New Mechanisms for Invocation Handling in Concurrent Programming Languages *

Mandy Chung and Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
{chungm,olsson}@cs.ucdavis.edu

Send correspondence regarding this paper to Olsson

October 2, 1998

Abstract

Invocation handling mechanisms in many concurrent languages have significant limitations that make it difficult or costly to solve common programming situations encountered in program visualization, debugging, and scheduling scenarios. This paper discusses these limitations, introduces new language mechanisms aimed at remedying these limitations, and presents an implementation of the new mechanisms. The examples are given in SR; the new mechanisms and implementation are an extension of SR and its implementation. However, these new mechanisms are applicable to other concurrent languages. They can augment or replace current invocation handling mechanisms.

Keywords: concurrent programming languages, invocation handling, language design, language implementation.

*This research is supported in part by the National Science Foundation under grant CCR-9527295.

1 Introduction

Many concurrent languages provide mechanisms that represent a communication channel — or *operation* [1] — and for generating (or invoking) and servicing invocations of operations. For example, synchronous or asynchronous message passing is provided in Ada [2], Concurrent C [3], CSP [4], Linda [5, 6], occam [7], Orca [8], and SR [9].

Invocation servicing mechanisms typically can service invocations from one of several operations. They may also allow explicit control in selecting which operation to service or in selecting which invocation for a particular operation to service. For example, Ada provides `select`/`accept` and SR provides input (`in`) statements. Some languages allow the selection of which invocation to service to be based on the values of an invocation's parameters. For example, SR's synchronization and scheduling expressions (**st** and **by** clauses) control which invocations are selectable and the order in which they are serviced. Some languages also provide a mechanism that gives the number of pending invocations for an operation, e.g., Ada's `COUNT` attribute and SR's '?' operator.

No existing mechanisms, however, provide a simple and efficient way to examine pending invocations of an operation or allow the selection decision to be based on more than just a single invocation's parameters. Examining invocations and "cross-invocation" or "cross-operation" selection are important in a number of real applications, including debugging, visualization, and scheduling algorithms. Solutions coded using current mechanisms often are cumbersome and inefficient.

This paper illustrates these shortcomings of existing invocation handling mechanisms and introduces new language mechanisms aimed at providing additional flexibility in invocation handling. The examples are given in SR and the new mechanisms are given as an extension to SR. However, the underlying concepts are language-independent and our work is applicable to other concurrent languages. We have built an initial implementation of the new mechanisms, which shows the new mechanisms have reasonable costs in cases of typical use.

The rest of this paper is organized as follows. Section 2 gives a brief overview of relevant SR language background. Section 3 outlines the general shortcomings of invocation handling mechanisms and illustrates the shortcomings via specific examples. Section 4 introduces our new mechanisms that provide additional, more expressive support for invocation handling. Sections 5 and 6 describe the implementation of the new mechanisms and their performance. Sections 7 and 8 discuss design issues and how our work can be applied to other languages. Finally, Section 9 concludes the paper. Further details on this work appear in [10].

2 Background: Invocation Handling in SR

The SR language provides a variety of mechanisms for writing parallel and distributed programs. Operations and the different ways to invoke and service them form the bases for SR's process interaction mechanisms. Refer to [9, 11] for a complete description of SR.

We focus here on operations serviced by SR's input (**in**) statement, which is used to service invocations as part of synchronous or asynchronous message passing. The examples below illustrate the key aspects of **in**.

Consider the code in Figure 1 for a server process that implements a shortest-job-next request allocator. The first arm of the **in** statement uses a synchronization expression (**st** clause) to accept

```
# shortest-job-next request allocator
process server
   var free := true
   do true ->
      in request(size) st free by size -> free := false
      [] release() -> free := true
      ni
   od
end
```

Figure 1: *Shortest-job-next request allocator.*

an invocation of `request` only when the server is free, and uses a scheduling expression (**by** clause) to select the shortest job among those pending invocations. The scheduling expression associated with `request` uses the invocation parameter `size` to determine which job's size is the smallest. Hence, on each iteration of the loop, the server process will service the invocation of `request` with the smallest `size` — but only if `free` is true — or it will service an invocation of `release`.

The final arm of an **in** statement can be an **else** arm. The statements associated with this arm will be executed if no invocation is selectable. For instance, consider the following code

```
# service all invocations of f for which x = 3
do true ->
   in f(x) st x = 3 ->
       ...
   [] else ->
      exit
   ni
od
```

This loop services all pending invocations of `f` whose parameter `x` is equal to 3. Each iteration of the loop services one such invocation, if there is one; otherwise, the else arm is executed and it exits the loop.

Generally, an input statement services invocations in first-come, first-served (FCFS) order according to their arrival time. This default order can be overridden by synchronization and scheduling expressions, as seen in the above examples.

SR's '?' operator returns the number of invocations currently pending for an operation. For example, it can be used to control a loop that services all pending invocations of operation `f`.

```
do ?f > 0 ->
   in f(x) ->
      ...
   ni
od
```

As another example, the '?' operator can be used to give preference in servicing one operation over another. For example, Figure 2 shows an **in** statement that can be used within a job scheduler to give preference to interactive requests over batch requests.

```
in interactive(...) -> ...
[] batch(...) st ?interactive = 0 -> ...
ni
```

Figure 2: *Job Scheduler (giving preference to interactive requests).*

SR's **forward** statement defers replying to a called invocation and instead passes on this responsibility to another operation. It does so by generating a new invocation. The forwarding process continues execution following **forward**. An example of the use of **forward** is the following (from [9]). Client processes make requests for service to a central allocator process. The allocator assigns a server process to the request by forwarding the invocation to it. To be more concrete, the allocator might represent a file server to which clients pass the names of files. The allocator determines, based on local information it maintains, on which server the requested file is located and forwards the client's request to the server, which typically would be located on a different machine. After the server services the forwarded invocation, it replies directly to the client.

Operations in SR programs can be shared (invoked or serviced) by more than one process. For example, a group of client processes may invoke a single operation in a server process. As another example, a group of worker processes may share a bag of tasks. The bag is represented by a single shared operation and each task is represented by an invocation. Each worker process repeatedly services a task from the bag.

3 Shortcomings of Invocation Handling Mechanisms

The two key shortcomings of current language mechanisms for invocation handling are:

- No *peeking*: Invocations can be examined only by actually servicing the invocation (or by employing very obscure and inefficient synchronization and scheduling expressions).

- Single-invocation based decisions: When choosing which invocation to service, only one invocation at a time can be considered within a synchronization or scheduling expression.

These limitations make it difficult to solve common programming situations encountered in program visualization, debugging, and some scheduling scenarios.

Linguistic mechanisms such as synchronization and scheduling expressions (e.g., **st** and **by** clauses in SR) are intended for examining invocation queues and for controlling selection of invocations to service. Although they are sufficient to solve many problems, they are limited in the two ways noted above.

Consider these limitations within SR. First, the only way to examine an invocation in SR is to actually service it, i.e., remove it from the queue of invocations and — given that it is not desirable to really service it — return it to the invocation queue by re-invoking the operation. As a result, code that needs to examine invocations is cumbersome and inefficient. Second, cross-invocation or cross-operation selection also requires examination of invocations. To implement this kind of scheduling problems using current SR's mechanisms often requires changes in the interface. That is, besides modifying the servicing code for the server, the code for the clients often needs to be changed as well to adapt the modified interface, for example, to invoke a new operation or to invoke the same operation name but with different parameterization.

The following subsections present specific examples to illustrate the above limitations and the kinds of applications where examining invocations is important.

3.1 First-in, First-out Order Peeking at Pending Invocations

During debugging, programmers often want to display pending invocations and their parameter values for a given operation. Similarly, visualizing or animating the execution of a concurrent program often requires displaying pending invocations. Such *peeking* should not disturb the invocations or their order on the operation's queue.

Consider, for example, an elevator controller simulation in which each person is represented as a process and the elevator controller is represented as a separate process. When a person process wants to take the elevator to another floor, it invokes operation `f` in the elevator controller; the controller services the invocation when it schedules the elevator to pick up this passenger. In a visualization of an execution of the simulation, the picture should show which people are waiting on which floors, i.e., the currently pending invocations for `f`.

One way to display all invocations of `f` is shown in Figure 3. Note how it services each invocation of `f` and then forwards that invocation back to `f`. This code uses **forward** to delay replying to the

```
op f(x: int)
var count := ?f # sets count to number of pending invocations of f
fa i:= 1 to count ->
   in f(x) ->
      write(x) # print out parameter
      forward f(x) # puts invocation back on queue
   ni
af
```

Figure 3: *Examining an operation without preserving arrival order of invocations.*

invoker until the invocation is really serviced (i.e., when the elevator picks up the passenger) rather than just examined. Its cost is high even though the invocation queue, in effect, does not change over this program fragment. Each invocation of `f` being examined is serviced once and re-invoked once.

A major shortcoming of the code in Figure 3 is that it does not necessarily preserve the arrival order of invocations of `f`. In particular, new invocations of `f` that arrive during execution of that code can end up at the front of the queue rather than at the back, or interspersed with old invocations. Thus, the seemingly simple act of examining invocations can have a nasty side effect that can result in confusion in debugging and can introduce unfairness (and possibly starvation) into how invocations are serviced.

To illustrate, consider the pending invocations of `f` at three different execution states shown in Figure 4. Each square represents one pending invocation of `f` containing the indicated value of the

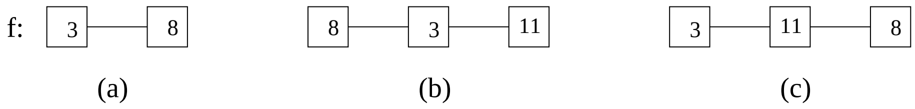

Figure 4: *Pending invocations of* `f` — *(a) before entering the loop (b) after the first iteration (c) after the second iteration.*

invocation parameter `x`. Initially, before examining `f` and executing the **fa** loop in Figure 3, invocations `f(3)`[1] and `f(8)` are pending as shown in Figure 4(a) and `count` is hence initialized to 2. During the

[1] The notation `f(3)` indicates an invocation of `f` with parameter 3. In our examples, each parameter value is unique, thus avoiding ambiguity in this notation.

first iteration of the **fa**, invocation `f(3)` is examined (via **in**) and then forwarded back to `f`. At the end of this iteration, a new invocation `f(11)` arrives and is appended onto the invocation queue, resulting in the invocation queue shown in Figure 4(b). The second iteration, which is the last iteration of the **fa**, examines invocation `f(8)` and then forwards it back to `f`. As a result, after the **fa** terminates, the new invocation `f(11)` ends up at the middle of the invocation queue (rather than at the back), which is shown in Figure 4(c). In this example, the first execution of the code in Figure 3 displays the correct output as described above (output of 3 and 8); however, printing the invocations again will show the different initial subsequence (output of 3, 11, and 8) seen in Figure 4(c).

The arrival order of invocations of `f` can be preserved, with additional programming effort, by using an auxiliary operation `aux_f`. Invocations of `f` are divided into two groups: invocations that have been examined and those that have not. All previously examined invocations are kept in `aux_f` while the rest remain in `f`. Unfortunately, this approach, like the code in Figure 3, is expensive because it services each invocation and then forwards it.

The code above assumes that only one process is examining and servicing invocation of a given operation. However, as noted in Section 2, more than one process can service invocations of a given operation. The above code can be modified to accommodate multiple processes, but at greater complexity and expense. See [10] for details.

3.2 Cross-invocation and Cross-operation Scheduling

As noted above, SR's input statement does not directly allow the decision as to which invocation is to be serviced to be based on multiple invocations or operations, a more general form of peeking. A number of fairly common and important problems, however, need exactly that capability to be implemented easily and efficiently. Below we present a few representative examples.

3.2.1 Lottery Scheduling

Consider the problem of implementing a lottery scheduling algorithm for resource management [12] in a system of user processes, which make requests of a manager process. Each request includes the user's identifier, *uid*. The manager process picks a uid randomly from the uids of all requests that are pending. In SR pseudocode, one possible structure for the manager process code is

```
do true ->
   in request(uid,...)
      st uid = "the uid chosen randomly from those in all pending requests" ->
      # service this request for some quantum
        ...
      if "more work left before this request can finish" ->
         # needs further service, so put it back on queue
         forward request(uid,...)
      fi
   ni
od
```

The above synchronization expression cannot be expressed directly in SR, so indirect means must be used. Figure 5 shows the code that implements the lottery scheduler. That code uses a loop that

```
var count := ?request
fa i := 1 to count ->
   in request(uid,...) ->
      # save the uid for later selection
        ...
      forward request(uid,...)
   ni
af
# set theuid to the randomly chosen one
theuid := random_uid()
in request(uid,...) st uid = theuid ->
   /* body as before */
ni
```

Figure 5: *Lottery scheduler.*

examines all invocations before the **in** statement, much like the code in Figure 3, and sets `theuid` to the randomly chosen one; the procedure `random_uid` chooses and returns one uid randomly from those uids saved in the **fa** loop. (For simplicity, the code assumes that there is at least one pending invocation of `request`.) The **in** statement then uses that variable in the synchronization expression to service that invocation.

3.2.2 Extended Job Scheduling

Consider extending the job scheduler in Figure 2 so that it gives preference to interactive requests unless there is a batch job submitted by the superuser (whose `uid` is 0). As with the lottery scheduler, there is no direct way in SR to solve this, but there are five noteworthy indirect solutions. The techniques used in these indirect solutions also work in many other similar cross-invocation and cross-scheduling scenarios.

Pre-in Loop As shown in Figure 6, a "pre-**in** loop", in the style suggested for the lottery scheduler (Figure 5), can be used to determine whether any superuser batch invocations are pending. Note that an auxiliary operation `aux_batch` is used in that code. Before servicing an invocation, the "pre-**in** loop" first counts the number of superuser batch invocations and then forwards all `batch` invocations to `aux_batch`. It services the superuser batch invocations from `aux_batch`, if any. If no such invocations are pending (i.e., `superbatch = 0`), it then services an invocation, as it did originally.

```
superbatch := 0
do true ->
   # check whether there are any superuser batch jobs
   do true ->
      in batch(uid,...) ->
         if uid = 0 -> superbatch++ fi
         forward aux_batch(uid,...)
      [] else ->
         exit
      ni
   od
   in aux_batch(uid,...) st uid = 0 ->  # superuser batch jobs
      superbatch--
      ...
   [] interactive(uid,...) st superbatch = 0 -> ...
   [] aux_batch(uid,...) st superbatch = 0 and ?interactive = 0 -> ...
   ni
od
```

Figure 6: *Extended job scheduler using a pre-in loop to examine superuser batch invocations.*

The above "pre-**in** loop" only examines invocations of `batch` without actually servicing them. Another kind of "pre-**in** loop" is also possible for the extended job scheduler, as shown in Figure 7.

```
superbatch := 0
do true ->
   # service all superuser batch jobs first
   do true ->
      in batch(uid,...) st uid = 0 ->
         ...
      [] else ->
         exit
      ni
   od
   in interactive(uid,...) -> ...
   [] batch(uid,...) st ?interactive = 0 -> ...
   ni
od
```

Figure 7: *Extended job scheduler using a pre-in loop to service superuser batch invocations.*

In contrast to the code in Figure 6, this "pre-**in** loop" services all superuser batch invocations instead of just examining them. No additional operation is needed in this code. This code is more efficient because invocations are not forwarded to another operation. However, this kind of "pre-**in** loop" can only be used when a synchronization expression in **in** can be used to identify which invocations to service. It could not be used, for example, for the lottery scheduler in Section 3.2.1 because all requests must be examined before selecting a request to service and a synchronization expression could not specify such selection.

Splitting The batch operation can be *split* [13], e.g.,

```
in interactive(uid,...) st ?superbatch = 0 -> ...
[] batch(uid,...) st ?superbatch = 0 and ?interactive = 0 -> ...
[] superbatch(uid,...) -> ...
ni
```

In this case, the interface has to be changed and the superuser process should invoke the operation `superbatch` instead of `batch` to submit a batch job.

Combining The batch and interactive operations can be *combined* into an array of operations [14], say `request`, with indices ranging over `SUPERBATCH`, `INTERACTIVE`, and `BATCH`, e.g.,

```
in (kind := SUPERBATCH to BATCH) request[kind](uid,...)
    st myturn(kind) ->
   ...
ni
```

This **in** will service a single invocation of `request`. Because elements of `request` are checked in non-deterministic order, the procedure `myturn` is used to enforce the desired ordering by checking whether any invocations with higher priority are pending, e.g.,

```
proc myturn(kind) returns ans
   ans := true
   # are there any pending higher priority invocations?
   # check up through kind's predecessor
   fa k := SUPERBATCH to pred(kind) ->
      if ?request[k] > 0 ->
         ans := false
         return
      fi
   af
end
```

The quantifier variable `kind` in the **in** statement is passed to `myturn` as a parameter. This procedure returns true if no invocations with priority higher than `kind` are pending; otherwise, it returns false. The **in** statement invokes `myturn` for each pending invocation until a selectable invocation with true return value from `myturn` is found, i.e., the highest priority invocation that is pending.

Similarly, the batch and interactive operations can be combined into a single operation with an additional parameter, indicating the priority of each request e.g.,

```
in p_request(uid,...,priority) by -priority -> # prioritized requests
   ...
ni
```

This **in** will service the invocation of `p_request` whose priority is the highest.

Nesting The batch and interactive operations can be serviced by nested **in** statements, e.g.,

```
in batch(uid,...) st uid = 0 -> ... # any superuser requests?
[] else ->
   in interactive(uid,...) -> ...
   [] batch(uid,...) st ?interactive = 0 -> ...
   ni
ni
```

This solution first checks whether any invocations of `batch` are from the superuser. If not, it then services an invocation of `interactive` or an invocation of `batch` if no interactive invocations are pending.

Preferences Another approach is to use preferences, as proposed in [15, 16, 14] and implemented in occam [7] and in an experimental version of SR [17]. For example, the extended job scheduler can be written

```
in [0] batch(uid,...) st uid = 0 -> ...
[] [1] interactive(uid,...) -> ...
[] [2] batch(uid,...) -> ...
ni
```

Here, we denote the preference assigned to each arm by the integer expression, lower-valued for higher-priority, at the start of each arm.

These approaches — "pre-**in** loop", splitting, combining, nesting, and preferences — have disadvantages. The "pre-**in** loop" is costly since it involves generating new invocations (see Section 3.1). The other approaches work well only when the number of choices is not too large, because each choice translates into a distinct arm of the input statement or a distinct operation index. Splitting or combining requires changes in the interface. Combining will not work easily if the original operations have different parameterizations. Nesting also incurs extra overhead due to its executions of separate **in** statements. Preferences require additional implementation effort and costs [17, 14]. "Pre-**in** loop", splitting, nesting, or preferences can require code to be repeated.

3.3 Other Motivating Examples

Sections 3.1 and 3.2 gave specific examples to illustrate the shortcomings. Other examples include:

Threshold Order Peeking Some applications require examining invocations in several different groups, each of which, for example, contains pending invocations whose parameter values are above a certain threshold, displaying one group of invocations at a time. Once all invocations of a group have been examined or no invocations of that group are pending, the threshold is then lowered to examine the next group of invocations. This kind of peeking differs from the peeking example in Section 3.1, where all pending invocations are examined as one whole group. Threshold peeking occurs in an airline boarding simulation in which each passenger is represented as a process and the plane controller is also represented as a separate process. Passengers are called for boarding according to their seat numbers. Passengers who sit with seats at the rear of the plane are boarded first, one row at a time.

Other Scheduling Scenarios Additional kinds of cross-invocation scheduling occur in real applications. *Two-dimensional scheduling* picks the least invocation based on invocation parameters x and y according to the usual ordering among pairs. This type of ordering occurs in priority scheduling, e.g., x represents class of user and y represents size of memory request; it also occurs in updating images, e.g., x and y represent coordinates on a screen. *Median scheduling* selects the *median* invocation based on arrival order or on parameter value. *Last-in, first-out scheduling* selects the most recently arrived invocation.

Multi-way Rendezvous The *multi-way rendezvous* [18] is a generalization of rendezvous in which more than two processes participate. More than a simple **in** statement is needed to provide this functionality.[19]

4 New Invocation Handling Mechanisms

Section 3 identified two general classes of problems for which current invocation handling mechanisms have shortcomings: those that require peeking and those that require cross-invocation or cross-operation selection.

To overcome those shortcomings, we introduce three new statements — **rd**, **mark**, and **take** — that deal with invocations. The **rd** (for "read", but pronounced as "R–D") statement reads invocations in an invocation queue without disturbing their order or contents. The **mark** statement marks the invocation currently being examined via **rd**. A marked invocation can be serviced later by a **take** statement.

4.1 Rd Statement

Syntactically, the **rd** statement looks similar to an **in** statement. Its last arm can be an else command.

Semantically, the **rd** statement is an iterator that reads all pending invocations, one per iteration, for operations appearing in the same **rd** in their arrival order. It treats the collection of invocations as a group rather than individually, as **in** does. In general, the first iteration of **rd** reads the oldest invocation and each subsequent iteration reads the next invocation until all pending invocations have been accessed. Each iteration neither removes an invocation nor modifies its contents. When no invocations are pending or all pending invocations have been read by the **rd**, the process is delayed until one of the operations appearing in the **rd** is invoked. If the **rd** statement contains an else command, then the else command's block of code is executed before the process is delayed. The else command is typically used to terminate, via an **exit** statement, the **rd** statement.

For example, the **rd** statement in Figure 8 examines all invocations *currently* pending for `f`.

```
# examining an operation while preserving arrival order of invocations
rd f(x) ->
   write(x) # print out parameter
[] else ->
   exit # terminate rd when no unexamined invocations remain
dr
```

Figure 8: *Examining an operation.*

The way in which **rd** iterates over the invocations pending on `f`'s invocation queue eliminates the potential of rearranging their order that was present in Figure 3. Invocations of `f` no longer need to be removed from and re-appended to the invocation queue to examine them. Without the else command, execution of the code in Figure 8 would not terminate. The executing process would read all pending invocations and then block until a new invocation of `f` arrives.

Similar to the **in** statement, **rd** can also employ synchronization and scheduling expressions to obtain more control over which invocation to examine and in what order. For example, the manager process in the lottery scheduler of Section 3.2.1 can use a scheduling expression to read pending requests in non-decreasing order of their invocation parameter `uid`:

```
rd request(uid,...) by uid ->
   # save unique uid for later selection
     ...
[] else ->
   exit
dr
```

Then, the manager process can pick a uid randomly and use an **in** statement to service a request, as it did in Figure 5.

4.2 Mark and Take Statements

The **mark** statement *marks* for possible later servicing an invocation currently being examined by an **rd** statement. The **take** statement services a previously marked invocation, namely the most recently marked invocation, by a given process, of the specified operation. The selected invocation is serviced by executing the associated block of code.

As a simple example of the mark and take statements, Figure 9 shows how they and the **rd** statement can simulate an **in** statement.

```
rd f(x) ->
   mark f
   take f(x) -> ... /* service invocation */ ekat
   exit
dr
```

Figure 9: **rd** *simulation of* **in**.

As another example, Figure 10 shows how to use **rd**, **mark**, and **take** to select the *median* invocation based on parameter value. This code examines pending invocations in sorted order via the **by** of the **rd**, counting until it reaches the invocation with the median value, which it then services. A solution to this problem expressed using **in** and preserving the order of invocations would have difficulties and costs similar to those described in Section 3.1.

As a final example of the use of the **mark** and **take** statements, Figure 11 shows a solution to the extended job scheduling problem from Section 3.2.2. The **rd** statement reads invocations of `batch` and `interactive`. Specifically, it examines all pending invocations of `batch` until there are no more or

```
do ?f > 0 -> # repeatedly service the median-valued invocation of operation f
   median := ?f/2+1 # recall ?f is the number of pending invocations of f
   count := 0
   # read invocations of f in the order of x
   rd f(x) by x ->
      if ++count = median ->
         # service the invocation of f for which x is the median
         mark f
         take f(x) -> ... /* service invocation */ ekat
         exit
      fi
   dr
od
```

Figure 10: *Median-value scheduling.*

it finds one from the superuser; it reads only the first invocation of `interactive` provided it has not already seen an invocation of `batch` from the superuser. It marks (at most) only the first invocation for each of the three kinds of service. The **rd** terminates via the **exit** statement in its else arm. The **exit** is executed if an invocation was marked; otherwise, the process blocks waiting for an invocation of `batch` or `interactive` to arrive before it executes an arm of the **rd**. This code avoids the problems of the solutions presented in Section 3.2.2 for this same example. First, this code does not modify the invocation queue when selecting an invocation to service via **rd**. Second, it does not require changes in the interface; processes service invocations of `batch` and `interactive` as they originally did. Last, it does not require code to be repeated for servicing superuser batch jobs and normal batch jobs. This solution is, however, lower-level in the sense that it explicitly marks and takes invocations and uses boolean flags to control doing so.

Other, higher-level solutions are also possible. For example, an **rd** could be used, in the style of a "pre-**in** loop", to determine whether any superuser invocations of `batch` are present. If so, the invocation could be serviced using **mark** and **take**; if not, the original **in** statement could be used.

The **take** statement can have an optional else arm, which is typically used to output error messages when the **take** statement fails to service an invocation. For example, suppose two processes are each executing the following **rd** statement at about the same time and mark the same invocation

```
var gotsuper := false, gotbatch := false, gotinteractive := false
# find the invocation to service
rd batch(uid,...) st not gotsuper ->
   if uid = 0 # any superuser requests?
      mark batch
      gotsuper := true
   [] uid != 0 & not gotbatch -> # mark first one
      mark batch
      gotbatch := true
   fi
[] interactive(uid,...) st not gotinteractive & not gotsuper ->
   mark interactive
   gotinteractive := true
[] else ->
   if gotsuper | gotbatch | gotinteractive -> exit fi
dr
# now service
if gotsuper | (gotbatch & not gotinteractive) ->
   take batch(uid,...) -> ... etak
[] else -> # i.e., gotinteractive
   take interactive(uid,...) -> ... etak
fi
```

Figure 11: *Extended job scheduler using* **rd**.

by executing

```
rd f(x) ->
   mark; exit
dr
```

If both processes attempt to take the marked invocation by executing

```
take f(x) ->
   ...
[] else ->
   write("take failed: marked invocation has been serviced")
ekat
```

then the second **take** statement to execute will fail. The else arm's block of code will then be executed. If the failed **take** statement does not contain an else arm, it terminates immediately and the executing process continues after the **ekat**. Unlike the **in** statement, the executing process does not block when the **take** statement does not service an invocation. Note that the code in Figures 10 and 11 assumed that invocations are not "stolen" by another process; see [10] for ways to deal with such concurrent access.

The **rd**, **mark**, and **take** statements can also be used without much difficulty to program solutions for threshold order peeking, other scheduling scenarios, and multi-way rendezvous described in Section 3.3; see [10] for details.

5 SRR Implementation

The SRR ("SR with **rd**") implementation extends the standard SR compiler and run-time support (RTS) [11, 9]. The RTS provides primitives for the generated code to invoke and service operations.

5.1 The Standard SR Implementation

The RTS maintains a queue of pending invocations for an operation that is serviced by **in** statements. For the invocation of an operation, the compiler generates codes for allocating an invocation block, filling in parameter values, and passing the invocation block to the RTS. When receiving the invocation block, the RTS appends the invocation onto the end of the queue associated with the operation and awakens the servicing process if it is waiting for such an invocation. The RTS will block the invoking process for a synchronous invocation, but not for an asynchronous invocation.

When a process executes an **in**, it must first obtain from the RTS exclusive access for the queue. It then searches the invocation list for an acceptable invocation. The executing process then requests the RTS to remove that invocation or to block itself if it found no acceptable invocation. When the process completes servicing the invocation, it invokes the RTS again, so the RTS can pass results back to the invoker or free up the invocation block.

The RTS provides locking of the invocation lists to avoid race conditions between, for example, an invoking process that is appending a new invocation and a servicing process that is removing an invocation.

5.2 Key SRR Implementation Issues

The iterative semantics of **rd** makes its implementation more complicated than the implementation of **in**, which deals with a single invocation. The RTS must maintain additional state information for each process executing an **rd** indicating which invocations it has already examined. The process executing **rd** acquires access to the invocation queue at the beginning of each iteration of the **rd** (i.e., before choosing an invocation for examination) and releases access before executing the command body.

Three other factors further complicate the implementation. First, as a given process is executing an **rd** for a given operation, other processes can modify the operation's invocation list — by adding invocations or by servicing (via **in** or **take**) invocations, possibly one that is currently being accessed by **rd**. Second, a process can use synchronization and scheduling expressions to read invocations in non-FCFS order. Third, a given operation can be examined by several processes simultaneously.

The SRR RTS uses an *examination queue* for the execution of each **rd** in the general case. An examination queue contains pointers to all unread invocations in the invocation queue in FCFS order. On each iteration of the **rd**, the process removes the pointer to the invocation selected for examination from the examination queue. The process also updates the examination queue to include the invocations that arrived after the previous iteration and to remove invocations that have been serviced (typically by other processes) during this iteration of **rd** but that were included in the examination queue.

The RTS also maintains, for each process executing an **rd**, pointers to two invocations: ICBE, the invocation that is currently being examined; and ILAST, the last invocation that has been inspected by the process in the invocation queue. (We distinguish between "inspected" and "examined". An invocation that has been inspected does not necessarily mean that it has been examined. An invocation is examined only when the command body associated with the **rd** is executed for that invocation.) ICBE is used to implement **mark** and ILAST is used in determining which invocations have not been inspected as new invocations are appended to the invocation queue.

The invocation queue can be modified by other processes while an **rd** is examining an invocation from the invocation queue. In particular, the ICBE or ILAST of an executing **rd** can be removed and freed by another process. The former case is straightforward: an additional flag in each invocation indicates whether or not it has been serviced and a reference count records the number of processes that are currently examining the invocation. The latter case is more complex. The RTS maintains in each invocation a list of processes that reference the invocation as an ILAST. When an invocation is serviced, the RTS informs each process in this list to update its ILAST pointer on its next iteration of **rd**.

A race condition can occur in accessing an invocation if one process is using **rd** to examine the invocation while another process is using **in** to service the invocation. To avoid this problem, **rd** can make a copy of the invocation block. However, dataflow analysis can identify when this invocation copying can be avoided, as occurs in many typical programs, including all tests reported on in Section 6.

5.3 SRR Implementation Refinement

Our SRR implementation optimizes many **rd** statements by dividing them into three disjoint types:

- *Unordered invariant* **rd**
- *Ordered invariant* **rd**
- *Variant* **rd**

An **rd** is *invariant* if the selection criteria for invocation examination (including the examination order) do not change over execution of the **rd**; i.e., the values of synchronization and scheduling expressions, if any, for each invocation do not change over execution of the **rd**. The terms "unordered" and "ordered" indicate whether or not the programmer *explicitly* specifies the examination ordering using a scheduling expression in the **rd**. An *unordered* invariant **rd** contains no scheduling expression, so the **rd** examines invocations in the normal FCFS invocation arrival order. An *ordered* invariant **rd** contains a scheduling expression whose value for any given invocation is the same across all iterations. An **rd** statement other than an invariant **rd** is a *variant* **rd**.

For example, consider the following **rd** statements:

```
rd f(x) -> ... dr                # unordered invariant
rd f(x) st x = 3 -> ... dr       # unordered invariant
rd f(x) by -x -> ... dr          # ordered invariant
rd f(x) st x > 0 by x -> ... dr  # ordered invariant
rd f(x) st x > t -> ... dr       # variant
```

The final statement is a variant **rd** because `t`, a local variable, can be modified over execution of the **rd**. Thus, an invocation that was not selectable for examination in past iterations of **rd** might become selectable in the next iteration.

The SRR compiler employs a simple and conservative analysis to determine statically whether or not an **rd** is invariant. If the synchronization and scheduling expressions of an **rd** reference only literals, constants, and invocation parameters and the **rd** contains no assignment statements, then the compiler considers the **rd** to be an invariant **rd**; thus, many commonly occurring **rd** statements are invariant. Otherwise, the compiler considers the **rd** to be a variant **rd**.

A variant **rd** requires the general case implementation using examination queues. Because the selection criteria for examination may change during its execution, each invocation may be accessed multiple times throughout the execution of the **rd**.

An ordered invariant **rd** requires a less costly implementation. Each process maintains a list of pointers only to *selectable* invocations for examination. The list is ordered by the value of the scheduling expression. On each iteration of the **rd**, the executing process asks the RTS for the first invocation in this ordered list. Each examined invocation is accessed only twice: the first access for including it into the ordered list and the second access for actual examining the invocation.

An unordered invariant **rd** has the least costly implementation. Each process maintains a pointer into the invocation queue indicating how far along the process has read through the queue. Each invocation is visited once. No examination queue is needed.

6 Performance

We evaluated the new mechanisms quantitatively using *micro-benchmarks* to measure the performance of individual mechanisms and *macro-benchmarks* to measure the performance of more realistic programs that consist of many language elements.

We ran the benchmarks on four UNIX systems: DEC Alpha 3000/400 (OSF 3.2), DECstation 260 (Ultrix 4.3), SGI Indigo (IRIX 5.3), and Sun SPARCstation 5 (SunOS). The implementations of SR and SRR use their own lightweight threads to simulate concurrency on these uniprocessors. We used the Unix *time* command to determine the total CPU times for the benchmark programs. To discount caching effects, we also measured the number of instruction cycles for the benchmark programs using *pixie*. All systems were lightly loaded when timing tests were run. Timing tests were run multiple times; variances between execution times were very small. Below, we summarize the results obtained on the Alpha. The overall results for the other systems were similar, although the specific results varied due to differing costs of context switching and memory allocation/deallocation.

6.1 Micro-benchmarks

Basic rd Overhead The implementation of **rd** requires additional RTS data structures and additional tests when generating and servicing invocations. We compared the performance of several programs using just the **in** statement when run with the SR and SRR implementations. The SRR performance depends on whether or not the operation also appears within an **rd** statement. Table 1 summarizes the cost of generating and servicing an invocation. Compared to the SR implementation,

Implementation	*Time(μs)*	*Cycles*
SR	21.582	653
SRR (operation does not appears in any **rd**)	25.402	692
SRR (operation appears in an **rd**)	26.160	707

Table 1: *Overhead due to* **rd** *implementation.*

the cost of invocation generation and service takes about 18% more time (6% more cycles) if the operation does not appear in any **rd** statement. If the operation appears in an **rd**, it takes 21% more time (8% more cycles).

Basic rd Cost The execution times of the different types of **rd** increase, as one would expect, by their relative implementation complexity: unordered invariant, ordered invariant, and variant. Table 2 shows the cost of examining a single invocation using an unordered invariant **rd**.[2] The

[2]This cost represents the average cost of examining a single invocation when examining, with a single **rd**, all pending invocations, the number of which varying from one to 1000.

Mechanism	*Time(μs)*	*Cycles*
rd	9.743	325

Table 2: *Cost of an unordered invariant* **rd**.

costs of the ordered invariant and variant **rd** statements are greater, and vary according to the exact synchronization and scheduling expressions used and the actual mix of invocations, which dictate how much scanning of the examination queue is needed.

in Simulation of rd We used two test programs to compare the performance of the real **rd** with the simulation using **in** of **rd** described in Section 3.1, in which each invocation being examined is serviced once and re-invoked once. One test program, in+forward, examines invocations without preserving their arrival order; the other test program, in+forward+order, examines invocations while preserving their arrival order. Table 3 shows the execution time of the simulation to examine one invocation. Compared to the execution time (and the execution cycles) for the real **rd** (Table 2),

Mechanism	*Time(μs)*	*Cycles*
in+forward	24.644	675
in+forward+order	31.440	727

Table 3: *Results of simulation of* **rd**.

in+forward takes about 2.53 (2.08) times as long, whereas in+forward+order takes about 3.23 (2.24) times as long.

rd Simulation of in As seen in Figure 9, the **rd** statement, together with **mark** and **take**, can simulate the input statement.

For simple input statements, the **rd** simulation takes 46% more time (39% more cycles) than the input statement. About half of this additional cost is due to the SRR program using separate mechanisms (**rd**, **mark** and **take**), which results in additional calls to RTS primitives; by contrast, the integrated **in** requires relatively few such calls. The other half of the additional cost is due to the overhead in the implementation of **rd**, described above.

The results for simulations of more complicated input statements — those with synchronization expression or scheduling expressions — depend on the particular expressions and the mix of invocations. The determining factor is that the implementation of **rd** does not need to re-search the entire invocation queue on each iteration, whereas the implementation of **in** often does. The **rd** implementation avoids the search by using either the examination queue or the ILAST pointer, depending on the type of **rd**. Table 4 summarizes some representative costs for servicing invocations with positive parameters. SEQ generates all invocations with positive parameter values. ALT generates invocations

Servicing Mechanism	*Invocation Sequence*	*Time(μs)*	*Cycles*
in+st	SEQ	22.918	641
in+st	ALT	291.118	17572
rd+st+mark+take	SEQ	32.333	920
rd+st+mark+take	ALT	41.500	1144

Table 4: *Results from simulation of* **in** *with synchronization expression.*

with alternating positive and negative parameter values.

6.2 Macro-benchmarks

We rewrote several realistic applications using the new SRR mechanisms. The applications include an elevator controller simulation, and programs that incorporate some of the schedulers mentioned earlier. Each of these applications required some examination of invocations. Our results show that the overall performance of some applications (e.g., the elevator controller simulation) were improved by 1–44%. However, the new mechanisms also slowed executions of some applications down by 36–270%. These differences are due to the same factors described for the micro-benchmarks. For example, some of the applications use **rd** to simulate an **in**. The performance of the macro-benchmarks was the best when we used **in** (rather than its **rd** simulation) for actually servicing invocations and **rd** only for examining invocations.

7 Design Alternatives

7.1 General Approaches

One, lower-level approach that we considered, but rejected, employs a new type — `inv` for invocation — as well as some special primitives on this type. Variables of type `inv` can point to invocations that match their declared parameterization. Various primitives to manipulate invocations and their queues would apply to `inv` variables, such as `getinv`, `grabinv`, `qlock`, and `qunlock`. The `getinv(f)` primitive returns a pointer to the next invocation of the specified operation `f`. The `grabinv(r)` primitive removes the invocation block pointed at by `r` from the invocation queue and returns it to the calling process. The `qlock` primitive locks the invocation queue associated with the specified operation whereas the `qunlock` primitive releases the lock.

The following code, for example, prints out all pending invocations of operation `f`, as in the code in Figure 3, but without disturbing their order.

```
var r: inv f
do (r := getinv(f)) != null -> # set r to point at next invocation of f
   write( r.x ) # print out parameter x of invocation r
od
```

As another example, consider how to locate and service the invocation of `batch` with a zero `uid` (as in Section 3.2.2). An initial attempt is:

```
var r: inv batch
do (r := getinv(batch)) != null and r.uid != 0 ->
   # do nothing
od
if r != null ->
   # service invocation
   # actually remove the invocation from the batch's invocation queue.
   grabinv(r)
      # actually service it and possibly send back reply
      ...
fi
```

For the `inv` code to be blocking (i.e., equivalent to the **in** code without the **else**), a new, "delay until next invocation arrives" primitive is needed; that primitive should allow waiting for an invocation of one of several operations.

Each of the above examples should explicitly lock its invocation queue. Otherwise, race conditions with another process accessing the queue are possible. In the two examples above, a `qlock(f)` and `qunlock(f)` pair could surround each code fragment, ensuring that the invocation queue does not change during execution of the fragment. Alternatively, finer grained control, closer to **in**'s semantics, could be obtained by locking just around each `getinv` and `grabinv`.

An advantage of this approach is that it would be fairly straightforward to implement as the new language primitives would be close to current primitives in the SR implementation. However, the approach has the significant drawback that it is lower-level and more error-prone than our new approach. For example, the programmer needs to ensure appropriate mutual exclusive access to invocation queues.

An even lower-level approach is to place invocations in a list that the programmer can manipulate using existing sequential language primitives and SR semaphores for locking as needed. Although this approach would provide maximum flexibility, its low level would lead to cumbersome programs. Moreover, this kind of approach can be inefficient since the programmer would need to write code to emulate the higher level abstractions [20].

7.2 Iterative Nature of Rd

Recall from Section 4.1 that **rd** is an iterator over a collection of pending invocations.

We considered a non-iterative **rd** statement, denoted here as **ni_rd** (pronounced as "nerd"), that reads only one invocation of the operations appearing in the **ni_rd**. More specifically, it reads the oldest pending invocation that has not been examined by the executing process. An implicit pointer is set to indicate how far along the invocation queue the process has examined; it is advanced to the next invocation of the specified operation on each execution of **ni_rd**. We also considered providing a new statement, `first`, whose semantics specify re-examination of an operation's invocation queue from the beginning.

However, even with `first`, the **ni_rd** cannot easily simulate some nested **rd** statements, e.g., those that compare invocations of the same operation, which are useful in some scheduling scenarios. Also, the **ni_rd** could not contain synchronization and scheduling expressions without incurring a very high implementation cost.

7.3 Blocking Else Command of Rd

As described in Section 4.1, **rd** does not terminate after executing its else command unless it executes an **exit**. The executing process blocks immediately after executing the else command and wakes up when new invocations arrive. This blocking behavior allows the programmer to change the selection criteria of an **rd** in its else command without terminating the **rd**. Otherwise, invocations that have not been read in the previous iterations would not be re-examined again, even if they satisfy the modified selection criteria. Such dynamic selection criteria are useful in, for example, threshold scheduling (Section 3.3).

To illustrate this semantics, consider the **rd**:

```
var n := 10
rd f(x) st x > n ->
   ...
[] else ->
   n := 0
dr
```

This **rd** implements a simplified form of threshold scheduling. It first examines invocations of `f` for which parameter `x` is greater than 10. When all such invocations have been examined, the process executes the statement `n := 0` and then blocks waiting for a new invocation to arrive. When a new invocation arrives, the process wakes up and then examines invocations of `f` for which parameter `x` is positive. Although the parameter values of the already read invocations are also positive, the process now examines only those that have not been read. If instead the semantics defined **rd** to terminate immediately after executing the else command, additional operations would have to be used to separate invocations that have been examined from those that have not. This blocking behavior avoids busy waiting for a selectable invocation to arrive.

7.4 Concurrent Access to Invocations

Recall that SRR permits more than one process to service or examine invocations of an operation (Sections 2 and 4). A process executing an **in** statement has exclusive access to the queue when it is deciding which invocation to service. The process relinquishes that exclusion after it makes its decision and has removed the invocation from the queue. Thus, a process servicing an invocation does not prevent other processes from, at the same time, examining the queue and servicing other invocations. The semantics of **rd** is similar to that of **in**. Thus, the **rd** statement can be nested with any **in** or **rd**.

We also considered another approach in which **rd** obtains exclusion at its start and releases exclusion at its end. Although simpler to implement than the approach we chose, a process executing an **rd** could execute a long time or even not terminate. Furthermore, this approach would reduce the potential concurrency between, for example, a process examining invocations via **rd** and one servicing invocations via **in**. It would also cause some nested **rd**'s to deadlock.

7.5 Modifying Parameters within rd

rd allows parameters of invocations to be modified. However, those changes do not affect the values of the parameters in the invocation for when it is later examined or serviced. (Thus, a naive implementation needs to copy an invocation for use within **rd**.) We chose this semantics to prevent one process examining an invocation and modifying its parameters from interfering with another such process or with another process that is servicing the same invocation using an **in**.

Another alternative, which we are currently considering, is to reflect such parameter modifications back to the parameters in the invocation (in which case **rd** would be a misnomer). For example, consider fair share [21] or aging schedulers that periodically update processes' priorities. Requests for service would naturally be represented as invocations to an operation with a parameter indicating priority. All such invocations could be scanned and the priority parameter updated via a simple **rd** statement. There are, however, two difficulties. First, the kind of interference mentioned above needs to be avoided. Second, it is possible that the invocation has already been serviced by another process at the time the examining process finishes with the invocation. One possible approach here is to apply copy-in, copy-out semantics for **rd**'s invocation parameters, but with the copy-out having no effect if the invocation has already been serviced.

8 Applicability to Other Languages

In earlier sections, we presented our work using SR as the base language. Our work applies equally well to other languages that have invocation handling mechanisms.

Ada, Concurrent C, occam, and Orca have invocation handling mechanisms similar to at least portions of SR's. They suffer from shortcomings similar to those discussed in Section 3: None permits a straightforward way to examine messages without servicing them. For example, Ada's requeue statement could be used in the same manner as SR's forward (Figure 3), but first-in, first-out order would not be guaranteed. The Linda [5, 6] primitives include `read`, which can be used to examine a matched tuple from the TS, the tuple space (i.e., pending invocations in our terms). However, it is nondeterministic, so it cannot be used to read through all pending tuples; the same tuple could be read repeatedly.

A new `collect` primitive [22] proposed for Linda allows groups of tuples to be moved between tuple spaces. When several processes execute `collect` and compete for the same tuples, the common tuples are then partitioned between these processes nondeterministically. This primitive, however, does not make invocation handling more flexible. Tuples can be examined only by actually removing the tuples and then inserting them back to the TS, similar to the approach used in Section 3.1. The `collect` primitive can only help move a group of tuples to another tuple space without using a loop. An alternative primitive, `forall`, is also proposed but rejected in [22]. This primitive allows iteration through elements of a tuple space. The semantics of our approach seem as though they would apply nicely to Linda.

We defined **rd** to be consistent with the existing constructs in SR. A similar construct in another language should also be defined to be consistent with the rest of that language.

For example, our work can be applied to Ada. In Ada, the `select` statement services an invocation from one of several entries, which are chosen nondeterministically. Ada permits only one process (task) to access invocations for a given operation (entry), does not provide scheduling expressions, and does not permit invocation parameters to appear in its equivalent of synchronization expressions. These simpler semantics simplifies the desired **rd** semantics. One possible definition of an **rd** for Ada would have similar nondeterministic behavior and therefore would have a simple implementation. The ordered invariant **rd** (Section 5.3) is a logical candidate.

Our work also applies to message passing libraries, such as PVM [23] and MPI [24]. PVM does not provide a peeking primitive. Although MPI does provide such a primitive (MPI_IPROBE), it is not integrated with mechanisms that give expressiveness similar to SR's **in** or SRR's **rd**. Hence, this low-level approach would share some of the problems described earlier (Section 7.1).

We are also exploring how to incorporate our invocation handling mechanisms in concurrent object-oriented languages (e.g., Java and concurrent variants of C++).

9 Conclusion

This paper discussed two significant limitations in invocation handling present in SR and in other languages. First, invocations can be examined only by actually servicing the invocation. Second, when selecting which invocation to service, only one invocation at a time can be considered within a synchronization or scheduling expression. These limitations make it difficult to solve common programming situations encountered in program visualization, debugging, and scheduling scenarios. Solutions to these problems using current mechanisms often result in cumbersome and inefficient code.

We then presented the new language mechanisms — **rd**, **mark**, and **take** — that improve invocation handling and overcome these difficulties. The examples given illustrate their use and their expressiveness. Our initial implementation of these new mechanisms (SRR) shows that the mechanisms have reasonable costs, at least in those cases of their typical use. We are refining the implementation and using it to obtain further feedback on the semantics and the costs. The new mechanisms can augment or replace a language's current invocation handling mechanisms.

This research has led us to consider other approaches to the general problem of invocation handling and further consider the tradeoffs between flexibility and simplicity. One issue is whether the mechanisms are indeed flexible enough. For example, **mark** allows only one invocation, per process, to be marked. We need to determine whether that is sufficient for most applications. Another issue is whether mechanisms such as **rd**, **mark**, and **take** should be the basic invocation handling mechanisms and **in** should be defined as an abbreviation for a commonly occurring pattern of their use. Doing so might also improve the implementation costs, which might be higher than they need to be because, to save implementation time, we implemented the new mechanisms as extensions of the current SR implementation (which is naturally biased toward **in**) rather than start from scratch. The implementation work has identified various optimizations that can be applied to invocation handling and examining mechanisms (in both SRR and SR).

References

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

[2] N. Gehani. *UNIX Ada Programming.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[3] N. Gehani and W.D. Roome. *The Concurrent C Programming Language.* Silicon Press, Summit, NJ, 1989.

[4] C.A.R. Hoare. "Communicating Sequential Processes". *Communications ACM*, 21(8):666–667, August 1978.

[5] N. Carriero and D. Gelernter. "Linda in Context". *Communications of the ACM*, 32(4):444–458, April 1989.

[6] D. Gelernter. "Generative Communication in Linda". *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[7] A. Burns. *Programming in Occam.* Addison Wesley, 1988.

[8] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. "Orca: A Language for Parallel Programming of Distributed Systems". *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

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

[10] M. Chung. *Invocation Viewing and Servicing in Concurrent Programming Languages: An Extension to SR.* Master's thesis, Dept. of Computer Science, University of California, Davis, March 1996.

[11] 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.

[12] C.A. Waldspurger and W.E. Weihl. "Lottery Scheduling: Flexible Proportional-Share Resource Management". In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, Monterey, California, November 1994. USENIX.

[13] 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.

[14] R.A. Olsson and C.M. McNamee. "Inter-Entry Section: Nondeterminism and Explicit Control Mechanisms". *Computer Languages*, 17(4):269–282, 1992.

[15] T. Elrad and F. Maymir-Ducharme. "Distributed Languages Design: Constructs for Controlling Preferences". In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 176–183, St. Charles, Illinois, August 1986.

[16] T. Elrad and F. Maymir-Ducharme. "Satisfying Emergency Communication Requirements with Dynamic Preference Control". In *Proceedings of Sixth Annual National Conference on Ada Technology*, March 14-17 1988.

[17] C. M. McNamee and W. A. Crow. "Inter-Entry Selection Control Mechanisms: Implementation and Evaluation". *Computer Languages*, 22(4):259–278, 1996.

[18] A. Charlesworth. "The Multiway Rendezvous". *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, February 1987.

[19] M. Coffin and R. A. Olsson. "An SR Approach to Multiway Rendezvous". *Computer Languages*, 14(4):255–262, 1989.

[20] R. A. Olsson. "Using SR for Discrete Event Simulation: A Study in Concurrent Programming". *SOFTWARE — Practice and Experience*, 20(12):1187–1208, December 1990.

[21] J. Kay and P. Lauder. "A Fair Share Scheduler". *Communications ACM*, 31(1):44–55, January 1988.

[22] P. Butcher, A. Wood, and M. Atkins. "Global Synchronisation in Linda". *Concurrency: Practice and Experience*, 6(6):505–516, September 1994.

[23] PVM. *Parallel Virtual Machine System (PVM) Version 3 Manual Pages*, 1992.

[24] *MPI: A Message-Passing Interface Standard (Version 1.1).* Message Passing Interface Forum, June 1995. http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.