Fairness in Shared Invocation Servicing*

Ronald A. Olsson†, Gregory D. Benson‡, Tingjian Ge†, and Aaron W. Keen§

†Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.
{olsson,get}@cs.ucdavis.edu

‡Department of Computer Science
University of San Francisco
San Francisco, CA 94117 U.S.A.
benson@usfca.edu

§Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407 U.S.A.
akeen@csc.calpoly.edu

January 14, 2003

Abstract

Invocation servicing is an important aspect of many concurrent programming languages. Some invocation handling mechanisms allow for multiway servicing by multiple processes. This paper addresses fairness with respect to choosing which invocation to service and fairness with respect to choosing which process to perform the servicing. It examines how these fairness issues have been resolved in the SR concurrent programming language. This paper presents a new approach that eliminates several key restrictions. The new approach has been implemented in JR, an extended Java that includes SR-like synchronization mechanisms. This paper discusses design and implementation issues and tradeoffs.

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

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

1 Introduction

A common model of concurrency presented by concurrent programming languages and message passing libraries consists of processes (or tasks or threads) that communicate via messages. A process may invoke an operation (or entry or port), passing it parameters as needed. The invocation will be serviced by a servicing process, which will wait until it finds a suitable invocation. The invoking process may delay until its invocation is serviced and the results returned (synchronous invocation) or it may continue its execution independently (asynchronous invocation). The exact semantics depends on the specific language or library. As examples, PVM [1], MPI [2], and the GNU Pth thread package [3, 4] provide simple asynchronous message passing; Linda [5] provides asynchronous message passing, too, but via a global tuplespace; CSP [6], occam [7], Ada [8], Communicating Java Threads [9], and Limbo [10] provide rendezvous; and SR [11], JR [12], Concurrent C [13], Hermes [14], and a synchronization class library for Java [15] provide asynchronous and synchronous message passing. Without loss of generality, this paper uses SR terminology in its discussion and SR-like syntax in its examples.

Two distinguishing characteristics of invocation handling mechanisms are:

- whether a multiway receive is allowed, i.e., whether a servicing process can wait for one of several operations to be invoked; and
- whether more than one process can service a given operation.

These characteristics are summarized in Table 1. The term "arm" refers to the part of an invocation handling construct that services one operation; thus, a multiway receive is represented by the use of an invocation handling construct with multiple arms, one for each operation. The combinations of characteristics given in the table suggest the notation that this paper will use: $x\mathrm{A}y\mathrm{P}$ refers to an invocation handling statement that can have x arms and that can be serviced by y processes. The entries in the above table represent 1A1P, 1ANP, MA1P, and MANP (where M and N are integers

Table 1: Combinations of invocation handling characteristics with example mechanisms

		Number of Servicing Processes 1	*Number of Servicing Processes* N
Number of Arms	1	receive on private operation MPI	receive on shared operation Linda
	M	select/accept statement Ada	input statement SR

greater than 1), which distinguish between the interesting cases of one or multiple arms and one or multiple servicing processes.

Each entry in Table 1 represents a useful mechanism. As examples,

- 1A1P represents a single process servicing a message with a simple receive statement.
- MA1P represents a single process rendezvousing with one of several other processes with an M-way receive statement.
- 1ANP represents N replicated worker processes servicing a request for work from a shared "bag of tasks" [16]. It also represents processes using a `P` semaphore operation to synchronize (i.e., each worker process attempts to receive an empty message, which is sent by a `V` semaphore operation).
- MANP represents N replicated servers each willing to handle one of M top-level invocations (e.g., get and put requests for HTTP).

Two kinds of fairness apply to invocation handling mechanisms:

- *invocation fairness*, which deals with the order in which invocations are serviced. For example, one possible semantics is that any given invocation will eventually be serviced, if processes are willing to service the operation sufficiently often.
- *process fairness*, which deals with the order in which processes are allowed to service invocations.

> For example, one possible semantics is that any given process waiting to service an operation will eventually be able to do so, if invocations of that operation arrive sufficiently often.

It is not necessary that a language provide these kinds of fairness. For example, a language can define semantics where the choice of invocation or process is non-deterministic. However, each of these kinds of fairness is often desirable, as follows.

Invocation fairness is often desirable. For 1A1P and 1ANP, invocation fairness is simple because each contains only a single arm. The invocations will typically be queued and serviced in that order (e.g., as in Ada). However, for MA1P and MANP, invocation fairness is more interesting because the invocation needs to be chosen from queues for multiple arms. Some invocation handling mechanisms make that choice non-deterministically (e.g., as in CSP, Ada, Communicating Java Threads, and Limbo); others make that choice fairly, based on the order of arrival of the invocations (e.g., as in SR and JR); yet others allow users to specify preferences among the arms (e.g., as proposed in [17, 18, 19] and implemented in occam and in an experimental version of SR [20]). Problems with mechanisms that do not provide invocation fairness are discussed in [19, 20].

Process fairness is often desirable. For 1A1P and MA1P, process fairness is trivial because only one process can service the invocations. However, for 1ANP and MANP, process fairness is more interesting because processes compete for invocations. As mentioned above, a semaphore is considered as 1ANP, so process fairness is needed to ensure fair semaphores; stronger fairness — e.g., first-come, first-served (FCFS) — might even be desirable.

Thus, both invocation fairness and process fairness for MANP are desirable. This paper discusses the semantic and implementation issues for MANP invocation handling statements that provide such fairness. It discusses how these issues have been resolved in SR — both earlier versions and more recent versions — and why changes were necessary between the approaches taken for these versions. It then presents a new approach to these issues and discusses the tradeoffs between the previous and new approaches. The new approach has been implemented in JR, an extended Java that includes

SR-like synchronization mechanisms. It eliminates the restrictions found in the earlier approaches and simplifies the implementation. The approach is also applicable to other languages or libraries for which such sharing is desired.

The rest of this paper is organized as follows. Section 2 presents practical examples of programs using each of the combinations for xAyP as well as other relevant features; in doing so, it introduces the key features of the SR programming language. Section 3 introduces the basic idea of using equivalence classes of operations by which processes coordinate their accesses to invocations. It discusses three specific variants and discusses their semantics and implementations. Section 4 presents a performance comparison between actual implementations of two of these variants. Section 5 discusses other alternative approaches, other variants and extensions of the work, and design and implementation tradeoffs between the various approaches. Finally, Section 6 concludes the paper.

2 Motivating Examples

This section presents practical examples of programs using each of the combinations for xAyP (Table 1) as well as other relevant features. The examples are written in the SR programming language, whose key features are introduced below. Because SR supports MANP, the most general combination, it also supports each of the others.

An SR program can be split into one or more address spaces called *virtual machines*, or simply VMs. The VMs can be located on a single physical machine or on multiple physical machines connected by a network. Each VM defines an address space on one physical machine. A VM is created dynamically and contains instances of *resources*, SR's modular component. Resource instances are created dynamically. Instances of VMs are referenced via VM capabilities and instances of resources are referenced via resource capabilities.

SR provides dynamic process creation, message passing, semaphores, remote procedure calls, and rendezvous through a mechanism called an *operation*. An operation can be viewed as providing a service. The code that services an operation is located in the body of a resource, which is split into units called `processes` or `procs`. Processes are created implicitly when the enclosing resource is created. Procs are instantiated when they are explicitly invoked.

This paper focuses 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`.

Example program: 1A1P. The code in Figure 1 declares a single operation, `f`, which is used by the sending and receiving process. SR's send statement is asynchronous.

Example program: MA1P. The code in Figure 2 is a sketch of a solution to the classic readers/writers problem; the code is based on that in [11]. A single process, the `RW_allocator`, services all the requests so that it can enforce the desired exclusive access to the database. It uses a single input statement, which has four arms; syntactically, the `[]` delimiter separates the arms. Two arms of its input statement use synchronization expressions (`st` clauses); each accepts a start invocation only when its synchronization expression is true.

Example program: 1ANP. The code in Figure 3 is a sketch of the use of a "bag of tasks" [16]. A group of worker processes share the bag. 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 and might itself place additional (sub-)tasks back into the bag. Consider Figure 4, which gives a variant of the input statement in Figure 3. It uses a scheduling expression (`by` clause) to select the smallest sized task among the pending invocations. Another example of a 1ANP program is one in which processes synchronize via semaphores, for which SR provides semaphore abbreviations for operations (`sem`), input statements (`P`), and sends (`V`).

Example program: MANP. The code in Figure 5 is a sketch of an HTTP server. Here, multiple server processes are each able to handle one request on each iteration of its loop. This structure is desirable to obtain concurrency in the handling of such requests;[1] i.e., multiple `http_server` processes are active concurrently. Invocations of `get_request` and `put_request` are serviced fairly because, as noted in Section 1, SR services invocations in FCFS order based on invocation arrival time.

In addition to the above examples, other variants of input statements use operation capabilities (described below). These usages also appear in practice and factor into the designs presented in Section 3.

Example program: operation capabilities. Operation capabilities serve as pointers to operations. They may be passed as parameters, assigned to variables, and used in invocation statements. (Note that an operation name acts as a constant operation capability.) For example, the code in Figure 6 shows a main resource that creates two instances of a compute resource. The main code passes a capability for its operation `done` to each instance of the compute resource. Those resource instances invoke `done` via their invocations of their `myans` parameters. The resource instances could also be placed on different virtual machines on different physical machines by making small changes (not shown) to `main`, thus yielding a distributed program.

Example program: varied number of arms. Some of the previous examples have shown several processes using identical input statements to service the same group of input statements. For example, each `http_server` in Figure 5 services `get_request` and `put_request`. It is also sometimes useful to have input statements service only some of a group of operations. For example, the code in Figure 5 can be extended, as shown in Figure 7, with specialized servers, so as to devote more processes to handle get requests than to handle put requests. That is, some processes can, as before, service either

[1] Multiple operations can sometimes be merged into a single one with a parameter to distinguish between them (e.g., a "request" operation with an extra parameter whose value is "get" or "put" in this example). The single operation would then be serviced in a single arm input statement, so the result would be 1ANP. However, such merging is not always easy or desirable, such as when the operations have differing numbers or types of parameters. [19]

kind of request, but some (the `http_server_get_only` processes) are dedicated to handling only get requests.

Example program: varied global and local operations. The above examples each used operations declared at the resource-level, i.e., global to all processes within the resource. Operations can also be declared within a process (or a block). For example, the code in Figure 3 can be modified, as shown in Figure 8, to shut down workers when computation has finished. The program creates 10 workers and records their capabilities for their `done` operations. To shut down the workers, the code invokes each worker's `done` operation. `bag` is still serviced by multiple worker processes, but the operation `done` is local to and serviced by only an individual worker process.

As seen in the examples in this section, operations in SR can be declared at the global level (at the top level within a resource such as `bag` in Figure 8) or at the local level (within a process or block, such as `done` in Figure 8). A global operation can be serviced by only one process (e.g., `start_read` in Figure 2) or by many processes (e.g., `bag` in Figure 3). The choice is determined by how the programmer writes the code. SR also provides arrays of operations. Arrays of operations are especially useful when each member of a family of processes should service its own global operation, as illustrated in Figure 9. Alternatively, an input statement can service *any* element of an array, as illustrated in Figure 10. A local operation is (typically) serviced by only one process — the one in which it is declared.[2] Other languages sometimes have different mechanisms to associate operations with processes or to specify which processes can service which operations.

In SR, an input statement generally services invocations in first-come, first-served (FCFS) order according to their arrival times. This default order can be overridden — and thereby invocation fairness affected — by synchronization and scheduling expressions, as seen, for example, in Figures 2 and 4.

[2] A local operation can also be serviced outside the defining process via servicing an operation capability for the operation (if allowed; see Section 3.2.)

3 Equivalence Class Approaches

As noted and illustrated in Sections 1 and 2, an operation can be serviced by input statements in more than one process. These processes thus compete to service the same invocations. To identify and control conflicts in access to invocations by such competing processes, operations are grouped into *classes* based on their appearance in input statements. These equivalence classes are used to provide both invocation fairness and process fairness (defined in Section 1).

The *Same* relation over the set of operations is defined as follows.

$$Same = \{(a, b)|\ a\ and\ b\ appear\ in\ the\ same\ input\ statement\} \tag{1}$$

Same is reflexive; i.e., (a, a) is in *Same* for each operation a. *Same* is obviously symmetric. The transitive closure of *Same*, then, is an equivalence relation, the equivalence classes of which are the classes of operations.

The domain of *Same* includes operations declared at the resource level as well as those declared at the local level. These different kinds of operations can appear within the same class. Note that the only way an operation is in a class by itself is if it is serviced only by input statements that service no other operation.

To illustrate classes, consider the resource in Figure 11. Note that these input statements are abstractions of the practical input statements in the examples in Section 2. The classes formed by the operations in `z` are {`a`,`e`,`g`}, {`b`,`c`}, {`d`}, {`f`}, and {`h`}. The class {`a`,`e`,`g`} results from `a` and `e` being serviced by the same input statement, and `a` and `g` being serviced by the same input statement. That is, (`a`,`e`) and (`a`,`g`) are in *Same*, and the only other pairs in same involving any of `a`, `e`, and `g` are (`a`,`a`), (`e`,`e`), and (`g`,`g`). The transitive closure of *Same* therefore contains the class {`a`,`e`,`g`}.

The above discussion gives a general idea of how operations can be formed into equivalence classes. However, the specific realizations of equivalence classes in concrete languages can vary according to the exact definition of what constitutes an operation, any restrictions on the usage of operations, and how and when equivalence classes are computed. These are described in detail in subsequent subsections.

The classes defined above identify patterns for classes that can exist during program execution; these classes are termed *dynamic classes.* These patterns determine what operations can be in the same dynamic class. That is, the execution of the program — during which resources are instantiated or destroyed, processes are created or completed, and local blocks are entered or exited — determines which operations are extant, which in turn determines which dynamic classes are extant and their exact membership. In Figure 11, for example, the instantiation of `z` results in three classes being created — one for `a`, one for `b` and `c`, and one for `d`. When a process is created or a block is entered, each of its local operations is assigned to a class that already exists or that must be created. Continuing the previous example, each time `p1` is instantiated, the new instance of `e` is added to the class containing `a` and a new class for the instance of `f` is created; each time an instance of `p1` completes, `e` is removed from the class containing `a` and `f`'s class ceases to exist. Consequently, the operations in an instance of a dynamic class can change during execution and can contain several instances of the same named operation; e.g., the dynamic class containing `a` contains all instances of `e`. Also, the number of classes for an instance of a resource can be unbounded. For example, an instance of the class for `f` is created for each instantiation of `p1` (only six in this program).

The semantics and the corresponding implementation for guaranteeing invocation fairness and process fairness (Section 1) can then be defined in terms of these dynamic classes. Each dynamic class can be represented by a *class structure,* which contains a queue of pending invocations of operations in the dynamic class, a flag indicating whether or not some process has access to the dynamic class, and a list of processes that are waiting to access the dynamic class. At most one process at a time is allowed to access the queues that contain invocations of operations in a given class structure. That is, for a given dynamic class, at most one process at a time can be selecting an invocation to service or appending a new invocation. Access to the invocations in a dynamic class is assumed to be fair in the sense that processes are given access in a first-come, first-served (FCFS) order; moreover, processes are given access to new invocations in FCFS order. Thus, a process waiting to access the invocations will eventually obtain access — in FCFS order — as long as all functions in synchronization and scheduling expressions in input statements eventually terminate.

To illustrate, again consider the code in Figure 11. Recall that `h` is in a class by itself and that `p2` has only one process instance. That process is the only one to service `h`; when it does, it competes for access to the associated class structure with only the processes that invoke `h` (none shown in Figure 11). Recall that `d`, too, is in a class by itself, but that `d` is serviced by instances of `p1` and `p2`. Those processes compete among themselves and any invokers of `d` for access to the class structure associated with `d`. Because `a`, `e`, and `g` are in the same class, instances of `p1` and `p2` compete for access to their invocations. Thus, if `p2` is accessing the associated class (call it C) to see whether an invocation of `a` or `g` is pending, then instances of `p1` cannot access C until `p2` has finished its access to C. Process `p1` still cannot access C even if C's invocation queue contains only an invocation of `e`, which `p1` can service, but `p2` cannot service. Such "lockout" ensures process fairness. Locking occurs only during invocation selecting, not during the subsequent actual servicing of the selected invocation, as, for example, indicated in the following code segment:

```
  # t's class structure is locked while a process selects an invocation of t.
in t(x) ->
  # but t's class structure is not locked here,
  # after the process has selected an invocation.
  write(x*10)
ni
```

The rest of this section presents three variations on exactly how the equivalence classes are computed in concrete languages. Subsequent subsections present EQV1 (Section 3.1), EQV2 (Section 3.2), and EQV3 (Section 3.3). Each of these variations provides invocation fairness and process fairness, but they differ in when they compute equivalence classes and in which specific language mechanisms they can handle.

3.1 EQV1: computed statically

SR versions 1 (as described in [21]) and 2.0-2.1 (as described in [11]) provide input statements that allow (only) the names of operations within the same resource instance, as have appeared in the examples so far in this paper. Therefore, these versions use precisely the relation *Same* as given in Equation 1. The unit of compilation for an SR program is the resource. The compiler statically determines the equivalence classes for each resource. All operations in a given equivalence class belong to the same resource instance, so the corresponding class structure and queues of invocations are located within the same virtual machine as the resource instance.

3.2 EQV2: computed statically, with (restricted) operation capabilities

SR versions 2.2 and beyond (as described in Appendix G of [11] and in [22]) provide two additional features related to operations that affect the computation of equivalence classes: operation capabilities in input statements and dynamic operations. (Figure 6 illustrated the declaration and *invocation* of operation capabilities, but not servicing invocations using operation capabilities. Dynamic operations are new.)

As an example, Figure 12 shows how a bag of tasks can be serviced by workers that might reside on different virtual machines. (In contrast, the workers in Figure 3 all reside within one resource instance and hence within one VM.) Each worker services the (remote) manager's `bag` operation via an operation capability in an input statement.

The use of an operation capability in an input statement must meet at least one of the following restrictions:

R1: the input statement has just one arm, which has no synchronization or scheduling expression, or

R2: the capability references an operation in the current resource.

These restrictions ensure that an input statement requires at most a single exchange of messages with a different virtual machine. (As with EQV1 in Section 3.1, each class structure and its invocation queues are located within the same virtual machine.)

An operation can also be created dynamically via an explicit `new` expression, which returns an operation capability for the new, anonymous operation. A dynamic operation can be serviced in an input statement via its operation capability. These operations exist until they are explicitly destroyed. One example of the use of a dynamic operation is a client/server application. The client first contacts a manager process. The manager creates an operation and passes a capability for it to both the client and a worker; the client then communicates via the operation directly with the worker.

These two new features affect the computation of equivalence classes. Note, though, that because dynamic operations are serviced only via their capabilities, their effect on the computation of equivalence classes is subsumed by that of allowing any operation capability in input statements.

As for EQV1 (Section 3.1), the compiler statically determines the equivalence classes for each resource. The two restrictions essentially ensure that the use of the operation capability has no effect on the computation of equivalence classes for the current resource as follows.

- For R1 (the input statement has just one arm), only an input statement with multiple arms will cause its operations to be placed in the same equivalence class.

- For R2 (the capability references an operation in the current resource), only operations from the same resource instance will be placed in the equivalence class. (The compiler generates code to perform a run-time check on the capability's value.)

However, the mere appearance of a capability in an input statement has a significant effect: all operations in the resource that might be assigned to that capability are placed in the same equivalence class as the other operations in the input statement. For example, consider the following code

```
op a(int), b(), d(), e()
var c: cap ()
c := e
 ...
in a(x) -> ...
[] c() -> ...
ni
 ...
in b() -> ni
in d() -> ni
in e() -> ni
 ...
```

The equivalence classes formed will be {a,e}, {b}, and {d}. The first equivalence class contains a and e because e is assigned to c and a and c appear together in the first input statement. However, that class does not contain b or d because neither is assigned to any capability and hence cannot be a possible value for c. To illustrate further, suppose the above code is changed to include

```
var g: cap ()
g := d
```

Then, the equivalence classes formed will be {a,d,e} and {b}. Even though g never appears in any input statement, the analysis considers assignment of an operation to *any* capability of a given type sufficient to include that operation as a possible value of *each* capability of that type. The Appendix gives an obscure example program where a similar change to the code can have a significant effect on the program's behavior.

To summarize, the computation is conservative, with the definition of *Same* now

$$Same = \{(a, b) |\ a \textit{ and } b \textbf{ might } \textit{appear in the same input statement}\}. \tag{2}$$

3.3 EQV3: computed dynamically, with (unrestricted) operation capabilities

The design of the JR language [12], an extended Java that includes SR-like synchronization mechanisms, resulted in a new approach. One goal was to eliminate the restrictions found in SR 2.2, i.e., R1 and R2 Section 3.2. The implication of that goal is that the compiler cannot (entirely) compute equivalence classes. More specifically, without these two restrictions, an input statement may contain several operation capabilities for operations defined in other resources[3] and the exact values of the operation capabilities are not known. Since the unit of compilation is the resource, the compiler would not easily be able to analyze the program to determine their values.

The new approach is to perform the equivalence class computation as the program executes. Operations are combined into equivalence classes based on their actual use in input statements, with equivalence classes being merged when an input statement that services a member from each is executed. The computation is exact, with the definition of *Same* now

$$Same = \{(a, b) |\ a \ and \ b \ \textbf{have} \ appeared \ in \ the \ same \ input \ statement\} \quad (3)$$

(In the above definition of *Same*, "have appeared" includes the input statement currently being executed.) Thus, the equivalence classes are computed dynamically, during program execution. This computation should not be confused with the dynamic classes mentioned previously.

To illustrate EQV3, consider again the resource in Figure 11. Suppose execution begins with a `p1` process executing its first two input statements.[4] Then, the equivalence classes are {`a`,`e`} and {`b`}. If `p2` then executes its first two input statements, the equivalence classes are {`a`,`e` `g`} and {`b`,`c`}. If `p2` then executes its next input statement, the equivalence classes are {`a`,`e` `g`}, {`b`,`c`}, and {`d`}. Note how the equivalence classes changed during execution and depend on the order in which processes execute, i.e., they are nondeterministic. In practice, the values of equivalence classes will quickly reach a "steady state" and such nondeterminism is not generally important. However, one can construct obscure examples where the order of execution is significant; for example, a variant of the program in the Appendix can nondeterministically place or not place `f` and `g` in the same equivalence class, which would affect overall program execution. (For example, the variant can use an if statement that

[3]In JR, the Java class construct replaces the SR resource construct. However, this paper uses the term resource to be consistent with terminology in earlier sections and to avoid potential confusion between a class construct and an equivalence class.

[4]As written, the program makes no invocations, so all processes would just block. For purposes of the present discussion, assume the program makes sufficient invocations so that no process blocks due to lack of an invocation.

is nondeterministically executed and whose body services `f` and `g` in a single input statement.)

In EQV1 and EQV2, the operations in a given equivalence class all belong to the same resource instance, so the corresponding class structure and queues of invocations are located within the same VM as the resource instance. In EQV3, the queues of invocations are still located within the VM in which the operation is defined, but the class structure may not be. If the class contains only operations that are defined on the same VM, then it is located on that VM. Otherwise, it is located in a centralized location.[5] Remote access is required to lock and unlock the class structure and, if the operation is located remotely, to obtain each invocation. The cost of remote access, then, is incurred only by those input statements that reference operations that are used remotely or that are in a class with a remote lock.

4 Performance

To see the performance implications of the different semantics, we measured the performance of an implementation of EQV2 and an implementation of EQV3. As described in Section 3, the implementation of EQV3 will, in general, require more run-time work than the implementation of EQV2. The performance data presented in this section confirms that expectation. The various micro-benchmarks indicate that the implementation of EQV3 requires 3-61% additional time, depending on the specifics of the benchmark. However, the macro-benchmarks indicate no significant difference because other computational costs in these programs dominate the overall execution costs. The rest of this section presents the details of these performance tests.

We used two implementations of SR. The first implementation, SR_{st}, is the standard SR implementation with static classes, i.e., it implements EQV2. The second implementation, SR_{dy}, is the

[5]For multi-VM SR or JR programs, such a central manager (*srx* or *jrx*) already exists to handle, for example, program termination detection and shutdown. Using the central manager to store the class structure, though, is a potential scalability concern.

standard SR implementation but modified for dynamic classes, i.e., it implements EQV3. (We used and compared SR implementations, rather than JR implementations, because the analysis needed for EQV2 is not easily performed within the framework of the current JR implementation.)

Section 3 described the general implementation requirements for EQV2 and EQV3. The significant difference in the implementations, and their performances, is that SR_{dy} must, as it executes each input statement, merge the classes of the operations in that input statement. The major cost in doing so is to merge the invocation queues for the classes. Invocations on the resulting, merged queue must be ordered by their arrival times so as to preserve the FCFS semantics in servicing them. Thus, SR_{dy} timestamps each invocation when the invocation is received; SR_{dy} uses logical timestamps[6]. In contrast, because SR_{st} never merges invocation queues, it does not need to timestamp invocations. (An alternative, which is used in the JR implementation, is not to physically merge the invocation queues. Instead, to find the oldest invocation, the implementation can compare the timestamps of the first invocation in the invocation queue for all operations that appears in the input statement. SR_{dy} merges invocation queues since that fits in better with the code it inherited from SR_{st}, where each class has a single queue of invocations.)

SR_{dy} uses one simple, but effective, optimization to reduce costs. Normally, at the beginning of executing each input statement, the run-time system (RTS) sorts the classes for the operations in that input statement. Sorting is needed because the locks for the classes need to be obtained in some consistent order to prevent the possibility of deadlock. However, that sorting and the rest of that code can be skipped if all the classes are the same. So, the RTS simply scans the classes to determine if they are all the same and, if so, then skips over the rest of the sorting and merging code. Section 5 discusses other possible and actual optimizations made in the JR implementation of EQV3.

The data presented are from tests run using the single processor (or uniprocessor) version of SR

[6]Timestamps using real clock time are possible, but they require a system call for each invocation, which is quite costly: 20-40% additional time in the micro-benchmarks, but no additional time in the macro-benchmarks where other costs dominate.

and the multiprocessor version, known as MultiSR [23]. We will refer to these versions, respectively, as SRu and SRm. As there are SRu and SRm versions for both SR_{st} and SR_{dy}, we have four implementations: SRu_{st}, SRu_{dy}, SRm_{st}, and SRm_{dy}. Each implementation must access internal RTS data structures with appropriate exclusion. For SRu, providing appropriate exclusion is simple because the RTS is entered only via calls from the generated code. However, SRm requires code to lock key data structures in case two processes enter the RTS at about the same time. This overhead will be seen in the data presented below: in some cases, programs will take longer in the multiprocessor versions than in the single processor versions. See Reference [24] for further discussion.

The data are from tests run on a 550 MHZ Pentium III processor PC running RedHat Linux 7.3 with the 2.4.18-18.7.xsmp kernel. This system is a dual processor, but the single processor versions of SR run SR processes (threads) on only a single processor at any time, so it is equivalent to running on a single processor for those tests. Each test was run numerous times and the variances were not significant.

To make the relative performances more apparent, we use the ratio of execution times multiplied by 100%, i.e.,

$$T_{SR_{dy}}/T_{SR_{st}} * 100\%$$

For brevity, we shall refer to this ratio as dy/st.

The micro-benchmark programs are named $B_{n,i+s,w}$. where

- n is the number of operations
- $i + s$ is the number of processes that invoke and service the operations. Most of the programs use $1+1$. Some of the programs use $2+2$, so that processes compete in generating and servicing invocations.
- w is a letter that represents when the invocation queues are merged (when run with SR_{dy})

– a: after all invocations are generated; i.e., no invocation is serviced until all have been generated.

– b: before any invocations are generated; i.e., no invocation is generated until invocation queues are forced to be merged, by using an input statement with an else clause at the start of the program.

– h: after half the invocations are generated.

Table 2: Micro-benchmark results

benchmark	SRu_{st} (sec)	SRu_{dy} (sec)	dy/st ratio (%)	SRm_{st} (sec)	SRm_{dy} (sec)	dy/st ratio (%)
$B_{2,1+1,a}$	1.04	1.25	120.1	1.66	2.05	123.4
$B_{2,1+1,b}$	1.04	1.13	108.6	1.65	1.72	104.2
$B_{2,1+1,h}$	1.04	1.16	111.5	1.65	1.79	108.4
$B_{2,2+2,h}$	0.95	1.05	110.5	2.53	2.58	101.9
$B_{3,1+1,a}$	1.60	2.05	128.1	2.50	3.28	131.2
$B_{3,1+1,b}$	1.63	1.82	111.6	2.49	2.98	119.6
$B_{3,1+1,h}$	1.58	1.90	120.2	2.49	3.14	126.1
$B_{3,2+2,h}$	1.49	1.65	110.7	3.64	3.78	103.8
$B_{10,1+1,a}$	5.37	8.68	161.6	8.43	11.82	140.2
$B_{10,1+1,b}$	5.38	7.03	130.6	8.48	9.52	112.2
$B_{10,1+1,h}$	5.33	7.59	142.4	8.43	10.70	126.9
$B_{10,2+2,h}$	5.24	6.38	121.7	12.39	12.80	103.3

Table 2 presents the data for the micro-benchmark programs. As can be seen, SR_{dy} requires additional time in all cases. It requires relatively more time as the number of operations increases, since the cost of merging depends on the number of operations. SR_{st} performs nearly identically for each set of $B_{n,i+s,w}$, i.e., for fixed n and $i+s$, but varying w. That is, SR_{st} is insensitive to when invocations are generated. On the other hand, SR_{dy} is sensitive to when invocations are generated since it needs to merge invocation queues, which can be costly. As seen in the data, SR_{dy} requires varying amount of time depending on how many invocations it needs to merge, in increasing order: $B_{n,1+1,b}$, $B_{n,1+1,h}$, and $B_{n,1+1,a}$. SR_{dy} performs relatively better for $B_{n,2+2,w}$ programs because processes compete for access to the class structure and might need to block until it becomes free. That is, compared to the executions of the $B_{n,1+1,w}$ programs, executions of these programs require more work not related to the merging of classes, so the performances of SR_{st} and SR_{dy} will be closer.

Table 3: Readers/Writers macro-benchmark results

benchmark	SRu_{st} (sec)	SRu_{dy} (sec)	dy/st ratio (%)	SRm_{st} (sec)	SRm_{dy} (sec)	dy/st ratio (%)
RWbare	1.74	1.84	105.7	4.78	4.71	98.5
RWnap	2.74	2.74	100.0	2.96	2.96	100.0
RWloop	1.10	1.10	100.0	2.06	2.10	98.9
*RWloop*2	9.74	9.76	100.2	8.35	8.34	99.8

Table 3 presents the data for the macro-benchmark programs. These programs are essentially the same as the Readers/Writers program in Figure 2, but they vary in their numbers of reader processes and writer processes, how many iterations each process executes, and whether they perform anything other than just synchronization.

- *RWbare*: processes perform just the synchronization (no other work). So, this program is essentially a micro-benchmark.
- *RWnap*: processes nap (i.e., sleep) to simulate some work activity (e.g., reading the database).
- *RWloop*: processes perform computational loops to simulate some work activity.
- *RWloop*2: same as *RWloop*, but with a different mix of processes and iterations.

These Reader/Writer programs can be characterized using the same subscript notation as for the micro-benchmark programs: $RWbare_{4,8+1,b}$, $RWnap_{4,8+1,b}$, $RWloop_{4,8+1,b}$, and $RWloop2_{4,6+1,b}$. Each program is closest to having invocation queue merging behavior b: At most one invocation per reader and writer process is generated before the `RW_allocator` executes its input statement for the first time. (That is, in the spirit of behavior b, the invocation queues are merged when they are very short.)

As can be seen, the times for SR_{st} and SR_{dy} are very close in all cases. Only for *RWbare* (essentially a micro-benchmark) does SRu_{dy} perform noticeably worse (but just by 5%). For the other cases, other computational costs (looping or napping) dominate overall program execution time, so the differences are negligible. The particular mix chosen for $RWloop2$ is such that the SRm executions show some speedup over the SRu executions. In other cases, the cost of the extra RTS locking needed by the SRm implementations (mentioned earlier) negated any gain from speedup.

We also ran some other benchmark programs on a network of workstations. For example, we ran a program similar to the program in Figure 12 to assess whether SR_{dy} introduces any additional overhead when servicing operation capabilities (in the restricted sense allowed in EQV2 semantics). In all cases, the results showed negligible differences (less than 1%) between the performances of SR_{st} and SR_{dy}: network costs dominate execution times.

5 Discussion

The dynamic computation of equivalence classes used for JR, described in Section 3.3, can be used for other languages in which similar semantics are desired. As concrete examples, it can easily be used for earlier and more recent versions of SR but with the semantics changed to reflect the dynamic computation, yet keeping the same restrictions for the different versions. That is, for earlier versions, only operation names can appear in input statements; for more recent versions, operation capabilities can also appear, but limited by the two restrictions (R1 and R2 in Section 3.2). For early versions, the compiler can check that only names of operations within the same resource instance appear in input statements and can generate code for dynamic class formation (similar to JR). During execution, the run-time system performs the equivalence class computation (similar to JR). Because each class is located entirely within the VM, the entire class structure (operation queues and class information) is kept within the VM. For more recent versions, the approach is similar. The differences are that the compiler now must allow capabilities for operations to appear in input statements, must check restriction R1, and must generate code to check R2 at run time (if R1 was not met for the given input statement). As above, the entire class structure is kept local to the VM. Of course, a new version of SR with JR semantics (i.e., no restrictions on operation capabilities) can also be defined and implemented; in fact, SR_{dy} from Section 4 is exactly that.

The equivalence class approach works well at identifying potential conflicts among competing processes. Consider the problem of providing only invocation fairness for $MANP$. One approach would have a process select an invocation by picking the invocation, for any of the arms, that arrived the earliest. A process would either need to lock all invocations queues or be prepared to repeat the selection process in case another process has removed the selected invocation. The former case requires that the invocation queues be locked according to some ordering of operations so as to avoid deadlock. The latter case is discussed in the following paragraph. Another, extreme approach would be to have a single lock for all invocation queues; however, the single lock would be a bottleneck in a multiprocessor implementation. (Also, the single lock would be more costly to implement if invocations of a given operation can be serviced on multiple VMs.) However, process fairness is not guaranteed in either of the above approaches, as processes compete entirely independently for access to the invocation queues. For process fairness, processes could wait in order of their arrival at input statements for access to the invocation queues. However, that kind of ordering is best maintained by a single queue. When a new invocation arrives, each process on that queue would need to be checked, in order, to see whether it can service the new invocation; such checking can be costly and many unproductive checks might be made.

One case mentioned above is for a process to repeat the selection process in case another process has removed the invocation it selected. Such a process can be given priority in its subsequent access to the invocation queues (when new invocations arrive). For example, it could be placed on a "victim queue", which would cause other processes to defer to it. Although this approach provides process fairness (fair, but not FCFS), it is fairly complex and its actual implementation is susceptible to race conditions.

Each of the equivalence class approaches provides FCFS invocation fairness and FCFS process fairness. To obtain such fairness, the semantics include notions of queues of invocations and processes. Weaker guarantees of fairness (e.g., invocations get serviced "eventually") can be specified more simply. (As noted above for "victim queues", their implementations might be even more complex.) However, stronger semantics (like FCFS) are sometimes desired, e.g., to provide FCFS semaphores or FCFS servicing of HTTP service or file server requests.

One natural variant of EQV3, besides combining equivalence classes, would be to also split them. One scheme is to remove an input statement's operations from the equivalence class after the input statement has selected an invocation. (More precisely, remove only those operations that no other input statement is currently trying to service; an implementation can use a reference counter for such.) The class can now be split if the remaining operations in the class do not appear together in any input statements. The definition of *Same* would now be

$$\begin{aligned} Same \;=\; & \{(a,b)|\ a \textit{ and } b \textit{ appear in the same input statement} \\ & \textit{for which some process is } \textbf{currently attempting to select } \textit{an invocation}\} \end{aligned} \tag{4}$$

An alternative scheme is to split periodically based on time, number of executions of input statements, etc. As an example of class splitting, suppose one process services operation `f` and another process services operation `g`, and the two operations are in different classes in different VMs. Further, suppose that to handle high demand at some later time a third process is created that services with a single input statement both operations `f` and `g`. Thus, the two classes would be merged and located centrally, so (more costly) remote access to the class and its invocations would be required. If the third process terminates after a while (e.g., when demand subsides), it might make sense to split the classes back to two separate ones on the two VMs; thus, access to invocations of `f` and `g` would again be local to each VM (and less costly). Although splitting might be justified in such a case, the question of exactly when to split has no clear answer. If splitting is attempted as part of execution of each input statement, then the overhead is likely to be excessive; if it is attempted periodically, then the semantics would not be very clear. Moreover, the need for splitting has not yet been observed in realistic programs and splitting would complicate the implementation. So, this variant does not seem to be justified.

The discussion of EQV3 (Section 3.3) suggests that some computation of equivalence classes must occur on each execution of an input statement. However, some optimizations are possible. One optimization is aimed at input statements whose operations do not change from one execution to the next, which is often the case. Such input statements contain just names of operations or operation capabilities whose values do not change. In the JR implementation, for example, each process maintains for each input statement within the process the list of operations that appear within the input statement. On each execution of the input statement, a new list is computed and checked against the current list of operations. If the lists are the same, then the equivalence classes will not be affected by this input statement, so no recomputation of equivalence classes is needed. A refinement of this optimization (not yet implemented in JR, though) is to check at translation time whether the input statement contains only names of operations within the same resource instance (and no operation capabilities). If so, then the equivalence class computation for this input statement can be performed only once. A further optimization would be to statically analyze the program to determine the set of values that operation capabilities might take on during execution. This analysis could use techniques similar to those developed in [25, 26]. However, the techniques require the analysis of the entire program (just before the program is executed or at the start of program execution) as opposed to just one resource at a time; so this optimization is not amenable to be included in implementations (like JR's) that compile one resource at a time.

One advantage of the EQV3 approach is that it imposes no restrictions — invocations can be serviced across different machines. For example, a variant of the HTTP server in Figure 5 could be distributed across several machines, with a process on each executing the two-way input statement. As another example, a variant of the code in Figure 12 could service `bag` invocations in order of their `size`, as in Figure 4. These kinds of input statements are useful. They are allowed by EQV3, but not by EQV1 or EQV2. (They do incur extra overhead for remote access, as discussed in Section 3.3.) This example also suggests another potential optimization for the EQV3 approach. For input statements that need to examine more than one invocation (e.g., to find an invocation that satisfies a synchronization expression) or the entire invocation queue (e.g., to find the invocation with the smallest `size` in the above example), more than one invocation at a time or the entire invocation queue can be transferred from the remote machine. However, this optimization would be more complicated than the current approach, which transfers one invocation at a time.

One difference between the static approaches (EQV1 and EQV2) and the dynamic approach (EQV3) is in how they deal with arrays of operations. The static approaches place all elements of an array of operations into the same equivalence class, whereas the dynamic approach treats each element as a separate operation. (The static approach cannot, in general, exactly determine which element an input statement will actually service, so it conservatively assumes all possible elements.) For example, consider operation `f` in Figure 9. The static approaches would place all elements of `f` into one class, whereas the dynamic approach would place each element of `f` into its own class. The latter could improve concurrency since it reduces the number of processes competing for access to the class structures. On the other hand, there would be no difference for `f` in Figure 10. Each approach would create a single equivalence class that contains all `N` elements of `f`.

One disadvantage of the EQV3 approach is the cost of computing the equivalence classes at run time. In EQV1 and EQV2, that computation is performed at compile time. (Section 4 discussed the performance implications of this difference.) Part of the motivation for the EQV3 approach was to simplify translation. In the JR implementation, the JR translator is fairly simple, mapping JR features to Java. The translator knows relatively little about operations, so it was advantageous to perform the equivalence class computation at run time. This approach, though, does mean that some of the optimizations on operations performed by the SR implementation are not performed by the JR implementation. For example, SR optimizes operations that are used as semaphores (no parameters, and invoked and serviced in ways consistent with semaphores); the queue of invocations is replaced by a counter and less expensive, specialized code to invoke and service semaphores is used. A similar optimization is not performed by the JR implementation because that would require the translator to perform analysis on the use of operations, which would complicate it.

6 Conclusions

This paper has discussed invocation fairness and process fairness, especially for $MANP$. It addressed the semantic and implementation issues for $MANP$ invocation handling statements and how these issues have been resolved in various languages. It presented the general idea of using equivalence classes of operations and how that idea has been used in extant languages. This paper has also presented limitations of those earlier approaches and showed how they can be removed by using with a new, dynamic approach to equivalence class computation. The new approach has been implemented in JR, an extended Java that includes SR-like synchronization mechanisms. This paper has also discussed design and implementation tradeoffs between the various approaches. It also analyzed the performance tradeoffs between the static and dynamic approaches. That analysis showed that the costs vary for the micro-benchmarks (3-61% additional time for the dynamic approach), but that the costs are about the same for the macro-benchmarks, where other computational costs dominate the overall execution costs.

Acknowledgments

Greg Andrews and Gregg Townsend were the primary designers of the changes between SR version 2.1 and SR version 2.2. Discussions with them have influenced the present work. Comments from the anonymous reviewers led to improvements in this paper.

Appendix: An (obscure) example program with unusual behavior

The program in Figure 13 illustrates how a slight change in a program can result, perhaps unexpectedly, in a major change in its execution. The program is legal under EQV2 and EQV3, but not under EQV1 because the program uses a capability in an input statement. However, this code is somewhat obscure and is unlikely to occur in a practical program.

The salient program details, under EQV2, are as follows. (The differences for EQV3 are discussed at the end of this appendix.) The `optype` declaration defines `T` to have a single integer parameter and no return type. A capability (`c`) and two operations (`f` and `g`) are then declared using `T`. The `send` statements invoke, without waiting for the invocations to be serviced, `f` twice (because `c` was set to point to `f`) and `g` once. Procedure `foo`, when called, services an invocation ($INPUT1$) of `g` and returns a boolean indicating the result of the comparison between `foo`'s and `g`'s parameters. $INPUT2$ services an invocation of `f` for which `foo(x)` returns true. $INPUT3$ services an invocation of `f` or `c`. Finally, $CAPDECL$ declares the capability `e`. Even though `e` is unused, how it is initialized affects program behavior. Consider two variants of this program: $PROGRAM1$, which initializes `e` to `f`, and $PROGRAM2$, which initializes `e` to `g`.

For $PROGRAM1$, `f` and `g` are in different equivalence classes because they never both appear (directly or transitively) in the same input statement and because `g` is never assigned to a capability. For $PROGRAM2$, `f` and `g` are in same equivalence class because

- `f` appears in an input statement along with `c`,
- `g` is assigned to `e`, and
- `c` and `e` have the same signature (i.e., type).

In this case, the equivalence class analysis is conservative and determines

- `g` is assigned to some capability (`e`) with signature `T`,

- `c` has signature `T` and is serviced in $INPUT3$, and

- `c` *could* be assigned `e` (although that does not actually happen in the program, the analysis assumes that it might).

Hence, `g` *could* be serviced in $INPUT3$, so `g` is considered equivalent to `f`.

When $PROGRAM1$ executes, it terminates normally with output of `first f got 33`, followed by `f 55` or `c 55`. The output is nondeterministic because $INPUT3$ nondeterministically selects which arm to execute, given that each arm refers to the same operation. (Note that if `e` did not appear at all in the program, the program would behave the same as $PROGRAM1$.)

When $PROGRAM2$ executes, it terminates abnormally with no regular output. The process executing $INPUT2$ locks the invocation queues for the equivalence class for `f` and `g` and then invokes procedure `foo` for the first invocation of `f`. Execution of $INPUT1$ cannot proceed because its operation (`g`) is in an equivalence class whose invocation queue is already locked. So, this program just "hangs".

Under EQV3, $PROGRAM1$ and $PROGRAM2$ will run just as $PROGRAM1$ ran under EQV2. In EQV3, the assignment of an operation to a capability does not affect the equivalence class computation; only the actual use of the capability's value (i.e., an operation) in an input statement does.

References

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

[2] *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.

[3] Ralf S. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. *USENIX Annual Technical Conference*, pages 239–250, June 2000.

[4] R. S. Engelschall. Gnu Pth. *http://www.gnu.org/software/pth/*, 2001.

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

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

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

[8] Intermetrics, Inc., 733 Concord Ave, Cambridge, Massachusetts 02138. *The Ada 95 Annotated Reference Manual (v6.0)*, January 1995. `ftp://sw-eng.falls-church.va.us/public/AdaIC/standards/95lrm_rat`.

[9] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. Communicating Java Threads. In *WoTUG 20*, pages 48–76, 1997.

[10] S. Dorward and R. Pike. Programming in Limbo. In *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON 1997)*, pages 245–250, San Jose, CA., 1997.

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

[12] A. W. Keen, T. Ge, J. T. Maris, and R. A. Olsson. JR: Flexible distributed programming in an extended Java. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 575–584, Phoenix, Arizona, April 2001.

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

[14] R. E. Strom et al. *Hermes: A Language for Distributed Computing.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[15] S. J. Hartley. *Concurrent Programming: The Java Programming Language.* Oxford University Press, 1998.

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

[17] 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.

[18] 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.

[19] R.A. Olsson and C.M. McNamee. Inter-entry selection: Nondeterminism and explicit control mechanisms. *Computer Languages*, 17(4):269–282, 1992.

[20] C. M. McNamee and W. A. Crow. Inter-entry selection control mechanisms: Implementation and evaluation. *Computer Languages*, 22(4):259–278, 1996.

[21] 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.

[22] G. M. Townsend. *Release Notes for Version 2.2 of SR.* University of Arizona, 1993. `http://www.cs.arizona.edu/sr/`.

[23] The SR programming language, version 2.3.2, August 1999. `http://www.cs.arizona.edu/sr/`.

[24] A. W. Keen, T. Ishihara, T. Li, E. F. Fodor, and R. A. Olsson. A comparison of concurrent programming and cooperative multithreading. *Concurrency and Computation: Practice & Experience*, 2003. to appear.

[25] C. M. McNamee and R. A. Olsson. Transformations for optimizing interprocess communication and synchronization mechanisms. *International Journal of Parallel Programming*, 19(5):357–387, October 1990.

[26] C. M. McNamee and R. A. Olsson. An attribute grammar approach to compiler optimization of intra-module interprocess communication. *International Journal of Parallel Programming*, 20(3):181–202, June 1991.

```
resource msg_pass()
  op f(int)
  process sender
    ...
    send f(3)
    ...
  end
  process receiver
    ...
    in f(x) -> write("got", x)  ni
    ...
  end
end
```

Figure 1: 1A1P program: simple send and receive

```
resource RW_demo()
  op start_read(), end_read(), start_write(), end_write()
  process RW_allocator
    var nr := 0, nw := 0
    do true ->
      in start_read() st nw = 0 -> nr++
      [] end_read() -> nr--
      [] start_write() st nr = 0 and nw = 0 -> nw++
      [] end_write() -> nw--
      ni
    od
  end
  process reader(i := 1 to 10)
    ...
    start_read()
    # read the DB
    end_read()
    ...
  end
  process writer(i := 1 to 10)
    ...
    start_write()
    # write the DB
    end_write()
    ...
  end
end
```

Figure 2: MA1P program: readers/writers

```
resource bag_demo()
  op bag(...)
  process other
    ...
    # place a task into the bag
    send bag(...)
    ...
  end
  process worker(i := 1 to 10)
    ...
    do ... ->
      # take a task (invocation) out of the bag and service it
      in bag(...) -> ... ni
    od
  end
end
```

Figure 3: 1ANP program: bag of tasks

```
      in bag(size,...) by size -> ... ni
```

Figure 4: Servicing `bag` invocations in order of their `size` parameters

```
resource http_demo
  op get_request(...), put_request(...)
  process http_server(i := 1 to 10)
    do true ->
      # service either request
      in get_request(...) -> ...
      [] put_request(...) -> ...
      ni
    od
  end
end
```

Figure 5: $MANP$ program: HTTP servers

```
resource main()
  import compute_res
  var w1, w2: cap compute_res
  op done(int)
  # start up 2 compute resources
  w1 := compute_res(done)
  w2 := compute_res(done)
  # wait for both workers to complete and sum their results
  var sum := 0
  fa i := 1 to 2 ->
    in done(ans) -> sum +:= ans ni
  af
  write("sum", sum)
end

# compute_res is passed an operation capability
# to which it returns its computed result
resource compute_res(myans: cap (int))
  process do_compute
    var myresult := 0
    # do some computation, placing the result in myresult
    ...
    send myans(myresult)
  end
end
```

Figure 6: Operation capabilities program: simple distributed computation

```
  # add this process to Figure 5
  process http_server_get_only(i := 1 to 4)
    do true ->
      in get_request(...) -> ... # handle get_request
      ni
    od
  end
```

Figure 7: Varied number of arms program: extended HTTP servers

```
resource bag_demo()
  op bag(...)
  var wdone[10]: cap ()
  # create 10 worker processes
  # and record in wdone their done capabilities
  fa i := 1 to 10 ->
    wdone[i] := worker(i)
  af

  process other
    ...
    # place a task into the bag
    send bag(...)
    ...
    # all work done,
    # so shut down the workers
    fa i := 1 to 10 ->
      send wdone[i]()
    af
  end

  proc worker() returns d: cap ()
    op done()
    d := done
    # when a worker executes the reply statement, it returns a cap for done
    # as its return value to the invoker (thereby allowing the invoker to
    # continue executing) and it continues executing after the reply.
    reply
    do true ->
      in bag(...)  -> ...  # service global op.
      [] done(...) -> exit # service local  op; then, break out of loop.
      ni
    od
  end

end
```

Figure 8: Varied global and local operations program: bag of tasks with termination

```
    op f[N](int)
    process q(i := 1 to N)
      ...
      in f[i] -> ... ni
      ...
    end
```

Figure 9: Array of operations: each `q` process services its own element of `f`

```
op f[N](int)
process r
  ...
  in (i := 1 to N) f[i] -> ... ni
  ...
end
```

Figure 10: Array of operations: the input statement services any element of `f`

```
resource z
    op a()
body z()
    op b(), c(), d()
    process p1(i := 1 to 6) # creates 6 p1 processes, each with different i
        op e(); op f()
        in b() -> ni
        in a() ->  [] e() ->  ni
        in d() -> ni
        in f() -> ni
    end
    process p2
        op g(), h()
        in b() ->  [] c() ->  ni
        in a() ->  [] g() ->  ni
        in d() -> ni
        in h() -> ni
    end
end
```

Figure 11: Simple resource used to illustrate equivalence classes

```
resource manager
  op bag(...) # visible outside resource
body manager(...)
  ...
end

# worker is passed a resource capability for the manager
# when the worker is instantiated (not shown)
resource worker(man: cap manager)
  ...
  # service an invocation of the manager's bag.
  # the operation capability man.bag refers to the bag operation
  # in the man instance of manager.
  in man.bag(...) -> ... ni
  ...
end
```

Figure 12: Bag of tasks with workers distributed on several machines

```
resource main()
  optype T = (int)
  var c: cap T
  op f, g: T

  c := f
  send c(33)
  send f(55)
  send g(77)

  procedure foo(x: int) returns z: bool
    #### INPUT1
    in g(y) -> z := y>x  ni
  end

  #### INPUT2
  in f(x) st foo(x) -> write("first f got", x) ni

  #### INPUT3
  in f(x) -> write("f",x)
  [] c(x) -> write("c",x)
  ni

  #### CAPDECL
  #### even though e is unused, how it is initialized affects program behavior.
  var e: cap T := f #### initialization for PROGRAM1
              #### g #### initialization for PROGRAM2

end
```

Figure 13: Program whose behavior depends on unused `cap` declaration