invocations are not necessarily serviced in FCFS order and, as a result, these optimizing transformations would no longer be semantic preserving. For the same reason, input statements using scheduling expressions are also ineligible for these optimizations.

Syntactic recognition of optimizable client/server programs is significantly more difficult than recognition of simple receive statement or proc implementations of operations as described above. Syntax tree pattern matching is required to provide a list of potentially optimizable input statements. The pattern matcher must recognize only those input statements directly embedded within an infinite loop within a single process and containing no scheduling expressions or synchronization expressions referencing operation invocation parameters. In addition, the code used to implement the operations must not contain a reply statement.

4.1.2 How Operations are Invoked

Having described the syntactic analysis necessary to determine how operations are implemented, we now describe the analysis necessary to determine how operations are invoked. In general, syntactic analysis is not adequate for this problem and we motivate the need for dataflow analysis. We first consider single resource programs and then discuss the extension of our analysis to multiple resource programs.

4.1.2.1 Single Resource Programs

Single resource programs are those programs that execute within a single resource instance. Many of these programs are easy to detect statically because they lack any statements performing explicit resource creation. In other cases, it is not possible to detect this situation. Figure 4.4 presents one such program. Static analysis may be inadequate to determine that the value of expression *exp* can never be true and, thus, that the create statement will never be executed. Because we are focusing our efforts on commonly occurring patterns, the analysis of this type of program is not considered. The rest of this section presents programs that illustrate optimization of single resource programs.

```
resource r
body r(x,y: cap ())
    ...
end r

resource simple
    import r
body simple()
    op a()
    op b()

    initial
        ...
        if exp ->
            r1 := create r(a,b)
        fi
    end initial
    ...
end simple
```

Figure 4.4: Potentially unrecognized single resource program.

4.1.2.1.1 Programs Using Direct Naming of Operations

The simplest case to be considered is programs using direct invocation of operations only. In these programs, syntactic analysis is adequate for determining the applicability of our optimizing transformations. For example, the program in Figure 4.5 requires only simple syntactic analysis to determine that operation `a` is semaphore optimizable because it is both parameterless and resultless, its implementation uses a simple receive statement, and all invocations of `a` use the send statement (see Section 3.2). The same simple analysis also can determine that the operation `b` is not semaphore optimizable because not all of its invocations use the send statement, e.g., `b` is invoked by a call statement in process `p1`. Note that process `p1` is a template for the many instances of `p1` that exist at run-time, but that static analysis of the template characterizes all invocations of operations `a` and `b` in all run-time instances. Because `simple` is a single resource program, all invocations and implementations of all operations are on the same virtual machine.

The required static analysis can be performed by maintaining lists of invocations and implementations (i.e., receive, input statement, or proc as described in the previous

```
resource simple
body simple()
    op a()
    op b()

    process p1(i:= 1  to 10)
        send a()
        send b()
        call b()
    end p1

    process p2
        do true ->
            receive a()
            receive b()
            receive b()
        od
    end p2

end simple
```

Figure 4.5: Single resource program using direct naming of operations with a semaphore optimizable operation.

section) in addition to examination of operation declarations.

Similar analysis of the program in Figure 4.6 reveals a possible optimization of the remote procedure call mechanism. Because the program is a single resource program, the invocations and implementations of operations `a` and `b` are all on the same virtual machine. In addition, the implementations of operations `a` and `b` are procs and both operations are directly invoked by call statements reflecting the invocation of the general remote procedure call mechanism. It appears that both call invocations can be optimized to a local call. However, recall the discussion of reply statements in the previous section. The occurrence of a reply statement within proc `b` makes any call invocation of that operation ineligible for optimization as a local procedure call. As a result, only the call invocation to operation `a` may be optimized to a simple procedure call. Again, static program analysis that determines operation implementations and lists of invocations can provide the information necessary to determine the optimizability of remote procedure calls in programs using these features. Unlike the semaphore optimization, which requires that *all* occurrences of an operation's

```
resource simple
body simple()
    op a(n: int)
    op b(n: int)

    initial
        call a(100)
        call b(5)
    end initial

    proc a(n)
        write(n)
    end a

    proc b(n)
        reply
        send a(n)
        write(n)
    end b

end simple
```

Figure 4.6: Single resource program with an optimizable remote procedure call.

```
resource simple
body simple()
    ...
    proc p1()
        op f()
        var c:  cap ()
        ...
        c:= f
        send c()
        ...
    end a
    ...
end simple
```

(a): Local capability variable

```
resource simple
body simple()
    var c:  cap ()
    ...
    proc p1()
        op f()
        ...
        c:= f
        send c()
        ...
    end a
    ...
end simple
```

(b): Global capability variable

Figure 4.7: Programs with local operations invoked through capability variables.

invocations be optimizable, the application of the remote procedure call optimization is occurrence-specific — some invocations can be optimized while others cannot.

4.1.2.1.2 Programs Using Capability Variables

The optimizations described in Chapter 3 can also be applied to many single resource programs that use capability variables. For programs that invoke operations through capability variables, the values that the capability variables may assume need to be determined. Because capability variables may be assigned one of a discrete, compile-time determinable set of operations, conventional dataflow techniques can be used. For a capability variable local to a process, traditional dataflow analysis is sufficient. Consider, for example, the program in Figure 4.7(a). The value of local capability variable `c` at the point of the send statement can be determined to be a capability for the local operation `f`. In contrast, for a shared capability variable, its possible values include all values assigned to it in all processes within the variable's scope. In Figure 4.7(b) the value of global capability variable `c` at any point in the execution of the program includes a capability for any local operation `f` associated with any instance of a process servicing an invocation of the proc `p1`. Since processes execute in an unknown order and multiple instances of processes invoking proc `p1`

```
op g(), h()

process p1(i:= 1  to 10)
    var c:  cap()
      ...
     c := g
    send c()
      ...
     c := h
    call c()
      ...
end p1

process p2
      ...
   receive g()
      ...
   receive h()
      ...
end p2
```

(a): Local capability variable

```
op g(), h()
var c:  cap()

process p1(i:= 1  to 10)
      ...
     c := g
    send c()
      ...
     c := h
    call c()
      ...
end p1

process p2
      ...
   receive g()
      ...
   receive h()
      ...
end p2
```

(b): Global capability variable

Figure 4.8: Code fragments invoking operations through capability variables.

may exist, we assume that a shared capability variable can hold any value assigned to it by any process at any time. This *shared value assumption* is a conservative and safe approach to approximating the values that a shared variable may assume.

Notice that the procs in Figures 4.7(a) and (b) are templates representative of the behavior of all possible instances of the proc `p1`, similar to the processes in Figure 4.5. However, unlike the processes in Figure 4.5 where the number of instances is bounded, there may be an arbitrary number of processes executing the proc `p1`. In both cases, analysis of the "template" is sufficient to characterize the behavior of all run-time instances.

As a second example of the shared value assumption, consider the code fragment presented in Figure 4.8(a). Since `c` is a local variable (i.e., each instance of `p1` gets its own copy of `c`), the send statement invokes `g` and the call statement invokes `h`. Provided that `g` meets the other requirements for semaphore optimizability, it can be so optimized. However, `h` cannot be optimized since it is invoked by a call statement.

Suppose that `c` is now declared as a shared variable, as in Figure 4.8(b). Since multiple instances of `p1` can co-exist, the value of `c` at the call (or send) statement can be either `g` or `h`. Hence, neither operation is semaphore optimizable. This is another example of conservative approximations, as there are at least two situations when the value of `c` is either specifically `g` or `h`, even when `c` is a global variable. One such situation occurs when the instances of process `p1` execute in serial order, e.g., when instances of `p1` synchronize with each other, perhaps through a global semaphore operation. A second situation occurs when the behavior of a given process is tied to a given process identifier as in the code below,

```
if i=3 ->
    c:= g
    send c()
fi
```

Our analysis currently does not extend to these optimizable cases.

4.1.2.1.3 Interprocess Analysis

Additional analysis — interprocedural or interprocess analysis — is required when capabilities for operations are passed as parameters. For example, consider the code fragment in Figure 4.9. `p1` passes a capability for operation `f` to `p2` through operation `q`. Since `p2` (potentially) calls `d`, the operation `f` is (potentially) called, making it ineligible for the semaphore optimization. It may, however, be eligible for remote procedure call optimization depending upon the implementation of `f` and other operations potentially invoked by the call invocation via capability parameter `d`. For example, if `f` is implemented by a proc with no reply statements and the proc resides in the same address space as the call invocation, then the call invocation can be optimized to a local call.

4.1.2.1.4 Arrays of operations

Programs using arrays of operations may also be examined for potential optimization. In Figure 4.10, for example, the entire array of operations can be optimized to an array of semaphores since each element meets the requirements for semaphore optimization.

```
op f(), q(d: cap ())
process p1
      ...
    send q(f)
      ...
end p1

process p2
      ...
    var d: cap ()
    receive q(d)
    call d()
      ...
end p2
 ...
```

Figure 4.9: Code fragment invoking operations through a capability parameter.

```
resource simple
body simple()
    const max := 20
    op a[1:max]()

    initial
        send a[1]()
    end initial

    process p1(n:= 1 to max)
        receive a[n]()
        ...
        if n<max ->
            send a[n+1]()
        fi
        ...
    end p1

end simple
```

Figure 4.10: Program with an array of semaphore optimizable operations.

```
resource simple
body simple()
    const max := 5
    op a[1:max]()
    op b[1:max]()

    initial
        var x: int
        send a[1]()
        send a[2]()
        send a[3]()
        send a[4]()
        call a[5]()
        ...
        call b[x]()
        ...
    end initial

    process p1(n:= 1  to max)
        receive a[n]()
        ...
    end p1

    process p2(m:= 1  to max)
        receive b[m]()
        ...
    end p2

end simple
```

Figure 4.11: Program where some, but not all, array operations are semaphore optimizable.

In the event that not all invocations to specific array elements are eligible for optimization, it may be possible to optimize some but not all operation elements. This situation is shown in Fig 4.11 where operations `a[1]` through `a[4]` can be optimized but `a[5]` cannot be. In addition, the call invocation to `b[x]` where `x` is some integer variable makes the optimization of any of the elements of `b` impossible unless it is possible to establish some constraints on the value of `x`. If `x` is a local variable, it may be possible to determine the set of possible values through traditional dataflow analysis. However, if `x` is a shared variable, the set of its possible values will be all values assigned to `x` in all processes (by the shared value assumption). In general, it is not clear that the interprocess dependency analysis necessary for the optimization of programs similar to that in Figure 4.11 is warranted. Global vari-

```
resource simple
    op a()
    op b()
body simple()
    process simpler
        receive a()
        receive b()
    end simpler

end simple

resource main
    import simple
body main()
    var caps:  cap simple
    initial
        fa i:= 1  to 10 ->
            caps:=  create simple()
            send caps.a()
            call caps.b()
        af
    end initial

end main
```

Figure 4.12: Single virtual machine, multiple resource program with direct invocation of operations across resource boundaries.

ables used as indices into arrays of operations is perceived by the author to be a form of language abuse not worthy of optimization efforts. See Chapter 7 for further discussion.

4.1.2.2 Multiple Resource Programs

This section describes the type of analysis necessary to perform optimizations across resource boundaries on programs that execute within a single virtual machine. Analogous to the analysis for single resource programs (Section 4.1.2.1), it is easy to determine statically many of the programs that will execute within a single virtual machine because they contain no program statements creating virtual machines. For example, Figure 4.12 shows a program with many resource instances all residing on the same virtual machine. Interresource analysis is required to determine that operation `a` can be replaced by a semaphore operation and that operation `b` cannot. Once again, the concept of a template is important. Many instances of resource `simple` are created at run-time, but static analysis of the program template provides sufficient information for optimization.

Interresource analysis must accommodate three possible situations. The simplest case is when a process executing within an instance of one resource directly invokes an operation implemented inside another resource. The operation name is qualified by a capability for the resource instance, as shown in Figure 4.12. Because resource `simple` is imported by resource `main`, the operations in `simple`'s specification may be invoked directly by `main`.

The second situation that must be considered is when capabilities for operations are passed to resources as parameters during resource creation. This situation is illustrated by the program in Figure 4.13. Capabilities for operations `j` and `k` are passed to resource `simple` at resource creation. These operations are then invoked through `simple`'s resource parameters `y` and `z`. Assuming that the other conditions for semaphore optimizability are met, it appears that both `j` and `k` are optimizable. We see below, however, that only operation `k` is actually semaphore optimizable.

The third situation that must be considered is when capabilities for operations are passed as parameters to operations serviced in different resources. These capability parameters can then be used to invoke operations from afar. Referring again to Figure 4.13, resource `main` directly invokes the operation `p1` in `simple` passing to it a capability for operation `j`. Operation `j` is then invoked through the capability parameter `x` inside proc `p1`, inside resource `simple`. Because this invocation uses a call statement, operation `j` is ineligible for semaphore optimization. Thus, only operation `k` is semaphore optimizable. However, the remote procedure call invocation, i.e., `caps.p1(j)`, can be optimized to a local call.

To determine the optimizability of these operations, values for resource capability parameters and operation capability parameters must be tracked. Because these programs are viewed as templates for actual run-time instances of resources and processes, any optimization must be valid for any possible parameter configuration. Again, this analysis is possible because the values being tracked belong to the finite set of operations declared within the program.

```
resource simple
    op p1(x: cap ())
body simple(y,z: cap ())
    initial
        send y()
        send z()
        ...
    end initial

    proc p1(x)
        call x()
    end p1
end simple

resource main
    import simple
body main()
    op j()
    op k()
    var caps: cap simple
    initial
        caps:= create simple(j,k)
        caps.p1(j)
        ...
    end initial
end main
```

Figure 4.13: Single virtual machine, multiple resource program with operations invoked through capabilities passed to resource at creation.

4.2 Multiple Virtual Machines Programs — Determining Co-residency

Many of our optimizations presented in Chapter 3 depend upon whether or not the resource instances invoking a given operation reside on the same virtual machine (i.e., in the same address space) as its implementation. This kind of analysis is more complicated than the intraresource analysis described in the previous section. It entails considering the forms of operation invocation. Operations invoked directly by name must be implemented in the same resource instance as their invoker. Operations invoked through operation or resource capability variables can be serviced in the same or a different resource instance; if serviced in a different resource instance, that instance may be located on the same or a different virtual machine.

The analysis required for determining whether resource instances reside on the same virtual machine examines resource creation statements and uses dataflow analysis to track resource creation patterns. As stated earlier, this analysis needs to determine only whether resource instances are located on the same virtual machine and not the exact virtual machine on which they are located.

To illustrate interresource analysis, consider the program fragment shown in Figure 4.14. It creates two new virtual machines and an instance of `r` and an instance of `s` on each. Each instance of `s` invokes operation `f` in the instance of `r` on the same virtual machine. This scenario is illustrated in Figure 4.15(a). These invocations can be optimized to use a local procedure call since they are within the same virtual machine.

To demonstrate when the local procedure call optimization is not applicable and to further exhibit the analysis that must be performed, consider the effect of changing the two statements in Figure 4.14 that create instances of `s` to

```
s1 := create s(r1) on vm2
s2 := create s(r2) on vm1.
```

The resource instances that communicate with one another are now located on different virtual machines, as illustrated in Figure 4.15(b). Invocations of `f` are not optimizable by using the local procedure call optimization since the invoking and invoked resource instances are in different address spaces.

```
resource main()
  import r, s
  initial
    var vm1, vm2: cap vm
    var r1, r2: cap r
    var s1, s2: cap s
    vm1 := create vm()
    vm2 := create vm()
    r1  := create r() on vm1
    r2  := create r() on vm2
    s1  := create s(r1) on vm1
    s2  := create s(r2) on vm2
  end initial
end

resource r
  op f(x: int)
body r()
  proc f(x)
    ...
  end
end

resource s
  import r
body s(cr: cap r)
  process p
    ...
    call cr.f(3)
    ...
  end
end
```

Figure 4.14: Multiple virtual machine program.

(a): Configuration of program in Figure 4.14

(b): Configuration of modified Figure 4.14 program

Figure 4.15: Effect of virtual and resource creation patterns on interprocess communication.

4.2.1 Determining When Resources Execute on the Same Virtual Machine

For programs with multiple resource instances, it becomes necessary to determine when resources reside in the same virtual machine and thus within the same shared address space. The following examples illustrate the additional analysis necessary to determine when interacting processes co-reside in shared address space. For the program in Figure 4.14, the dataflow analysis must now track patterns for possible virtual machine creation. These patterns are representative of run-time configurations and it is necessary to associate one of these patterns with the resource capability variables being tracked by dataflow analysis. That is, each resource pattern being instantiated has associated with it a virtual machine pattern. For the program in Figure 4.14 simple dataflow analysis yields two virtual machine patterns, say `vmpat1` and `vmpat2`. In contrast, the program in Figure 4.16 yields one virtual machine pattern, say `vmpat3`. Although many instances of `vmpat3` may exist at run-time, only one template for virtual machine creation occurs within the program. Specifically, that template is a virtual machine with templates for resources `r` and `s`. In some cases, it is not possible to determine virtual machine patterns and the resources residing within them. Figure 4.17 presents one such program. Resource `r1` is associated with a virtual machine pattern, say `vmpat4`; however, resource `r2` is associated with either virtual machine pattern `vmpat4` or with another virtual machine pattern, say `vmpat5`, generated by the create statement inside the if statement. In this situation, our analysis must assume that resources `r1` and `r2` are not co-resident on the same virtual machine. These programs, however, are not representative of commonly occurring and/or acceptable coding styles and, as such, are not considered prime targets for optimization.

The occurrence of virtual machine creation statements within looping constructs can also result in situations where resources are created on virtual machines with non-determinable patterns. Consider the program in Figure 4.18 where the virtual machine creation statement before the for-all statement is associated with a *pattern* attribute, say `vmpat6`, and the virtual machine creation statement within the for-all statement is associated with the pattern attribute, say `vmpat7`. When `vm1` is referenced in the resource creation statements, it is not possible, in general, to determine which virtual machine pattern, `vmpat6`

```
resource main()
  import r, s
  initial
    var vm1:  cap vm
    var r1:  cap r
    var s1:  cap s
    fa i:= 1  to 10 ->
      vm1 :=  create vm()
      r1  :=  create r()  on vm1
      s1  :=  create s(r1)  on vm1
    af
  end
end

resource r
  op f(x:  int)
body r()
  proc f(x)
    ...
  end
end

resource s
  import r
body s(cr:  cap r)
  process p
    ...
    call cr.f(3)
    ...
  end
end
```

Figure 4.16: Multiple virtual machine program with a single virtual machine creation pattern.

```
resource main()
  import r, s
  initial
   ...
    var vm1:  cap vm
    var r1:  cap r
    var s1:  cap s
    vm1 :=  create vm()
    r1  :=  create r()  on vm1
    if   exp  ->
      vm1 :=  create vm()
    fi
    s1  :=  create s(r1)  on vm1
    af
  end
end

resource r
  op f(x:  int)
body r()
  proc f(x)
    ...
  end
end

resource s
  import r
body s(cr:  cap r)
  process p
    ...
    call cr.f(3)
    ...
  end
end
```

Figure 4.17: Multiple virtual machine program with a nondeterminable virtual machine creation pattern.

```
resource main()
  import r, s
  initial
    var vm1: cap vm
    var r1: cap r
    var s1: cap s
    vm1 := create vm()
    fa i:= 1 to 10 ->
      r1  := create r() on vm1
      s1  := create s(r1) on vm1
      vm1 := create vm()
    af
  end
  ...
end main

  ...
```

Figure 4.18: Program fragment with problematic virtual machine creation patterns.

or `vmpat7`, is assigned to `vm1`; thus, we must make conservative assumptions. We assume that upon entering a looping construct the value of any virtual machine capability variable assigned to by a creation statement within that construct is *unknown* until that creation statement is executed or the variable is assigned to by some other means, e.g., an assignment statement.

This loop assumption ensures that our analysis of program behavior makes safe, although conservative, assumptions concerning possible run-time values for virtual machine capability variables.

4.3 Optimization Requirements

The examples presented in the earlier sections of this chapter illustrate the need to characterize invocations and implementations of operations before optimizations can be applied to a program. For example, when a statement invokes an operation through a capability variable, it is necessary to record that the possible values for that variable are so "related". In addition, it is necessary to determine and record whether or not the virtual machine for the invoking process is the same as the virtual machine for the servicing process.

To describe these requirements more formally, we present several definitions. These definitions are relevant to all of the optimizations. The following subsections contain additional definitions relevant to specific optimizations.

First, enumerate the occurrences, $s_1, ..., s_n$, of the send invocation statements in a program, and the occurrences, $c_1, ..., c_m$, of the call invocation statements in a program.

Define S_i to be the set of operations potentially invoked by an instance of the send statement s_i. Similarly, define C_j to be the set of operations potentially invoked by an instance of the call statement c_j. Then, define S to be a multiset of all S_i and define C to be a multiset of all C_j, i.e.,

$$S = \{S_1, ..., S_n\} \text{ and } C = \{C_1, ..., C_m\}.$$

S and C are multisets because multiple occurrences of an invocation may potentially invoke the same set of operations. These sets define an equivalence relation, R, "potentially invoked by the same invocation".

Next, define $S_{i,same}$ to be the set of operations that can potentially be invoked by a given send statement, where invoking and servicing processes are executing on the same virtual machine. Similarly, define $C_{j,same}$ to be the set of operations that can potentially be invoked by a given call statement, where invoking and servicing processes are executing on the same virtual machine.

In contrast, define $S_{i,diff}$ to be the set of operations that can potentially be invoked by a given send statement, where the invoking and servicing processes are possibly executing on different virtual machines. Similarly, define $C_{j,diff}$ to be the set of operations that can potentially be invoked by a given call statement, where the invoking and servicing processes are possibly executing on different virtual machines. Then define S_{same} to be the multiset of all $S_{i,same}$, C_{same} to be the multiset of all $C_{j,same}$, S_{diff} to be the multiset of all $S_{i,diff}$, and C_{diff} to be the multiset of all $C_{j,diff}$. That is,

$$S_{same} = \{S_{1,same}, ..., S_{n,same}\} \text{ and } C_{same} = \{C_{1,same}, ..., C_{m,same}\}$$

and

$$S_{diff} = \{S_{1,diff}, ..., S_{n,diff}\} \text{ and } C_{diff} = \{C_{1,diff}, ..., C_{m,diff}\}.$$

This provides equivalent definitions for S and C. Each element of S is the union of corresponding set elements of S_{same} and S_{diff} and each element of C is the union of corresponding set elements of C_{same} and C_{diff}. That is,

$$S = \{S_{1,same} \cup S_{1,diff}, ..., S_{n,same} \cup S_{n,diff}\}$$

and

$$C = \{C_{1,same} \cup C_{1,diff}, ..., C_{m,same} \cup C_{m,diff}\}.$$

The following subsections contain definitions relevant to particular optimizing transformations, as well as formalizations defining the applicability requirements for each of the transformations.

4.3.1 Semaphore Optimization Requirements

Semaphore optimizable operations can be characterized more formally given the following additional definitions. Recall that sets S and C both define an equivalence relation, R, "potentially invoked by the same invocation". Denote by R^* the reflexive, transitive closure of the relation R. Let S' be the set of all operations invoked by sends and C' the set of all operations invoked by calls — i.e.,

$$S' = \bigcup_{y \in S} y \quad \text{and} \quad C' = \bigcup_{y \in C} y \ .$$

Let S'_{diff} be the set of all operations invoked by sends, where invoking and servicing processes are possibly on different virtual machines — i.e.,

$$S'_{diff} = \bigcup_{y \in S_{diff}} y \ .$$

In addition, denote the set of parameterless operations by P and the set of operations serviced by receive statements (or the equivalent input statement form) by Q. Then the set of semaphore optimizable operations is defined as

$$\{x \in S' \mid x \notin S'_{diff} \wedge x \in P \wedge x \in Q \wedge x \notin C' \wedge (x\, R^*\, w \;\Rightarrow\; (w \notin S'_{diff} \wedge w \notin C' \wedge w \in Q))\}.$$

4.3.2 RPC-to-PC Optimization Requirements

Because the optimization of remote procedure call is occurrence-specific, each occurrence of a call statement must be examined for possible optimization. Each set in C corresponds to an occurrence of a call statement within a program and each of these occurrences is independently analyzed. An occurrence of a call statement c_i is optimizable if all members of the set C_i are implemented by procs with no reply statements and if the set of all interprocess interactions through this occurrence is between processes residing on the same virtual machine. Let T be the set of operations implemented by a proc with no reply statements. Then, the set of occurrences of optimizable call statements is

$$\{c_i \mid C_{i,diff} = \{\} \wedge (x \in C_{i,same} \Rightarrow\ x \in T)\}.$$

.

4.3.3 Client/Server Optimization Requirements — Rendezvous and Asynchronous Message Passing

The analysis requirements for the client/server transformations described in Chapter 3 are more complex than for the optimizations described above. In addition to identifying invocations and implementations of operations, a significant amount of syntactic pattern matching is required to identify optimizable server processes. The following definitions are used to characterize the application of this transformation.

First, enumerate the occurrences, $in_1, ..., in_l$, of input statements in a program, and let IN_i be the multiset of operations serviced by the occurrence of input statement in_i. Then, define IN to be the set of sets IN_i — that is,

$$IN = \{IN_1, ..., IN_l\}.$$

IN is a multiset because SR permits multiple implementations of operations using the input statement or the receive statement.

Then enumerate the occurrences, $din_1, ..., din_h$, of input statements that are eligible for client/server optimization. Recall from Section 4.1 that these input statements must occur immediately within the scope of an infinite loop that occurs directly inside a process as shown below.

```
process ...
    statements
    do true ->
        statements
        input statement
        statements
    od
end
```

These input statements are subject to additional constraints: no reply statements within the the input statement's associated bodies of code, no synchronization expressions referencing operation parameters, and no scheduling expressions.

An input statement reflecting an optimizable client/server using rendezvous can be characterized as an input statement that corresponds to an occurrence din_i. In addition, the operations implemented by the input statement are not implemented by any other input or receive statements[2] and for all such operations they must be invoked only by call statements within the same shared address space. The set of input statements reflecting optimizable client/server patterns is characterized more formally below. Because the type of applicable optimizing transformation depends upon whether the operations serviced by the server process are invoked synchronously, asynchronously, or both as described in Chapter 3, we provide three characterizations for the client/server optimizations. The first requires that all client processes invoke operations serviced by the server synchronously via call statements, as in the program for *readers/writers* presented in Figure 3.8 of Chapter 3.

$$\{in_i | \exists j :: \; in_i = din_j \wedge ((x \in IN_i \wedge i \neq k) \Rightarrow x \notin IN_k) \wedge (x \in IN_i \Rightarrow (x \notin S' \wedge x \notin C'_{diff}))\}.$$

Similarly, the set of input statements reflecting optimizable client/servers using asynchronous message passing, as in Figure 3.9 of Chapter 3, is defined as follows:

$$\{in_i | \exists j :: \; in_i = din_j \wedge ((x \in IN_i \wedge i \neq k) \Rightarrow x \notin IN_k) \wedge (x \in IN_i \Rightarrow (x \notin C' \wedge x \notin S'_{diff}))\}.$$

[2]Recall from Chapter 2 that the SR language prohibits implementation of operations by both procs and input statements.

For hybrids, using both rendezvous and asynchronous message passing, as in the *readers/writers* example in Figure 3.11, the set of optimizable input statements is defined as follows:

$$\{in_i \mid \exists j :: in_i = din_j \wedge ((x \in IN_i \wedge i \neq k) \Rightarrow x \notin IN_k) \wedge$$
$$(x \in IN_i \Rightarrow ((x \notin S' \wedge x \notin C'_{diff}) \vee (x \notin C' \wedge x \notin S'_{diff})))\}.$$

This definition, in essence, is a combination of the two previous definitions.

4.4 Summary

The analysis requirements for automatic application of our optimizing transformations necessitate both syntactic and dataflow analysis. Syntactic analysis can be used to determine how operations are implemented. However, how they are invoked can require dataflow analysis along with interprocess and intermodule analysis to track values of capability variables and parameters. For programs executing on a single virtual machine the above analysis suffices to determine optimizability.

For programs executing on multiple virtual machines additional analysis is required. Co-residency of program resource instances can in some cases be determined providing additional optimizations for these programs.

Several observations or assumptions are fundamental to our analysis. The most fundamental is that SR programs provide templates for operations, processes, resources, and virtual machines. We can characterize the behavior of all run-time instances of an operation by examining the static code. More specifically, we can statically determine all possible values for all operation capability variables because each can only be assigned one of a discrete compile-time determinable set of operation templates, and assignment is the only operation that can be performed on these variables. Similarly, we can statically determine all possible values for all resource capability variables because each can only be assigned one of a discrete compile-time determinable set of resource templates. Finally, we observe that virtual machine templates or patterns can be tracked statically for many commonly occurring situations.

The following chapter describes, in greater detail, the algorithms used to perform dataflow analysis and to track operation, virtual machine, and resource templates.

Chapter 5

Transformation Algorithms

In Chapter 3, we presented source-level transformations that improve the behavior of high-level interprocess communication and synchronization mechanisms. As described in Chapter 4, the applicability of these optimizing transformations depends upon information that must be obtained through significant program analysis. Both syntactic and dataflow analysis are required. For example, simple syntactic analysis is necessary to determine which operations are parameterless. Similarly, simple syntactic analysis can be used to determine how operations are implemented. More complex syntactic analysis is required to determine optimizable input statements associated with client/server programs. Dataflow analysis is required to track virtual machine patterns, resource co-residency, and the potential values of capability variables used to invoke operations. This dataflow analysis must include both interprocess and interresource analysis. In this chapter, we present the transformation algorithms used to perform the necessary syntactic and dataflow analysis.

5.1 Overview of Transformation Algorithms

Because both syntactic and dataflow information was necessary for our analysis and because we wished to do our analysis at a high level, we chose to use the *method of attributes* introduced by Babich and Jazayeri [BJ78]. This method performs dataflow analysis on a parse tree representation of the program instead of on the more typical *control flow graph*. Our techniques extend the efforts of Babich and Jazayeri to concurrent languages and to interprocess and interresource analysis.

Figure 5.1 presents an overview of our transformer. Our optimizing transformer

Figure 5.1: Overview of optimizing transformer.

has four components as described below.

Syntactic analyzer The syntactic analysis uses an attributed grammar to both parse the source program and to produce an abstract syntax tree decorated with syntactically determinable information, e.g., how operations are implemented and which operations are parameterless.

Dataflow analyzer The dataflow analysis necessary to determine all possible invocations of all operations within a program module includes both interprocess and interresource analysis in three conceptually distinct phases.

- Phase one requires a single pass over the source program and is responsible for the identification of virtual machine creation patterns and their effective ranges. Although conceptually a separate pass, this pass can be combined with the single pass required for the syntactic analysis described above.
- Phase two is responsible for the identification of resource creation and, when possible, the association of a virtual machine pattern with a created resource. In addition, phase two is responsible for determining all of the possible values for capability variables used within a program. This phase iterates over a pruned view of the decorated abstract syntax tree, computing dataflow attributes until the decorated tree reaches a stable state — a fixed point. This pruned or sliced view includes all statements that potentially affect capability variable values.
- Phase three uses the information generated in phase two — the values that capability variables potentially hold — to determine how operations are invoked. This phase requires a single pass over a different view of the abstract syntax tree that includes all operation invocation statements.

Transformer The transformer takes the abstract syntax tree generated by the dataflow analyzer and applies the transformations. In some cases, compiler directives are added to the tree; in other cases, entire subtrees are replaced with subtrees reflecting the transformed program.

Re-sourcer The re-sourcer generates new source code yielding the transformed program.

The following sections describe, in greater detail, the approaches we use for both syntactic and dataflow analysis. We focus only on selected interesting attribute definitions used in this analysis; a comprehensive set of definitions is provided in Appendix A. Finally, we briefly describe the transformer and re-sourcer.

5.2 Syntactic Analysis

The initial pass over the source program uses attribute grammars to create an abstract syntax tree and to generate typical parsing tables such as the symbol table. In addition, the syntactic analyzer collects additional information to aid the transformation process. Recall from Chapter 4 that this information must include: P, the set of parameterless operations implemented by a receive statement or an equivalent input statement; T, the set of operations implemented by a proc with no embedded reply statements; and IN_i, the set of operations serviced by an input statement in_i. Syntactic determination of din_i, the input statement occurrences that are eligible for client/server optimization requires more complex analysis, namely, a tree pattern matching algorithm.

5.3 Dataflow Analysis

Our presentation of the dataflow analysis algorithms assumes the presence of the symbol table. In addition, several functions are assumed to facilitate presentation; some of these functions access this table. These functions appear in Table 5.1.

The following sections present the algorithms for the three phases of our dataflow analysis.

5.3.1 Phase One — Virtual Machine Creation Patterns

Phase one involves a single pass over the source program; this pass can be combined with the syntactic analysis described in the previous section. This phase determines the values of several attributes: *globalvars*, *pattern*, *res*, *vmvars*, and *loopcnt*. The attribute *globalvars* is attached to each resource node and contains all globally declared — shared —

Function	**Description**
curres	returns the name of the resource being analyzed
$save(node, attribute)$	*node* — current parse tree node *attribute* — attribute associated with current parse tree node saves the current value of *attribute* in the specified *node*
gensym(vmpat)	*vmpat* — a constant string returns the next in a sequence of unique virtual machine pattern identifiers using *vmpat* as prefix
removelocals(defs)	*defs* — a set of capability variable definitions returns the definitions for all global, i.e., shared variables only
pass1	returns true if phase two analysis is in its first iteration or pass over the abstract syntax tree and false otherwise
formal(x,p)	x — an operation, capability variable for an operation or resource, or qualified operation p — an actual invocation or an implementation parameter returns the possible formal equivalents of the actual parameter — used to propagate possible parameter bindings
resequiv(z,r,a)	z — a resource capability variable r — a resource a — an operation returns the operation in resource r that is structurally equivalent to the operation a declared in the resource for which z is a capability
input(p)	p — an actual parameter returns true if p is an input parameter and false otherwise
output(p)	p — an actual parameter returns true if p is an output parameter and false otherwise
op(p)	p — an actual parameter returns true if p is an operation and false otherwise
capvar(p)	p — an actual parameter returns true if p is a capability variable and false otherwise
qualop(p)	p — an actual parameter returns true if p is a qualified operation (written as z.a) and false otherwise
qres(p)	p — an actual parameter that is a qualified operation returns the resource capability variable, i.e., the qualifier
qop(p)	p — an actual parameter that is a qualified operation returns the operation
inv(p)	p — an actual parameter returns true if p is an invocation and false otherwise
invop(p)	p — an actual parameter that is an invocation returns the set operations possibly invoked

Table 5.1: Descriptions of functions used to facilitate presentation of attribute definitions used in dataflow analysis.

variables for that resource. It is required for the dataflow analysis in phase two. The attribute *pattern* is associated with the parse tree node for virtual machine creation statements and contains a unique pattern identifier — one for each occurrence of a virtual machine creation statement. These patterns represent templates for virtual machines. The attribute *res* is attached to the program node of the parse tree and contains a *(resource, virtual machine pattern)* pair representing the program's main or initial resource and a pattern identifier for the implicitly created virtual machine associated with the resource. The description of attributes *vmvars* and *loopcnt* is more complex. In phase two of the dataflow analysis, we use attribute definitions to propagate virtual machine templates associated with virtual machine capability variables. These definitions are then used to associate virtual machine patterns with resources created by resource creation statements. Dataflow definitions for virtual machine template propagation are affected by virtual machine creation statements; however, they are also affected by simple assignment statements and flow of control constructs.

To illustrate the effect of flow of control constructs on virtual machine capability propagation, recall the program in Figure 4.18 in Chapter 4. It contains resource and virtual machine creation statements within a looping structure; references to the virtual machine capability variable precede the assignment to the variable, making the value of the variable statically undeterminable. Recall also that we handle this situation in our analysis through a simplifying *loop assumption.* The implementation of this assumption collects, for each looping structure within a program, a list, *vmvars*, of all virtual machine capability variables assigned to via a creation statement within the body of the loop. This list is then used in phase two to assign the virtual machine pattern "unknown" to all variables in the list at the beginning of the looping construct. The attribute *loopcnt* reflects the nesting depth of looping structures and is used to determine when the attribute *vmvars* must be computed.

Phase one analysis is performed on a *slice* of the abstract syntax tree. For our purposes, a tree slice is that portion of the parse tree necessary to compute the attributes under consideration.[1] For example, declarations local to a proc or process do not affect the value of phase one attributes and are ignored. Similarly, most assignment statements and operation invocation and implementation statements do not require analysis in this phase.

To illustrate the attributes *globalvars*, *pattern*, *res*, *vmvars*, and *loopcnt*, consider the program in Figure 5.2, which is similar to the program in Figure 4.16 in Chapter 4. A fragment of the parse tree slice analyzed in this phase appears in Figure 5.3. Tree nodes are decorated with phase-one attributes. This fragment includes only the resource `main` because that is the only resource producing interesting phase one attribute values.

The value of the attribute `pattern` attached to the tree node for the create statement is generated by a utility that generates sequences of unique identifiers. The definitions used to generate these attribute values appear in Appendix A.1.

5.3.2 Phase Two — Resource Creation and Propagation of Virtual Machine, Resource, and Operation Templates

Recall from Section 5.1 that phase two iterates over the parse tree slice generated by phase one, generating values for attributes *defs*, *res*, and *bnd* until a fixed point is reached. The attribute *defs* contains template definitions for capability variables as a set of triples of the form

((capability variable, vm pattern),(value, vm pattern),flag)

where value is either an operation, resource, or virtual machine pattern depending upon the type of capability variable being defined and flag is "l" or "g" representing a local or global variable respectively. As in phase one, the attributes *res* contains *(resource, virtual machine pattern)* pairs. The attribute *bnd* is a set of triples containing parameter binding information to aid interprocess and interresource analysis. Each triple has the form

((formal operation or resource parameter, vm pattern),(value, vm pattern),flag)

[1] Our tree slices reflect a syntactic program decomposition rather than the more difficult semantic program decomposition described in [Wei84].

```
resource main()
  import r, s
  initial
    var vm1: cap vm
    var r1: cap r
    var s1: cap s
    fa i:= 1 to 10 ->
      vm1 := create vm()
      r1  := create r() on vm1
      s1  := create s(r1) on vm1
    af
  end initial
end main

resource r
  op a(y: cap())
body r()
  process p1
      receive a(z)
      ...
      send z()
      ...
  end p1
end r

resource s
  import r
body s(x: cap r)
  process p2
      op b()
      ...
      call x.a(b)
      ...
      receive b()
  end p2
end s
```

Figure 5.2: Program with many run-time virtual machine instances but one virtual machine pattern.

Figure 5.3: Slice of abstract syntax tree decorated with phase one attributes.

where value is either an operation, resource, or virtual machine pattern template, and flag is "in" for a value input to an operation or resource, and "out" for a value that is output from an operation invocation. A formal operation parameter is the formal parameter that appears in the declaration for the operation, rather than the one that appears in the implementation of the operation. Similarly, a formal resource parameter is the parameter associated with the resource definition. A parameter that is both an input and an output parameter will have multiple entries in *bnd*.

Recall from Chapter 4 that for a program invoking operations through capability variables, the values that the variables may assume, i.e., operations, need to be determined. The key is that capability variables may be assigned one of a discrete, compile-time determinable set of operation patterns.[2] This fact allows conventional dataflow techniques to be used to track their potential values. For a capability variable local to a process, traditional dataflow analysis is sufficient. However, for a capability variable shared by several processes, traditional dataflow techniques must be extended to handle the shared variable assumption.

When capabilities for operations are passed as parameters, additional interprocedural or interprocess analysis is required. Because processes execute in an unknown order, we assume that any invocation of an operation may be serviced by any implementation of that operation. This is in contrast to the typical *call graph* used for interprocedural analysis in sequential languages. Moser makes a similar observation concerning the Ada programming language [Mos90]. Since our transformations are applied to templates for both resources and operations, all potential parameter values must be accommodated. This constraint has the effect of simplifying interprocess analysis; thus a global set of potential operation and resource bindings is sufficient.

[2]Although the actual values tracked are technically patterns for instances of operations, for simplicity, we refer to them as operations.

5.3.2.1 Pattern Propagation

Operation patterns potentially assigned to capability variables, denoted by the attribute *defs*, must be determined for each invocation of the capability variables. Determining these potential patterns requires dataflow techniques similar to those used for constant propagation described by Aho, Sethi, and Ullman [ASU86]. We summarize some of these equations below to provide some background for our own dataflow definitions. Let $defs_{in}(S)$ denote the attribute definitions that are possible before executing a particular statement S. Then, $defs_{out}(S)$ denotes the attribute definitions following execution of S. In general, the value of the attribute *defs* is affected by any statement S that affects the value of a variable; $defs_{out}(S)$ is equal to the definition generated by S unioned with the set of previously valid definitions, $defs_{in}(S)$, minus the set of definitions *killed* by the definition generated in S. For example, for the statement *S:* `x := a+b`

$$defs_{out}(S) = defs_{in}(S) - kill(defs_{in}(S), x) \cup (x := a + b).$$

Flow of control constructs also affect these definitions. Conservative assumptions are typically used to ensure that *defs* reflects all possible definitions. For example, for the statement *S:* `if e then` S_1 `else` S_2 `fi`

$$defs_{out}(S) = defs_{out}(S_1) \cup defs_{out}(S_2).$$

Aho, Sethi, and Ullman [ASU86] also point out the circular nature of the dataflow equations for looping structures. For example, for the statement *S:* `do e` S_1 `od`

$$defs_{in}(S_1) = defs_{in}(S) \cup defs_{out}(S_1)$$

where $defs_{out}(S_1)$ is a function of $defs_{in}(S_1)$.

In concurrent languages, the presence of global variables, whose definitions must propagate through the entire program, creates additional circularities. Similarly, the need for interprocess and interresource analysis results in additional circularities. We resolve these circularities by iterating over the decorated parse tree until a fixed point is reached. The number of iterations required to reach a fixed point depends not only upon the program being analyzed, but also upon the order in which program resources are analyzed. Further discussion of iteration bounds appears in Section 5.3.2.1.3.

We use conventional dataflow techniques extended to handle shared variables along with interprocess analysis and interresource analysis to propagate virtual machine, resource, and operation patterns. The remainder of this section describes the attribute definitions used to handle shared variables and then to propagate pattern values: first for virtual machines and resources, and then for operations.

5.3.2.1.1 Shared Variables

Conventional dataflow equations such as those described above hold for capability variables that are local to a proc or process. However, for globally declared capability variables, which may be shared, the shared value assumption described earlier dictates that new definitions not *kill* previous definitions. Thus, in this case, $defs_{out}(S)$ is equal to the definition generated by S unioned with the set of previously valid definitions, i.e., for a statement S

$$defs_{out}(S) = defs_{in}(S) \cup \ \{\text{definitions generated by } S\}.$$

The circularity caused by shared variables manifests itself when we examine the dataflow definition for $defs_{in}(S)$ where

$$defs_{in}(S) = defs_{in}(S) \cup defs_{out}(S).$$

5.3.2.1.2 Virtual Machine and Resource Pattern Propagation

To illustrate the propagation of virtual machine and resource patterns, consider again the program in Figure 5.2. Figures 5.4, 5.5, and 5.6 illustrate the effect of phase two analysis on resources `main`, `s`, and `r` respectively. Figure 5.4 reflects attribute values following a single phase two pass over resource `main`. Figure 5.5 reflects attribute values following subsequent analysis of resource `s`. Finally, Figure 5.6 reflects attribute values for resource `r` assuming prior analysis of resources `main` and `s`.

Examining Figure 5.4 more closely, the value of *defs* is affected by the for-all statement, the virtual machine create statement, and the resource create statements. The following are attribute definitions used to represent the effect of the for-all statement on the attributes *defs*, *res*, and *bnd*.

$$
\begin{aligned}
S \;:\;& \texttt{fa} \ldots \texttt{->}\ S_1\ \texttt{af} \\
& if\ pass1\ then \\
& \quad defs_{in}(S_1) = defs_{in}(S) \cup \\
& \qquad \{((x,u),(unk,nil),k) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge x \in save(S, vmvars)\} \\
& else \\
& \quad defs_{in}(S_1) = defs_{in}(S) \cup save(S_1, defs) \\
& defs_{out}(S) = defs_{out}(S_1) \cup defs_{in}(S) \\
& save(S, defs) = defs_{out}(S)_{in} \\
& res_{in}(S_1) = res_{in}(S) \\
& res_{out}(S) = res_{out}(S_1) \\
& bnd_{in}(S_1) = bnd_{in}(S) \\
& bnd_{out}(S) = bnd_{out}(S_1) \\
& globalvars_{in}(S_1) = globalvars_{in}(S)
\end{aligned}
$$

The $pass1$[3] flag present in the above definition is used to distinguish the first phase two pass from subsequent passes. In this first pass of phase two, the virtual machine capability variables assigned to within the for-all statement are initially assigned the "unknown" value, *unk*. In addition, the value of $defs_{out}(S)$ is saved for use in subsequent passes. This statement does not affect the values of attributes *bnd* and *res*, hence their values upon completion of *S* are equal to their values before execution of *S*.

The occurrence of the virtual machine create statement within the for-all statement *kills* the definition in *defs* that binds *unk* to the variable `vm1` (because the variable is local)

[3] To facilitate presentation, *pass1* is tested outside the attribute definitions rather than included as part of a predicate within an attribute definition. The set *globalvars* is also accessed in this manner.

Figure 5.4: Phase two parse tree slice for resource `main` in the program in Figure 5.2.

Figure 5.5: Phase two parse tree slice for resource **s** in the program in Figure 5.2.

Figure 5.6: Phase two parse tree slice for resource **r** in the program in Figure 5.2.

and assigns it the virtual machine pattern `vmpat1`. The attribute definitions used by the virtual machine machine create statement appear below.

$$
\begin{array}{lll}
S & : & x := \texttt{create}\, \texttt{vm}() \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
& & \qquad \{((x,u),(pattern,nil),l) |\ r = curres \wedge (r,u) \in res_{in}(S)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(unk,nil),g) | (y,u) \in res_{in}(S)\} \\
& & res_{out}(S) = res_{in}(S) \\
& & bnd_{out}(S) = bnd_{in}(S)
\end{array}
$$

Again, this statement does not affect the values of attributes *bnd* and *res*.

Attribute definitions for resource creation statements are similar to those for virtual machine creation, however they are more complex primarily due to the presence of resource parameters. The definitions are shown below.

$$
\begin{array}{lll}
S & : & x := \texttt{create}\, y(p_1, \ldots, p_n)\ \texttt{on}\ z \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
& & \qquad \{((x,u),(y,v),l) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(v,nil),?) \in defs_{in}(S)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(y,unk),g) | (y,u) \in res_{in}(S)\} \\
& & res_{out}(S) = res_{in}(S) \cup \\
& & \quad \{(y,v) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(v,nil),?) \in defs_{in}\} \\
& & bnd_{out}(S) = bnd_{in}(S) \cup \\
& & \quad \{((formal(y,p_i),u),(p_i,u),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge op(p_i)\} \cup \\
& & \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
& & \quad \{((formal(y,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge \\
& & \qquad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
& & \quad \{((formal(y,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{array}
$$

The resource creation statement affects not only the value of *defs*, but also the values of *res* and *bnd*. The attribute *res* is updated to include *(resource, virtual machine pattern)* pairs for all possible values of the virtual machine capability variable used in the statement. The attribute *bnd* is updated to include all possible capability parameter values that are passed to the resource being created. This interresource analysis is complex; it is described in greater detail in the next section. The "?" that appears in the local/global field of a *defs* triple in the attribute definition for *defs* indicates a "don't care" situation, i.e., the value may be either "l" or "g". Additional phase two attribute definitions are provided in Appendix A.2.

5.3.2.1.3 Operation Pattern Propagation

Propagation of operation capability variable values, i.e., operation patterns, must also be determined. Again, we use conventional dataflow techniques extended to handle shared variables, and interprocess and interresource analysis.

The following are selected dataflow equations used to track the values of operation capability variables and parameters within a program module. Others appear in Appendix A.2. The value of an operation capability variable is affected by assignment statements. The value of an operation capability parameter is affected by invocation statements, operation implementations, and resource creation statements.

Assignment statements use four possible forms: direct assignment of an operation to a capability variable, indirect assignment of the possible values for one capability variable to another capability variable, assignment of an operation qualified by a resource capability to a capability variable, and finally assignment of a return value for an operation invocation to a capability variable. We present the attribute definitions for the first three cases below. The fourth case is discussed in the following section. The following conventions are used for the grammar productions: x is a capability variable, a is an operation name, y is a capability variable, z is a capability for a resource, and the p_i are capability parameters. The phase one attribute *globalvars* is used to distinguish the handling of local and shared variables.

The first set of attribute definitions is for the direct assignment of an operation to a capability variable.

$$
\begin{array}{lll}
S & : & x := a \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
& & \qquad \{((x,u),(a,u),l) |\ r = curres \wedge (r,u) \in res_{in}(S)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(a,u),g) |\ r = curres \wedge (r,u) \in res_{in}(S)\} \\
& & res_{out}(S) = res_{in}(S) \\
& & bnd_{out}(S) = bnd_{in}(S)
\end{array}
$$

This assignment statement does not affect the values of attributes *res* or *bnd*; hence, their values upon completion of S are simply equal to their values before execution of S. The value of the attribute *defs*, however, must be augmented to include the binding of operation a to variable x for each virtual machine pattern associated with the resource in which x resides.

The second set of attribute definitions below is for indirect assignment, i.e., the assignment of the possible values for one capability variable to another capability variable. In this case, the value of *defs* is augmented to include the possible bindings of values associated with variable y to the variable x for each virtual machine pattern associated with the resource in which x resides.

$$
\begin{array}{lll}
S & : & x := y \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
& & \qquad \{((x,u),(a,v),l) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((y,u),(a,v),?) \in defs_{in}(S)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup
\end{array}
$$

$$\{((x,u),(a,v),g)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((y,u),(a,v),?) \in defs_{in}(S)\}$$
$$res_{out}(S) = res_{in}(S)$$
$$bnd_{out}(S) = bnd_{in}(S)$$

The third set of attribute definitions is assignment of an explicitly qualified operation to an operation capability variable. The qualified operation is of the form $z.a$ where z is a resource capability variable and a is an operation. In this case, the value of *defs* is augmented to include the possible bindings of operations associated with $z.a$ to the variable x for each virtual machine pattern associated with the resource in which x resides. The set of operations associated with $z.a$ is determined first by finding the values associated with resource capability variable z from the input value of *defs*. Given that variable z is statically declared to be a capability for a particular resource, that a is an operation within that resource, and that r_1 is a possible resource value associated with z, the function *resequiv* finds the operation within r_1 that is structurally equivalent to operation a. Structural equivalence of resources is not a widely used feature, but our analysis must accommodate its presence.

$$
\begin{array}{lll}
S & : & x := z.a \\
 & & if\ x \notin globalvars_{in}(S)\ then \\
 & & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
 & & \qquad \{((x,u),(b,v),l)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(r_1,v),?) \in defs_{in}(S) \wedge \\
 & & \qquad\quad b = resequiv(z, r_1, a)\} \\
 & & else \\
 & & \quad defs_{out}(S) = defs_{in}(S) \cup \\
 & & \qquad \{((x,u),(b,v),g)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(r_1,v),?) \in defs_{in}(S) \wedge \\
 & & \qquad\quad b = resequiv(z, r_1, a)\} \\
 & & res_{out}(S) = res_{in}(S) \\
 & & bnd_{out}(S) = bnd_{in}(S)
\end{array}
$$

The last type of assignment statement — assignment to an operation capability variable through invocation of an operation returning a capability as a result — involves both interprocess and interresource analysis and is discussed in the next section.

5.3.2.1.4 Propagation Across Process and Resource Boundaries

To propagate operation, resource, and virtual machine patterns across process and resource boundaries, invocation statements, implementation statements, and resource creation statements require attribute definitions tracking input and output capability parameter values. Invocations and implementations have several forms. In general, invocations include all call and send statements and occurrences of operation function calls within expressions. The invocations of interest in our analysis include those invocations returning a capability as a result and those invocations with capability parameters. Operation invocations may be directly through an operation name, indirectly through a capability parameter, or through an operation qualified by a resource capability variable. Operations are implemented by a proc, or by input, or receive statements. All forms of implementation reference the operation directly. Below we present the attribute definitions necessary to track operation, resource, and virtual machine patterns through operation invocation and implementation for several cases. First, we present in detail the definitions for assignment statements invoking an operation returning a capability as a result. All three invocation forms are described. Next, we present the attribute definitions for a call statement invocation. Finally, we present definitions for the receive statement implementation.

The first set of attribute definitions are for assignment to a capability variable through direct operation invocation. These definitions vary from those presented for simple assignment in several ways. First, the attribute *defs* must be augmented to include all possible values returned by the invocation. This information is tracked in the attribute *bnd* during analysis of operation implementation. The definitions below illustrate the extraction of this information from *bnd* as part of the computation of *defs*. In addition, the values associated with the attribute *bnd* are affected when the invocation involves capability parameters. Values associated with actual capability parameters are bound to the formal parameter appearing in the declaration of the operation. These bindings are used during the analysis of operation implementations. Actual capability parameters may be in one of several forms: an operation, a capability variable, a qualified operation, or yet another invocation returning a capability as a result. Each of these cases is reflected in the definition of the attribute *bnd* below.

$$
\begin{array}{l}
S \;:\; x := a(p_1,\ ...,\ p_n) \\
\quad if\ x \notin globalvars_{in}(S)\ then \\
\quad\quad defs_{out}(S) = (defs_{in}(S) - \\
\quad\quad\quad kill(defs_{in}(S), x) - \\
\quad\quad\quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i) |\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
\quad\quad\quad \{((x,u),(b,v),l) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge ((a,u),(b,v),out) \in bnd_{in}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad else \\
\quad\quad defs_{out}(S) = defs_{in}(S) \cup \\
\quad\quad\quad \{((x,u),(b,v),g) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge ((a,u),(b,v),?) \in bnd_{in}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad res_{out}(S) = res_{in}(S) \\
\quad bnd_{out}(S) = bnd_{in}(S) \cup \\
\quad\quad \{((formal(a,p_i),u),(p_i,u),in) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge op(p_i)\} \cup \\
\quad\quad \{((formal(a,p_i),u),(b,v),in) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(a,p_i),u),(b,v),in) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge \\
\quad\quad\quad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(a,p_i),u),(b,v),in) |\ r = currres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{array}
$$

The second form of assignment using operation invocation involves an invocation through a capability variable. The definitions below vary from those above in that invocations of all possible values of the capability variable y must be accounted for. That is, all possible return values for all values of y must be associated with the variable x in the attribute *defs*. Similarly, bindings of actual parameters to formal declaration parameters must include the formal declaration parameters associated with all possible invocation operations.

$$
\begin{array}{l}
S \;\;:\;\; x := y(p_1,\; ...,\; p_n) \\
\quad if\ x \notin globalvars_{in}(S)\ then \\
\quad\quad defs_{out}(S) = (defs_{in}(S) - \\
\quad\quad\quad kill(defs_{in}(S), x) - \\
\quad\quad\quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i) |\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
\quad\quad\quad \{((x,u),(b,w),l) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((y,u),(c,v),?) \in defs_{in}(S) \wedge ((c,v),(b,w),out) \in bnd_{in}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad else \\
\quad\quad defs_{out}(S) = defs_{in}(S) \cup \{((x,w),(b,v),g) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((y,u),(c,v),?) \in defs_{in}(S) \wedge ((c,v),(b,w),?) \in bnd_{in}(S)\} \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad res_{out}(S) = res_{in}(S) \cup \\
\quad bnd_{out}(S) = bnd_{in}(S) \cup \\
\quad\quad \{((formal(y,p_i),u),(p_i,u),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge op(p_i)\} \cup \\
\quad\quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(y,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i,r,qop(p_i)) \wedge \\
\quad\quad\quad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(y,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{array}
$$

The third form of assignment using operation invocation involves an invocation through a qualified operation. The definitions below vary from those above in that invocations of all possible operations $z.a$ must be accounted for. Given resource capability variable z, possible values associated with z, and the operation a, the function *resequiv* provides a list of operations potentially invoked by the invocation statement. All possible return values for these operations must be associated with the variable x in the attribute *defs*. Similarly, bindings of actual parameters to formal declaration parameters must include the formal declaration parameters associated with all possible invocation operations.

$$
\begin{array}{l}
S \;\; : \;\; x := z.a(p_1,\ ...,\ p_n) \\
\quad if\ x \notin globalvars_{in}(S)\ then \\
\quad\quad defs_{out}(S) = (defs_{in}(S) - \\
\quad\quad\quad kill(defs_{in}(S), x) - \\
\quad\quad\quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i) |\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
\quad\quad\quad \{((x,u),(c,w),l) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(r_1,v),?) \in defs_{in}(S) \wedge \\
\quad\quad\quad\quad b = resequiv(z, r_1, a) \wedge ((b,v),(c,w),out) \in bnd_{in}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad else \\
\quad\quad defs_{out}(S) = defs_{in}(S) \cup \{((x,w),(b,v),g) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad ((y,u),(r_1,v),?) \in defs_{in}(S) \wedge b = resequiv(z, r_1, a) \wedge ((b,v),(c,w),?) \in bnd_{in}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
\quad\quad\quad \{((p_i,u),(b,v),g) |\ p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
\quad res_{out}(S) = res_{in}(S) \\
\quad bnd_{out}(S) = bnd_{in}(S) \cup \\
\quad\quad \{((formal(z.a,p_i),u),(p_i,u),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge op(p_i)\} \cup \\
\quad\quad \{((formal(z.a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(z.a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge \\
\quad\quad\quad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
\quad\quad \{((formal(z.a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
\quad\quad\quad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{array}
$$

Below we present the attribute definitions used to propagate capabilities for another invocation form, the call statement. Actual capability parameter values must be bound for all input parameters and values for output parameters are extracted from the attribute *bnd* and associated with actual parameters in the attribute *defs*. Although the invocation may use any one of the three forms described above, we present here definitions for an invocation through a qualified operation. Appendix A.2 contains a complete set of attribute definitions for operation invocation and implementation.

$$
\begin{aligned}
S \;:\;& \texttt{call}\ z.a(p_1,\ \ldots,\ p_n) \\
& defs_{out}(S) = (defs_{in}(S) - \\
&\quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i) |\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
&\quad \{((p_i, u), (b, v), l) |\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad ((formal(z.a, p_i), u), (b, v), out) \in bnd_{out}(S)\} \cup \\
&\quad \{((p_i, u), (b, v), g) |\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad ((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\} \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S) \cup \\
&\quad \{((formal(z.a, p_i), u), (p_i, u), in) |\ r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad input(p_i) \wedge op(p_i)\} \cup \\
&\quad \{((formal(z.a, p_i), u), (b, v), in) |\ r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad input(p_i) \wedge capvar(p_i) \wedge ((p_i, u), (b, v), ?) \in defs_{in}(S)\} \\
&\quad \{((formal(z.a, p_i), u), (b, v), in) |\ r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge \\
&\qquad ((qres(p_i), u), (b, v), ?) \in defs_{in}(S)\} \\
&\quad \{((formal(z.a, p_i), u), (b, v), in) |\ r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad inv(p_i) \wedge c \in invop(p_i) \wedge ((c, u), (b, v), out) \in bnd_{in}(S)\}
\end{aligned}
$$

Finally, we present attribute definitions tracking actual parameter bindings through an implementation. The definitions below apply to the receive statement. Attribute definitions propagating capabilities through input statements and procs appear in Appendix A.2.

$$
\begin{aligned}
S \;:\;& \texttt{receive}\ a(p_1,\ \ldots,\ p_n) \\
& defs_{out}(S) = (defs_{in}(S) - \\
&\quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i) |\ capvar(p_i) \wedge input(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
&\quad \{((p_i, u), (b, v), l) |\ r = curres \wedge (r, u) \in res_{in}(S) \wedge \\
&\qquad ((formal(z.a, p_i), u), (b, v), in) \in bnd_{in}(S)\} \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S)
\end{aligned}
$$

To illustrate the effect of these attribute definitions, consider once again the program in Figure 5.2. An operation is invoked through a capability parameter in resource `r`. The value of this capability variable can only be determined by examination of the parameter values assigned to `z` through an invocation of operation `a`. The invocation in resource `s` that sends a capability for the operation `b` is one such invocation. The attribute definitions for the send statement affect the attribute *bnd*, recording the binding of `b` to the formal declaration parameter `y`. The attribute definitions for the implementation of operation `a`, i.e., the receive statement, associate the binding of values associated with formal parameter `y` to implementation parameter `z` and the result appears in the attribute *defs*. Thus, when the send statement in resource `r` is invoked, the possible values for the implementation parameter `z` can be determined.

This approach to interprocess and interresource analysis assumes that the results of an operation invocation are possible results for any invocation of an operation. This seemingly expedient assumption does not provide any significant liability in our analysis. This is because we are interested in optimizing commonly occurring interprocess communication and synchronization patterns. All of the programs that we have examined for potential optimization require rather simple propagation analysis. See Chapter 7 for further discussion of this issue.

5.3.2.1.5 Iteration Bounds

As stated earlier in this chapter, the number of iterations required to reach a fixed point depends upon the content of the program resources being analyzed and the order in which program resources are analyzed. For example, consider again the program in Figure 5.2. If the phase two analysis is performed in resource `main, s, r` order, then only one pass over the source program is actually required to reach a stable state and one additional pass is required to assure that the fixed point has been reached. However, if the resources are analyzed in the reverse order, three passes are required to reach a stable state plus one more to confirm the fixed point. The difference is that in the first case the analysis propagates resource creation and parameter binding information in the same order that the program propagates this information; whereas in the second case, resources are examined before binding information has been determined. In general, resources should be analyzed in reverse compilation order, i.e., the order in which resources need to be compiled as determined by import statements[4].

Theoretically, the number of iterations required to reach a fixed point can be quite large; in practice however, the number is quite small. While an upper bound on the number of iterations can be characterized in several ways, it is most simply characterized as follows. Each capability variable in a given program can be assigned from a finite set of values; let *N(x)* be the number of operations that are legally assignable to the capability variable x. Then, each place where x can be assigned a value (i.e., assignment statement or invocation with variable or result parameter) requires at most *N(x)* iterations to obtain all possible values of x. The total number of iterations is then the sum over all assignments to all capability variables plus one additional iteration to assure that a fixed point has been reached. It is possible to construct artificial programs for which the phase two analysis achieves this bound. In practice, however, the algorithm reaches a fixed point on most of our test programs within two to four iterations. See Section 7.2 for further discussion of this issue.

5.3.3 Phase Three — Operation Invocation Analysis

Phase three scans a different slice of the parse tree — one that includes all operation invocations — and computes the values of four attributes S_{same}, S_{diff}, C_{same}, and C_{diff}. Recall from Chapter 4 that S_{same}, S_{diff}, C_{same}, and C_{diff} are multisets of sets of operations

[4]Automatic determination of compilation order for SR is possible and has been implemented as the SRM utility [OW89].

— one set for each occurrence of a call or a send statement in a program. Each element of S_{same} is the set of operations that can potentially be invoked by a given send statement where the invocation and the invoked operation co-reside on the same virtual machine. Similarly, each element of S_{diff} is a set of operations that can be invoked by a given send statement where the invocation and the invoked operation cannot be guaranteed to co-reside on the same virtual machine. The elements of C_{same} and C_{diff} are similarly defined.

These multisets are affected primarily by invocation statements. The attribute equations for send invocation statements appear below. Send statements can have one of three forms: a direct invocation through an operation, an indirect invocation through an operation capability variable, and an invocation of an operation qualified by a resource capability variable. All three of these cases are illustrated below.

S : `send` $a(...)$

$$C_{same_{out}}(S) = C_{same_{in}}(S)$$
$$C_{diff_{out}}(S) = C_{diff_{in}}(S)$$
$$S_{same_{out}}(S) = S_{same_{in}}(S) \cup \{a\}$$
$$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup \{\}$$

S : `send` $y(...)$

$$C_{same_{out}}(S) = C_{same_{in}}(S)$$
$$C_{diff_{out}}(S) = C_{diff_{in}}(S)$$
$$S_{same_{out}}(S) = S_{same_{in}}(S) \cup \{x|\ ((y,u),(x,u),?) \in save(S, defs) \wedge u \neq unk\}$$
$$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup \{x|\ ((y,u),(x,v),?) \in save(S, defs) \wedge u \neq v\}$$

S : `send` $z.a(...)$

$$C_{same_{out}}(S) = C_{same_{in}}(S)$$
$$C_{diff_{out}}(S) = C_{diff_{in}}(S)$$
$$S_{same_{out}}(S) = S_{same_{in}}(S) \cup$$
$$\{b|\ r = currres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,u),?) \in save(S, defs) \wedge u \neq unk \wedge$$
$$b = resequiv(z,s,a)\}$$
$$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup$$
$$\{b|\ r = currres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,v),?) \in save(S, defs) \wedge u \neq v \wedge$$

$b = resequiv(z, s, a)\}$

The attribute equations for call statements are similar to those above, but they are more complex because they handle output parameters as well as input parameters. These definitions are provided in Appendix A.3.

Consider again the program in Figure 5.2. The parse tree slice including all invocation statements shown in Figure 5.7 illustrates the effect of these attribute definitions. (Resource `main` is not shown since it has no invocation statements; hence its attribute values are all {}.)

5.3.4 Virtual Machine Patterns — A Non-optimizable Example

Consider the program in Figure 5.8. Two virtual machine creation statements correspond to two virtual machine patterns. Notice that the communication between resources `r` and `s` crosses virtual machine boundaries similar to the example illustrated by Figure 4.15(b) in Chapter 4. Our analysis must prohibit the application of our optimizing transformations.

Phase one dataflow analysis identifies two virtual machine creation statements and generates pattern identifiers for each, say `vmpat1` and `vmpat2`. No looping constructs are present, thus the attribute *vmvars* is empty and not interesting. In phase two, analysis of the resource creation statements affects the values of attributes *defs*, *res*, and *bnd*. Upon completion of a single analysis pass over the resource `main`, the values of these attributes are as follows:

$$\begin{aligned}
defs &= \{((r1, initvm), (r, vmpat1), l), ((r2, initvm), (r, vmpat2), l),\\
&\quad\; ((s1, initvm), (s, vmpat2), l), ((s2, initvm), (s, vmpat1), l)\}\\
res &= \{(r, vmpat1), (r, vmpat2), (s, vmpat1), (s, vmpat2), (main, initvm)\}\\
bnd &= \{((x, vmpat2), (r, vmpat1), in), ((x, vmpat1), (r, vmpat2), in)\}.
\end{aligned}$$

Figure 5.7: Phase three parse tree slice for program in Figure 5.2.

```
resource main()
  import r, s
  initial
    var vm1, vm2: cap vm
    var r1, r2: cap r
    var s1, s2: cap s
    vm1 := create vm()
    vm2 := create vm()
    r1  := create r()  on vm1
    r2  := create r()  on vm2
    s1  := create s(r1)  on vm2
    s2  := create s(r2)  on vm1
  end initial
end main

resource s
  import r
body s(x: cap r)
  process p2
      op b()
      ...
      call x.a(b)
      ...
      receive b()
  end p2
end s

resource r
  op a(y: cap())
body r()
  process p1
      receive a(z)
      ...
      send z()
      ...
  end p1
end r
```

Figure 5.8: Program with interprocess communication crossing virtual machine boundaries.

Notice how the *bnd* associates the resource parameter with a machine pattern different from the pattern for the resource being created. The effect of this difference propagates through our attribute definitions.

Upon subsequent analysis of resource `s` (still in phase two), the values of attributes *defs* and *bnd* are affected as follows:

$$\begin{aligned}
defs &= \{((x, vmpat1), (r, vmpat2), g), ((x, vmpat2), (r, vmpat1), g)\} \\
res &= \{(r, vmpat1), (r, vmpat2), (s, vmpat1), (s, vmpat2), (main, initvm)\} \\
bnd &= \{((x, vmpat2), (r, vmpat1), in), ((x, vmpat1), (r, vmpat2), in), \\
&\quad ((y, vmpat1), (b, vmpat2), in), ((y, vmpat2), (b, vmpat1), in)\}.
\end{aligned}$$

The value of *res* remains unchanged. After phase two analysis of resource `r`, we have the following attribute values:

$$\begin{aligned}
defs &= \{((z, vmpat1), (b, vmpat2), l), ((z, vmpat2), (b, vmpatl), l)\} \\
res &= \{(r, vmpat1), (r, vmpat2), (s, vmpat1), (s, vmpat2), (main, initvm)\} \\
bnd &= \{((x, vmpat2), (r, vmpat1), in), ((x, vmpat1), (r, vmpat2), in), \\
&\quad ((y, vmpat1), (b, vmpat2), in), ((y, vmpat2), (b, vmpat1), in)\}.
\end{aligned}$$

Figure 5.9 presents the final phase three attribute parse tree slice illustrating the effect of analysis of a program whose processes potentially communicate across virtual machine boundaries. Phase three analysis determines that communicating instances of resources `r` and `s` do not co-reside on the same virtual machine. The attribute *defs* associated with each invocation statement provides this information. For example, the value of *defs* associated with the send statement in resource `r` indicates that when an operation is invoked from an instance of `r` on a virtual machine whose template is `vmpat1`, the operation being invoked is an instance of operation `b` located on a virtual machine whose template is `vmpat2`. That is, the communication is not between processes executing within the same virtual machine.

Figure 5.9: Phase three parse tree slice for program in Figure 5.8.

5.4 Transformer

The transformer uses the attributed parse tree generated by phase three of the dataflow analysis and the information generated by the the syntactic analysis to determine the applicability of the optimizing transformations. Determination is made using the formal characterizations presented in Section 4.3 of Chapter 4.

Semaphore and local procedure call optimizations are the easiest to determine: they simply require examination of the operations potentially invoked by each invocation statement within the program along with additional static information, such as which operations are parameterless. Upon identifying a semaphore optimizable operation the transformer appends the compiler directive `SEM` to the appropriate invocation and implementation statement nodes in the attributed parse tree. For a local procedure call optimization, the transformer appends the compiler directive `LOCAL` to the appropriate invocation statement node in the attributed parse tree.

For the client/server optimizations, more complex tree pattern matching is required to determine potentially optimizable situations. Our prototype is written in Prolog, a language that provides tree pattern matching facilities. Using these facilities, optimizable client/server patterns can be identified. Once identified, the entire subtree associated with the server process is replaced by a subtree with the transformed server. A single process node is replaced by multiple proc nodes — one for each arm in the server's optimizable input statement. In addition, all client process invocation parse tree nodes are appended with the `LOCAL` compiler directive.

The order in which these optimizations are searched for and identified must be considered. Our approach is to apply optimizations providing the most benefit for the least effort first. The simple semaphore and local procedure call optimizations meet this criteria. Since they affect a non-overlapping set of operations, the order in which they are applied is not important. The more difficult client/server optimizations are considered only after all semaphore and local procedure call optimizations have been applied.

Another reason for this order of application is that the set of semaphore and the set of client/server optimizable operations may be overlapping. Simple semaphore optimization provides a better optimization than the client/server optimization and is best applied first.

Identification of additional optimizations may have an affect on the order of application. For further discussion, see Chapter 7.

5.5 Re-Sourcer

The re-sourcer takes as input the modified parse tree produced by the transformer. It simply traverses the tree emitting source code to yield the transformed program. The tree pattern matching facilities present in our implementation language, Prolog, make this a trivial task.

Chapter 6

Implementation and Performance

In this chapter, we describe the Prolog-based implementation of the transformation schema presented in Chapter 5. Prolog was chosen as the implementation language for several reasons. Tree pattern matching is inherent in the language. Prolog Definite Clause Grammars provide a convenient tool for parsing and evaluating attributes in the syntactic and dataflow analysis stages of our transformation approach. Syntactic tree slices are easily obtained. The actual transformations, which involve tree modification, are relatively easy to implement in Prolog; the re-creation of the source program is a trivial task. The following sections describe the Prolog-based implementation, provide preliminary performance results, describe implementation limitations, and reflect upon some of our experiences with the Prolog language.

6.1 Prolog-based Implementation

The organization of the Prolog implementation mirrors the transformation schema presented in Figure 5.1 in Chapter 5.

The syntactic analyzer uses Prolog's Definite Clause Grammars to parse source programs, create an abstract syntax tree, and compute syntactically determinable attributes. To illustrate the ease with which this is done, consider the Prolog clause below.

```
resource_comp-- >
```

```
[resource],[X],
spec,
body.
```

To recognize a resource_comp, it is necessary to satisfy the goals on the right-hand-side of the rule, i.e., recognize the token "resource", followed by an identifier which is bound to the variable X, followed by a resource spec clause, followed by a resource body clause. [1]

Attributes can be associated with grammar rules constructing the abstract syntax tree as a by-product of the parsing process. The clause below illustrates the construction of a syntax tree node for resource_comp. This node has three values associated with it: X the resource name, Y the subtree for the resource specification, and Z the subtree for the resource body.

```
resource_comp(resource_comp(X,Y,Z))-- >
    [resource],[X],
    spec(Y),
    body(Z).
```

Additional attributes may be attached to the grammar rules thus enabling the tracking of syntactically determinable data such as P, the set of parameterless operations that is used to determine the applicability of the semaphore optimization (see Section 4.3.1). The clauses below illustrate how this set is generated using attributes. The attribute P2 in the rule for resource_comp actually receives its values from the rule for decl, where the rule recognizing operation parameters returns the value nil when an operation has no declared parameters. The goals appearing within braces, in the rule for decl, represent semantic actions to be performed when the syntactic structure for a decl is recognized.

[1] Prolog conventions state that identifiers beginning with an upper-case alphabetic character represent program variables and those beginning with lower-case characters represent constants or clauses. Specific SR tokens, such as reserved words or identifiers, are enclosed in brackets.

```
resource_comp(resource_comp(X,Y,Z),P2)-- >
    [resource],[X],
    spec(Y,[],P1),
    body(Z,P1,P2).

spec(spec(X,Y),P1,P2)-- >
    spec_comp(X,P1,P2),
    params(Y).

spec_comp(spec_comp(X,Y),P1,P2)-- >
    decl(X,P1,P2),
    spec_comp(Y).

decl(decl(X,Y),P1,P2)-- >
    [op],[X],
    params(Y),
    {(Y == nil- >
        P2 = [X|P1].
    ;P2 = P1
    )}.
```

The rules provided above have been simplified to facilitate presentation. For example, a `decl` in the actual implementation includes many types of declarations (variable, capability, pointer, etc.) in addition to an operation declaration. Similarly, a `spec_comp` includes components other than declarations, e.g., import statements. The grammar for the entire SR language requires approximately 450 clauses and 1200 lines of code.

Phase one of the implementation operates on the abstract syntax tree created during the syntactic analysis. As described in Chapter 5, this phase identifies virtual machine creation patterns. Although we suggested in Chapter 5 that this phase of the analysis can be combined with the syntactic analysis, our implementation uses two distinct passes to simplify program development. Our phase one analyzer currently handles all aspects of the SR language except the concurrent invocation statement. The phase one analyzer is short — approximately 250 clauses and 350 lines of code.

The iterator, phase two of the dataflow analysis, takes the decorated parse tree as input. It re-evaluates the *defs*, *res*, and *bnd* values for the tagged tree nodes creating a new parse tree with updated values for *defs*, *res*, and *bnd* until the evaluation causes no change in the values of these attributes. Most of the SR language features have been implemented in phase two. Both interprocess and interresource analysis have been implemented, tracking patterns for operations, resources, and virtual machines across process and resource boundaries. However, several omissions should be noted. As in phase one, the concurrent invocation statement is not implemented. In addition, analysis of three other language features is not implemented: next and exit statements, actual operation invocation parameters that are operation invocations themselves, and capability arrays. The phase two analyzer, the most complex part of the transformation analysis, uses roughly 750 clauses and more that 900 lines of code.

Phase three analysis, as described in Chapter 5, generates the sets of operations invoked by each occurrence of a call or send invocation statement. Given the attribute *defs* generated in the phase two analysis and attached to each invocation tree node, the phase three attributes are easy to compute. Phase three consists of approximately 250 clauses and 350 lines of code.

Finally, the transformer uses the attributes generated by both the syntactic analysis and the dataflow analysis to determine the applicability of our optimizing transformations as specified in Section 4.3. First, the set of semaphore optimizable operations is determined. Then, the syntax tree is traversed identifying all semaphore optimizable invocations and implementations. For each optimizable instance, an internal compiler directive is attached to the syntax tree. Next, the set of operations whose invocations represent remote procedure calls potentially implemented as local procedure calls is determined. Then the syntax tree is traversed once again flagging the appropriate invocations with compiler directives identifying the invocations as LOCAL. The third type of transformation, the client/server transformations, is more complex. There are many different forms for client/server optimization including client/servers using rendezvous, client/servers using asynchronous message passing, and client/servers using a combination of both. Our implementation currently transforms only client/servers using rendezvous. Transforming the other forms requires slight modifications of the existing code and are planned for future implementation.

6.2 Performance

The optimizer, in its current form, is intended to show feasibility. The program code is not optimized and no attempt has been made to make the code efficient. For example, the use of cuts to reduce the number of choice points is tentative and inconsistent. Two implementation decisions have already turned out to be less than ideal. The first is the decision to make the symbol table nonglobal. The symbol table is carried as an attribute during parsing and attached to all tree nodes that may potentially require its presence during the dataflow analysis. This representation is convenient for scoped languages like SR. Unfortunately, this results in far too many copies of the symbol table attached to too many nodes and ultimately in bloated memory requirements. The second is the input mechanism currently being used to input SR source programs into a Prolog token stream. In an effort to rapidly produce an implementation, it was decided to use the Unix tool Lex to scan an SR source program and to produce a Prolog clause that assigns a list of tokens to a Prolog program variable. The Prolog implementation then compiles this clause before any program analysis is performed. The time required to compile this clause is significant for programs of any size and, more importantly, clauses have a maximum size limitation, which translates into approximately 650 lines of SR code. This limitation is clearly a serious one for anything other than a prototype or toy system. We discuss this issue further in the next section.

We present our preliminary performance results in Table 6.1. SR programs with up to 600 lines of code were processed by the optimizer and the CPU times for execution of the analysis and transformation tasks were recorded using the Prolog built-in `statistics`

CPU time (secs)

optimizer phase	lines of program code						
	50	100	200	300	400	500	600
program input	1.400	3.300	8.100	11.900	17.833	22.283	30.500
syntactic analysis	.133	.233	.400	.634	.766	.916	1.117
phase one	.000	.017	.033	.033	.050	.067	.067
phase two	.117	.216	.333	.516	.600	.650	.767
phase three	.016	.150	.217	.284	.333	.316	.383
transformation	.000	.083	.150	.267	.284	.317	.317
re-sourcing	.033	.067	.167	.250	.283	.350	.433
total	1.699	4.066	9.400	13.884	20.149	24.899	33.584
total less input	.299	.766	1.300	1.984	2.316	2.616	3.084
Memory Usage (Kbytes)							
program	429	435	443	448	453	457	461
data	524	983	1,835	2,293	2,490	2,949	3,277
total	953	1,418	2,278	2,741	2,943	3,406	3,738

Table 6.1: Preliminary performance results for prototype optimizer.

predicate. The numbers reported are for Quintus Prolog Release 3.0 running on a lightly loaded SPARCstation with 8 Mbytes of memory. Program and data space memory requirements for the transformation of these programs are also reported. Greater than 50% of the program statements in the test programs employed interprocess communication and synchronization. That is, those statements required examination during the dataflow phases of the optimization. Clearly, most real programs contain a significantly lower percentage of such statements; the analysis costs would therefore be significantly lower, as we have observed for other programs not reported here.

The poor choice of input mechanism is reflected in the large CPU times reported for program input. Similarly, the large memory usage is in part due to our implementation's inefficient use of this resource.

The data in Table 6.1 indicate that the relationship between program analysis time (i.e., "total less input" in Table 6.1) and program size is linear. Extrapolating this data indicates that an SR program with 10,000 lines of code could be optimized in approximately

50 secs, ignoring input time. Unfortunately, the data in Table 6.1 does not show a clear relationship between memory usage and program size. Thus, extrapolation of memory usage for a 10,000 line program is not so clear. The numbers given above are not consistent enough to provide any significant hints as to memory versus program size growth patterns except that memory will increase. This inconsistency may be due to the way in which Quintus Prolog allocates memory and does its garbage collection. For example, memory may be allocated in increasingly larger blocks to reduce anticipated requests for this resource. Clearly, an effort must be made to reduce memory requirements. One possible approach is to convert the symbol table to a global structure, thereby reducing the many copies currently attached to various parse tree nodes.

As noted earlier in this section, these statistics are preliminary and are very dependent upon the Quintus Prolog implementation. The reported results are not necessarily representative of all programs or all Prolog implementations. We present the results merely to provide some insights into the optimizer's performance. To summarize, the execution times, less program input costs, are encouraging. The space requirements are not conclusive yet, but they may present a potential problem. We continue to investigate these issues, but first we must re-implement the input mechanism so that we can handle larger programs.

6.3 Implementation Limitations

The implementation of our optimizer, in its current form, demonstrates the feasibility of our optimizations and the techniques used to apply them. Several modifications and/or additions are necessary to turn this prototype into an actual optimization tool. First, several SR language features remain to be implemented. These include the next and exit statements, the use of operation invocations in actual invocation parameters, concurrent invocation statement, and capability arrays. The first two features were omitted in the current implementation because they do not directly affect the performance of the mechanisms being optimized, they were not particularly interesting, and their implementation would have prolonged development of the prototype. Their implementation requires additional time but not additional analysis techniques. However, the second two features, concurrent invocation and capability arrays, require additional analysis techniques. The optimization of the concurrent invocation statement is of particular interest and further investigation is planned. The second modification involves the mechanism used to input the original SR source into the Prolog optimizer. This input mechanism can be changed to more efficiently do its own tokenizing and thus bypass the cost of clause compilation described in the previous section. Some additional effort is required to solve the size limitations. The most reasonable approach would be to create distinct token lists for each resource in the SR program. Finally, the existing SR compiler must be modified in two ways. The compiler must be able to handle the compiler directives generated by the optimizer and to generate the appropriate intermediate code. These modifications will require a relatively small effort.

6.4 Reflections on Prolog

The choice to use Prolog was made during the early stages of this project before any implementation work was attempted. The need for high-level features to perform list and tree manipulation as well as tree pattern matching was anticipated. We had a pretty good idea of what would be involved in a C implementation and it was not appealing: C provides no high-level list or tree manipulation mechanisms. Icon [GG90], the other language that we considered, was more appealing than C. It has built-in mechanisms to handle both lists and trees, but it still lacks the tree pattern matching capabilities that we needed. Prolog advertised all of the features that we desired. Our knowledge of Prolog was at the time minimal. Thus, another factor entering into our decision was curiosity: we were interested to see if the advertised benefits were in fact real. Our experience tells us that they are. In addition to the anticipated benefits, there have been several unforeseen benefits from the decision to use Prolog, including the Definite Clause Grammars used to parse programs and to propagate attributes. Once the decision to use Prolog was made, it became unclear whether or not the choice of language was directing other choices as well. The decision to use the method of attributes for the dataflow analysis was largely due to the ease with which attribute grammars can be handled in Prolog. The actual expression of specific optimization constraints is also handled easily in Prolog; however, their characterization is probably more independent of language choice.

Although grammars, attributes, parse tree pattern matching, and the characterization of optimization constraints are all handled easily by Prolog, we do note one unforeseen drawback affecting program development and evolution. Often, the structure of a data item is exposed when the item is passed as an argument to a clause. This is usually done so that Prolog's unification mechanism provides a form of clause selection. For example, in the clause below, the structure of the syntax tree is exposed.

```
resource_comp(resource_comp(X,Y,Z))-- >
    [resource],[X],
    spec(Y),
    body(Z).
```

In particular, the tree node for `resource_comp` has three children, `X`, `Y`, and `Z`. If the structure of that node is modified to have four children, all clauses that reference this node must then be adjusted to reflect the new structure. Some form of data hiding is possible, but it seems counter to the nature of Prolog programming and also introduces unwanted overhead. [O'K90]

Chapter 7

Discussion and Future Work

As stated by Aho, Sethi, and Ullman, "the best transformations are those that yield the most benefit for the least effort."[ASU86] They state further that optimizing transformations must preserve the semantic meaning of the original source program, that they must, on average, improve the run-time performance of programs, and finally, that the time spent in additional program analysis must be worth the effort. The first two criteria are satisfied easily. We described, in detail, in Chapters 3 and 4 the constraints on the application of our transformations that *informally* ensure their semantic correctness. We discuss *formal* verification of the transformations in Section 7.4. In addition, Chapter 3 cites references for performance measurements for our simpler transformations including semaphore optimization and remote-to-local procedure call. Additional measurements are presented for the more complex client/server optimizations using rendezvous and asynchronous message passing. These measurements provide convincing evidence of the improvement effected by our transformations. The third criterion — the additional program analysis must be worth the effort — is a more subjective issue. Clearly, the program analysis required for our transformations is significant. In general, for programs run only once or twice, the analysis is certainly not worth the effort. One exception, however, is programs with hard real-time execution constraints that cannot be met without optimization. For these programs and others running in a production environment, the improvement afforded by our optimizing transformations could be not only significant, but also necessary and worth the cost of a single optimization effort following program debugging and testing.

In this chapter, we discuss the merits and limitations of our transformations and our analysis techniques. We identify areas for future research that will provide a more complete approach to optimization of interprocess communication and synchronization mechanisms, as well as potential enhancements to our existing system. In addition, we describe the relevance of this effort to concurrent programming languages other than SR.

7.1 Optimizing Transformations

The source-level transformations presented in Chapter 4 have been applied to SR programs. Although the examples presented there use only single resource programs, they can be applied to multiple resource programs as well. The semaphore optimization and remote-to-local procedure optimization require communication between processes executing within resources residing in the same shared address space. In the case of the client/server model, clients may reside in different resources, assuming that the multiple resources reside in the same shared address space. Although the client/server transformations are intended for use in a shared memory environment, they are also semantically valid in a distributed environment when all of the inactive procedures or active processes used to implement the server are located (as is reasonable) on one processor. The calls to the procedures from the client tasks simply become remote procedure calls. For client/server models using rendezvous, whether or not these transformations to inactive procedures provide any reduction in execution time in a distributed environment depends upon the relative cost of rendezvous and remote procedure call. The optimizer could consider these costs as a condition for application of the transformations. Certainly no significant reduction in overall execution time will be achieved since the dominant cost is associated with the communication between machines rather than with the calling and servicing mechanisms at either end of the communication. However, the transformed server process will execute more efficiently than the original server.

These source-level transformations optimize commonly occurring patterns of interprocess communication. However, additional approaches to optimization can be used to extend the types of optimization that can be performed. We describe several of these below.

7.1.1 Additional Approaches to Optimization

Several different approaches can be used to extend our existing compiler optimization of interprocess communication mechanisms. One such approach uses problem-specific semantic information in the form of programmer supplied directives or assertions to aid optimization. This approach remains at the source level. Another approach is to examine the relationship between program source code and the underlying run-time support system used to implement interprocess communication mechanisms. This approach can require significant modification of the compiler's code generator and the run-time support system itself.

7.1.1.1 Use of Assertions

Programmer supplied directives or assertions can be used to extend the range of applicable transformations. Consider once again the *readers/writers* problem first appearing in Figure 2.4 of Chapter 2. The invariant assertion typically used to describe this problem is

$$nr \geq 0 \wedge nw \geq 0 \wedge nw \leq 1 \wedge (nr = 0 \vee nw = 0)$$

where nr is the number of readers and nw is the number of writers. With intelligent analysis of this assertion and the input statement synchronization expressions used to constrain the servicing of reader and writer requests for service in the solution appearing in Figure 2.4, the synchronization expressions can be removed as follows. The assertion yields three possible situations: $nr = 0 \wedge nw = 0$, $nr > 0 \wedge nw = 0$, or $nr = 0 \wedge nw = 1$. Given these three possible cases, we can construct a new server using a do statement with one arm for each case. The guard on each arm tests to see whether its case is currently valid; the body

```
process server
    do nr = 0 & nw = 0 ->
        in read_request() -> nr++
         [] write_request() -> nw++
         [] read_release() -> nr--
         [] write_release() -> nw--
        ni
     [] nr > 0 & nw = 0 ->
        in read_request() -> nr++
         [] read_release() -> nr--
         [] write_release() -> nw--
        ni
     [] nr = 0 & nw = 1 ->
        in read_release() -> nr--
         [] write_release() -> nw--
        ni
    od
end server
```

Figure 7.1: Server code derived from assertion-based transformation.

of each arm then executes an input statement that services only those operations valid in that case. These input statements no longer need synchronization expressions. The operations to be included in these input statements can be determined by examining only their synchronization expressions. Figure 7.1 contains the server transformed using this scenario. Notice that operations `read_release` and `write_release` are not constrained by synchronization expressions and, thus, they appear in all input statements. Additional analysis of the bodies of code within the input statement arms can be used to eliminate unnecessary implementations of these operations. For example, in the first case where $nr = 0 \wedge nw = 0$, execution of either statement `nr--` or `nw--` would violate the invariant assertion; thus, neither operation is eligible for servicing in this case.

Section 3.4.2.2.3 presents a transformation and performance results for the *readers/writers* problem that could be derived using these assertion-based techniques. The transformation analysis is assuredly more sophisticated than the analysis required for any of the optimizations presented in this dissertation. In addition, assertions must either be provided by the programmer or derived automatically by the analyzer. However, we feel that additional effort is warranted in this area, particularly because this type of optimization is often independent of shared memory/distributed environment and because the timing results for this type of optimization show, on the average, an improvement in run-time performance (see Figures 3.12 and 3.13).

7.1.1.2 Additional Source-level Transformations

The investigation of additional source-level transformations is of great interest. One such kind of optimization is analogous to procedure *cloning* in interprocedural optimization [CKT86]. Procedure cloning replaces a procedure in the original source code by several specialized procedures in the object code. One motivation for doing so is to reduce testing of parameter values. For example, if the original procedure takes different actions based on the value of one parameter, a specialized procedure can be created for each of the possible actions, thus eliminating the parameter passing and testing. In the context of our transformations, we could clone processes or resources, although we currently do not. The implementation of procedure cloning requires the dataflow analysis of conventional variables. This type of optimization warrants further effort.

7.1.1.3 Run-Time Support Dependent Optimizations

The relationship between a program and the underlying run-time support system provides another opportunity to expand our current range of optimizations. These new optimizations typically require new or modified run-time support system routines and modification of the compiler's code generator since the generated code must interact with the run-time system. Several such optimizations are currently under investigation. One is an optimization of asynchronous message passing operations with parameters. This optimization is similar to the semaphore optimization in that it requires that communication is between processes in the same shared address space. The difference is that this optimization permits parameter values to be passed between the communicating processes, thus, semaphores alone are not sufficient. To enable this optimization, the run-time support system must provide routines implementing message passing in shared memory, in addition to the already present general routines that currently implement message passing operations. A routine to send messages would simply place the message in shared memory, update some pointers to the message area, and then signal a semaphore associated with the operation. The routine to service a message would wait for the operation semaphore, and then extract the first message in the shared memory area and update pointers to the memory area. It then becomes the task of the compiler to determine which run-time support routines — the general ones or the new specialized ones — should be invoked.

Another such optimization involves operations implemented with scheduling expressions. Ideally, in this situation, pending invocations of the operation should be queued in scheduling expression order to eliminate list traversal time. In general, this optimization is not possible, and pending invocations are queued in FCFS order. However, in some cases where the scheduling expression is a simple function of an invocation parameter, the invocation queue can be maintained in scheduling expression order. More precisely, such ordering is desirable only when all implementations of an operation in all arms of all input statements use scheduling expressions that service invocations in the same order. The run-time support must supply specialized routines maintaining the ordered invocation queues.[McN89]

Yet another optimization involving specialized run-time support routines is applicable only to the SR language. Recall that SR places pending invocations for operations serviced by the same input statement on the same queue. There are some circumstances where this implementation leads to unnecessary inefficiencies. For example, consider once again the *readers/writers* problem. All invocations for all operations are placed on the same pending queue. Consider the situation where a `writer` process has been given permission to write the database and many `reader` processes invoke the `read_request` operation before the `writer` process releases the database by invoking `write_release`. These invocations are all queued on the same pending queue and each time a `read_request` is made the server is awakened to see if the request can be serviced. Each time the server is awakened, it examines pending invocations in order until either it finds one that it can service or it determines that it can service none. When the `writer` process finally releases the database, all prior pending invocations are examined before the `write_release` invocation is selected. A more efficient approach would be to determine which operations are eligible for servicing based on their synchronization expressions and to have direct access to lists of pending invocations for these specific operations. (Note that this is possible only if the synchronization expressions are not a function of specific invocation parameters.) Direct access to invocations for specific operations can be obtained through the use of threaded pending queues. These queues provide both a single thread for all operations serviced by the input statement, and individual threads for each specific operation. Threaded queues have been implemented for an experimental version of SR. This version uses these queues for all pending queues and the overhead introduced by the additional threads was roughly 1-2% for programs where we expected to see no advantage. There was noticeable improvement in programs using arrays of operations where we expected to see an improvement (a 14% run-time reduction in some cases). To date, no compiler analysis exists to determine appropriate use of these specialized pending queues. We anticipate additional effort in this area.

One additional optimization under consideration relates to the implementation of message passing described in Section 3.2. SR permits multiple implementations of operations. Thus, many processes may block (on an operation process queue) attempting to service the same operation. When the operation is invoked, the existing run-time support system awakens all the processes that are blocked attempting to service the operation. There are many easily identifiable instances where the run-time support system could correctly awaken only the first blocked process. We anticipate a significant performance improvement from this optimization.

Further effort is required to determine additional optimizations and appropriate application of those described above. The presence of additional optimizations may affect the performance and applicability of those presented in this dissertation.

7.2 Program Analysis Techniques

Chapter 4 described the requirements for intraresource and interresource analysis used to determine the applicability of our optimizing transformations. The requirements differ for the different transformations. The procedure call transformation optimizes specific invocations. That is, only some invocations may be replaced by local procedure call, the others remain unchanged and use the general invocation mechanism; in particular, those that are on different virtual machines are not changed. In contrast, the semaphore transformation requires that *all* invocations of a specific operation meet the requirements stated earlier. The client/server transformations are similar in that they require that all invocations of a given operation (in all resource instances) be either call or send invocations.

Most of these requirements are relatively straightforward to check. However, information indicating how each operation is invoked requires dataflow analysis, interprocess, and interresource analysis.

Our dataflow analysis techniques propagate values for operation, resource, and virtual machine capability variables within processes and across process and resource boundaries. In essence, we are treating the dataflow problem as a problem of *constant propagation*. This is possible because we treat operations, resources, and virtual machines as templates or patterns for run-time instances. Thus, operation capability variables can be assigned one of a discrete, statically determinable set of operation templates. Similarly, resource capability variables can be assigned one of a discrete, statically determinable set of resource templates. Virtual machine capability variables are different in that templates are not, in general, statically determinable. However, it is possible, in some cases, to identify virtual machine patterns and propagate their values. In other cases, it is necessary to indicate that the values are unknown. This approach is conservative but necessary to ensure correct application of the transformations.

The iterative algorithm used to propagate capability variable values is theoretically much worse than reality. This is because conventional programming techniques do not typically produce programs exhibiting worst case behavior. For example, the analysis of the *readers/writers* example presented in Figure 2.4 of Chapter 2, typical of many client/server problems, requires only two iterations to reach a fixed point. The analysis of the contrived program presented in Figure 7.2, however, requires five iterations (the predicted worst case) if program analysis is performed in left-to-right (top-to-bottom) order. It takes one iteration for the operation `a` to propagate from process `p5` to proc `p4`'s parameter `z`. It takes another iteration for `a` to propagate from proc `p4` to proc `p3`'s parameter `y`, and so forth. The four capability parameters each requires a set of potential values. Each parameter can assume at most one value, `a`, therefore at most four iterations are required to reach a stable state and one additional iteration is necessary to confirm that a fixed point has been reached.

Although the analysis of this simple example achieves the worst case bound described in Chapter 5, the analysis of more complex programs with larger numbers of operations typically will not. This is because, in most cases, not all variables can assume all possible operations as values, and because multiple values are often propagated in one iteration. The worst case bound simply serves to assure that a fixed point will be reached.

7.2.1 Analysis Limitations

The dataflow analysis techniques described in Chapter 5 have several limitations. First, we chose to focus on the dataflow analysis of interest to our research. Thus, the analysis is applied only to capability variables; conventional variables are ignored. The addition of dataflow analysis involving traditional variables would facilitate the recognition of optimizable forms that are currently not possible. For example, the bounded client/server model in Figure 7.3 is optimizable if our analysis can determine bounds on the number of

```
resource contrived()
    ...
    proc p1(w)
        ...
        send w()
        ...
    end p1

    proc p2(x)
        ...
        send p1(x)
        ...
    end p2

    proc p3(y)
        ...
        send p2(y)
        ...
    end p3

    proc p4(z)
        ...
        send p3(z)
        ...
    end p4

    process p5
        op a()
        ...
        send p4(a)
        receive a()
        ...
    end p5
end contrived
```

Figure 7.2: Contrived program for which analysis achieves worst case iteration bound.

```
process server
   fa i := 1  to 10 ->
        in oper() -> ...  ni
        ...
    af
end server

process client(i := 1  to 10)
        ...
        call oper()
        ...
end client
```

Figure 7.3: Bounded client/server model.

operation invocations. These bounds must match the bounds in the server process.

Another limitation of our existing dataflow techniques involves abnormal loop control statements, i.e., next and exit statements. We currently provide no dataflow definitions for these statements, although we expect to do so in the near future.

Another more interesting issue involves our approach to interprocess analysis. The techniques described in Chapter 5 assume that the results of an operation invocation are possible results for any invocation of an operation. This assumption is reasonable when the values returned as a result of an operation invocation are independent of operation input parameters. However, when results are dependent upon input parameters, our techniques are overly conservative. For example, consider the program fragment in Figure 7.4. Our current techniques will bind both operations `g` and `h` to the `result` parameter of `p2`. It is possible, with additional analysis of conventional variables and a more sophisticated approach to propagating parameter values, that our interprocess analysis could be more precise. Our analysis would have to make all invocation bindings occurrence-specific. In order to determine occurrence-specific output bindings, the operation implementation must be analyzed once for each set of invocation occurrence bindings. Although analysis of

```
op p2(i: int; res x: cap())
   ...
process p1
       ...
    call p2(1,x)
    send x()
       ...
    call p2(2,x)
    call x()
       ...
end p1

proc p2(i,result)
   op g(), h()
       ...
   if i=1 ->
       result := g
   []  else ->
       result := h
   fi
       ...
end p2
```

Figure 7.4: Program fragment requiring invocation occurrence-specific analysis.

conventional variables certainly require this level of effort, it is not clear if the optimization of commonly occurring patterns of interprocess communication and synchronization requires such an effort.

Finally, as noted in Chapter 6, two language features require further study before they are incorporated into our optimization tool. These include arrays of capabilities and the concurrent invocation statement. Static analysis of capability arrays requires the analysis of conventional variables used as indices. Further investigation of programs using these constructs is required before we attempt their optimization. Similarly, the concurrent invocation statement requires further study before techniques are developed for its optimization. We suspect that this construct will yield significant opportunity for optimization at the run-time support level, but this remains to be verified.

7.3 Relevance to Other Languages

As stated in Chapter 3, our optimizing transformations are applicable to SR, to Concurrent C, and in some cases, to Ada. For the client/server transformations, Ada's semantics for guarded selection prohibits elimination of the server process, making these transformations inappropriate as an automatic compiler optimization. However, they are still appropriate, in some cases, as high-level source-to-source transformations for Ada. Concurrent C, being similar to SR, is also a candidate for automatic optimization. Ada and Concurrent C both have a nondeterministic selection policy making their transformations more efficient than those implementing SR's FCFS selection policy.

Guarded operations with parameters may be transformed as long as their synchronization expressions are independent of operation parameters. This restriction is relevant for SR and Concurrent C, but not Ada where guards may not reference operation parameters.

Some of our transformations depend on the availability of inexpensive semaphores in the base language or its implementation. Such is the case in Ada, which provides semaphores as a package. We also assume that the cost of rendezvous is higher than that for procedure calls, semaphores, and context switching. That is certainly the case in a number of languages [Clap86,AtOl88].

The SR language continues to evolve. Version 2 [AO, Ols91] enables the programmer to declare operations and variables within a virtual machine and thus global to any resources also created within that virtual machine. The presence of these global operations and variables will, in some cases, simplify, but not eliminate the need for program analysis to ensure appropriate application of our transformations. Because global operations can be declared, fewer operations will need propagation through capability variables or parameters. However, capability variables and parameters are still required to propagate operation, resource, and virtual machine capabilities across virtual machine boundaries, and may be used within virtual machine boundaries. Thus, dataflow, interprocess, and interresource analysis tracking capabilities will continue to be necessary in optimizing SR version 2 programs.

7.4 Formal Correctness of Transformations

Although we describe constraints necessary to ensure the correct application of our optimizing transformations, the justification that these constraints are necessary and sufficient is informal. Current interest in compiler verification dictates the need for a formal proof of the semantic correctness of our transformations. In general, defining the formal semantics of concurrent languages is a difficult problem [AFD80, Lam80, OG76]. The problem is particularly difficult for the SR language because of its range of communication and synchronization mechanisms and its dynamic process and resource structure. Although some progress has been made [HO], a complete axiomatic description of the SR language does not exist. Therefore formal proofs of the transformations are not yet possible.

7.5 Debugging Issues

One drawback of employing source-to-source transformations, as advocated in this dissertation, is that they can make debugging more difficult. In particular, the program as written by the programmer is not the same as the one being executed and debugged. The problem of debugging optimized programs has been investigated for sequential programs [Hen82]. Debugging concurrent programs, however, is more difficult [GMGK84, MH89], and the transformations described in this paper can exacerbate the difficulties. Specifically, the transformations might have: added or eliminated processes; added, eliminated, or modified operations and their invocations; or altered the order in which processes execute. One solution is to debug the original untransformed program. However, some concurrent programming errors arise only when processes execute in a particular order. If the transformations affect that order, then debugging the transformed program is not the same as debugging the original program. This problem is worthy of further investigation.

7.6 Optimizer as a Separate Tool

We decided to make the optimizer a separate tool rather than an integral part of the SR translation process primarily because the SR language was and still is undergoing a revision and the existing compiler was being completely rewritten. Thus, using the SR compiler would have bound us to a nearly obsolete version of the language. Not wanting to be dependent upon the progress of the new compiler, we chose to develop the optimizer as an independent tool. Having worked with the implementation for some time, it is clear that the optimizer could in fact be adapted to include the entire parsing and code generation process, similar to the compiler written in Prolog for a "toy" Algol-like language described by Warren [War80]. The optimizer already parses, generates the symbol table, and performs a significant amount of program analysis performed by the actual SR compiler. To create a stand-alone optimizing compiler would require additional error handling and recovery during the parsing phase, additional syntactic analysis currently not performed by the optimizer, and finally intermediate code generation. All of these tasks are fairly easily incorporated into the framework of the existing optimizer; we plan to further investigate this issue.

Chapter 8

Summary

This dissertation provides, what the author believes to be, the first comprehensive effort to optimize high-level interprocess communication and synchronization mechanisms at the compiler level. In Chapter 3, we described source-level transformations for mechanisms including asynchronous message passing, rendezvous, and remote procedure call. We describe the simple semaphore optimization of asynchronous message passing, the remote-to-local procedure call optimization, and the optimization of commonly occurring client/server models. These optimizing transformations require that communicating processes reside in the same shared address space. We present new transformations for client/server models using rendezvous and asynchronous message passing. These new transformations significantly improve the performance of programs where communicating processes reside in the same shared address space. All of the transformations require significant program analysis to automatically determine their applicability. Additional transformations, valid in a distributed environment, are also discussed. Their impact on performance is not as consistent, although, on average, they do improve performance. In addition, they are more difficult to automate.

In Chapter 4, we discussed the many language issues affecting the automatic application of the optimizing transformations and characterized formally the constraints for each of the optimizations presented in Chapter 3. The analysis requirements for automatic application of our optimizing transformations necessitate both syntactic and dataflow analysis.

Syntactic analysis can be used to determine how operations are implemented. However, how they are invoked can require dataflow analysis along with interprocess and intermodule analysis to track values of capability variables and parameters. For programs executing on a single virtual machine, the above analysis suffices to determine optimizability. For programs executing on multiple virtual machines, additional analysis is required. Co-residency of program resource instances can in some cases be determined providing additional optimizations for these programs.

Several observations or assumptions are fundamental to our analysis. The most fundamental is that SR programs provide templates for operations, processes, resources, and virtual machines. We can characterize the behavior of all run-time instances of an operation by examining the source code. More specifically, we can statically determine all possible values for all operation capability variables because each can only be assigned one of a discrete compile-time determinable set of operation templates and assignment is the only operation that can be performed on these variables. Similarly, we can statically determine all possible values for all resource capability variables because each can only be assigned one of a discrete compile-time determinable set of resource templates. Finally, we observe that virtual machine templates or patterns can be tracked statically for many commonly occurring situations.

The application of these source-level transformations necessitates the extension of data-flow techniques to concurrent languages. In Chapter 5, we presented new techniques for handling shared variables and the extension of interprocedural analysis to interprocess analysis. The use of attribute grammars to perform data-flow analysis extends previous uses for attribute grammars. Perhaps the most difficult task has been the analysis required to determine when process interactions are between processes residing in the same shared address space. For languages allowing dynamic process creation, as SR does, this problem is not solvable in the general case; however, it is possible to identify specific situations where interacting processes are guaranteed to reside in the same shared address space.

In Chapter 6, we described an actual implementation of the automated application of the optimizing transformations for the SR language. Although currently incomplete, a large portion of the language is implemented and this implementation illustrates the feasibility of our analysis techniques.

In Chapter 7, we discussed the merits and limitations of our optimizations and analysis techniques. We note that the optimization analysis is significant and should not be performed until program development has been completed. One drawback of these transformations is that they complicate the debugging process.

Key issues for future research, as mentioned in Chapter 7, include a more comprehensive approach to optimization of interprocess communication and synchronization mechanisms including run-time support dependent optimizations, interprocess analysis enhancements, and the formal semantics of SR and the verification of the transformations. In addition, we are interested in assertion-aided transformation and the performance of our transformations when applied to languages such as Ada and Concurrent C. Finally, our implementation requires enhancements to accommodate the entire SR language and we plan to improve its performance with the intention of using it as a real-world tool.

Appendix A

Attribute Definitions

The following sections present the attribute definitions used in the dataflow analysis algorithms described in Chapter 5. For completeness, the definitions presented in Chapter 5 are repeated below along with the additional definitions required to fully implement the dataflow algorithms. Three sets of attribute definitions are presented, corresponding to the three phases of the dataflow algorithm.

The relevant production rules for SR's grammar are presented using the form

left-hand side : right-hand side

For simplicity, many of SR's production rules do not appear below; those omitted are generally not relevant to the attribute computation. For example, no rules for local declarations are present as they do not affect the values of our attributes. Table A.1 provides a brief description of the nonterminal symbols used in the production rules for our SR attributed grammar. The attribute definitions associated with each production rule are listed below the relevant rule. Table A.2 provides a brief description of each of the attributes used in the definitions that follow. Refer to Chapter 5 for more complete descriptions of these attributes and their functions.

These attribute or dataflow definitions assume the existence of several predefined functions and constants. Tables A.3 and A.4 provide brief descriptions of these functions and constants.

Nonterminal	Description
Rlist	a list of resources
R	a resource
Body	a list of code bodies within a resource
B	a body of code within a resource
D	a declaration of a global variable
Comp	a component within a resource (a proc or process)
S	a list of statements or a statement
Arm	an arm of an input statement

Table A.1: Description of nonterminal symbols in SR's attributed grammar.

Attribute	Description
lastres	the main resource in a program — generally the last resource; the only resource that is created implicitly
globalvars	the global variables declared within a resource
loopcnt	reflects the static depth of looping constructs
vmvars	the virtual machine capability variables assigned to via create statements within the body of a looping construct
pattern	a unique virtual machine pattern identifier
res	(resource, virtual machine pattern) pairs, affected primarily by resource creation statements
defs	((variable,virtual machine pattern),(value,virtual machine pattern),global/local flag) triples representing potential capability variable values
bnd	binding of input and output parameters to formal parameters for operation or resource, used to perform interprocess analysis
C_{same}	operations invoked and implemented on the same virtual machine for each occurrence of a call statement
C_{diff}	operations invoked and implemented on different virtual machines for each occurrence of a call statement
S_{same}	operations invoked and implemented on the same virtual machine for each occurrence of a send statement
S_{same}	operations invoked and implemented on different virtual machines for each occurrence of a send statement

Table A.2: Description of attributes.

Function	**Description**
curres	returns the name of the resource being analyzed
$save(node, attribute)$	*node* — current parse tree node *attribute* — attribute associated with current parse tree node saves the current value of *attribute* in the specified *node*
gensym(vmpat)	*vmpat* — a constant string returns the next in a sequence of unique virtual machine pattern identifiers using *vmpat* as prefix
removelocals(defs)	*defs* — a set of capability variable definitions returns the definitions for all global, i.e., shared variables only
pass1	returns true if phase two analysis is in its first iteration or pass over the abstract syntax tree and false otherwise
formal(x,p)	x — an operation, capability variable for an operation or resource, or qualified operation p — an actual invocation or an implementation parameter returns the possible formal equivalents of the actual parameter — used to propagate possible parameter bindings
resequiv(z,r,a)	z — a resource capability variable r — a resource a — an operation returns the operation in resource r that is structurally equivalent to the operation a declared in the resource for which z is a capability
input(p)	p — an actual parameter returns true if p is an input parameter and false otherwise
output(p)	p — an actual parameter returns true if p is an output parameter and false otherwise
op(p)	p — an actual parameter returns true if p is an operation and false otherwise
capvar(p)	p — an actual parameter returns true if p is a capability variable and false otherwise
qualop(p)	p — an actual parameter returns true if p is a qualified operation (written as z.a) and false otherwise
qres(p)	p — an actual parameter that is a qualified operation returns the resource capability variable, i.e., the qualifier
qop(p)	p — an actual parameter that is a qualified operation returns the operation
inv(p)	p — an actual parameter returns true if p is an invocation and false otherwise
invop(p)	p — an actual parameter that is an invocation returns the set operations possibly invoked

Table A.3: Descriptions of functions used to facilitate presentation of attribute definitions used in dataflow analysis.

Constant	**Description**
`initvm`	unique virtual machine pattern associated with the main resource for a program, the only implicitly created resource
?	represents a “don’t care” value

Table A.4: Descriptions of predefined constants used in attribute definitions.

A.1 Attribute Definitions for Phase One of Dataflow Analysis

$Prog \;:\; Rlist$

$save(Prog, res) = \{lastres_{out}(Rlist), \texttt{initvm}\}$

$Rlist \;:\; R$

$lastres_{out}(Rlist) = lastres_{out}(R)$

$Rlist \;:\; Rlist_1\, R$

$lastres_{out}(Rlist) = lastres_{out}(R)$

$R \;:\; \texttt{resource}\, x\, Spec\, Body$

$globalvars_{in}(Body) = \{\}$
$lastres_{out}(R) = x$
$save(R, globalvars) = globalvars_{out}(Body)$

$Body \;:\; B$

$globalvars_{in}(B) = globalvars_{in}(Body)$
$globalvars_{out}(Body) = globalvars_{out}(B)$

$Body \;:\; Body_1 \;\; B$

$globalvars_{in}(Body_1) = globalvars_{in}(Body)$
$globalvars_{in}(B) = globalvars_{out}(Body_1)$
$globalvars_{out}(Body) = globalvars_{out}(B)$

$B \;:\; D$

$globalvars_{in}(D) = globalvars_{in}(B)$
$globalvars_{out}(B) = globalvars_{out}(D)$

$B \;:\; Comp$

$loopcnt_{in}(Comp) = 0$
$vmvars_{in}(Comp) = \{\}$

$$globalvars_{out}(B) = globalvars_{in}(B)$$

D : $\texttt{var}\, x_1, \ldots, x_n \, : \texttt{cap vm}$

$$globalvars_{out}(D) = globalvars_{in}(D) \cup \{x_1, \ldots, x_n\}$$

D : $\texttt{var}\, x_1, \ldots, x_n \, : \texttt{cap}\, y$

$$globalvars_{out}(D) = globalvars_{in}(D) \cup \{x_1, \ldots, x_n\}$$

The following attribute definitions are for relevant program statements (virtual machine create statements) and flow of control constructs.

$Comp$: $\texttt{process}\, p \, \ldots \, S \, \texttt{end}\, p$

$$loopcnt_{in}(S) = loopcnt_{in}(Comp)$$
$$loopcnt_{out}(Comp) = loopcnt_{out}(S)$$
$$vmvars_{in}(S) = vmvars_{in}(Comp)$$
$$vmvars_{out}(Comp) = vmvars_{out}(S)$$

$Comp$: $\texttt{proc}\, p(p_1, \ldots, p_n) \, S \, \texttt{end}\, p$

$$loopcnt_{in}(S) = loopcnt_{in}(Comp)$$
$$loopcnt_{out}(Comp) = loopcnt_{out}(S)$$
$$vmvars_{in}(S) = vmvars_{in}(Comp)$$
$$vmvars_{out}(Comp) = vmvars_{out}(S)$$

$Comp$: $\texttt{initial}\, S \, \texttt{end initial}$

$$loopcnt_{in}(S) = loopcnt_{in}(Comp)$$
$$loopcnt_{out}(Comp) = loopcnt_{out}(S)$$
$$vmvars_{in}(S) = vmvars_{in}(Comp)$$
$$vmvars_{out}(Comp) = vmvars_{out}(S)$$

$Comp$: $\texttt{final}\, S \, \texttt{end final}$

$$loopcnt_{in}(S) = loopcnt_{in}(Comp)$$
$$loopcnt_{out}(Comp) = loopcnt_{out}(S)$$
$$vmvars_{in}(S) = vmvars_{in}(Comp)$$
$$vmvars_{out}(Comp) = vmvars_{out}(S)$$

$S \quad : \quad S_1 \; S_2$

$$loopcnt_{in}(S_1) = loopcnt_{in}(S)$$
$$loopcnt_{in}(S_2) = loopcnt_{out}(S_1)$$
$$loopcnt_{out}(S) = loopcnt_{out}(S_2)$$
$$vmvars_{in}(S_1) = vmvars_{in}(S)$$
$$vmvars_{in}(S_2) = vmvars_{out}(S_1)$$
$$vmvars_{out}(S) = vmvars_{out}(S_2)$$

$S \quad : \quad x := \texttt{create}\,\texttt{vm}()$

$$pattern(S) = gensym(vmpat)$$
$$save(S, pattern) = pattern(S)$$
$$vmvars_{out}(S) = if\,loopcnt_{in}(S) > 0\,then\,vmvars_{in}(S) \cup \{x\}\,else\,\{\}$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$

$S \quad : \quad \texttt{fa} \ldots \texttt{->} \; S_1\,\texttt{af}$

$$loopcnt_{in}(S_1) = loopcnt_{in}(S) + 1$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$
$$vmvars_{in}(S_1) = \{\}$$
$$vmvars_{out}(S) = vmvars_{in}(S) \cup vmvars_{out}(S_1)$$
$$save(S, vmvars) = vmvars_{out}(S_1)$$

$S \quad : \quad \texttt{do} \ldots \texttt{->} \; S_1 \,[]\, \ldots \,[]\, \ldots \; \texttt{->} \; S_n\,\texttt{od}$

$$loopcnt_{in}(S_i) = loopcnt_{in}(S) + 1,\ \forall i = 1\ to\ n$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$
$$vmvars_{in}(S_i) = \{\},\ \forall i = 1\ to\ n$$
$$vmvars_{out}(S) = vmvars_{in}(S) \cup\ (\cup_{i=1\ to\ n} vmvars_{out}(S_i))$$
$$save(S, vmvars) = \cup_{i=1\ to\ n} vmvars_{out}(S_i)$$

$S \quad : \quad \texttt{do} \ldots \texttt{->} \; S_1 \,[]\, \ldots \,[]\, \texttt{else} \, \texttt{->} \; S_n\,\texttt{od}$

$$loopcnt_{in}(S_i) = loopcnt_{in}(S) + 1,\ \forall i = 1\ to\ n$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$
$$vmvars_{in}(S_i) = \{\},\ \forall i = 1\ to\ n$$
$$vmvars_{out}(S) = vmvars_{in}(S) \cup\ (\cup_{i=1\ to\ n} vmvars_{out}(S_i))$$
$$save(S, vmvars) = \cup_{i=1\ to\ n} vmvars_{out}(S_i)$$

$S \quad : \quad \texttt{if} \ldots \texttt{->} \; S_1 \,[]\, \ldots \,[]\, \ldots \; \texttt{->} \; S_n\,\texttt{fi}$

$$loopcnt_{in}(S_i) = loopcnt_{in}(S),\ \forall i = 1\, to\, n$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$
$$vmvars_{in}(S_i) = vmvars_{in}(S),\ \forall i = 1\, to\, n$$
$$vmvars_{out}(S) = \cup_{i=1\ to\ n} vmvars_{out}(S_i)$$

S : `if ... -> ` S_1 `[] ... [] else -> ` S_n `fi`

$$loopcnt_{in}(S_i) = loopcnt_{in}(S),\ \forall i = 1\, to\, n$$
$$loopcnt_{out}(S) = loopcnt_{in}(S)$$
$$vmvars_{in}(S_i) = vmvars_{in}(S),\ \forall i = 1\, to\, n$$
$$vmvars_{out}(S) = \cup_{i=1\ to\ n} vmvars_{out}(S_i)$$

A.2 Attribute Definitions for Phase Two of Dataflow Analysis

$Prog$: $Rlist$

$$res_{in}(Rlist) = save(Prog, res)$$
$$res_{out}(Prog) = res_{out}(Rlist)$$
$$bnd_{in}(Rlist) = save(Prog, bnd)$$
$$bnd_{out}(Prog) = bnd_{out}(Rlist)$$
$$save(Prog, res) = res_{out}(Rlist)$$
$$save(Prog, bnd) = bnd_{out}(Rlist)$$

The following are attribute definitions for program components — resources.

$Rlist$: R

$$res_{in}(R) = res_{in}(Rlist)$$
$$res_{out}(Rlist) = res_{out}(R)$$
$$bnd_{in}(R) = bnd_{in}(Rlist)$$
$$bnd_{out}(Rlist) = bnd_{out}(R)$$

$Rlist$: $Rlist_1\ R$

$$res_{in}(Rlist_1) = res_{in}(Rlist)$$
$$res_{in}(R) = res_{out}(Rlist_1)$$
$$res_{out}(Rlist) = res_{out}(R)$$
$$bnd_{in}(Rlist_1) = bnd_{in}(Rlist)$$
$$bnd_{in}(R) = bnd_{out}(Rlist_1)$$
$$bnd_{out}(Rlist) = bnd_{out}(R)$$

R : `resource` $x\ Spec\ Body$

$$defs_{in}(Body) = save(R, defs)$$
$$save(R, defs) = defs_{out}(Body)$$
$$res_{in}(Body) = res_{in}(R)$$
$$res_{out}(R) = res_{out}(Body)$$
$$bnd_{in}(Body) = bnd_{in}(R)$$
$$bnd_{out}(R) = bnd_{out}(Body)$$

$$globalvars_{in}(Body) = save(R, globalvars)$$

The following are attribute definitions for components within resources — procs and processes.

$Body \;:\; B$

$$\begin{aligned}
&defs_{in}(B) = defs_{in}(Body)\\
&defs_{out}(Body) = defs_{out}(B)\\
&res_{in}(B) = res_{in}(Body)\\
&res_{out}(Body) = res_{out}(B)\\
&bnd_{in}(B) = bnd_{in}(Body)\\
&bnd_{out}(Body) = bnd_{out}(B)\\
&globalvars_{in}(B) = globalvars_{in}(Body)
\end{aligned}$$

$Body \;:\; Body_1 \;\; B$

$$\begin{aligned}
&defs_{in}(Body_1) = defs_{in}(Body)\\
&defs_{in}(B) = defs_{out}(Body_1)\\
&defs_{out}(Body) = defs_{out}(B)\\
&res_{in}(Body_1) = res_{in}(Body)\\
&res_{in}(B) = res_{out}(Body_1)\\
&res_{out}(Body) = res_{out}(B)\\
&bnd_{in}(Body_1) = bnd_{in}(Body)\\
&bnd_{in}(B) = bnd_{out}(Body_1)\\
&bnd_{out}(Body) = bnd_{out}(B)\\
&globalvars_{in}(Body_1) = globalvars_{in}(Body)\\
&globalvars_{in}(B) = globalvars_{in}(Body)
\end{aligned}$$

$B \;:\; Comp$

$$\begin{aligned}
&defs_{in}(Comp) = defs_{in}(B)\\
&defs_{out}(B) = defs_{out}(Comp) - removelocals(Defs_{out}(Comp))\\
&res_{in}(Comp) = res_{in}(B)\\
&res_{out}(B) = res_{out}(Comp)\\
&bnd_{in}(Comp) = bnd_{in}(B)\\
&bnd_{out}(B) = bnd_{out}(Comp)\\
&globalvars_{in}(Comp) = globalvars_{in}(B)
\end{aligned}$$

$Comp \quad : \quad \texttt{process}\, p\ \ldots\ S\ \texttt{end}\, p$

$$
\begin{aligned}
& defs_{in}(S) = defs_{in}(Comp) \cup save(S, defs)\\
& defs_{out}(Comp) = defs_{out}(S)\\
& save(S, defs) = defs_{in}(S)\\
& res_{in}(S) = res_{in}(Comp)\\
& res_{out}(Comp) = res_{out}(S)\\
& bnd_{in}(S) = bnd_{in}(Comp)\\
& bnd_{out}(Comp) = bnd_{out}(S)\\
& globalvars_{in}(S) = globalvars_{in}(Comp)
\end{aligned}
$$

$Comp \quad : \quad \texttt{proc}\, p(p_1,\ \ldots, p_n)\ S\ \texttt{end}\ p$

$$
\begin{aligned}
& defs_{in}(S) = defs_{in}(Comp) \cup save(S, defs) \cup\\
& \quad \{((p_i, u), (a, v), l) |\ input(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Comp) \wedge\\
& \qquad ((formal(p, p_i), u), (a, v), in) \in bnd_{in}(S)\}\\
& defs_{out}(Comp) = defs_{out}(S)\\
& save(S, defs) = defs_{in}(S)\\
& res_{in}(S) = res_{in}(Comp)\\
& res_{out}(Comp) = res_{out}(S)\\
& bnd_{in}(S) = bnd_{in}(Comp)\\
& bnd_{out}(Comp) = bnd_{out}(S) \cup\\
& \quad \{((formal(p, p_i), u), (a, v), out) |\ output(p_i) \wedge r = curres \wedge (\ r, u) \in res_{in}(Comp) \wedge\\
& \qquad ((p_i, u), (a, v), ?) \in defs_{out}(S)\}\\
& globalvars_{in}(S) = globalvars_{in}(Comp)
\end{aligned}
$$

$Comp \quad : \quad \texttt{proc}\, p(p_1,\ \ldots, p_n)\ \texttt{returns}\ z\ S\ \texttt{end}\ p$

$$
\begin{aligned}
& defs_{in}(S) = defs_{in}(Comp) \cup save(S, defs) \cup\\
& \quad \{((p_i, u), (a, v), l) |\ input(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Comp) \wedge\\
& \qquad ((formal(p, p_i), u), (a, v), in) \in bnd_{in}(S)\}\\
& defs_{out}(Comp) = defs_{out}(S)\\
& save(S, defs) = defs_{in}(S)\\
& res_{in}(S) = res_{in}(Comp)\\
& res_{out}(Comp) = res_{out}(S)\\
& bnd_{in}(S) = bnd_{in}(Comp)\\
& bnd_{out}(Comp) = bnd_{out}(S) \cup\\
& \quad \{((formal(p, p_i), u), (a, v), out) |\ output(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Comp) \wedge\\
& \qquad ((p_i, u), (a, v), ?) \in defs_{out}(S)\} \cup\\
& \quad \{((p, u), (b, v), out) |\ r = curres \wedge (r, u) \in res_{in}(Comp) \wedge ((z, u), (b, v), ?) \in defs_{out}(S)\}\\
& globalvars_{in}(S) = globalvars_{in}(Comp)
\end{aligned}
$$

$Comp$: `initial` S `end initial`

$$defs_{in}(S) = defs_{in}(Comp) \cup save(S, defs)$$
$$defs_{out}(Comp) = defs_{out}(S)$$
$$save(S, defs) = defs_{in}(S)$$
$$res_{in}(S) = res_{in}(Comp)$$
$$res_{out}(Comp) = res_{out}(S)$$
$$bnd_{in}(S) = bnd_{in}(Comp)$$
$$bnd_{out}(Comp) = bnd_{out}(S)$$
$$globalvars_{in}(S) = globalvars_{in}(Comp)$$

$Comp$: `final` S `end final`

$$defs_{in}(S) = defs_{in}(Comp) \cup save(S, defs)$$
$$defs_{out}(Comp) = defs_{out}(S)$$
$$save(S, defs) = defs_{in}(S)$$
$$res_{in}(S) = res_{in}(Comp)$$
$$res_{out}(Comp) = res_{out}(S)$$
$$bnd_{in}(S) = bnd_{in}(Comp)$$
$$bnd_{out}(Comp) = bnd_{out}(S)$$
$$globalvars_{in}(S) = globalvars_{in}(Comp)$$

The following are attribute definitions for components within procs/processes — statements.

S : $S_1 \ S_2$

$$defs_{in}(S_1) = defs_{in}(S)$$
$$defs_{in}(S_2) = defs_{out}(S_1) \cup save(S, defs)$$
$$defs_{out}(S) = defs_{out}(S_2)$$
$$res_{in}(S_1) = res_{in}(S)$$
$$res_{in}(S_2) = res_{out}(S_1)$$
$$res_{out}(S) = res_{out_2}(S)$$
$$bnd_{in}(S_1) = bnd_{in}(S)$$
$$bnd_{in}(S_2) = bnd_{out}(S_1)$$
$$bnd_{out}(S) = bnd_{out}(S_2)$$
$$globalvars_{in}(S_1) = globalvars_{in}(S)$$
$$globalvars_{in}(S_2) = globalvars_{in}(S)$$

S : `fa` $\ldots$ `->` S_1 `af`

$if\ pass1\ then$
$defs_{in}(S_1) = defs_{in}(S) \cup$
$\{((x,u),(unk,nil),k) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge x \in save(S, vmvars)\}$
$else$
$defs_{in}(S_1) = defs_{in}(S) \cup save(S_1, defs)$
$defs_{out}(S) = defs_{out}(S_1) \cup defs_{in}(S)$
$save(S, defs) = defs_{out}(S)_{in}$
$res_{in}(S_1) = res_{in}(S)$
$res_{out}(S) = res_{out}(S_1)$
$bnd_{in}(S_1) = bnd_{in}(S)$
$bnd_{out}(S) = bnd_{out}(S_1)$
$globalvars_{in}(S_1) = globalvars_{in}(S)$

S : `do` $\ldots$ `->` S_1 [] $\ldots$ [] $\ldots$ `->` S_n `od`

$defs_{in}(S_i) = defs_{in}(S) \cup save(S, defs),\ \forall i = 1\ to\ n$
$defs_{out}(S) = defs_{in}(S) \cup (\cup_{i=1\ to\ n} defs_{out}(S_i))$
$save(S, defs) = defs_{out}(S)$
$res_{in}(S_i) = res_{in}(S),\ \forall i = 1\ to\ n$
$res_{out}(S) = res_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} res_{out}(S_i))$
$bnd_{in}(S_i) = bnd_{in}(S),\ \forall i = 1\ to\ n$
$bnd_{out}(S) = bnd_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} bnd_{out}(S_i))$
$globalvars_{in}(S_i) = globalvars_{in}(S),\ \forall i = 1\ to\ n$

S : `do` $\ldots$ `->` S_1 [] $\ldots$ [] `else` `->` S_n `od`

$defs_{in}(S_i) = defs_{in}(S) \cup save(S, defs),\ \forall i = 1\ to\ n$
$defs_{out}(S) = \cup_{i=1\ to\ n} defs_{out}(S_i)$
$save(S, defs) = defs_{out}(S)$
$res_{in}(S_i) = res_{in}(S),\ \forall i = 1\ to\ n$
$res_{out}(S) = res_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} res_{out}(S_i))$
$bnd_{in}(S_i) = bnd_{in}(S),\ \forall i = 1\ to\ n$
$bnd_{out}(S) = bnd_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} bnd_{out}(S_i))$
$globalvars_{in}(S_i) = globalvars_{in}(S),\ \forall i = 1\ to\ n$

S : `if` $\ldots$ `->` S_1 [] $\ldots$ [] $\ldots$ `->` S_n `fi`

$defs_{in}(S_i) = defs_{in}(S),\ \forall i = 1\ to\ n$
$defs_{out}(S) = defs_{in}(S) \cup (\cup_{i=1\ to\ n} defs_{out}(S_i))$
$res_{in}(S_i) = res_{in}(S),\ \forall i = 1\ to\ n$
$res_{out}(S) = res_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} res_{out}(S_i))$

$bnd_{in}(S_i) = bnd_{in}(S),\ \forall i = 1\ to\ n$
$bnd_{out}(S) = bnd_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} bnd_{out}(S_i))$
$globalvars_{in}(S_i) = globalvars_{in}(S),\ \forall i = 1\ to\ n$

S : `if ... ->` S_1 `[] ... [] else ->` S_n `fi`

$defs_{in}(S_i) = defs_{in}(S),\ \forall i = 1\ to\ n$
$defs_{out}(S) = \cup_{i=1\ to\ n} defs_{out}(S_i)$
$res_{in}(S_i) = res_{in}(S),\ \forall i = 1\ to\ n$
$res_{out}(S) = res_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} res_{out}(S_i))$
$bnd_{in}(S_i) = bnd_{in}(S),\ \forall i = 1\ to\ n$
$bnd_{out}(S) = bnd_{in}(S)\ \cup\ (\cup_{i=1\ to\ n} bnd_{out}(S_i))$
$globalvars_{in}(S_i) = globalvars_{in}(S),\ \forall i = 1\ to\ n$

S : $x :=$ `create vm()`

$if\ x \notin globalvars_{in}(S)\ then$
$\quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup$
$\quad\quad \{((x,u),(pattern,nil),l)|\ r = curres \wedge (r,u) \in res_{in}(S)\}$
$else$
$\quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(unk,nil),g)|(y,u) \in res_{in}(S)\}$
$res_{out}(S) = res_{in}(S)$
$bnd_{out}(S) = bnd_{in}(S)$

S : $x :=$ `create` $y(p_1,\ldots,p_n)$ `on` z

$if\ x \notin globalvars_{in}(S)\ then$
$\quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup$
$\quad\quad \{((x,u),(y,v),l)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(v,nil),?) \in defs_{in}(S)\}$
$else$
$\quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(y,unk),g)|(y,u) \in res_{in}(S)\}$
$res_{out}(S) = res_{in}(S) \cup$
$\quad \{(y,v)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(v,nil),?) \in defs_{in}\}$
$bnd_{out}(S) = bnd_{in}(S) \cup$
$\quad \{((formal(y,p_i),u),(p_i,u),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge$
$\quad\quad input(p_i) \wedge op(p_i)\} \cup$
$\quad \{((formal(a,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge$
$\quad\quad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\}$
$\quad \{((formal(y,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge$
$\quad\quad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i,r,qop(p_i)) \wedge$
$\quad\quad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\}$
$\quad \{((formal(y,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge$

$$inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}$$

The following attribute definitions are for assignments to capability variables for operations, resources, and virtual machines. These assignment statements have several possible forms: direct assignment of an operation to a capability variable, assignment of a capability variable to another capability variable, assignment of a operation qualified by a resource capability to a capability variable, and invocation of an operation returning a capability as a result. In the latter case — invocation of an operation returning a capability as a result — the invocation has three possible forms: directly through an operation, indirectly through a capability variable, and finally, through an operation qualified by a capability for a resource. We present the attribute definitions for assignment below. The following conventions are used for the grammar productions: x is a capability variable, a is an operation, y is a capability variable, z is a capability for a resource, and the p_i are capability parameters.

$$
\begin{aligned}
S \;:\;\; & x := a \\
& if\ x \notin globalvars_{in}(S)\ then \\
& \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x)) \ \cup \\
& \qquad \{((x,u),(a,u),l) \mid r = curres \wedge (r,u) \in res_{in}(S)\} \\
& else \\
& \quad defs_{out}(S) = defs_{in}(S) \cup \{((x,u),(a,u),g) \mid r = curres \wedge (r,u) \in res_{in}(S)\} \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S)
\end{aligned}
$$

$$
\begin{aligned}
S \;:\;\; & x := y \\
& if\ x \notin globalvars_{in}(S)\ then \\
& \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x)) \ \cup \\
& \qquad \{((x,u),(a,v),l) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge ((y,u),(a,v),?) \in defs_{in}(S)\} \\
& else \\
& \quad defs_{out}(S) = defs_{in}(S) \cup \\
& \qquad \{((x,u),(a,v),g) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge ((y,u),(a,v),?) \in defs_{in}(S)\} \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S)
\end{aligned}
$$

$$
\begin{array}{lll}
S & : & x := z.a \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - kill(defs_{in}(S), x))\ \cup \\
& & \qquad \{((x,u),(b,v),l)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(r_1,v),?) \in defs_{in}(S) \wedge \\
& & \qquad\quad b = resequiv(z, r_1, a)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup \\
& & \qquad \{((x,u),(b,v),g)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((z,u),(r_1,v),?) \in defs_{in}(S) \wedge \\
& & \qquad\quad b = resequiv(z, r_1, a)\} \\
& & res_{out}(S) = res_{in}(S) \\
& & bnd_{out}(S) = bnd_{in}(S)
\end{array}
$$

$$
\begin{array}{lll}
S & : & x := a(p_1,\ ...,\ p_n) \\
& & if\ x \notin globalvars_{in}(S)\ then \\
& & \quad defs_{out}(S) = (defs_{in}(S) - \\
& & \qquad kill(defs_{in}(S), x) - \\
& & \qquad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i)|\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
& & \qquad \{((x,u),(b,v),l)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((a,u),(b,v),out) \in bnd_{in}(S)\} \cup \\
& & \qquad \{((p_i,u),(b,v),l)|\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
& & \qquad \{((p_i,u),(b,v),g)|\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
& & else \\
& & \quad defs_{out}(S) = defs_{in}(S) \cup \\
& & \qquad \{((x,u),(b,v),g)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge ((a,u),(b,v),?) \in bnd_{in}(S)\} \cup \\
& & \qquad \{((p_i,u),(b,v),l)|\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup \\
& & \qquad \{((p_i,u),(b,v),g)|\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad\quad ((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \\
& & res_{out}(S) = res_{in}(S) \\
& & bnd_{out}(S) = bnd_{in}(S) \cup \\
& & \quad \{((formal(a,p_i),u),(p_i,u),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge op(p_i)\} \cup \\
& & \quad \{((formal(a,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
& & \quad \{((formal(a,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge \\
& & \qquad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
& & \quad \{((formal(a,p_i),u),(b,v),in)|\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& & \qquad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{array}
$$

$S \;:\; x := y(p_1,\ ...,\ p_n)$

$if\ x \notin globalvars_{in}(S)\ then$

$defs_{out}(S) = (defs_{in}(S) -$

$kill(defs_{in}(S), x) -$

$(\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i)|\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup$

$\{((x, u), (b, w), l)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((y, u), (c, v), ?) \in defs_{in}(S) \wedge ((c, v), (b, w), out) \in bnd_{in}(S)\} \cup$

$\{((p_i, u), (b, v), l)|\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\} \cup$

$\{((p_i, u), (b, v), g)|\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\}$

$else$

$defs_{out}(S) = defs_{in}(S) \cup \{((x, w), (b, v), g)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((y, u), (c, v), ?) \in defs_{in}(S) \wedge ((c, v), (b, w), ?) \in bnd_{in}(S)\}$

$\{((p_i, u), (b, v), l)|\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\} \cup$

$\{((p_i, u), (b, v), g)|\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\}$

$res_{out}(S) = res_{in}(S) \cup$

$bnd_{out}(S) = bnd_{in}(S) \cup$

$\{((formal(y, p_i), u), (p_i, u), in)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$input(p_i) \wedge op(p_i)\} \cup$

$\{((formal(a, p_i), u), (b, v), in)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$input(p_i) \wedge capvar(p_i) \wedge ((p_i, u), (b, v), ?) \in defs_{in}(S)\}$

$\{((formal(y, p_i), u), (b, v), in)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i, r, qop(p_i)) \wedge$

$((qres(p_i), u), (b, v), ?) \in defs_{in}(S)\}$

$\{((formal(y, p_i), u), (b, v), in)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$inv(p_i) \wedge c \in invop(p_i) \wedge ((c, u), (b, v), out) \in bnd_{in}(S)\}$

$S \;:\; x := z.a(p_1,\ ...,\ p_n)$

$if\ x \notin globalvars_{in}(S)\ then$

$defs_{out}(S) = (defs_{in}(S) -$

$kill(defs_{in}(S), x) -$

$(\cup_{i=1\ to\ n}\{kill(defs_{in}(S), p_i)|\ capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup$

$\{((x, u), (c, w), l)|\ r = curres \wedge (r, u) \in res_{in}(S) \wedge ((z, u), (r_1, v), ?) \in defs_{in}(S) \wedge$

$b = resequiv(z, r_1, a) \wedge ((b, v), (c, w), out) \in bnd_{in}(S)\} \cup$

$\{((p_i, u), (b, v), l)|\ capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\} \cup$

$\{((p_i, u), (b, v), g)|\ capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r, u) \in res_{in}(S) \wedge$

$((formal(a, p_i), u), (b, v), out) \in bnd_{out}(S)\}$

$else$

$$defs_{out}(S) = defs_{in}(S) \cup \{((x,w),(b,v),g) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$((y,u),(r_1,v),?) \in defs_{in}(S) \wedge b = resequiv(z,r_1,a) \wedge ((b,v),(c,w),?) \in bnd_{in}(S)\} \cup$$
$$\{((p_i,u),(b,v),l) \mid capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup$$
$$\{((p_i,u),(b,v),g) \mid capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\}$$

$$res_{out}(S) = res_{in}(S)$$

$$bnd_{out}(S) = bnd_{in}(S) \cup$$
$$\{((formal(z.a,p_i),u),(p_i,u),in) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$input(p_i) \wedge op(p_i)\} \cup$$
$$\{((formal(z.a,p_i),u),(b,v),in) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\}$$
$$\{((formal(z.a,p_i),u),(b,v),in) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i,r,qop(p_i)) \wedge$$
$$((qres(p_i),u),(b,v),?) \in defs_{in}(S)\}$$
$$\{((formal(z.a,p_i),u),(b,v),in) \mid r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}$$

The following are attribute definitions for call/send invocation statements. The definitions shown apply to direct invocations of the operation a. Invocation statements may also invoke operations through capability variables, or through qualified operations. The attribute definitions for these cases are simply variations of those shown below. Parameters p_i can be operations, capability variables, qualified operations, or operation invocations returning a capability as a result. Additionally, parameter invocations can invoke operations directly, through a capability variable, or through a qualified operation. Each of these cases is reflected in the definition of the attribute bnd.

S : `call` $a(p_1,\ \ldots,\ p_n)$

$$defs_{out}(S) = (defs_{in}(S) -$$
$$(\cup_{i=1\ to\ n}\{kill(defs_{in}(S),p_i) \mid capvar(p_i) \wedge output(p_i) \wedge p_i \notin globalvars(S)\})) \cup$$
$$\{((p_i,u),(b,v),l) \mid capvar(p_i) \wedge p_i \notin globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\} \cup$$
$$\{((p_i,u),(b,v),g) \mid capvar(p_i) \wedge p_i \in globalvars(S) \wedge r = curres \wedge (r,u) \in res_{in}(S) \wedge$$
$$((formal(a,p_i),u),(b,v),out) \in bnd_{out}(S)\}$$

$$res_{out}(S) = res_{in}(S)$$

$$
\begin{aligned}
& bnd_{out}(S) = bnd_{in}(S) \cup \\
& \quad \{((formal(a,p_i),u),(p_i,u),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge op(p_i)\} \cup \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i,r,qop(p_i)) \wedge \\
& \qquad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{aligned}
$$

S : $\texttt{send}\ a(p_1,\ ...,\ p_n)$

$$
\begin{aligned}
& defs_{out}(S) = defs_{in}(S) \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S) \cup \\
& \quad \{((formal(a,p_i),u),(p_i,u),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge op(p_i)\} \cup \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge capvar(p_i) \wedge ((p_i,u),(b,v),?) \in defs_{in}(S)\} \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad input(p_i) \wedge qualop(p_i) \wedge b = resequiv(qres(p_i,r,qop(p_i)) \wedge \\
& \qquad ((qres(p_i),u),(b,v),?) \in defs_{in}(S)\} \\
& \quad \{((formal(a,p_i),u),(b,v),in) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad inv(p_i) \wedge c \in invop(p_i) \wedge ((c,u),(b,v),out) \in bnd_{in}(S)\}
\end{aligned}
$$

The following are attribute definitions for receive and input implementation statements. *bnd.*

S : $\texttt{receive}\ a(p_1,\ ...,\ p_n)$

$$
\begin{aligned}
& defs_{out}(S) = (defs_{in}(S) - \\
& \quad (\cup_{i=1\ to\ n}\{kill(defs_{in}(S),p_i) |\ capvar(p_i) \wedge input(p_i) \wedge p_i \notin globalvars(S)\})) \cup \\
& \quad \{((p_i,u),(b,v),l) |\ r = curres \wedge (r,u) \in res_{in}(S) \wedge \\
& \qquad ((formal(z.a,p_i),u),(b,v),in) \in bnd_{in}(S)\} \\
& res_{out}(S) = res_{in}(S) \\
& bnd_{out}(S) = bnd_{in}(S)
\end{aligned}
$$

S : $\texttt{in}\ Arm_1\ []\ \ldots\ Arm_m\ \texttt{ni}$

$$
\begin{aligned}
& defs_{in}(Arm_i) = defs_{in}(S)\ \forall i = 1\ to\ m \\
& defs_{out}(S) = \cup_{i=1\ to\ m}\ defs_{out}(Arm_i)
\end{aligned}
$$

$res_{in}(Arm_i) = res_{in}(S)\ \forall i = 1\ to\ m$
$res_{out}(S) = \cup_{i=1\ to\ m}\ res_{out}(Arm_i)$
$bnd_{in}(Arm_i) = bnd_{in}(S)\ \forall i = 1\ to\ m$
$bnd_{out}(S) = \cup_{i=1\ to\ m}\ bnd_{out}(Arm_i)$

S : `in` Arm_1 [] ... Arm_m [] $Elsearm$ `ni`

$defs_{in}(Arm_i) = defs_{in}(S)\ \forall i = 1\ to\ m$
$defs_{in}(Elsearm) = defs_{in}(S)$
$defs_{out}(S) = \cup_{i=1\ to\ m}\ defs_{out}(Arm_i)$
$res_{in}(Arm_i) = res_{in}(S)\ \forall i = 1\ to\ m$
$res_{out}(S) = \cup_{i=1\ to\ m}\ res_{out}(Arm_i)$
$bnd_{in}(Arm_i) = bnd_{in}(S)\ \forall i = 1\ to\ m$
$bnd_{out}(S) = \cup_{i=1\ to\ m}\ bnd_{out}(Arm_i)$

Arm : $a(p_1,\ ...,\ p_n)\ ...\ -> S$

$defs_{in}(S) = defs_{in}(Arm) \cup save(S, defs) \cup$
$\{((p_i, u), (a, v), l)|\ input(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Comp) \wedge$
$((formal(a, p_i), u), (a, v), in) \in bnd_{in}(S)\}$
$defs_{out}(Arm) = defs_{out}(S)$
$save(S, defs) = defs_{in}(S)$
$res_{in}(S) = res_{in}(Arm)$
$res_{out}(Arm) = res_{out}(S)$
$bnd_{in}(S) = bnd_{in}(Arm)$
$bnd_{out}(Arm) = bnd_{out}(S) \cup$
$\{((formal(a, p_i), u), (a, v), out)|\ output(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Arm) \wedge$
$((p_i, u), (a, v), ?) \in defs_{out}(S)\}$
$globalvars_{in}(S) = globalvars_{in}(Arm)$

Arm : $a(p_1,\ ...,\ p_n)$ `returns` $z\ ...\ -> S$

$defs_{in}(S) = defs_{in}(Arm) \cup save(S, defs) \cup$
$\{((p_i, u), (a, v), l)|\ input(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Comp) \wedge$
$((formal(a, p_i), u), (a, v), in) \in bnd_{in}(S)\}$
$defs_{out}(Arm) = defs_{out}(S)$
$save(S, defs) = defs_{in}(S)$
$res_{in}(S) = res_{in}(Arm)$
$res_{out}(Arm) = res_{out}(S)$
$bnd_{in}(S) = bnd_{in}(Arm)$
$bnd_{out}(Arm) = bnd_{out}(S) \cup$
$\{((formal(a, p_i), u), (b, v), out)|\ output(p_i) \wedge r = curres \wedge (r, u) \in res_{in}(Arm) \wedge$
$((p_i, u), (b, v), ?) \in defs_{out}(S)\} \cup$

$$\{((a,u),(b,v),out)|\ r = curres \wedge (r,u) \in res_{in}(Arm) \wedge$$
$$((z,u),(b,v),?) \in defs_{out}(S)\}$$
$$globalvars_{in}(S) = globalvars_{in}(Arm)$$

$Elsearm$: `else ->` S

$$defs_{in}(S) = defs_{in}(Arm) \cup save(S, defs)\}$$
$$defs_{out}(Arm) = defs_{out}(S)$$
$$save(S, defs) = defs_{in}(S)$$
$$res_{in}(S) = res_{in}(Arm)$$
$$res_{out}(Arm) = res_{out}(S)$$
$$bnd_{in}(S) = bnd_{in}(Arm)$$
$$bnd_{out}(Arm) = bnd_{out}(S)$$
$$globalvars_{in}(S) = globalvars_{in}(Arm)$$

Although not shown return and reply statements are handled using several additional attributes and functions providing access to the symbol table. More specifically, when a return or reply statement is encountered the operation performing the return or reply is identified, a list of its output and return parameters obtained, and the current values of these parameters added to the attribute *bnd*.

A.3 Attribute Definitions for Phase Three of Dataflow Analysis

$Prog \quad : \quad Rlist$

$$
\begin{aligned}
&C_{same_{in}}(Rlist) = \{\} \\
&C_{same_{out}}(Prog) = C_{same_{out}}(Rlist) \\
&C_{diff_{in}}(Rlist) = \{\} \\
&C_{diff_{out}}(Prog) = C_{diff_{out}}(Rlist) \\
&S_{same_{in}}(Rlist) = \{\} \\
&S_{same_{out}}(Prog) = S_{same_{out}}(Rlist) \\
&S_{diff_{in}}(Rlist) = \{\} \\
&S_{diff_{out}}(Prog) = S_{diff_{out}}(Rlist)
\end{aligned}
$$

The following are attribute definitions for program components — resources.

$Rlist \quad : \quad R$

$$
\begin{aligned}
&C_{same_{in}}(R) = C_{same_{in}}(Rlist) \\
&C_{same_{out}}(Rlist) = C_{same_{out}}(R) \\
&C_{diff_{in}}(R) = C_{diff_{in}}(Rlist) \\
&C_{diff_{out}}(Rlist) = C_{diff_{out}}(R) \\
&S_{same_{in}}(R) = S_{same_{in}}(Rlist) \\
&S_{same_{out}}(Rlist) = S_{same_{out}}(R) \\
&S_{diff_{in}}(R) = S_{diff_{in}}(Rlist) \\
&S_{diff_{out}}(Rlist) = S_{diff_{out}}(R)
\end{aligned}
$$

$Rlist \quad : \quad Rlist_1 \; R$

$$
\begin{aligned}
&C_{same_{in}}(Rlist_1) = C_{same_{in}}(Rlist) \\
&C_{same_{in}}(R) = C_{same_{out}}(Rlist_1) \\
&C_{same_{out}}(Rlist) = C_{same_{out}}(R) \\
&C_{diff_{in}}(Rlist_1) = C_{diff_{in}}(Rlist) \\
&C_{diff_{in}}(R) = C_{diff_{out}}(Rlist_1) \\
&C_{diff_{out}}(Rlist) = C_{diff_{out}}(R) \\
&S_{same_{in}}(Rlist_1) = S_{same_{in}}(Rlist) \\
&S_{same_{in}}(R) = S_{same_{out}}(Rlist_1) \\
&S_{same_{out}}(Rlist) = S_{same_{out}}(R)
\end{aligned}
$$

$S_{diff_{in}}(Rlist_1) = S_{diff_{in}}(Rlist)$
$S_{diff_{in}}(R) = S_{diff_{out}}(Rlist_1)$
$S_{diff_{out}}(Rlist) = S_{diff_{out}}(R)$

R : `resource` x *Spec Body*

$C_{same_{in}}(Body) = C_{same_{in}}(R)$
$C_{same_{out}}(R) = C_{same_{out}}(Body)$
$C_{diff_{in}}(Body) = C_{diff_{in}}(R)$
$C_{diff_{out}}(R) = C_{diff_{out}}(Body)$
$S_{same_{in}}(Body) = S_{same_{in}}(R)$
$S_{same_{out}}(R) = S_{same_{out}}(Body)$
$S_{diff_{in}}(Body) = S_{diff_{in}}(R)$
$S_{diff_{out}}(R) = S_{diff_{out}}(Body)$

The following are attribute definitions for components within resources — procs and processes.

$Body$: B

$C_{same_{in}}(B) = C_{same_{in}}(Body)$
$C_{same_{out}}(Body) = C_{same_{out}}(B)$
$C_{diff_{in}}(B) = C_{diff_{in}}(Body)$
$C_{diff_{out}}(Body) = C_{diff_{out}}(B)$
$S_{same_{in}}(B) = S_{same_{in}}(Body)$
$S_{same_{out}}(Body) = S_{same_{out}}(B)$
$S_{diff_{in}}(B) = S_{diff_{in}}(Body)$
$S_{diff_{out}}(Body) = S_{diff_{out}}(B)$

$Body$: $Body_1 \;\; B$

$C_{same_{in}}(Body_1) = C_{same_{in}}(Body)$
$C_{same_{in}}(B) = C_{same_{out}}(Body_1)$
$C_{same_{out}}(Body) = C_{same_{out}}(B)$
$C_{diff_{in}}(Body_1) = C_{diff_{in}}(Body)$
$C_{diff_{in}}(B) = C_{diff_{out}}(Body_1)$
$C_{diff_{out}}(Body) = C_{diff_{out}}(B)$
$S_{same_{in}}(Body_1) = S_{same_{in}}(Body)$
$S_{same_{in}}(B) = S_{same_{out}}(Body_1)$

$$S_{same_{out}}(Body) = S_{same_{out}}(B)$$
$$S_{diff_{in}}(Body_1) = S_{diff_{in}}(Body)$$
$$S_{diff_{in}}(B) = S_{diff_{out}}(Body_1)$$
$$S_{diff_{out}}(Body) = S_{diff_{out}}(B)$$

B : $Comp$

$$C_{same_{in}}(Comp) = C_{same_{in}}(B)$$
$$C_{same_{out}}(B) = C_{same_{out}}(Comp)$$
$$C_{diff_{in}}(Comp) = C_{diff_{in}}(B)$$
$$C_{diff_{out}}(B) = C_{diff_{out}}(Comp)$$
$$S_{same_{in}}(Comp) = S_{same_{in}}(B)$$
$$S_{same_{out}}(B) = S_{same_{out}}(Comp)$$
$$S_{diff_{in}}(Comp) = S_{diff_{in}}(B)$$
$$S_{diff_{out}}(B) = S_{diff_{out}}(Comp)$$

$Comp$: `process` p ... S `end` p

$$C_{same_{in}}(S) = C_{same_{in}}(Comp)$$
$$C_{same_{out}}(Comp) = C_{same_{out}}(S)$$
$$C_{diff_{in}}(S) = C_{diff_{in}}(Comp)$$
$$C_{diff_{out}}(Comp) = C_{diff_{out}}(S)$$
$$S_{same_{in}}(S) = S_{same_{in}}(Comp)$$
$$S_{same_{out}}(Comp) = S_{same_{out}}(S)$$
$$S_{diff_{in}}(S) = S_{diff_{in}}(Comp)$$
$$S_{diff_{out}}(Comp) = S_{diff_{out}}(S)$$

$Comp$: `proc` $p(p_1, \ldots, p_n)$ S `end` p

$$C_{same_{in}}(S) = C_{same_{in}}(Comp)$$
$$C_{same_{out}}(Comp) = C_{same_{out}}(S)$$
$$C_{diff_{in}}(S) = C_{diff_{in}}(Comp)$$
$$C_{diff_{out}}(Comp) = C_{diff_{out}}(S)$$
$$S_{same_{in}}(S) = S_{same_{in}}(Comp)$$
$$S_{same_{out}}(Comp) = S_{same_{out}}(S)$$
$$S_{diff_{in}}(S) = S_{diff_{in}}(Comp)$$
$$S_{diff_{out}}(Comp) = S_{diff_{out}}(S)$$

$Comp$: `initial` S `end initial`

$$C_{same_{in}}(S) = C_{same_{in}}(Comp)$$
$$C_{same_{out}}(Comp) = C_{same_{out}}(S)$$

$$C_{diff_{in}}(S) = C_{diff_{in}}(Comp)$$
$$C_{diff_{out}}(Comp) = C_{diff_{out}}(S)$$
$$S_{same_{in}}(S) = S_{same_{in}}(Comp)$$
$$S_{same_{out}}(Comp) = S_{same_{out}}(S)$$
$$S_{diff_{in}}(S) = S_{diff_{in}}(Comp)$$
$$S_{diff_{out}}(Comp) = S_{diff_{out}}(S)$$

$Comp$: `final` S `end final`

$$C_{same_{in}}(S) = C_{same_{in}}(Comp)$$
$$C_{same_{out}}(Comp) = C_{same_{out}}(S)$$
$$C_{diff_{in}}(S) = C_{diff_{in}}(Comp)$$
$$C_{diff_{out}}(Comp) = C_{diff_{out}}(S)$$
$$S_{same_{in}}(S) = S_{same_{in}}(Comp)$$
$$S_{same_{out}}(Comp) = S_{same_{out}}(S)$$
$$S_{diff_{in}}(S) = S_{diff_{in}}(Comp)$$
$$S_{diff_{out}}(Comp) = S_{diff_{out}}(S)$$

The following are attribute definitions for components within procs/processes — statements.

S : S_1 S_2

$$C_{same_{in}}(S_1) = C_{same_{in}}(S)$$
$$C_{same_{in}}(S_2) = C_{same_{out}}(S_1)$$
$$C_{same_{out}}(S) = C_{same_{out}}(S_2)$$
$$C_{diff_{in}}(S_1) = C_{diff_{in}}(S)$$
$$C_{diff_{in}}(S_2) = C_{diff_{out}}(S_1)$$
$$C_{diff_{out}}(S) = C_{diff_{out}}(S_2)$$
$$S_{same_{in}}(S_1) = S_{same_{in}}(S)$$
$$S_{same_{in}}(S_2) = S_{same_{out}}(S_1)$$
$$S_{same_{out}}(S) = S_{same_{out}}(S_2)$$
$$S_{diff_{in}}(S_1) = S_{diff_{in}}(S)$$
$$S_{diff_{in}}(S_2) = S_{diff_{out}}(S_1)$$
$$S_{diff_{out}}(S) = S_{diff_{out}}(S_2)$$

S : `fa` ... `->` S_1 `af`

$$C_{same_{in}}(S_1) = C_{same_{in}}(S)$$

$C_{same_{out}(S)=C_{same_{out}}}(S_1)$
$C_{diff_{in}}(S_1) = C_{diff_{in}}(S)$
$C_{diff_{out}(S)=C_{diff_{out}}}(S_1)$
$S_{same_{in}}(S_1) = S_{same_{in}}(S)$
$S_{same_{out}(S)=S_{same_{out}}}(S_1)$
$S_{diff_{in}}(S_1) = S_{diff_{in}}(S)$
$S_{diff_{out}(S)=S_{diff_{out}}}(S_1)$

S : `do` $\ldots$ `->` S_1 [] $\ldots$ [] $\ldots$ `->` S_n `od`

$C_{same_{in}}(S_i) = C_{same_{in}}(S),\ \forall i := 1\ to\ n$
$C_{same_{out}}(S) = C_{same_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} C_{same_{out}}(S_i))$
$C_{diff_{in}}(S_i) = C_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} C_{diff_{out}}(S_i))$
$S_{same_{in}}(S_i) = S_{same_{in}}(S),\ \forall i := 1\ to\ n$
$S_{same_{out}}(S) = S_{same_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{same_{out}}(S_i))$
$S_{diff_{in}}(S_i) = S_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{diff_{out}}(S_i))$

S : `do` $\ldots$ `->` S_1 [] $\ldots$ [] `else` `->` S_n `od`

$C_{same_{in}}(S_i) = C_{same_{in}}(S),\ \forall i := 1\ to\ n$
$C_{same_{out}}(S) = C_{same_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} C_{same_{out}}(S_i))$
$C_{diff_{in}}(S_i) = C_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} C_{diff_{out}}(S_i))$
$S_{same_{in}}(S_i) = S_{same_{in}}(S),\ \forall i := 1\ to\ n$
$S_{same_{out}}(S) = S_{same_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{same_{out}}(S_i))$
$S_{diff_{in}}(S_i) = S_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{diff_{out}}(S_i))$

S : `if` $\ldots$ `->` S_1 [] $\ldots$ [] $\ldots$ `->` S_n `fi`

$C_{same_{in}}(S_i) = C_{same_{in}}(S),\ \forall i := 1\ to\ n$
$C_{same_{out}}(S) = C_{same_{in}}(S) \cup (\cup_{i=1\ to\ n} C_{same_{out}}(S_i))$
$C_{diff_{in}}(S_i) = C_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} C_{diff_{out}}(S_i))$
$S_{same_{in}}(S_i) = S_{same_{in}}(S),\ \forall i := 1\ to\ n$
$S_{same_{out}}(S) = S_{same_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{same_{out}}(S_i))$
$S_{diff_{in}}(S_i) = S_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)\ \cup\ (\cup_{i=1\ to\ n} S_{diff_{out}}(S_i))$

S : `if` $\ldots$ `->` S_1 [] $\ldots$ [] `else` `->` S_n `fi`

$C_{same_{in}}(S_i) = C_{same_{in}}(S),\ \forall i := 1\ to\ n$
$C_{same_{out}}(S) = \cup_{i=1\ to\ n} C_{same_{out}}(S_i)$
$C_{diff_{in}}(S_i) = C_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$C_{diff_{out}}(S) = \cup_{i=1\ to\ n} C_{diff_{out}}(S_i)$
$S_{same_{in}}(S_i) = S_{same_{in}}(S),\ \forall i := 1\ to\ n$
$S_{same_{out}}(S) = \cup_{i=1\ to\ n} S_{same_{out}}(S_i)$
$S_{diff_{in}}(S_i) = S_{diff_{in}}(S),\ \forall i := 1\ to\ n$
$S_{diff_{out}}(S) = \cup_{i=1\ to\ n} S_{diff_{out}}(S_i)$

The following are attribute definitions for call/send invocation statements.

S : `send` $a(...)$

$C_{same_{out}}(S) = C_{same_{in}}(S)$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)$
$S_{same_{out}}(S) = S_{same_{in}}(S) \cup \{a\}$
$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup \{\}$

S : `send` $y(...)$

$C_{same_{out}}(S) = C_{same_{in}}(S)$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)$
$S_{same_{out}}(S) = S_{same_{in}}(S) \cup \{x|\ ((y,u),(x,u),?) \in save(S, defs) \wedge u \neq unk\}$
$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup \{x|\ ((y,u),(x,v),?) \in save(S, defs) \wedge u \neq v\}$

S : `send` $z.a(...)$

$C_{same_{out}}(S) = C_{same_{in}}(S)$
$C_{diff_{out}}(S) = C_{diff_{in}}(S)$
$S_{same_{out}}(S) = S_{same_{in}}(S) \cup$
$\{b|\ r = curres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,u),?) \in save(S, defs) \wedge u \neq unk \wedge$
$b = resequiv(z,s,a)\}$
$S_{diff_{out}}(S) = S_{diff_{in}}(S) \cup$
$\{b|\ r = curres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,v),?) \in save(S, defs) \wedge u \neq v \wedge$
$b = resequiv(z,s,a)\}$

S : `call` $a(...)$

$C_{same_{out}}(S) = C_{same_{in}}(S) \cup \{a\}$
$C_{diff_{out}}(S) = C_{diff_{in}}(S) \cup \{\}$

$S_{same_{out}}(S) = S_{same_{in}}(S)$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)$

S : `call` $y(...)$

$C_{same_{out}}(S) = C_{same_{in}}(S) \cup \{x|\ ((y,u),(x,u),?) \in save(S, defs) \wedge u \neq unk\}$
$C_{diff_{out}}(S) = C_{diff_{in}}(S) \cup \{x|\ ((y,u),(x,v),?) \in save(S, defs) \wedge u \neq v\}$
$S_{same_{out}}(S) = S_{same_{in}}(S)$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)$

S : `call` $z.a(...)$

$$C_{same_{out}}(S) = C_{same_{in}}(S) \cup \{b|\ r = curres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,u),?) \in save(S, defs) \wedge u \neq unk \wedge b = resequiv(z,s,a)\}$$
$$C_{diff_{out}}(S) = C_{diff_{in}}(S) \cup \{b|\ r = curres \wedge (r,u) \in save(S, res) \wedge ((z,u),(s,v),?) \in save(S, defs) \wedge u \neq v \wedge b = resequiv(z,s,a)\}$$
$S_{same_{out}}(S) = S_{same_{in}}(S)$
$S_{diff_{out}}(S) = S_{diff_{in}}(S)$

Bibliography

[Ada83] American National Standards Institute. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A.

[AFD80] K. Apt, N. Francez, and W. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[And89] G. R. Andrews. A method for solving synchronization problems. *Science of Computer Programming*, 13(1989/90)(1):1–21, December 1989.

[And91] G.R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[AO] G.R. Andrews and R.A. Olsson. *Concurrent Programming in SR.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA. to appear.

[AO88] M. S. Atkins and R. A. Olsson. Performance of multitasking and synchronization mechanisms in the programming language SR. *SOFTWARE — Practice and Experience*, 18(9):879–895, September 1988.

[AO89] G. R. Andrews and R. A. Olsson. Report on the SR programming language, version 1.1. Technical Report CSE-89-11, University of California, Davis, Division of Computer Science, May 1989.

[AOC+88] G. R. Andrews, R. A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM*

Transactions on Programming Languages and Systems, 10(1):51–86, January 1988.

[AS83] G. R. Andrews and F. B. Schneider. Concepts and notation for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis Part I. Exhaustive analysis. *Acta Informatica*, 10:245–264, 1978.

[BLW87] A. Burns, A. M. Lister, and A. J. Wellings. *A Review of Ada Tasking*, volume 262 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[Bri78] P. Brinch Hansen. Distributed processes: A concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[Bur88] A. Burns. *Programming in occam*. Addison Wesley, 1988.

[CDV+86] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze. Toward real-time performance benchmarks for Ada. *Communications of the ACM*, 29(8):760–781, 1986.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[CKT86] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[Fel79] J. A. Feldman. High level programming for distributed computing. *Communications of the ACM*, 22(6):353–368, June 1979.

[GG90] R. E. Griswold and M. T. Griswold. *The Icon Programming Language (Second Edition).* Prentice Hall, Englewood Cliffs, NJ, 1990.

[GMGK84] H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[GR89] N. Gehani and W. D. Roome. *The Concurrent C Programming Language.* Silicon Press, New Jersey, 1989.

[Hen82] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[HI85] T. Hikita and K. Ishihata. A method of program transformation between variable sharing and message passing. *SOFTWARE — Practice and Experience*, 15(7):677–692, July 1985.

[Hil82] P. N. Hilfinger. Implementation strategies for Ada tasking idioms. In *Proceedings of the AdaTEC Conference on Ada*, pages 26–30, October 1982.

[HN80] A. N. Habermann and I. R. Nassi. Efficient implementation of Ada tasks. Technical Report CMU-CS-80-103, Carnegie-Mellon University, Department of Computer Science, January 1980.

[HO] D. T. Huang and R. A. Olsson. Axiomatic semantics of SR. unpublished notes.

[JA82] A. Jones and A. Ardo. Comparative efficiency of different implementations of the Ada rendezvous. In *Proceedings of the AdaTEC Conference on Ada*, pages 212–223, October 1982.

[Lam80] L. Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, June 1989.

[McN89] C. M. McNamee. Toward the optimization of interprocess communication mechanisms. Technical Report CSE-89-24, University of California, Davis, Division of Computer Science, September 1989.

[MH89] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MO90a] C. M. McNamee and R. A. Olsson. An attribute grammar approach to compiler optimization of intra-module interprocess communication. Technical Report CSE-90-44, University of California, Davis, Division of Computer Science, November 1990. submitted for publication.

[MO90b] C. M. McNamee and R. A. Olsson. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming*, 19, October 1990. to appear.

[Mos90] L. E. Moser. Data dependency graphs for Ada programs. *IEEE Transactions on Software Engineering*, 16(5):498–509, May 1990.

[MP90] S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–105–II113, 1990.

[MRS89] Ada Runtime Environment Working Group MRSTI Task Force — Association for Computing Machinery Special Interest Group for Ada. *A Model Runtime System Interface for Ada*, January/February 1989.

[OG76] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[O'K90] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, Cambridge, MA, 1990.

[Ols90] R. A. Olsson. Using SR for discrete event simulation: A study in concurrent programming. *SOFTWARE — Practice and Experience*, 20(12):1187–1208, December 1990.

[Ols91] R. A. Olsson. Parallelism, distribution, and synchronization in SR. In *Proceedings of the IEEE Computer Society's COMPCON Spring 1991*, pages 407–412, February 1991.

[OM] R. A. Olsson and C. M. McNamee. Inter-entry section: Nondeterminism and explicit control mechanisms. *Computer Languages.* to appear.

[OW89] R.A. Olsson and G.R. Whitehead. A simple technique for automatic recompilation in modular programming languages. *SOFTWARE — Practice and Experience*, 8:757–773, August 1989.

[REMC81] E. S. Roberts, A. Evans, C. R. Morgan, and E. M. Clarke. Task management in Ada — A critical evaluation for real-time multiprocessors. *SOFTWARE — Practice and Experience*, 11:1019–1051, 1981.

[RS88] J. H. Reif and S. A. Smolka. The complexity of reachability in distributed communicating processes. *Acta Informatica*, 25(9):333–354, 1988.

[Sch82] J. Schauer. Vereinfachung von prozess-systemen durch sequentialisierung. Technical Report Bericht 30/82, Universität Karslruhe, Institut für Informatik II, 1982. Dissertation.

[Ste80] D. R. Stevenson. Algorithms for translating Ada multitasking. *SIGPLAN Notices*, 15(11):166–175, November 1980.

[Ves89] S. Vestal. Mixing coroutines and processes in an Ada tasking implementation. *Ada Letters*, IX(2):90–101, March/April 1989.

[War80] D. H. D. Warren. Logic programming and compiler writing. *SOFTWARE — Practice and Experience*, 10:97–125, 1980.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wei90] R. P. Weicker. An overview of common benchmarks. *IEEE Computer*, 23(12):65–75, December 1990.