Dynamic Reconfiguration of Distributed Applications

Christine R. Hofmeister

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Applications requiring concurrency or access to specialized hardware are naturally written as distributed applications, where each software component (*module*) can execute on a different machine, and modules interact via *bindings*. In order to make changes to very long-running applications or those that must be continuously available, we must *dynamically* change the application. *Dynamic reconfiguration* of a distributed application is the act of changing the configuration of the application as it executes. Examples of configuration changes are replacing a module, moving a module to another machine, and adding or removing modules from the application. The most challenging aspect of dynamic reconfiguration is that an application in execution has state information, both within the modules and within the communication channels between modules. This state information may need to be transferred from the old configuration to the new in order to reach an application state compatible with the new configuration. Thus, in addition to requiring a mechanism for changing the configuration during execution, dynamic reconfiguration requires that modules be able to divulge and install state information, and requires a mechanism for coordinating the communication during reconfiguration. Prior to this work, all systems supporting some form of dynamic reconfiguration have given the application programmer no support nor even guidelines for capturing and restoring an application's state information. We have developed a machine-independent method for installing this functionality in the application, given a set of reconfiguration points designated by the programmer. This new technique has been implemented as part of the general framework we have developed to support dynamic reconfiguration of distributed applications. These reconfiguration capabilities were implemented on top of existing operating systems and compilers, requiring no modifications to either. They support dynamic reconfiguration for applications composed of mixed languages, communicating via message passing, running on a heterogeneous distributed platform.

This research was supported by the National Science Foundation under contract NSF CCR-9021222. This paper is the author's PhD dissertation, as supervised by James M. Purtilo.

Chapter 1

Introduction

Dynamic reconfiguration of a distributed application is the act of changing the configuration of the application as it executes. Dynamic reconfiguration is needed in order to make changes to very long-running applications or those that must be continuously available. Examples of configuration changes are replacing a software component (module), moving a module to another machine, and adding or removing a module from the application.

The most challenging aspect of dynamic reconfiguration is that an application in execution has state information, both within the modules and within the communication channels between modules. This state information may need to be transferred from the old configuration to the new in order reach an application state compatible with the new configuration. Thus, in addition to requiring a mechanism for changing the configuration during execution, dynamic reconfiguration requires that modules be able to divulge and install state information, and requires a mechanism for coordinating the communication during reconfiguration.

We view a software application as being a system of interoperating processes, where each process is implemented by one module, i.e., a collection of individual data and program units. Module interfaces that are bound to one another represent communication channels between the processes. These communication channels, or bindings, together with the modules themselves, comprise the application structure. The application's geometry describes how this structure is mapped onto a heterogeneous distributed architecture. Within this distributed application framework, programmers need reliable techniques for managing three general types of changes:

Module implementations: The system's overall structure remains the same, but a user may require alteration to one of the individual modules. For example, experimenters may wish to replace some program unit with another that implements a different algorithm, in order to study the impact on performance; system administrators may wish to replace or repair device drivers without loss of service; and software engineers, responsible for enhancing a long-running program, may need to extend an application's functionality without losing persistent state within the executing program.

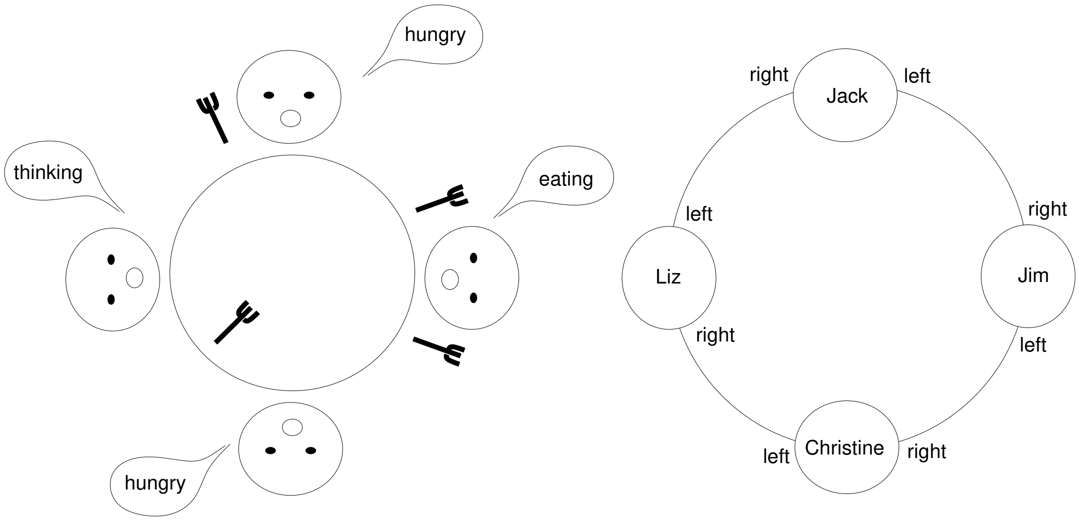


Figure 1.1: The Dining Philosopher Problem.

Structure: The system's logical structure (also called either the modular structure or the topology) may change. The bindings between module interfaces may be altered, new modules may be introduced, and other modules may be removed. Of course, structural changes may in turn require alterations to the implementation of modules, as described above. Users may introduce entirely new capabilities to an existing application.

Geometry: The logical application structure may remain fixed, but the mapping of that structure onto a distributed architecture — that is, the geometry — may change. Geometric reconfiguration is useful for load balancing, software fault tolerance, adaptation to changes in available communication resources, and relocation of processes in order for them to access guarded resources.

To illustrate these requirements, we shall introduce an example, a distributed version of the well-known dining philosophers problem. The dining philosophers problem is a resource allocation problem in which mutual exclusion must be preserved and resources must be allocated fairly. The resources in this case are forks, each of which is shared between a pair of philosophers. The group of dining philosophers is seated around a circular table with a single fork between each pair (Figure 1.1, left). Each diner thinks for a while, then gets hungry and tries to eat. In order to eat, a diner must have exclusive use of its two adjacent forks, so no neighboring philosophers can eat at the same time. After eating, the diner returns to thinking, thus beginning the cycle again.

Our implementation of this problem uses the decentralized algorithm developed by Chandy and Misra [10]. The details of this algorithm are not critical to our purpose here, so we show only the pseudo-code for a diner in Figure 1.2. Our original example has four diners, each a separate process, passing forks and requests for forks on bindings between pairs of diners (Figure 1.1,

```
initialize diner state to HUNGRY;
initialize left fork state;
initialize right fork state;

main() {
   if (status is special) set initial values so that graph is acyclic;

   while (1) {

      update left fork state;

      update right fork state;

      if (HUNGRY and conditions are right) start EATING;
      else if (done EATING) start THINKING;
      else if (done THINKING) become HUNGRY;
   }
}
```

Figure 1.2: Pseudo-code for `diner.c`.

right). Because the algorithm is decentralized, the protocol for sharing forks is contained in each diner, and is based entirely on a diner's local state.

We illustrate this problem on an existing distributed programming system, POLYLITH [27]. In order to run this example on a heterogeneous network using POLYLITH, the user provides an application specification, a simple description of the application's modular structure. The application specification describes the attributes of each module, including its interfaces, and defines the bindings between them. Given an application specification and implementations for the modules, POLYLITH is responsible for packaging and invoking processes, and for coercing data representation, synchronization, and marshalling of data during communication. Figure 1.3 shows the application specification for the diner example.

We can now describe each of the possible forms of reconfiguration in terms of this example:

Module implementations: An example of individual module reconfiguration is to replace one of the diners with a verbose diner, one that displays detailed information about its activities. Whereas the original diner says only whether it is eating, thinking, or hungry, the verbose diner also provides information about the forks and requests for forks. In order to perform this replacement without losing the fair allocation and mutual exclusion properties of the application, the old diner's state information must be used to initialize the verbose diner.

Structure: One way to change the structure of this application is to add a new diner. Again, in order to preserve the mutual exclusion properties, the new diner must be initialized with

```
service "diner" : {
    implementation : {  binary : "/world/Users/crh/diner.out" }
    algebra : { "STATUS=($S)" }
    client "left" : { string } accepts { string }
    function "right" : { string } returns { string }
}

orchestrate "diners" : {
    tool "Jim" : "diner $S=special"
    tool "Christine" : "diner $S=regular"
    tool "Liz" : "diner $S=regular"
    tool "Jack" : "diner $S=regular"
    bind "Jim left" "Christine right"
    bind "Christine left" "Liz right"
    bind "Liz left" "Jack right"
    bind "Jack left" "Jim right"
}
```

Figure 1.3: Application Specification for Dining Philosophers.

appropriate state information. But in this case the new diner's initial state is based on the state of its two future neighbors.

Geometry: An example of geometric reconfiguration is to move a diner from its original host to another host. If both hosts are of like architecture and operating system, then the migration is a straightforward engineering operation. However, heterogeneity defeats existing migration techniques. To deal with this problem, we use the same technique as for changing a module implementation: we capture the diner's state before removing it, then use that state information to initialize a new version created on the target machine.

There are a large number of activities that must be coordinated before a user can begin to capture and manipulate the state of a running process. Any environment to support general dynamic program reconfiguration in the presence of heterogeneity must meet the following requirements:

1. **Communication across heterogeneous hosts:** Especially because of the presence of heterogeneity of architectures and languages, programmers need a reliable way to coerce the representation of data that is transmitted during both normal communication and any reconfiguration.

2. **Current configuration is accessible:** Users must have a notation for identifying the program components or attributes that they wish to reconfigure, and they must be able to visualize the current state and geometry of a running program. Because the configuration is no longer static, users cannot reliably reconfigure a program without understanding what processes are currently being employed and where they are running.

3. **Bindings are not compiled into modules:** In order to isolate the locality of dynamic changes, modules should not explicitly name the other modules with which they communicate. To do so would require that modules be replaced simply because their bindings had changed.

4. **No covert communication among modules:** The execution environment must ensure that all communication between processes can be controlled by the external agent responsible for reconfiguration. If processes are allowed to communicate by a private channel, then a subsequent reconfiguration involving one of the processes may fail to update all dependencies — as a result, a module may find itself trying to access a non-existent resource.

5. **Ability to add and remove modules and bindings:** The execution environment must provide a mechanism for manipulating the application components dynamically. The basic operations are to add and remove modules and bindings; other operations, such as replacement, can be built from these basic operations.

6. **Access to messages in transit:** When changes are made dynamically, there may be messages in transit across bindings. These must either be accessible to the reconfigurer, or there must be a mechanism for ensuring that no messages are in transit during reconfiguration.

7. **Mechanism for synchronizing activities:** When dynamic reconfiguration activities involve more than one module, changes may need to be synchronized so that application properties (such as freedom from deadlock) are preserved.

8. **Access to module's state information:** The dynamic reconfiguration mechanism must ensure that all relevant information characterizing a process is captured and represented. This can include state information that is cached on behalf of the process in the underlying operating system. The primary example of this type of information is the table of open file descriptors that the operating system maintains for each process.

Our approach to meeting the above requirements is to build upon the existing POLYLITH software interconnection system [27]. POLYLITH provides users with an environment for easily constructing large (and possibly distributed) applications for use in heterogeneous execution environments. For these reasons, POLYLITH is a natural platform for reconfigurable applications.

The POLYLITH bus organization satisfies our requirements concerning coercion of data's representation in a heterogeneous system. The bus manages data transformation during normal communication, and this same coercion mechanism serves to relocate the process state on other hosts.

The design philosophy of POLYLITH separates the configuration of the application (as in Figure 1.3) from the implementation of the modules (Figure 1.2). This configuration information directs the application start-up, and is accessible at run time for directing dynamic changes. In addition, bindings are given as part of the configuration information, and are hidden from the particular module implementations. Thus dynamic binding changes are done at the configuration level, not within the modules.

The bus abstraction also helps assure programmers that processes do not communicate by private channels. All modules built using the POLYLITH system will only communicate via the bus. The bus protocol notifies each process of its symbolic name, but never passes it an 'absolute' name for other modules. Since, by design, no application component communicates directly with other modules, these components cannot be affected by reconfiguration of other modules. Once a new version of a module has been created, the bus directs subsequent communication to the new version, abandoning the old version.

Thus the first four requirements for a reconfiguration environment are met by the POLYLITH system. The remaining four items were not originally supported by POLYLITH. Our approach was to supplement the original POLYLITH system by adding reconfiguration primitives to support items 5. (add/remove modules and bindings), 6. (access messages in transit), and 7. (synchronize reconfiguration activities). These new facilities are described in Chapter 2.

The last two items in the list of requirements are the most challenging. First consider a geometric change: this case amounts to replacing an executing module with the same module, perhaps compiled for a different architecture. The process state of the module must be captured in an abstract format, transferred to the other machine, then restored in the new module. A module implementation change is also a form of replacement, but in this case the process state must be transformed into a state appropriate for the new implementation of the module.

The main contributions of this work are in providing support for module participation during dynamic reconfiguration. We have defined an approach to capturing and restoring persistent state that is supported by the reconfiguration primitives, and discovered a machine-independent method for capturing and restoring the activation record stack. Chapter 2 covers module replacement when the process state can be ignored, and Chapter 3 describes how modules can participate during reconfiguration by capturing and restoring their process states.

With replacement, synchronization is not a significant problem because only one module is involved. Structural changes may also require capturing and restoring the process state, but these changes may in addition require synchronizing the reconfiguration activities among several modules. Chapter 4 explores different approaches to synchronizing dynamic reconfiguration. The final chapter summarizes this work, and the appendix contains a detailed description of the new reconfiguration primitives described in Chapter 2.

Chapter 2

Application-level Replacement Activities

In this chapter, we define the reconfiguration primitives needed to perform general dynamic reconfiguration. These reconfiguration primitives operate at the application level: they provide all services that take place outside the modules. We show how these are used for replacement in a simple Producer/Consumer example, where the module state does not need to be captured or restored. Finally, we describe a reconfiguration CATALYST, a special module that can be installed in any application to provide dynamic replacement of other modules. The CATALYST is built from the reconfiguration primitives.

2.1 Dynamic Reconfiguration Primitives

The reconfiguration primitives are extensions to POLYLITH system; they provide application-level support for reconfiguration tasks by allowing users to suspend communication between modules during reconfiguration, alter the configuration of the application, and transfer state information from one module to another. A particular dynamic reconfiguration activity, such as replacement, consists of a sequence of these primitives, and can be initiated by any module of the application, or by a third party. Figures 2.1, 2.2, and 2.3 show the three groups of reconfiguration primitives, which use the same approach to applying changes: first get a capability for applying the change (`mh_obj_cap`, for example), next make a series of edits to describe the change (`mh_edit_...`), then apply the change atomically (`mh_chg_obj`).

First we examine the primitives that change the application configuration (Figures 2.1 and 2.2). Since an application is composed of modules and bindings, it is natural to describe reconfiguration in terms of changes to these modules and bindings: specifically, adding and deleting modules and bindings.

In addition to operations for making configuration changes, the reconfiguration framework must

`mh_obj_cap`	
`(&m,n)`	Get capability to module `n`
`(&m,NULL)`	Get capability to a new module
`mh_edit_if`	
`(&m,"add",if)`	Add specified interface to `&m`
`(&m,"del",if)`	Remove specified interface from `&m`
`mh_edit_objattr`	
`(&m,"add",attr,val)`	Add or replace attribute value for `&m`
`(&m,"del",attr,NULL)`	Remove specified attribute from `&m`
`mh_edit_ifattr`	
`(&m,"add",if,attr,val)`	Add or replace attribute value for interface `if` of `&m`
`(&m,"del",if,attr,NULL)`	Remove attribute from interface `if` of `&m`
`mh_chg_obj`	
`(&m,"add")`	Add module `&m`
`(&m,"del")`	Remove module `&m`

Figure 2.1: Polylith Reconfiguration Primitives for Altering Modules

`mh_bind_cap (&b,appl)`	Get capability for altering bindings in `appl`
`mh_edit_bind`	
`(&b,"add",n1,if1,n2,if2)`	Add binding between interfaces `n1 if1, n2 if2`
`(&b,"del",n1,if1,n2,if2)`	Delete binding between `n1 if1, n2 if2`
`(&b,"cpo",n1,if1,n2,if2)`	Copy messages queued for `n1 if1, n2 if2`
`(&b,"rmq",n1,if1,NULL,NULL)`	Remove messages queued for `n1 if1`
`mh_rebind (&b)`	Apply all binding changes specified in `&b`

Figure 2.2: Polylith Reconfiguration Primitives for Altering Bindings

`mh_hold_cap (&h,appl)`	Get capability for holding objects
`mh_edit_hold`	
`(&h,NULL,n,if)`	Hold interface `if` of module `n`
`(&h,"obj",n,NULL)`	Hold module `n`
`mh_hold (&h)`	Apply all holds specified in `&h`
`mh_rlse (&h)`	Release all holds specified in `&h`
`mh_objstate_move`	Induce `&m1` to divulge its state via `if1`
`(&m1,if1,&m2,if2)`	and forward it to `&m2 if2`

Figure 2.3: Primitives for Synchronizing Reconfiguration

provide operations for synchronizing reconfiguration activities. If a binding is deleted while a message is in transit, the message may be only partially transmitted. Another potential problem arises with multiple binding changes: if a module has multiple interfaces, these must be rebound simultaneously so that messages on these interfaces are consistent with each other.

The group of primitives in Figure 2.3 provides synchronization for reconfiguration by holding interfaces or modules at the application level. When a hold is applied to an interface, the module attempting communication over that interface is blocked. Similarly, a held module is blocked upon attempting any POLYLITH bus service. One of the parameters of `mh_edit_hold` indicates whether unread messages will be moved to another interface.

The synchronization operations allow us to hold a group of interfaces, perform the reconfiguration, then release the same group. Holding a group of interfaces "freezes" portions of the application in a consistent state, allowing modules to continue execution, but preventing communication with other modules along those held interfaces.

Purely structural changes (adding or deleting modules, and changing bindings) can be done without any support from within the modules' implementations. But many reconfiguration changes involve changes at the module level, either to replace the implementation of the module, or to move the module to another host. These module-level changes require the module's participation in capturing its process state. In order to coordinate the module-level activities, the reconfiguration framework includes an operation that signals a module to divulge state information on a particular interface, then moves that state information to an interface of another module. This `mh_objstate_move` operation will be described in Chapter 3, when we describe how to capture and restore module state.

2.2 The Producer/Consumer Example

In this section, we introduce a simple Producer/Consumer application in order to demonstrate the

```
producer()
    s = next input string
    While (s is not the empty string) do
        send s on the "produce" interface
        s = next input string
    end while
end producer


consumer()
    while (true) do
        receive message s from the "consume" interface
        write s to output
    end while
end consumer
```

Figure 2.4: Modules for the Producer/Consumer Problem

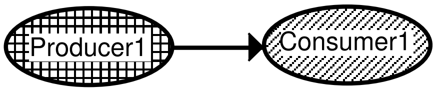


Figure 2.5: Initial Configuration for the Producer/Consumer Application

reconfiguration primitives. This application is an example of replacement where the module does not have to be captured and restored. In the application's initial configuration, a producer module accepts a character string from standard input and writes the string to its outgoing interface, repeating this until an empty string is entered to terminate the application. A consumer module reads a character string from its incoming interface and writes it to standard output, and repeats this until the producer terminates. Pseudo-code for these two modules is shown in Figure 2.4. Initially, the outgoing interface of the producer is bound to the incoming interface of the consumer. (The graphic representation of our initial configuration is shown in Figure 2.5.) POLYLITH buffers messages on the binding between these two interfaces; thus the consumer has no internal buffer, and the producer does not synchronize with the consumer.

The application specification for the Producer/Consumer example is shown in Figure 2.6. It defines an application consisting of two modules, a `producer` and a `consumer`, bound together on their corresponding interfaces.

The producer or consumer module can be replaced with a module on another machine or a module that uses a different binary. The structure of the application does not change during this reconfiguration, but a new module must be created and bound into the application, then the old module must be deleted. Figure 2.7 shows how the reconfiguration primitives are used to replace the consumer module of the Producer/Consumer application.

```
service "producer" : {
    implementation : { binary : "./producer.out" }
    source "produce" : { string }
}

service "consumer" : {
    implementation : { binary : "./consumer.out" }
    sink "consume" : { string }
}

orchestrate "application" : {
    tool "producer1" : "producer"
    tool "consumer1" : "consumer"
    bind "producer produce" "consumer1 consume"
}
```

Figure 2.6: Application Specification for the Producer/Consumer Problem

```
mh_obj_cap (&old,"consumer1");
mh_obj_cap (&new,"consumer1");
mh_edit_objattr (&new,"add","BINARY","./consumer2.out");

mh_hold_cap (&h,"");
mh_edit_hold (&h,NULL,"producer1","produce");
mh_edit_hold (&h,NULL,"consumer1","consume");
mh_hold (&h);

mh_bind_cap (&b,"");
mh_edit_bind (&b,"del","producer1","produce",&old,"consume");
mh_edit_bind (&b,"add","producer1","produce",&new,"consume");
mh_edit_bind (&b,"cpo",&old,"consume",&new,"consume");
mh_rebind (&b);

mh_chg_obj (&new,"add");
mh_chg_obj (&old,"del");
mh_rlse (&h);
```

Figure 2.7: Reconfiguration Events for Replacing the Consumer

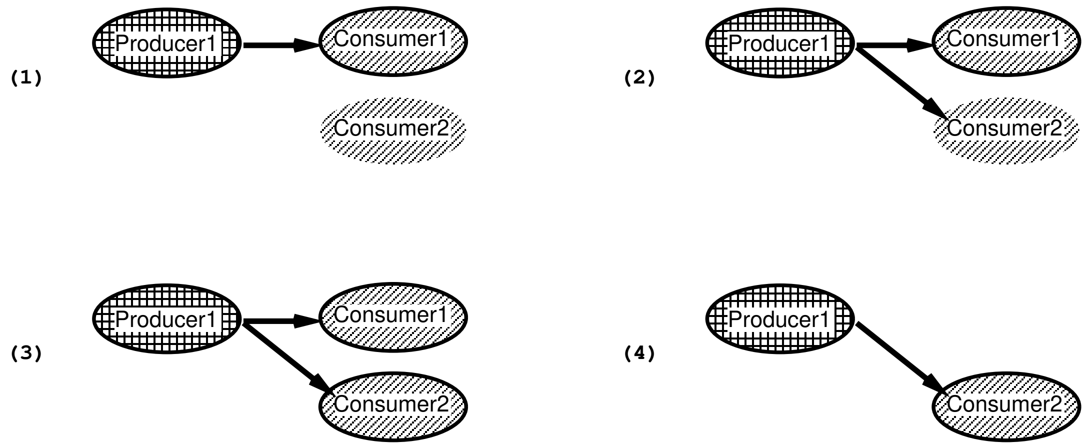


Figure 2.8: Replacing the Consumer

Our basic rebind operation allows us to move messages buffered on a binding to another binding, and to rebind a group of bindings atomically. We need both these features for this reconfiguration: we must move any messages on the old binding to the new binding before deleting the old; and we must delete the old binding and add the new at the same time, to avoid a temporary state of having two bindings between producer and consumer or having none. Because reading from and writing to an interface are also atomic operations, the rebind occurs between sending or receiving messages; rebinding will never happen while a message is being sent or received.

The timing of a replacement reconfiguration is critical because a module is being deleted from the configuration. In fact, we cannot delete a module at an arbitrary time and guarantee that no data will be lost during the reconfiguration. If we want to be able to interrupt the module and do the reconfiguration immediately instead of waiting for the module to reach a desirable program state, we must relax our correctness requirements. Inconsistencies are allowed during reconfiguration, as long as consistency returns when the reconfiguration is complete.

One inconsistency that can appear in this application is that reconfiguration can occur after the consumer has received its input string but before it has written that string to standard output. The new version of the consumer is created and bound to the producer, and the old version is deleted along with the string that it had not yet displayed. We could wait for a while before deleting the original consumer, giving it time to write the string, but how long do we wait? No length of time will be long enough for all cases, except an infinite wait. An infinite wait is not acceptable; it is not practical to keep all former versions of a module running throughout the life of an application.

Although there is a chance that messages will be lost if a module is interrupted for replacement, once the old version of the module has been deleted, the replacement is complete, and the application again works correctly.

2.3 The Catalyst

Writing the detailed sequence of steps needed to replace a module is a tedious task, prone to minor errors when done manually. This task is automated with the CATALYST module, which examines the current configuration of the application in order to determine how to bind the new module to existing ones. The CATALYST can replace more than one module at a time; in the Producer/Consumer example, both the producer and the consumer can be replaced at one time. Thus a new module may be bound to an existing module or to a new module, depending on which modules are being replaced. To decide where a new module should be bound, the CATALYST examines the original bindings: if the module it was originally bound to is also being replaced, the CATALYST binds the new module to its corresponding new module. Otherwise, if there is no change to the module to which it was originally bound, the CATALYST binds the new module to the original.

The CATALYST is built from the basic reconfiguration operations discussed in Section 2.1. It is part of the resulting software application but remains inactive until a reconfiguration is requested. At such a time, the CATALYST performs the requested reconfiguration, then reverts to its inactive state until another reconfiguration is requested (Figure 2.9).

Using a CATALYST, the reconfiguration described in the previous section is done by entering "`replace consumer1`." The CATALYST asks for a binary and site for the new module, creates a new version of the module (at the new site if one is specified), moves all bindings from the old version to the new, deletes the old version, and starts the new version. The site specifies the machine where the module will execute; if no site is specified, the new module will run on the same machine as the original consumer module.

In this chapter, the CATALYST was used for replacement without module participation, but we will show in Chapter 3 that it works equally well when module participation is required. The characteristics of the Producer/Consumer application suggest general characteristics of modules that are reconfigurable without any participation from the application modules:

- The module's program state, including both data structures and program counter, can be safely discarded during reconfiguration. As an example, in an application where the module continuously provides updated information and an occasional lost message can be tolerated, that module can be replaced without its participation.
- The module requires no special initialization when it is created dynamically. Modules that must restore data passed from the old version do not qualify, nor do modules whose state depends on other modules. In an application where a module shares a token with others, we can discard the program state of one of these modules during replacement, but the new module must be initialized using state information from the others in order to reach a consistent application state after reconfiguration.

$\mathrm{M} \equiv \{\ \mathrm{m} \mid \mathrm{m}$ is a reconfigurable module $\}$
$\mathrm{m}_i \equiv$ interface i of module m
$\mathrm{R} \equiv$ the set of modules replaced, where $\mathrm{R} \subseteq \mathrm{M}$
$\mathrm{R}' \equiv$ the set of modules added

Each time through the loop, M is replaced by $(\mathrm{M} - \mathrm{R}) \cup \mathrm{R}'$

```
catalyst()
    Loop infinitely
        get R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        for each m ∈ R
(1)         get name, binary, site for m′
            create m′
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        for each m′ ∈ R′
            for each interface m′_i
(2)             find the r_j that m_i is bound to
                if r ∈ R
                    then bind m′_i to r′_j
                    else bind m′_i to r_j
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        for each m′ ∈ R′
(3)         start up m′
            remove m
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    end loop
end catalyst
```

Figure 2.9: CATALYST: the Reconfiguration Module

- There is no synchronization between the reconfigurable module and its neighbors. For example, modules that use a procedure call/return protocol do not fit this characteristic. If the client module is waiting for a return message from the server module when the server is interrupted for replacement, the new server will not give the expected return message and the client will deadlock.

While most applications will not be as simple as the Producer/Consumer example, many applications will contain individual modules that can be replaced without module participation. One general type of producer is a module that serves as an interface for gathering real-time data. A module that continuously sends out temperature, pressure, or load data can usually be replaced without module participation.

Chapter 3

Module Participation during Replacement

Dynamic reconfiguration may require a module's participation in one of the following ways:

- A module may need to **divulge its program state**, for initializing another module after reconfiguration. The program state can include things such as data structures, program counter, and other run-time information.
- A module may need to **perform special initialization** when it is created dynamically. During replacement, the new module may need to be initialized so that it has the same program state as the original module. A newly created module may need to initialize its state based on the current state of other modules in the application.
- Instead of allowing reconfiguration at any time, a module may need to **delay the reconfiguration** until it reaches a program state where application consistency can be maintained; this state is the reconfiguration point.

Current research recognizes that a general solution to dynamic reconfiguration requires module participation. The dynamic reconfiguration environment provided by Conic [22] [23] supports the application-level reconfiguration activities of adding or deleting modules and the bindings between them, but requires the programmer to manually adapt a module to participate during reconfiguration. The reconfiguration framework of Conic is separated into configuration-level concerns and application-level concerns. Configuration-level activities are independent of the algorithms, protocols, and states of the application, and are guaranteed to leave the system in a consistent state (where consistency is defined in terms of the application). To ensure consistency after reconfiguration, the modules in the application must be programmed to respond to the configuration level commands `unlink`, `link`, and `passivate`, corresponding to the three types of module participation listed above.

There are other distributed environments that support limited reconfiguration, namely replacement, and these also require the programmer to provide module participation. Durra [5] [6] and Argus [7] have facilities for incorporating fault tolerance into an application, and they use these facilities to help support dynamic reconfiguration. But even with these facilities, module participation must be written by the programmer in order to perform reconfiguration. In Durra, module participation is required when reconfiguration is triggered by a `raise_signal` call. In Argus, there are two different methods of replacement. One method uses the same approach we take: `encode` and `decode` operations are used to support module participation. The other approach requires the user to transfer state from an old guardian (module) to a new one.

Podus, the updating environment described in [15], requires that module participation be provided by the programmer when program state must be transferred from an old version to a new one. The other aspect of module participation, that of delaying the reconfiguration until a suitable time, is handled outside the module, by invoking replacement only when the module is inactive. This approach is feasible only because interactions among modules are restricted to procedure-call semantics and calling cycles are disallowed.

3.1 Abstract Process State

To support general dynamic reconfiguration activities, the characterization of the process state must be in an abstract format. On a multiprogrammed computer, programs are continually swapped into and out of main memory, and with each swap a process state must be saved and another restored. This capture/restoration of process state is machine-specific, but it serves as a model for an abstract characterization of the process state. Assuming a static-scoped language and single-threaded modules, the items comprising a process state are:

- program counter
- static data– in data area
- dynamic data– in activation record (AR) stack
- temporary values– in registers or AR stack
- procedure call/return information– in AR stack
  - return address
  - control link to restore context upon return
  - register values to restore upon return
- user-allocated data– in heap
- file descriptors, process status information– accessible only to kernel

The temporary or intermediate values used when a statement in a high-level language is compiled into multiple machine language statements may not be compatible with the values used when compiling into a different machine language. Thus the abstract program state must be captured between high-level statements. In addition, optimizing compilers sometimes cache variables in registers to reduce the number of writes to memory. Our approach is to let the compiler handle referencing these variables correctly.

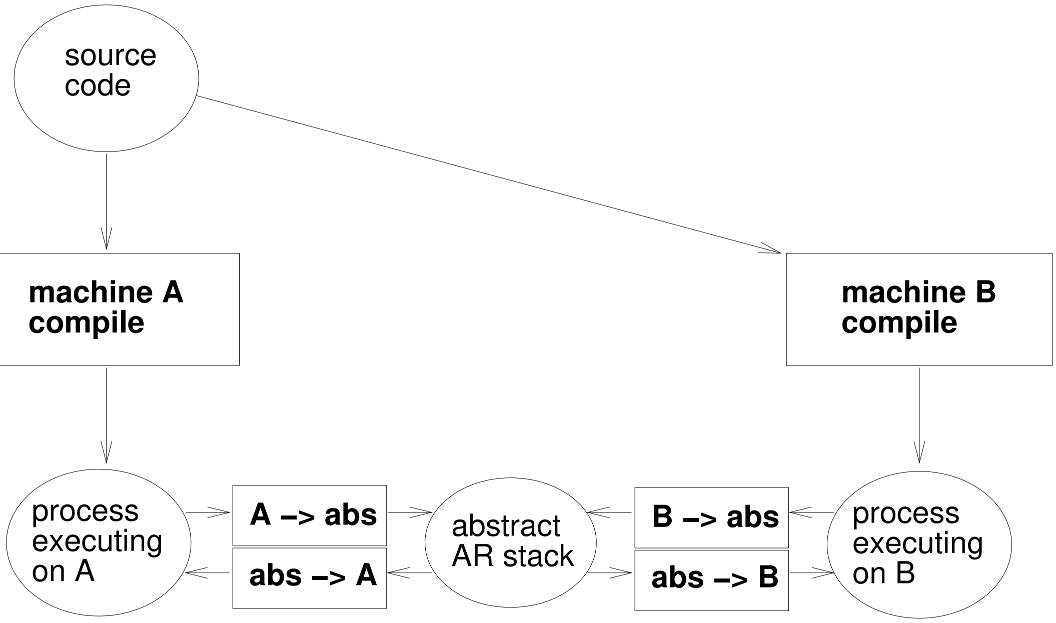


Figure 3.1: Obvious Approach to Capturing and Restoring the Abstract Process State

Because the process status information is machine-specific, we do not attempt to restore this information in the new process. File descriptors are an essential part of the process state, but this information is usually accessible only to the kernel of the operating system, so we do not automatically capture them at this time. The data stored in the heap is dynamically allocated by the programmer. At the present time, the programmer must write code to capture and restore heap data structures and to regain access to files.

When a process executes a procedure call, the process state makes a logical "context switch" by pushing a new frame onto the activation record stack. This new activation record contains the procedure parameters, variables local to the procedure, and various pointers for accessing non-local variables and for restoring the old context after the procedure completes. Upon returning from the procedure, the topmost frame is popped from the activation record stack. Although all static-scoped languages follow this approach, the exact format of the activation records depends on the compiler and the architecture of the machine.

The obvious approach to capturing and restoring state is to write machine-specific programs that translate the run-time stack and data areas into an abstract format, then back to another machine-specific format (Figure 3.1). Our solution adds a layer of abstraction: these translation programs are written in a high-level language, namely the same language used to write the module. Thus all machine-specific details are generated by the standard compilers provided with the machine.

The original program is augmented with source-level statements that can capture and restore the data area and any activation record stack that reaches a reconfiguration point. This transformed

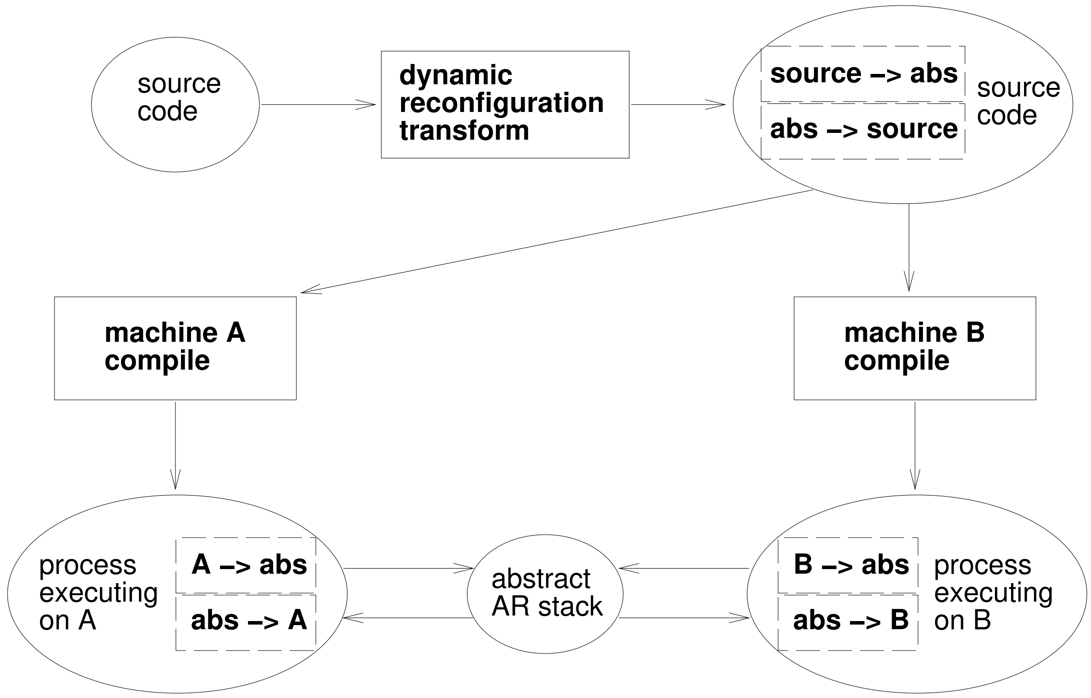


Figure 3.2: Our Approach to Capturing and Restoring the Abstract Process State

source program is compiled in the normal way, resulting in a program that translates a machine-specific process state into an abstract format and vice versa (Figure 3.2).

3.2 Global Data and the Program Counter

Our first step in addressing the difficult problem of capturing and restoring the abstract process state is to examine the global data and the program counter. In this section, we describe how to capture and restore the following highlighted items from the list of items constituting the process state:

- **program counter**
- **static data– in data area**
- ◦ dynamic data– in activation record (AR) stack
- ◦ temporary values– in registers or AR stack
- ◦ procedure call/return information– in AR stack
  - ◦ return address
  - ◦ control link to restore context upon return
  - ◦ register values to restore upon return
- ◦ user-allocated data– in heap
- ◦ file descriptors, process status information– accessible only to kernel

The approach is described in Section 3.2.1, then in Section 3.2.2 it is applied to the Dining Philosopher Example introduced earlier. With this example, we show how the CATALYST module is used for replacement with module participation.

3.2.1 Transmitting ADTs

As a first step for capturing and restoring the module state, we use the approach for transmitting abstract data types (ADTs) presented in [16]. In this work, Herlihy and Liskov extend the ADT concept to include the new operations **encode** and **decode**, which map a particular implementation of the ADT into an external (canonical) representation and vice versa.

In this section, we show how this approach is useful for capturing and restoring global data in the module state, demonstrating this on a module that implements a stack data type. Figure 3.3 shows the basic operations of the stack, **push, pop, create, full**, plus the new **encode** and **decode** operations. In the Herlihy/Liskov scheme, the **encode** and **decode** operations are invoked in the same way as all other operations; this interface is shown in left half of Figure 3.4.

For reconfiguration, instead of simply transmitting the ADT, we must transmit the state of the entire module, which includes the state of the program counter. The reasonable place to capture the state of the stack module is between operations, when it is not servicing a request. So we place the reconfiguration point at the end of the `while` loop (Figure 3.4(right side)), invoking the **encode** operation at this point. The invocation of the **encode/decode** is not statically determined, as it is in the case of ADT transmission, but is determined dynamically. Even though the implementation we have given is not asynchronous, the encode message can arrive at any time. The stack module checks for an encode message at each reconfiguration point; this is equivalent to allowing the encode messages to arrive asynchronously, with the stack module immediately moving to the reconfiguration point by finishing its current operation.

For restoring the module state, the module must **decode** the ADT, then resume execution at the reconfiguration point. In order to perform this special initialization, the module checks whether it has a special status value of `clone`, and if so, proceeds with the special initialization. This status attribute is a convenient mechanism for detecting modules that need special initialization, and it is our convention throughout this work. Other mechanisms, such as a command line argument, could also be used.

The general format for special initialization is to check the status attribute, and if it has a value of `clone`, to restore the data (**decode**), then jump to the reconfiguration point. In this module, resuming execution at the beginning of the program is equivalent to resuming at the reconfiguration point. Thus we have implicitly captured and restored the program counter by controlling where the state is captured and where execution resumes upon restoration.

With these reconfiguration capabilities installed, this stack module, which implements the stack with a linked list, could be replaced by a stack module that uses an array implementation.

```
/* internal representation */
struct inode {
   char item;
   struct inode *next; };
typedef struct inode *istack;

/* external representation */
struct xstack {
   int size; char *item; };

encode(i_s,x_s)
istack *i_s;
struct xstack *x_s;
{  int i;   istack t;

   t = *i_s;
   for (i=0; t!=NULL; i++) {
      x_s->item[i] = t->item;
      t = t->next;
      }
   x_s->item[i] = NULL;
   x_s->size = i;
   send (x_s);
}

decode(i_s,x_s)
istack *i_s;
struct xstack *x_s;
{  int i;

   receive (&x_s);
   create (i_s);
   for (i=x_s->size-1; i>=0; i--) {
      push (i_s, x_s->item[i]);
   }
}
```

```
push(s,j)
istack *s;  /* stack can't be full */
char j;
{  istack t;
   t = malloc(sizeof(struct inode));
   t->item = j;
   t->next = *s;
   *s = t;
}

char pop(s)
istack *s; /* stack can't be empty */
{  char item; istack t = *s;
   *s = (*s)->next;
   item = t->item;
   free(t);
   return (item);
}

create(s)
istack *s;
{  while (*s != NULL) { pop(s); } }

int empty(s)
istack *s;
{  return (*s==NULL); }

int full(s)
istack *s;
{  int i;  istack t = *s;
   for (i=0; t!=NULL; i++) {
      t = t->next;
   }
   return (i==STKSIZE);
}
```

Figure 3.3: Stack Operations

```c
char s[256];
char xstack_buf[256];
struct xstack x_s={0,xstack_buf};

main(argc,argv)
int argc;
char **argv;
{
 while (1) {

   if (push requested) ...
   else if (pop requested) ...
   else if (create requested) ...
   else if (empty requested) ...
   else if (full requested) ...
   else if (encode requested) ...
   else if (decode requested) ...
 }
}
```

```c
char s[256];
char xstack_buf[256];
struct xstack x_s={0,xstack_buf};

int reconfig_requested=0;

main(argc,argv)
int argc;
char **argv;
{

 if (status is clone)
   decode(s, &x_s);

 signal(SIGHUP,catch_reconfig);

 while (1) {

   if (push requested) ...
   else if (pop requested) ...
   else if (create requested) ...
   else if (empty requested) ...
   else if (full requested) ...

   if (reconfig_requested)
      encode(s, &x_s);
 }
}

catch_reconfig()
{ reconfig_requested = 1; }
```

Figure 3.4: Two Interfaces for the Stack Module

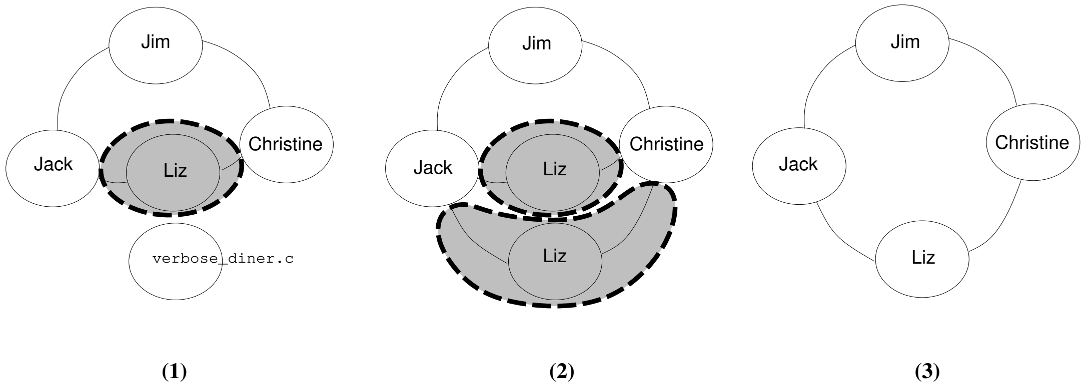


Figure 3.5: Replacing a Diner with a Verbose Diner.

3.2.2 Replacing a Dining Philosopher

The stack example from the previous section demonstrates how the program counter and global data can be captured and restored, and in this section we show how this approach fits into the rest of the reconfiguration activities for replacement. In the Introduction, we described a scenario where one of the diners is dynamically replaced with a verbose diner. Here we give the details of this reconfiguration activity. The replacement is accomplished by creating a new verbose diner module, copying the state from the old diner to the new, binding the verbose diner into the application, and removing the old diner (Figure 3.5).

The CATALYST is enhanced to perform replacement with module participation; it does this by invoking the `mh_objstate_move` reconfiguration primitive. Figure 3.6 shows the reconfiguration events performed by the CATALYST, which coordinates application-level changes and module participation for replacing modules.

First the CATALYST gains access to the specification for the existing module, in order to determine its attributes and eventually delete the module and bindings. This module specification contains the same items as those supplied in the original configuration specification (Figure 1.3), but it corresponds to the current configuration, which could have been changed dynamically. After acquiring access to the old diner and creating a new diner, the new diner is given an updated `BINARY` attribute, specifying the implementation of the new diner as a verbose diner, and a `STATUS` attribute, indicating that the new diner must restore its state upon start up.

Next the rebinding commands are prepared. The `mh_struct_objnames` command returns an array containing the names of the module's interfaces, which are passed to the `mh_struct_ifdest` and `mh_struct_ifsources` commands in order to determine the current bindings. Bindings to the old module's interfaces are replaced by bindings to the new module's interfaces of the same name. The rebinding commands are applied all at once, after the old module has divulged its state.

The `mh_objstate_move` operation signals the old module to divulge its state, waits until the old

```
mh_obj_cap(&old,"Liz");                           /* access old module */
mh_obj_cap(&new,"Liz");                           /* create new module */
mh_edit_objattr(&new,"add","BINARY","verbose_diner.out");
mh_edit_objattr(&new,"add","STATUS","clone");

mh_bind_cap(&b);                           /* prepare binding commands */
mh_struct_objnames(&old,if,&num_if);
for (i=0; i<num_if; i++) {
                                                   /* rebind outgoing */
    mh_struct_ifdest(&old,if[i],bind,&num_bind);
    for (j=0; j<num_bind; j++) {
        mh_edit_bind(&b,"del",&old,if[i],bind[j],NULL);
        mh_edit_bind(&b,"add",&new,if[i],bind[j],NULL);
    }
                                                   /* rebind incoming */
    mh_struct_ifsources(&old,if[i],bind,&num_bind);
    for (j=0; j<num_bind; j++) {
        mh_edit_bind(&b,"del",bind[j],NULL,&old,if[i]);
        mh_edit_bind(&b,"add",bind[j],NULL,&new,if[i]);
        mh_edit_bind(&b,"cpq",&old,if[i],&new,if[i]);
    }
}
                         /* get state from old module, send it to new */
mh_objstate_move(&old,"encode",&new,"decode");

mh_rebind(&b);                               /* apply binding commands */
mh_chg_obj(&new,"add");                         /* start up new module */
mh_chg_obj(&old,"del");                           /* remove old module */
```

Figure 3.6: Replacement with Module Participation

module has complied, and sends this state to the new module. The old diner sends its process state on interface `encode` then blocks indefinitely. The old diner's state is sent to the `decode` interface of the new diner, which is not yet active. This accomplishes the state transfer from the old module to the new, except for messages that may be queued for the old diner. These queued messages are copied to the new diner in the rebinding phase: here the old module's bindings are removed, bindings for the new module are added, and queued messages are copied from the old to the new. Now that the state of the old diner and its bindings has been copied to the new diner, the old module is deleted and the new one is started.

Applying the binding changes atomically simplifies the reconfiguration task, both by reducing the number of steps required and by making it easier to reason about the reconfiguration. Notice that we did not need any `mh_hold` primitives in this scenario: the old module blocks after encoding its state, effectively holding itself. But the modules bound to this old module can continue sending messages to it, and without atomic rebinding, we would have to hold both ends of each binding destined for replacement.

The geometric reconfiguration described in the Introduction is to move a diner to another host. This reconfiguration is almost identical to replacing a module with another implementation; the difference is that instead of changing the `BINARY` attribute, we change the `MACHINE` attribute to specify a different host name. The POLYLITH platform, designed to accommodate heterogeneity, handles all underlying details.

The reconfiguration events shown in Figure 3.6 are a simplified version of the CATALYST, which is parameterized to accept a module name and attributes. We have also left out the details of determining whether to bind the new module to another new module or an existing one. The CATALYST can be used to replace one or more modules in any application, provided the modules have been prepared to participate during reconfiguration.

We have not yet discussed the old and new diners' participation in this replacement scenario. Figure 3.7 shows the code for `diner.c` after it has been modified to support state capture and restoration for replacement. (When comparing this to Figure 1.2, the amount of new code may seem substantial; but while we abstracted away all details of the original algorithm, we included the details of the reconfiguration aspects.) To support replacement, the module provides **encode** and **decode** operations to capture and restore its own process state.

During reconfiguration, the

```
mh_objstate_move(&old,"encode",&new,"decode")
```

command first binds the first module's `encode` interface to the new module's `decode` interface, then signals the first module to divulge its state. The diner module is prepared to receive the signal with procedure `catch_reconfig()` (Figure 3.7), and it turns on the flag `reconfig_requested`. The purpose of this flag is to delay the **encode** operation until the diner reaches a reconfigurable state. After returning from the signal handler, the diner continues normal execution until it reaches the bottom of the main `while` loop, where it performs the `encode` operation and blocks.

```
initialize diner state to HUNGRY;
initialize left fork state;
initialize right fork state;
reconfig_requested = 0;

catch_reconfig() { reconfig_requested = 1 }

main() {
   if (status is special) set initial values so that graph is acyclic;
   else if (status is clone) receive diner state, left fork state,
      and right fork state on interface decode;
   signal(SIGHUP,catch_reconfig);

   while (1) {

      update left fork state;

      update right fork state;

      if (HUNGRY and conditions are right) start EATING;
      else if (done EATING) start THINKING;
      else if (done THINKING) become HUNGRY;

      if (reconfig_requested) {
         send diner state, left fork state, and right fork state on interface encode;
         block;
      }
   }
}
```

Figure 3.7: Reconfigurable Version of diner.c.

By delaying the **encode** operation, we have in effect defined the process state to include the program counter, with its value set to the end of the loop.

Because this diner's `encode` interface is temporarily bound to the new diner's `decode` interface, the process state is sent to the new diner. Recall that the final reconfiguration steps are to remove the old diner and start up the new one. The new diner has a `STATUS` attribute of `clone`, so when it is started up, its first action is to perform the **decode** operation. Since the program counter was at the end of the main while loop when the state was captured, there is no need for an explicit `goto` the end of the loop; execution resumes at the beginning of the loop.

3.3 The Activation Record Stack

In the previous section, the variables used in capturing and restoring process state were global data, and there was a single reconfiguration point, located in procedure `main`. Next we show how to capture and restore the module state in the general case, which includes capturing and restoring local data (data allocated on the activation record stack), and includes reconfiguration points located in called procedures.

The approach we take is similar to the technique proposed in [31] for heterogeneous process migration, where by compiling a special program that restores the process state, the authors force the compiler to manage the machine-specific details of restoring the activation record stack. We use the compiler to restore and capture the activation record stack, without making any changes to the compiler or operating system.

Using this approach we can capture and restore the following highlighted items in the process state:

- ◦ program counter
- ◦ static data– in data area
- **• dynamic data– in activation record (AR) stack**
- **• temporary values– in registers or AR stack**
- **• procedure call/return information– in AR stack**
  - **• return address**
  - **• control link to restore context upon return**
  - **• register values to restore upon return**
- ◦ user-allocated data– in heap
- ◦ file descriptors, process status information– accessible only to kernel

The example in Section 3.3.1 illustrates how this activation record capture and restoration work, Section 3.3.2 explains the algorithm used in the transformation, and Section 3.3.3 describes the Dynamic Reconfiguration Transform (DRT) tool that implements this source-to-source transformation.

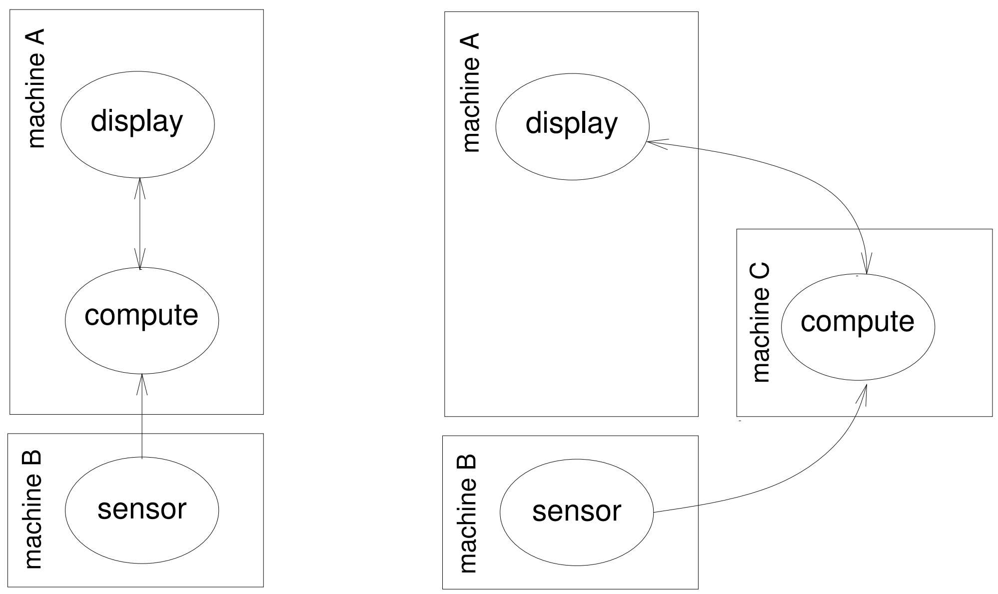


Figure 3.8: The Monitor Example Before Reconfiguration (left); After Reconfiguration (right)

3.3.1 The Monitor Example

The Monitor example is a distributed application containing three modules, each of which can be distributed to a different machine. The reconfiguration performed in this example is to move one of the modules to another machine while the application executes. The starting configuration is shown in Figure 3.8 (left): module `sensor` produces temperature values at regular intervals, module `display` requests a value then displays it, and upon request module `compute` performs a computation on a group of temperature values and returns the result. In the ending configuration shown in Figure 3.8 (right), the `compute` module has been relocated to another machine.

The computation performed in the `compute` module is merely to average a group of temperature values. However, to best illustrate the mechanism used to capture and restore the activation record stack, we have used a recursive algorithm to perform this computation, placing the reconfiguration point within the recursive procedure. Thus moving the `compute` module during execution requires capturing the state of the activation record stack in the midst of these recursive calls.

The POLYLITH configuration specification for this application is shown in Figure 3.9. In addition to the standard information included in an application specification, the reconfiguration points (in this case, `R`) are specified. This could be incorporated into the application specification, but for now the reconfiguration points are specified in a separate file. The reconfiguration point specified corresponds to a label `R` inserted by the programmer into the source code for module `compute`. For the current prototype these labels are unique across all modules in the application, but the reconfiguration points could easily be qualified by module name and even by procedure

```
service "display" {
    implementation : { binary : "./display.out" }
    client "temper" : {integer} accepts {^float}
}

service "compute" {
    implementation : { binary : "./compute.exe" }
    function "display" : {^integer} returns {float}
    sink "sensor" : {^integer}
}

service "sensor" {
    implementation : { binary : "./sensor.exe" }
    source "out" : {integer}
}

orchestrate "monitor" {
    tool "display"
    tool "compute"
    tool "sensor"
    bind "display temper" "compute display"
    bind "sensor out" "compute sensor"
}


reconfiguration point = R
```

Figure 3.9: Application Configuration Specification

name.

The source code for the `compute` module is shown in Figure 3.10. It loops forever, checking for requests on the `"display"` interface. If one arrives, it recursively computes the average of `n` temperature values read from the `"sensor"` interface. When no requests are pending, it discards any available temperature values by trivially computing the average of one value.

When reconfiguration is requested, the `compute` module executes until it reaches reconfiguration point `R`. Because the reconfiguration point is inside the recursive procedure, at reconfiguration time there will be one or more activation record for this procedure at the top of the stack. Since the recursive procedure could have been called from one of three places within the module, the penultimate activation record in the stack can correspond to any one of these three calls.

During reconfiguration, it is the responsibility of the module to delay reconfiguration until the appropriate point, package up its state, and install state in a dynamically created module. In the monitor example, module `compute` is moved to another machine, so this module must be prepared

```
main(argc, argv)
int argc; char **argv;
{  int n; double response; void compute();

   mh_init (&argc, &argv, NULL, NULL);
   while (1) {
      /* handle requests for temp */
      while (mh_query_ifmsgs("display")) {
         mh_read("display","i",NULL,NULL,&n);
         /* compute avg of n temps */
         compute (n, n, &response);
         mh_write("display","F",NULL,NULL,response);
      }
      /* keep sensor buffer clear */
      if (mh_query_ifmsgs("sensor")) {
         compute (1, 1, &response);
      }
      sleep(2);
   }
}

void
compute(num, n, rp)
int num, n; double *rp;
{  int temper;

   if (n<=0) { *rp = 0.0; return; }
   compute (num, n-1, rp);
R: mh_read ("sensor","i",NULL,NULL,&temper);
   *rp = *rp + ((double)temper / (double)num);
}
```

Figure 3.10: Original Compute Module

to participate during reconfiguration. Our method of providing this module participation is to pre-process the source program for the module, adding code to capture and restore the process state at the reconfiguration point specified by the programmer. When the reconfiguration point is located in a procedure other than the main procedure, the state of each procedure in the activation record stack must be captured and restored.

Figures 3.11 and 3.12 show the source code for module `compute` after the module participation statements (in slanted typeface) have been automatically inserted. When a reconfiguration signal is captured (Figure 3.12), the flag `reconfig_requested` is turned on, and the module continues executing until it reaches the block of code just above the reconfiguration point `R`. Inside this capture block, the `reconfig_requested` flag is turned off, the `capture_stack` flag is set, the state specified by the programmer is captured, and the procedure returns, thus completing the capture and pop of the top activation record.

The procedure containing the reconfiguration point could have been called from one of three statements, labeled here by `L1, L2, or L3`. Immediately following each of these three statements is another capture block. This block is executed when the `capture_stack` flag is set: it simply captures the local state and returns. The differences between the capture blocks in the main and those in procedure `compute` are that the local state is different, and that the main contains an `mh_encode()` to send the captured state outside the module. The capture block just after `L3` will execute once for each recursive call on the activation record stack. The bottom activation record is captured and popped by one of the capture blocks in the main.

In each of the calls to `mh_capture`, in addition to the local variables, an integer `1, 2, 3,` or `4` is captured. This integer value corresponds to a label marking the statement where execution should resume during restoration. Thus these integers represent an abstract program counter.

During restoration, the same source program is executed for module `compute`, but it is assigned a status such that the flag `mh_restoring` is turned on. This flag remains on, triggering the restore blocks, until the activation record stack has been completely rebuilt. Each restore block restores the local state and jumps to the statement where execution should resume, thus restoring the program counter. When the final activation record is restored, the `mh_restoring` flag is turned off, and execution resumes at the reconfiguration point.

Since the code for divulging and installing program state is part of the source program, the compiler takes care of dereferencing variables from the activation record stack and of rebuilding the activation record stack during restoration. Thus the module thread is captured and restored without explicit reference to the program counter or to any of the call/return information stored in the activation record stack.

Because the state is captured and restored between statements in the high-level language, temporary or intermediate values used in a computation are never part of the process state. Variables cached in registers will be correctly captured and restored, because the capture and restore statements are embedded in the source program, and the compiler takes care of referencing these variables correctly.

```
main(argc, argv)
int argc; char **argv;
{ int n; double response; void compute();
   mh_init (&argc,&argv,NULL,NULL);
/* ------------------------------------------------ begin restore -------- */
   if (strcmp(mh_get_status(),"clone")==0) mh_restoring=1; else mh_restoring=0;
   if (mh_restoring) {
      mh_decode(); mh_restore("iif",&mh_location,&n,&response);
      if (mh_location==1) goto L1;
      if (mh_location==2) goto L2;
   }
   signal(SIGHUP,mh_catch_reconfig);
/* -------------------------------------------------- end restore --------- */
   while (1) { /* handle requests for updated temperature */
      while (mh_query_ifmsgs ("display")) {
         mh_read("display","i",NULL,NULL,&n);
L1:      compute (n, n, &response);
/* ------------------------------------------------ begin capture -------- */
         if (capture_stack) {
            mh_capture("IIF",1,n,response); mh_encode();
            return; }
/* -------------------------------------------------- end capture --------- */
         mh_write("display","F",NULL,NULL,response);
      }
      if (mh_query_ifmsgs ("sensor")) {
L2:      compute (1, 1, &response);
/* ------------------------------------------------ begin capture -------- */
         if (capture_stack) {
            mh_capture("IIF",2,n,response); mh_encode();
            return; }
/* -------------------------------------------------- end capture --------- */
      }
      sleep(2);
   }
}
```

Figure 3.11: Procedure `main` Prepared for Reconfiguration

```
void compute(num, n, rp)
int num, n; double *rp;
{ int temper;
/* ------------------------------------------------ begin restore -------- */
   if (mh_restoring) {
      mh_restore ("iiif",&mh_location,&num,&n,rp);
      if (mh_location==3) goto L3;
      if (mh_location==4) {
         mh_restoring=0; signal(SIGHUP,mh_catch_reconfig);
         goto R; }
} /* ------------------------------------------------ end restore --------- */
   if (n<=0) { *rp=0.0; return; }
L3: compute (num, n-1, rp);
/* ------------------------------------------------ begin capture -------- */
   if (capture_stack) {
      mh_capture("IIIF",3,num,n,*rp);
      return; }
/* ------------------------------------------------- end capture --------- */
/* ------------------------------------------------ begin capture -------- */
   if (reconfig_requested) {
      reconfig_requested=0; capture_stack=1;
      mh_capture("IIIF",4,num,n,*rp);
      return; }
/* ------------------------------------------------- end capture ----------*/
R: mh_read ("sensor","i",NULL,NULL,&temper);
   *rp = *rp + ((double)temper / (double)num);
}

void mh_catch_reconfig() { reconfig_requested=1; }
```

Figure 3.12: Recursive Procedure `compute` Prepared for Reconfiguration

3.3.2 Source Code Transformation

The monitor example described in Section 3.3.1 illustrates how the activation record stack is captured and restored. This section describes the general technique for transforming a source program with reconfiguration points specified by the programmer into a reconfigurable source program. Several issues that must be resolved for the general case did not arise in the monitor example. A program may have more than one reconfiguration point; in such a case, the question is whether each reconfiguration point must have its own capture and restore blocks, or all reconfiguration points can share the same capture and restore blocks. A more subtle problem is the potential discrepancy between the local state used to restore the activation record stack and the local state when the original procedure call was made.

The static call graph of a program contains a node for each procedure and function in the program, and a directed edge from node **a** to node **b** if and only if the source code for procedure `a` contains a call to procedure `b`. All nodes in this graph have one or more incoming edges except for the node corresponding to the `main` procedure, which has only outgoing edges. Figure 3.13 shows an example program and its corresponding static call graph. (The line numbers S_i are unique in order to simplify the explanation; in practice, the file name distinguishes between duplicate line numbers.) The static call graph is determined by examining the source program, not by analyzing its run-time behavior. At any particular time during program execution, the frames contained in the activation record stack correspond to a path in the static call graph originating at node **main**. Thus the static call graph describes all possible activation record stacks.

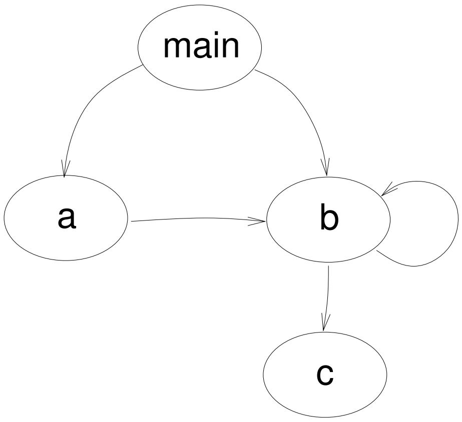
Because we allow reconfiguration to occur only at reconfiguration points and not at any arbitrary point in the program execution, when reconfiguration occurs, only procedures containing a reconfiguration point can be at the top of the activation record stack. Thus only procedures which could be below these on the activation record stack need be instrumented for reconfiguration. In terms of the static call graph, only nodes on paths starting at **main** and ending at a procedure containing a reconfiguration point are of concern; these nodes and edges define a subgraph of the original static call graph. Each of the procedures in this subgraph, including the main and the procedure containing the reconfiguration point, must be prepared for reconfiguration.

The first step in preparing a program for reconfiguration is to augment this subgraph of the static call graph. The augmented subgraph, called the reconfiguration graph, contains an edge for each procedure call, and each edge is labeled with the line number of the call. Thus if procedure `main` calls `a` in two different statements, there are two edges from **main** to **a**. The reconfiguration graph also contains a new node, named **reconfig**, and an edge from each reconfiguration point to the **reconfig** node, annotated with the line number of the reconfiguration label. In addition, the edges in the reconfiguration graph are numbered consecutively, so each edge is labeled (i, S_i), where i is an integer and S_i is a line number. These edges define the places where the program state is potentially captured and restored. Figure 3.13 shows the reconfiguration graph and its corresponding program.

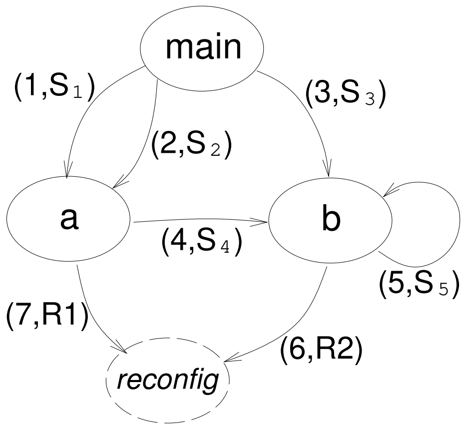
The second step is to install code to capture and restore the program state. The state may be captured at various places within the procedure, but it is always restored at the beginning of the

main() {	a() {	b() {	c() {
. . .	. . .	. . .	. . .
S_1 a(x_1)	R1:	S_5 b(x_5)	}
. . .	. . .	. . .	
S_2 a(x_2)	S_4 b(x_4)	R2:	
. . .	. . .	. . .	
S_3 b(x_3)	}	S_8 c(x_8)	
. . .		. . .	
}		}	

Sample Program



Static Call Graph



Reconfiguration Graph

Figure 3.13: Example of Static Call Graph and Reconfiguration Graph

Capture Block for Reconfiguration Edge (j,R):

```
    if (reconfig_requested) {
        reconfig_requested = 0;
        capture_stack = 1;
        mh_capture (j, local vars);
        return;
    }
R:
```

Capture Block for Edge (i,S_i):

```
Si: f(x);
    if (capture_stack) {
        mh_capture (i, local vars);
        return;
    }
Li:
```

Figure 3.14: Capture Blocks

procedure. Thus each node in the reconfiguration graph will receive a single restore block and one or more capture blocks. For each edge originating at that node, restore code is inserted into the restore block, and a capture block is installed at the line number associated with that edge.

There are two kinds of capture blocks that must be inserted in order to reconfigure. The first is used at a reconfiguration point, and the second is used to capture the state of the activation record stack. The only difference between these two types of blocks is that they are triggered by different flags (Figure 3.14). The flag set in the reconfiguration signal handler triggers the blocks installed at reconfiguration points, and these blocks set the flag which triggers the blocks installed for activation record capture.

For each edge terminating at node **reconfig**, a capture block for that reconfiguration point is installed immediately preceding the reconfiguration label installed by the programmer. For each remaining edge i associated with statement S_i in the reconfiguration graph, a label `Li` is inserted at the statement immediately following S_i, and a capture block is installed immediately preceding label `Li`.

Notice that a single capture block is installed after each procedure call that could be interrupted by a reconfiguration. In our example, `main`'s call to `a` in line S_1 could be interrupted by a reconfiguration at `R1` (in procedure `a`) or at `R2` (in procedure `b`). Capturing the state of `main` at S_1 does not depend on which reconfiguration point triggered the capture, so reconfiguration points can share capture blocks.

```
if (mh_restoring) {
    mh_restore (&mh_location, local vars);
    restore code for each edge
}
```

Restore Code for Edge (i,S_i):

```
    if (mh_location==i) {
        f(x);
        goto Li;
    }
```

Restore Code for Reconfiguration Edge (j,R):

```
    if (mh_location==j) {
        mh_restoring = 0;
        install reconfiguration signal handler
        goto R;
    }
```

Figure 3.15: Restore Block

We have not yet discussed how to automatically determine which variables should be captured in these capture blocks. At the present time, the programmer provides this information as part of the specification of a reconfiguration point. For capturing the state of the activation record stack, the relevant variables are the parameters and local variables of a procedure. Data-flow analysis could be used to determine the set of live variables at each capture block; variables are live when they contain values that will (or might) be used in the remainder of the program. The primary difficulty in automatically determining which variables to capture arises with pointer variables. Since pointers are addresses, they must be translated into an abstract format for capture and restoration. For example, a pointer variable containing an explicit address would be translated into a variable that points to the n^{th} character of a string located at some symbolic address.

A restore block is inserted at the top of each procedure present in the reconfiguration graph. Inside this restore block the local state is restored, then there is restore code for each edge originating at that node (Figure 3.15). Again, the restore code for a reconfiguration point differs slightly from the code for restoring the activation record stack. If the state capture was triggered by reconfiguration point `R`, then the activation record stack has been completely restored and after jumping to `R`, reconfiguration is complete. If the state capture was triggered by a return from a procedure call interrupted by reconfiguration, then the restore code repeats the procedure call and jumps to the statement immediately following the original procedure call.

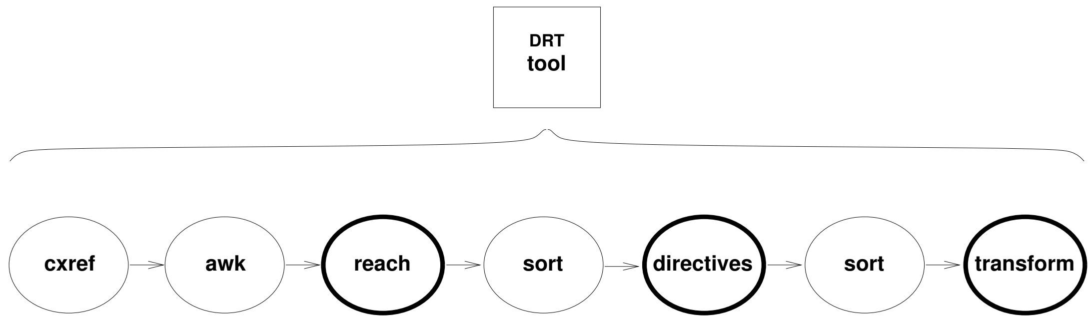


Figure 3.16: Dynamic Reconfiguration Transform Tool

Here arises the question of whether it is acceptable to simply repeat the original procedure calls when restoring. The local state at the time of the original procedure call is not guaranteed to be the same as the local state at the time it is captured, because the called procedure may change the values of variables that are visible to the callee. Thus the values of the arguments when the procedure is originally called can be different from their values when the procedure is reinvoked during restoration. On the surface this discrepancy appears to be acceptable, since during restoration the first action of the called procedure is to restore its own local state, including all parameter values, so the values of arguments passed during restoration are inconsequential. However, a problem can arise when the arguments are not scalars but are expressions. When the original procedure call is repeated during restoration, these expressions are evaluated with the restored state, and their evaluation can cause a run-time error that did not arise when they were evaluated with the original state. The solution to this problem is to not repeat the original procedure call, but to modify the call by substituting dummy arguments for expressions whose evaluation could result in a run-time error. The data types of these dummy arguments are determined by the types declared in the parameter list of the procedure.

3.3.3 The Dynamic Reconfiguration Transform Tool

In this section, we describe the Dynamic Reconfiguration Transform (DRT) tool, built for performing the source-to-source transform. Our goal was to implement the algorithm from Section 3.3.2, automating the installation of capture and restore blocks, not automating the entire state capture.

The approach used for the DRT tool was to use existing Unix tools (`cxref, awk, sort`) plus some programs written specifically for this transform (Figure 3.16). The primary difficulty in generating the static call graph lies in parsing the original source program. To avoid having to locate or develop an adequate C grammar for one of the standard analysis tools (Yacc, NewYacc), we chose to use the cross-reference information produced by `cxref` to deduce the static call graph. In addition to being quick to develop, the advantage of this approach is that the parsing matches that of the compiler. The drawback to this approach is that the output is line-oriented, giving rise to some restrictions for this prototype: the reconfiguration label must be located at the beginning of a physical line, and functions that can reach a reconfiguration point cannot be used as expressions (the value returned must be assigned to a temporary variable, which can then be used inside an expression).

The output generated by `cxref`, which is in a report format, is sent through `awk` in order to put it into a record format. The next step is to extract certain portions of the cross-reference information. The cross-reference information lists where each variable, procedure, or label is defined and used, giving the line number, procedure, and file for each entry. Thus we can determine the procedure in which the reconfiguration label is located, and all the procedures that can reach it. The custom program `reach` takes the output of `awk`, plus the list of reconfiguration points, and saves only the cross-reference information for procedure calls that can reach a reconfiguration point.

This reachability information is then sorted so that we can generate reconfiguration directives in a single pass over the original source program. The directives are generated, one capture directive and one restore directive for each reconfiguration point and for each procedure call that could reach a reconfiguration point.

Because the restore directives are grouped together at the beginning of a procedure, while the capture directives are scattered throughout the procedure (at reconfiguration points and at calls to procedures that can reach a reconfiguration point), these directives are sorted. Then, in program `transform`, these sorted directives are applied to the original source program in a single pass, creating the transformed source program.

3.3.4 Discussion

The large body of work on process migration deals mainly with homogeneous process migration, in which the process state is captured in a machine-specific format and restored on a machine of the same architecture [14]. The main examples of this approach are Amoeba [24], Charlotte [2], DEMOS/MP [25], MOS [4], Sprite [12], and the V system [32]. Emerald [20] uses the same basic approach, although here the unit of granularity is an object, not a process. These systems all rely on an operating system designed or customized to support migration, whereas our goal was to develop techniques that operate above the operating system. But the most fundamental difference between the homogeneous process migration work and ours is that they have no need to represent the process state in an abstract format, so their approach to capturing and restoring state is to copy the executing image.

Our approach does not use checkpointing, in which the entire state of the process is saved periodically, and execution is rolled back to the most recent checkpoint in order to restore the process. Instead, when a reconfiguration is requested, the process continues executing until it reaches the next reconfiguration point. Thus the run-time cost is merely that of periodically testing the flags installed for reconfiguration. The cost of capturing the process state is paid only when a reconfiguration is performed, instead of at regular intervals during execution.

Because in checkpointing the run-time cost is that of capturing the process state at regular intervals, and a low run-time cost is desired, work on this topic generally assumes that the process state is saved by copying it out to secondary storage [11, 21, 29, 30]. This of course depends on the state being restored on a machine of the same architecture and operating system, whereas the focus of our work is to remove those dependencies.

In our approach, only the flags are tested at regular intervals. The frequency of flag testing depends on how many reconfiguration points are inserted, and where they are placed. In order for a module to quickly respond to a reconfiguration request, the reconfiguration points must be located within the most frequently executed code. However, for applications with an execution time on the order of days rather than seconds, placing reconfiguration points where they will be checked regularly is more important than placing them where they will be checked frequently. Putting reconfiguration points in deeply-nested procedures or in procedures that are called from many places increases the occurrence of reconfiguration flags in the source code, but the execution cost due to testing these flags depends on how often the procedure is actually invoked.

By virtue of where a reconfiguration point is placed, it could prohibit certain compiler optimizations such as code motion. Because these reconfiguration techniques are intended for long-running or continuously available applications, a reconfiguration delay measured in seconds rather than micro-seconds may be perfectly acceptable. If so, reconfiguration points should be placed outside of computationally intensive loops or procedures, so that the code executed most often can be optimized as much as possible.

A reconfiguration platform supports dynamic updates at a particular level of atomicity: updates can be atomic at the module level, at the procedure level, or at the statement level. If the reconfiguration is atomic at the module level, it means that modules execute atomically with respect to reconfiguration; a module cannot be updated while it is executing. Reconfiguration without module participation, as discussed in Chapter 2, is atomic at the module level because module state cannot be captured and restored (although there can be a partial execution of a module).

Podus, a system that supports updates with procedure-level atomicity is described in [15]. This system is restricted to updating a program without moving it from the original machine. The program is updated by replacing each procedure when it is not executing. To maintain consistency between the old version and the new during the replacement, the update is performed from the bottom up, by allowing a procedure to be replaced only after all the procedures it invokes have been replaced. The implications of this update strategy are that programs written in a top-down style will be updated more successfully than those that are not modularized. When changes to the program are restricted to the lower-level procedures, updates can be performed quickly, but when the higher-level procedures have changed, the update cannot complete until these procedures are inactive. For example, when the `main` procedure has changed, the update cannot complete until the program terminates. When a procedure has some cached state that must be installed in the new version, the programmer must write a special routine to do this.

A platform that preserves atomicity at the statement level could support reconfiguration either at every statement, or at certain reconfiguration points specified by the programmer. We take the latter approach, as does the reconfiguration framework of Conic [23]. In Conic, reconfiguration activities are separated into configuration level concerns and application level concerns. Configuration level activities are independent of the algorithms, protocols, and states of the application, and are guaranteed to leave the system in a consistent state, where consistency is defined in terms of the application. To ensure consistency after reconfiguration, the programmer writes code for the modules to respond to the configuration level commands `passivate`, `unlink`, and `link`. These commands correspond to the module-level activities of moving to a compatible state, capturing process state, and restoring process state.

In contrast to the Conic work, work by Hollander and Silberman addresses the problem of capturing the process state in an abstract format [19]. Although the goal of their work is to support heterogeneous process migration, not a full spectrum of dynamic reconfiguration activities, the `Migration/Recovery` points they use are similar to our reconfiguration points. However, in their approach, preparing the process state for migration is done primarily at migrate time, not at compile time as in our approach. They cannot do the preparation statically because their approach relies on examining the activation record stack in order to determine the call chain. Of course, direct examination of the activation record stack requires knowledge of its machine-specific layout, so this approach is not machine-independent. In this respect, it is similar to the work described in [13].

The method proposed in [31] supports heterogeneous process migration (moving a module) between every possible statement. These migration points are the places where the abstract state defined by the high-level source program and the state in the binary correspond. To capture the process state at one of these migration points, they propose using a procedural interface to an existing source-level debugger. At migration time, a machine-independent migration program would be generated, compiled, and executed on the target machine. The migration program first reconstructs global and heap data, then rebuilds the activation record stack by executing a sequence of calls to special procedures, which are modified versions of the procedures in the activation record stack at migration time. The modified procedures initialize local variables, call the next modified procedure in the call stack, and arrange to resume execution in the original procedure. Thus the details of the code and data translation are hidden in the compilers for each machine.

This approach is summarized in Figure 3.17. For capturing the activation record stack a separate, machine-specific translation program is required, (as in Figure 3.1), but for restoring the activation record stack the translation is embedded in the source program (as in Figure 3.2).

The main advantages of our work over the technique described in [31] are that the state is captured abstractly and that the run-time cost is much lower. Their goal is to support heterogeneous process migration, requiring that migration points be transparent to the programmer and that they be available at as many places in the source code as possible. Our goal is to support dynamic reconfiguration: we expect the programmer to be involved in selecting a small set of reconfiguration points. Because the number of reconfiguration points is relatively small, we can prepare the program to capture the process state in addition to restoring it, instead of using one

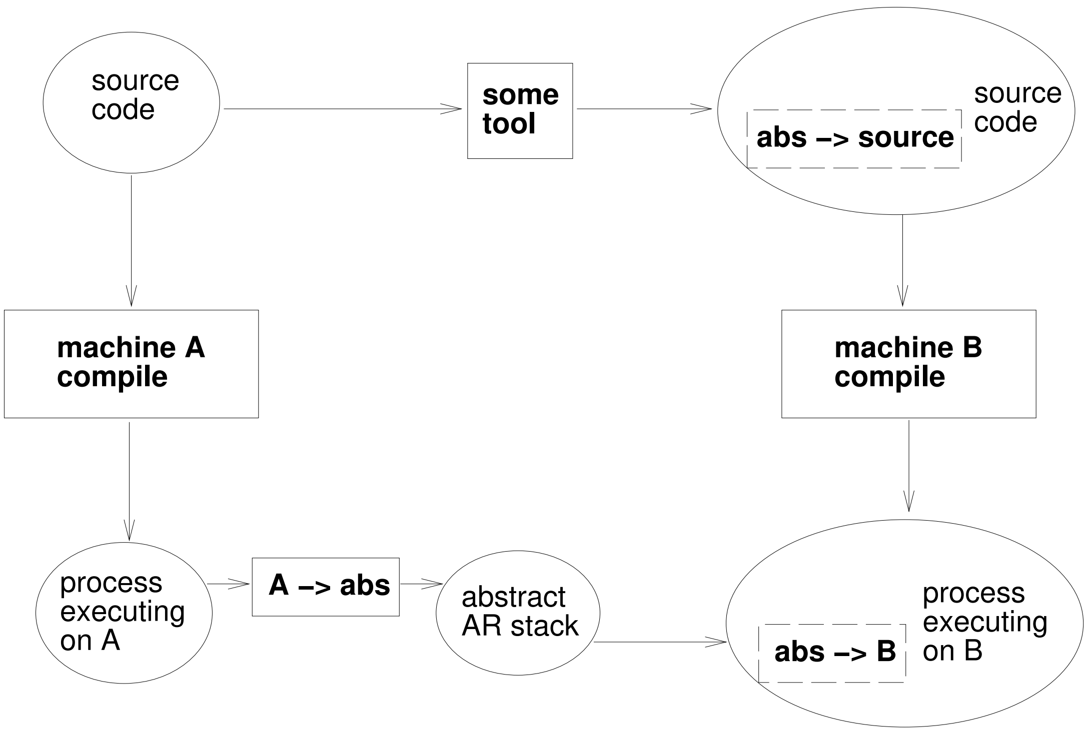


Figure 3.17: Theimer/Hayes Approach to Capturing and Restoring the Abstract Activation Record Stack

technique for capturing process state and a different technique for restoring it. For the same reason, we can prepare the program for all possible reconfigurations when the original program is compiled. In contrast, Theimer and Hayes prepare the program for one specific migration when that migration is requested, by generating and compiling the new source program at run time.

3.4 Files and the Heap

The approach presented up to this point shows how to capture and restore static and dynamic data (in the data area and in the activation record stack), procedure call/return information, and the program counter. This includes everything from the list in Section 3.1 except for the following highlighted items:

- ◦ program counter
- ◦ static data– in data area
- ◦ dynamic data– in activation record (AR) stack
- ◦ temporary values– in registers or AR stack
- ◦ procedure call/return information– in AR stack
  - ◦ return address
  - ◦ control link to restore context upon return
  - ◦ register values to restore upon return
- • **user-allocated data– in heap**
- • **file descriptors, process status information– accessible only to kernel**

The example in this section shows how data from the heap and files can be captured and restored.

This application is a regenerative simulation used for studying disk I/O behavior [9]. This example was our first experience adapting for reconfiguration a previously existing application, one that was developed without any forethought of providing dynamic reconfiguration capabilities. The simulation's execution time varies from minutes to hours, depending on the command-line arguments given at start-up. With dynamic reconfiguration, a long simulation could be replaced with a faster version, without losing the work computed by the slower simulation.

The pseudo-code for the simulation is shown in Figure 3.18, and the label `R` indicates our desired reconfiguration point. The lines in slanted typeface are the capture and restore blocks inserted by the DRT tool. Inside these capture and restore blocks the data local to the procedure must be captured (restored), and the blocks installed in the `main` procedure must also capture (restore) the global data.

After inserting the capture and restore blocks, we examined the program to determine the global and static data. Figure 3.19 lists the different files used in this application and the global and static data declared in each one. Of the six files, three contain data that must be captured and restored (`sim.c, reg.c, and sched.fclrp.c`). In each of these three files, we provided **encode**

```
main()
{  local declarations
   restore block
   read cmd line arguments

   for (read increments)
      for (write increments) {
L2:      run_simulation();
         capture block
      }
   }
}

run_simulation()
{  local declarations
   restore block
   initialize

   while (more events and termination criteria not met) {
      switch (event type) {
         case A: ...
         case B: ...
         case C: ...
            event_q_decap();
            capture block
R:          next_service();
            break;
         case D: ...
      }
   }
   printstats();
}
```

Figure 3.18: Regenerative Simulation

sim.c:	global data:
	scalars
	3 linked list data structures allocated from heap
	queue of events
	queue of disk reads
	queue of disk writes
	2 instances of a simple structure
	2 files
	random number generator
	local data: scalars
reg.c:	static data: scalars
sched.fclrp.c:	global data: scalars
serv_time.c:	no global or static data
poisson.c:	no global or static data
distr.c:	no global or static data

Figure 3.19: Structure of Simulation Files

and **decode** routines that capture and restore the data declared in that file. These routines are invoked in the capture and restore blocks of the `main` procedure.

Most of the global and static variables are simple scalars, which pose no problem when capturing and restoring their values. However, there are three linked list data structures declared in `sim.c`, and these data structures are allocated from the heap. Our approach to capturing these linked lists is to traverse the list and transmit each item in the list. For restoring the lists, we read each item and rebuild the list in the same manner that it was originally built. When the data structure is implemented with accessors for inserting and removing an item (instead of in-lining the operations), it is simple to flatten and transmit the data structure. Figure 3.20 shows how one of these linked lists is captured and restored; this particular data structure was implemented with a `queue_event` operation but no corresponding `dequeue_event` operation, so restoring the linked list is simpler than capturing it.

The two files used in this program are used for output only, so we capture the files simply by closing them, and restore by re-opening them with `append`. This assumes that the file system is cross-mounted when the module replacement is across machines. If cross-mounting were not feasible, the file would have to be physically moved to the new machine.

The random number generator used in this application is another example of process state that is not immediately accessible. In restoring the state, if the seed used to initialize the random number generator is the same as the original seed or is given some arbitrary value, then the sequence

```
capture:
   n = 0;
   e = eventq;
   while (e != (EVENT *)NULL) {
      f_buf[n] = e->ev_time;
      i_buf[n] = e->ev_type;
      e = e->ev_next;
      n++;
   }
   send n, i_buf, f_buf;


restore:
   receive n, i_buf, f_buf;
   eventq = (EVENT *)NULL;
   for (k=0; k<n; k++) {
      queue_event (i_buf[k], f_buf[k]);
   }
```

Figure 3.20: Capturing and Restoring Data from the Heap

of numbers is not random. The state of the random number generator must be captured, and that state used to initialize the random number generator during restoration. Random number generators generally include such operations, so these would be invoked in the capture and restore blocks.

This example is not intended to imply that we have an automated approach for capturing and restoring files and heap data. At this time these must be addressed on a case-by-case basis, but this application demonstrates that they can be successfully captured and restored given the reconfiguration primitives we have provided.

This application also provided the opportunity to determine the performance impact of installing reconfiguration capabilities in a module. We measured the execution time of the program first in its original form and then with the capture and restore blocks installed (but no reconfiguration performed). There was no detectable difference in execution time between the original and the transformed program, so the run-time cost of testing the flags guarding the capture/restore blocks is not significant.

Chapter 4

Structural Changes

All previous dynamic reconfiguration scenarios are examples of module replacement, where modules and bindings may be added and removed, but the net effect of the reconfiguration is to leave the structure intact. For module replacement, synchronization is of concern only within the module being replaced: the module's process state may need to be captured and restored so that its new version interacts correctly with the rest of the application.

In this chapter, we discuss examples of dynamic reconfiguration where the structure of the application changes. For these structural changes, the reconfiguration activities may require synchronization between two or more modules. In addition to requiring the process states of individual modules to be captured and restored, a structural change may require these process states to be compatible before reconfiguration can occur.

The three examples in this chapter require different degrees of synchronization in order to perform a structural dynamic reconfiguration. The first example, replicating the producer in the Producer/Consumer application, needs no synchronization. The second example, adding a diner in the Dining Philosopher application, uses the `mh_hold` primitives to synchronize reconfiguration activities. The third example, repartitioning in a Jacobi relaxation application, requires the modules to cooperate in moving to compatible states before reconfiguring.

4.1 Replicating the Producer

Given the Producer/Consumer application described in Section 2.2, a simple structural reconfiguration is to replicate the producer module and its binding (Figure 4.1). Replication of a module and its bindings is done in three steps, labeled (1), (2), and (3) in the figure. First a new module is created with the same attributes and interfaces as the original, but with a new name. Second, the new module is given the same bindings as the original. The third and final step is to start the execution of the new module. Replicating the producer does not require synchronization because no existing module or binding is deleted, changed, or interrupted; we are simply adding a new

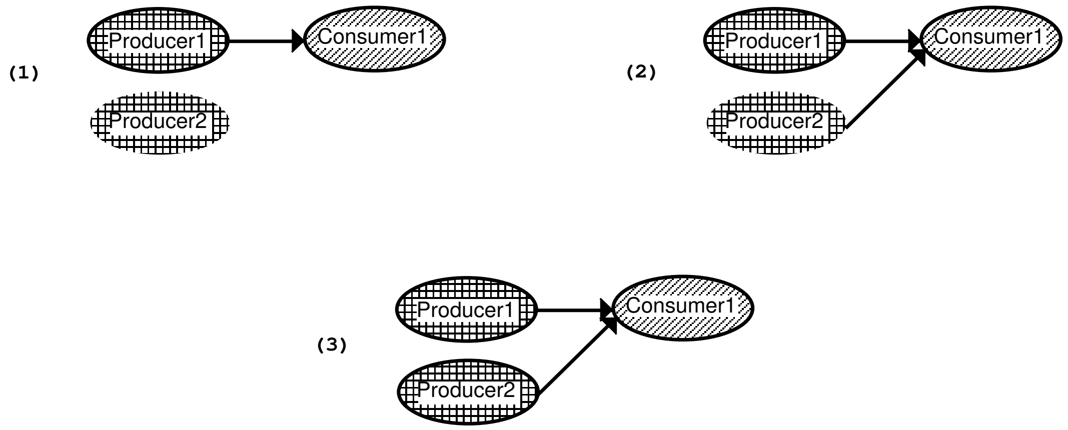


Figure 4.1: Replicating the Producer Module

module and new binding to the application.

The reconfigured application works correctly (works as it would have if the new configuration were the original configuration) because of certain characteristics of the producer and consumer. The new producer begins execution exactly as desired: it prompts for an input string. It needs no special coordination with the consumer because both producers are bound to the same interface on the consumer; the consumer need not know that two different modules are sending messages to it.

Because the replacement of a producer or consumer requires no synchronization or state capture/restoration, it is not surprising that replicating the producer is correspondingly simple. However, replicating the consumer is not trivial, because the semantics of this reconfiguration are ambiguous. The producer's messages could be broadcast to both consumers, or they could alternate between the consumers, or they could be sent randomly to one of the two, etc.

4.2 The Uninvited Diner

Next we examine a structural change that does require synchronization and state capture/ restoration, although this is accomplished in a very different way than for the module replacement examples. In the Introduction, the example we gave of a structural change was to add a diner to the Dining Philosophers application. This is done by creating a new diner, binding it into the application, and giving it an appropriate initial state. One approach to initializing the new diner is to wait until its future neighbors reach some known state, then initialize the new diner accordingly. Our approach is instead to initialize the new diner with a composite of its two neighbors' states, as shown in Figure 4.2. The shaded portion of the initial application configuration (left) corresponds to the state we are capturing. This shaded portion is duplicated to arrive at the final configuration (right). The advantage of this approach is that the new diner can be added

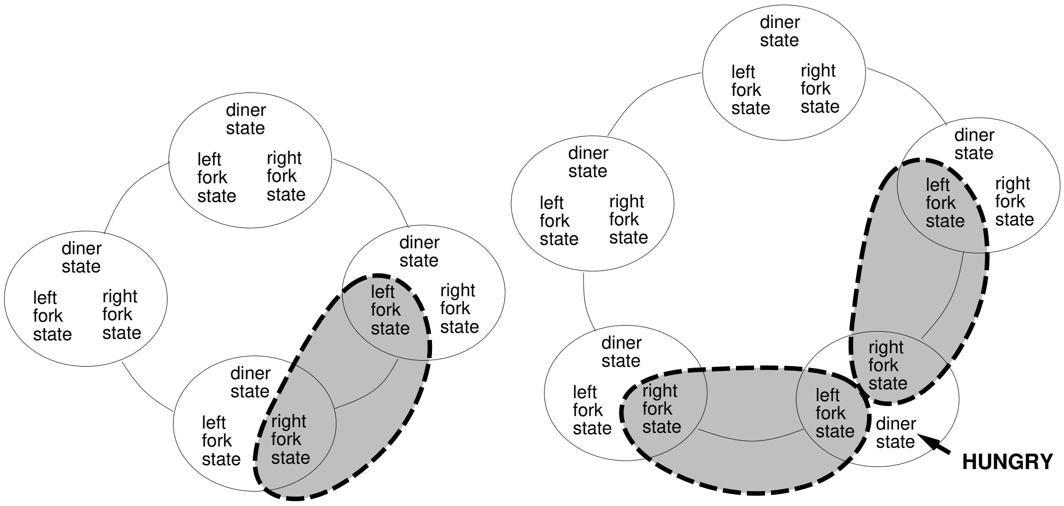


Figure 4.2: Adding an Uninvited Diner.

immediately, without waiting for the application to reach some predetermined state.

The sequence of events in this reconfiguration scenario is shown in Figure 4.3. The new diner is the same as the original diners, except for its `NAME` and `STATUS` attributes. In this reconfiguration we use the state of two modules plus the state of the binding between them to initialize the new diner; this composite state must be consistent, meaning that it must reflect a correct application state. When the `mh_objstate_move` primitive is invoked for each neighbor, there is no guarantee that the two neighbors will divulge their states at the same time. Thus we must freeze a portion of the application before calling `mh_objstate_move`, and release it only after the two neighbors have divulged their states. The `mh_edit_hold` is applied to the two interfaces within the shaded portion of Figure 4.2; this prevents the two neighbors from making any further communication until these interfaces are released. In addition to removing the existing (bi-directional) binding and adding two new ones, the binding changes include copying queued messages to the appropriate interface of the new diner.

This reconfiguration scenario presents an approach to capturing and restoring process state that is distinctly different from the case where a diner is being replaced. For replacement, we capture and restore the full state of the module, including the program counter. In order to add a diner, we capture the partial state of two different modules, and do not capture the program counter. But in initializing the new module, we use the two partial states and appropriate default values to create a composite state. Figure 4.4 shows what is added to `diner.c` and `verbose_diner.c` in order to support state capture and restoration for this reconfiguration scenario.

During reconfiguration, the

```
mh_obj_cap (&new,left_neighbor);                    /* create the new diner */
mh_edit_objattr (&new,"add","NAME",newname);
mh_edit_objattr (&new,"add","STATUS","composite");

mh_hold_cap (&hcap,NULL);                                 /* hold neighbors */
mh_edit_hold (&hcap,NULL,left_neighbor,"right");
mh_edit_hold (&hcap,NULL,right_neighbor,"left");
mh_hold (&hcap);
                                                       /* bind in new diner */
mh_bind_cap (&bcap,NULL);
mh_edit_bind (&bcap,"del",left_neighbor,"right", right_neighbor,"left");
mh_edit_bind (&bcap,"del",right_neighbor,"left", left_neighbor,"right");
mh_edit_bind (&bcap,"add",left_neighbor,"right", &new,"left");
mh_edit_bind (&bcap,"add",&new,"left",          left_neighbor,"right");
mh_edit_bind (&bcap,"add",&new,"right",         right_neighbor,"left");
mh_edit_bind (&bcap,"add",right_neighbor,"left", &new,"right");
mh_edit_bind (&bcap,"cpo",right_neighbor,"left", &new,"left");
mh_edit_bind (&bcap,"cpo",left_neighbor,"right", &new,"right");
mh_rebind (&bcap);
                      /* get state from neighbors and send it to new diner */
mh_objstate_move (left_neighbor,"right_fork_state",&new,"right_fork_state");
mh_objstate_move (right_neighbor,"left_fork_state",&new,"left_fork_state");

mh_chg_obj(&new,"add");                               /* start up new diner */
mh_rlse (&hcap);                                       /* release neighbors */
```

Figure 4.3: Reconfiguration Events for the Uninvited Diner.

```
initialize diner state to HUNGRY;
initialize left fork state;
initialize right fork state;

catch_reconfig() {
    send left fork state on interface left_fork_state;
    send right fork state on interface right_fork_state;
}

main() {
    if (status is special) set initial values so that graph is acyclic;
    else if (status is clone) {
        receive left fork state on interface left_fork_state;
        receive right fork state on interface right_fork_state;
    }
    signal(SIGHUP,catch_reconfig);

    while (1) {

        update left fork state;

        update right fork state;

        if (HUNGRY and conditions are right) start EATING;
        else if (done EATING) start THINKING;
        else if (done THINKING) become HUNGRY;
    }
}
```

Figure 4.4: The Uninvited Diner: `diner.c`.

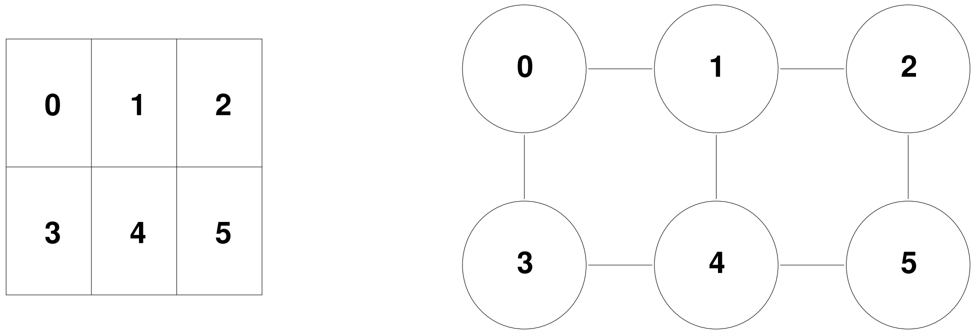


Figure 4.5: Jacobi Example with Six Partitions (left); Application Structure (right).

```
mh_objstate_move (&left_neighbor, "right_fork_state",
&new, "right_fork_state")
```

command (Figure 4.3) binds the two `right_fork_state` interfaces together, and signals the left neighbor to divulge its right fork state. A similar command is directed to the right neighbor. Upon receiving the signal, each diner sends its fork state immediately, then resumes normal execution. The new diner, with a `STATUS` of `clone`, begins by getting each fork state from the appropriate interface. Its initial diner state is defined to be `HUNGRY`, since this state is compatible with any combination of fork states. We know that mutual exclusion and fair allocation have been preserved because the new module's initial state is consistent with each of its neighbors' states, its diner state of `HUNGRY` is compatible with its fork states, and after initialization it follows the same protocol rules as all other diners.

4.3 Jacobi Relaxation

The third example of a structural change arises in a scientific application that uses a Jacobi relaxation technique for solving partial differential equations. The original sequential version is solved by initializing a large matrix, performing a computation on the matrix, then repeating this computation for a set number of iterations. This application is amenable to a distributed solution by dividing the matrix into partitions, each of which executes concurrently as a separate module. The modules are connected to each other in a grid pattern, and each module must swap boundary values with its neighbors at every iteration. For example, the distributed version with six partitions has six modules, named `"0"` through `"5"`, connected by the bindings shown in Figure 4.5. The reconfiguration activity we provide is to repartition the data dynamically, by gathering the data from all partitions, then dividing it into new partitions (Figure 4.6).

Each module executes the code shown in Figure 4.7. In each iteration there is a communicate phase followed by a compute phase, and the reconfiguration point (designated by the label `R`) is located at the compute phase. When a reconfiguration request is received by a module, it delays sending its state until execution reaches the reconfiguration point. Thus when the modules divulge

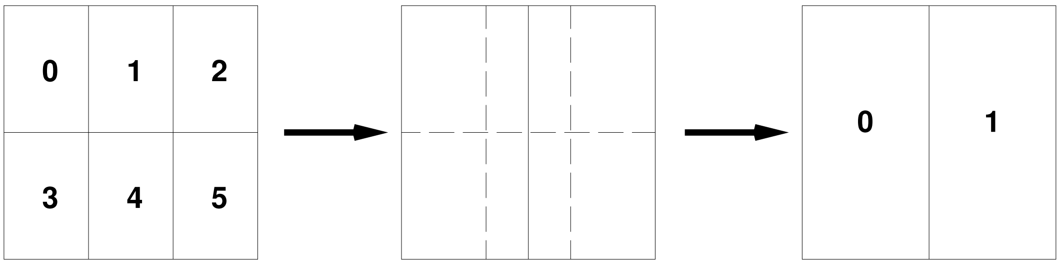


Figure 4.6: Repartitioning the Matrix.

```
main() {
   int i;

   initialize my partition;

   for (i=1; i<=numiter; i++) {

R:    send boundary values to all neighbors;
      receive boundary values from all neighbors;
      relax();
      compute_error();
   }
}
```

Figure 4.7: Pseudocode for the Jacobi module.

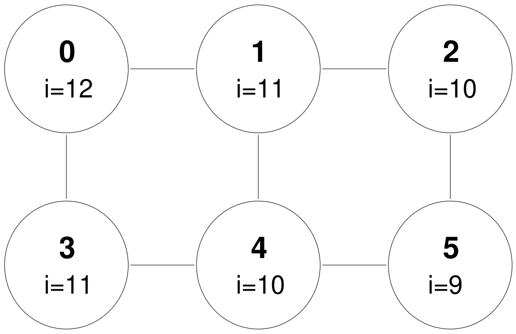


Figure 4.8: Possible Application State for Jacobi Relaxation.

their state for repartitioning, they will all be at the same place in the code, at the communication phase.

In all prior examples, delaying reconfiguration until the module reaches the reconfiguration point is sufficient for achieving a consistent application state, but this is not sufficient for the repartitioning example. In the Jacobi application, even though the modules communicate at each iteration, they are not guaranteed to be executing the same iteration upon reaching their respective reconfiguration points.

For example, with six partitions (two rows and three columns), the modules could be executing the iterations shown in Figure 4.8. This is the application state that would result from the following events, assuming that initially all modules are on the 9^{th} iteration, and all have sent values to their neighbors:

1. modules $0-4$ receive values in the 9^{th} iteration, execute the computation phase, move to the 10^{th} iteration and send values
2. modules $0, 1, 3$ receive values in the 10^{th} iteration, execute the computation phase, move to the 11^{th} iteration and send values
3. module 0 receives values from 1 and 3 in the 11^{th} iteration, executes the computation phase, moves to the 12^{th} iteration and sends values

Thus although neighbors can be at most one iteration apart, the iteration distances can accumulate.

If reconfiguration occurs when the original modules are on different iterations, then the composite matrix created from those modules is meaningless. All modules must be at the same iteration before repartitioning can take place. The reconfigurable state in this application is defined to be the state where all modules are at their reconfiguration points, and all modules are at the same iteration. When reconfiguration is requested, the modules must agree upon the iteration at which they will divulge their state, then continue normal processing until they reach this iteration.

```
int reconfig=0, restoring=0;
int global_i=-1, found_global_i=0;

catch_reconfig()  { reconfig = 1; }
encode() { send my partition on interface encode; }
decode() { receive my partition on interface decode; }

main()  {
   int i;

   if (status is clone) {
      decode();
      receive i on interface decode;
      signal (SIGHUP, catch_reconfig);
      goto R;
   }

   initialize my partition;
   signal(SIGHUP,catch_reconfig);

   for (i=1; i<=numiter; i++)  {

      if (reconfig) {
         if (!found_global_i) {
            send my id and i on interface reconfig;
            receive global_i and found_global_i on interface reconfig;
         }
         if (found_global_i) && (i==global_i) {
            reconfig = 0 ;
            encode() ;
            send i on interface encode;
         }
      }
R:    send boundary values to all neighbors;
      receive boundary values from all neighbors;
      relax();
      compute_error();
   }
}
```

Figure 4.9: Jacobi Module Adapted for Repartitioning.

Figure 4.9 shows the code for each module, adapted for reconfiguration. Once the module has received the reconfiguration signal, at every subsequent iteration it checks to see whether it has reached the reconfigurable state, where the reconfigurable state is defined as

```
(found_global_i) && (i == global_i)
```

Until the value for `global_i` has been determined, the module must advertise its own value of `i` at every iteration.

The reconfiguration events for repartitioning the matrix are shown in Figure 4.10. After all the original modules have been told to divulge their state, the value of `global_i` is determined by first getting the next iteration of every module, then setting `global_i` to the maximum of these values. The value of `global_i` is sent to all modules, which will divulge their state upon reaching the appropriate iteration. After the state of the old modules has been received, the matrix is divided into new partitions and sent to the new modules.

This application demonstrates another issue that can arise in reconfiguration: sometimes the reconfigurable state cannot be determined solely by a module's local state information. Global state information may be required in order for a module to determine its reconfigurable state. In this case, first the application must determine the appropriate global state at which to reconfigure, then each module must do its part in establishing that global state before moving to its reconfiguration point.

```
for (n=0; n < old_num_nodes; n++) {            /* get access to old nodes */
   mh_obj_cap (&(old_caps[n]), name[n]);
}
for (n=0; n < new_num_nodes; n++) {                   /* create new nodes */
   mh_obj_cap (&(new_caps[n]), name[0]);
   mh_edit_objattr (&(new_caps[n]), "add", "NAME", name[n]);
   mh_edit_objattr (&(new_caps[n]), "add", "MACHINE", site[n]);
   mh_edit_objattr (&(new_caps[n]), "add", "STATUS", "clone");
   mh_edit_objattr (&(new_caps[n]), "add", "CMDLINEARGS", new_args);
}
for (n=0; n < old_num_nodes; n++) {     /* tell old nodes to divulge state */
   mh_objstate_move (&(old_caps[n]),"encode","filter","decode");
}

   /* determine the appropriate value for global_i, send it to all modules */

global_i = -1;
found_global_i = false;
for (n=0; n < old_num_nodes; n++) next_i[n] = -1;
num_checked_in = 0;
while (num_checked_in < old_num_nodes) {
   receive node, i;
   if (next_i[node]==-1) num_checked_in++;
   next_i[node] = ++i; /* must do at least one more iteration */
   send global_i, found_global_i to &(old_caps[node]);
}
global_i = max (next_i, old_num_nodes);
found_global_i = true;
for (n=0; n < old_num_nodes; n++) {
   receive node, i;
   send global_i, found_global_i to &(old_caps[node]);
}

filter_in (N, old_rows, old_cols);           /* read state from old nodes */
for (n=0; n < old_num_nodes; n++) {                   /* remove old nodes */
   mh_chg_obj (&(old_caps[n]), "del");
}
config (N, new_rows, new_cols);           /* bind new nodes to each other */
for (n=0; n < new_num_nodes; n++) {                 /* start up new nodes */
   mh_chg_obj (&(new_caps[n]), "add");
}
filter_out (N, new_rows, new_cols);            /* send state to new nodes */
```

Figure 4.10: Reconfiguration Events for Repartitioning the Matrix.

Chapter 5

Summary

In extending the POLYLITH environment with reconfiguration primitives, we have corroborated the conclusions drawn in [23] about the basic activities that a dynamic reconfiguration environment must support. However, our reconfiguration environment supports modules written in standard procedural languages (such as C or Pascal), and the POLYLITH platform supports heterogeneous languages on heterogeneous machines.

The main focus of this thesis is providing support for module participation during reconfiguration. To this end, we have first defined an approach to capturing and restoring global data (the **encode, decode** operations) that is supported by the reconfiguration primitives, and second, discovered a mechanism for capturing and restoring the activation record stack that is machine-independent. Given reconfiguration points specified by the programmer, we automatically place the capture and restore blocks needed. Within these capture/restore blocks, we know what data should be captured/restored, although we do not generate the code to capture/restore individual variables or data structures.

Another contribution of this work is exploring the extent of module participation required during dynamic reconfiguration. When replacing a module, module participation is limited to capturing and restoring the local state. When performing a structural change, modules may need access to global state information during dynamic reconfiguration.

It is not realistic to expect to support general dynamic reconfiguration without preparing the application beforehand (statically). This is in contrast to the approach taken for supporting process migration.

5.1 Performance Issues

There are several aspects of performance that arise in dynamic reconfiguration. The most important of these is how adapting an application for dynamic reconfiguration affects the normal execution of the application. The techniques described in Chapter 3 for adapting modules to participate during reconfiguration affect normal execution only through the capture and restore blocks inserted in the module. The contents of these blocks does not affect execution speed, because the blocks are entered only when dynamic reconfiguration occurs. Thus the capture and restore blocks affect execution speed only via the global flag used to guard a block.

As discussed in Section 3.3.4, the frequency with which these flags are tested depends on how many reconfiguration points are defined, and where they are located. A general guideline for placing reconfiguration points is to put them where they will be checked regularly. In the simulation application described in Section 3.4, we compared the execution time of the original program with the execution time of the version adapted for reconfiguration, and the performance impact was not detectable.

The placement of the reconfiguration points also has an impact on the size of the process state that must be captured and restored. The set of variables that are live (those containing a value that may be needed in the future) at the reconfiguration point is a superset of the variables that must be captured and restored. However, a reasonable approximation is to capture all global variables plus the local variables declared in procedures that can reach the reconfiguration point. The size of the process state also affects the time it takes to perform dynamic reconfiguration, since this process state must be transmitted and installed in another module.

The length of time from initiation of dynamic reconfiguration until the reconfiguration is complete depends on the cost of performing the configuration-level activities, in addition to the time it takes for modules to move to a reconfigurable state and transmit their process state. In our implementation, each module is a heavy-weight process, so the cost of creating and deleting modules reduces to the cost of creating and killing processes. Bindings are logical entities mapped onto a single physical connection (a socket), so the cost of rebinding depends mainly on the number of modules involved, not the number of logical bindings between modules. In addition, many of these configuration-level activities can occur while the modules are moving to a reconfigurable state and divulging their process state.

5.2 Open Problems

Before concluding, we describe some of the open problems in dynamic reconfiguration. One of these is the problem of automatically determining what type of reconfiguration activity should be installed in an application, and what conditions should trigger the reconfiguration. An example of this is determining when a module should be moved and where it should be placed in order to support load balancing.

Another issue outside the scope of this thesis is determining what the semantics of the application require for a reconfigurable state. We have shown an example where each module needed global state information in order to establish a reconfigurable state (the Jacobi relaxation example). For other applications, moving to the reconfiguration point is sufficient, but the location of the reconfiguration point must be determined. Other applications can lie between these two extremes, perhaps needing the cooperation of one other module in order to establish the reconfigurable state.

Although we have developed a tool that performs all application-level activities for replacing a module in any application (the CATALYST), we have not attempted to provide a similar tool for performing arbitrary reconfiguration activities, such as changing the structure of the application in some way. Thus another open problem is finding a general way to support structural changes in which module participation requires access to the global state.

It is our conviction that much remains to be learned by applying existing dynamic reconfiguration techniques and approaches to real applications. This experience is essential for the areas of research we have just described.

In order to study module participation during dynamic reconfiguration, we use a statically-scoped procedural language, `C`. Our reason for focusing on this type of language is the prevalence of statically-scoped procedural languages. Although the tools we developed would need to be adapted for other languages of this type, the techniques are directly applicable. For modules written in other types of programming languages, the approach to capturing and restoring persistent state (cf. global variables in `C`) carries over, but we would not expect the approach to capturing and restoring run-time structures to be directly applicable.

Given modules written in statically-scoped procedural languages, there are several possible guidelines for placing reconfiguration points. One possibility is to use data-flow analysis to determine where the state (the set of live variables) is smallest. This would minimize the size of the state that is captured and restored, but does not take into consideration how frequently the reconfiguration points are checked. Control-flow analysis could be used to place reconfiguration points outside of loops, but control-flow and data-flow analysis are expensive to perform and current tools are not robust enough for analyzing arbitrary programs. More promising possibilities are to place reconfiguration points at procedure boundaries or at communication points, places where the module interacts with other modules. This approach takes advantage of the programmer's procedural and communication abstractions.

5.3 Conclusions

Dynamic reconfiguration is critically important for long-running applications or those that must be continuously available, in order to perform maintenance and to adapt the software to a changing environment. As the physical environment changes, an application with dynamic reconfiguration capabilities can make corresponding changes, in order to improve the performance of the application, or simply to maintain adequate functionality. Not all changes can be anticipated a priori, but when an application is equipped with general dynamic reconfiguration capabilities, it is provided with a mechanism for adding new reconfiguration capabilities.

In this work we have examined the module participation aspect of dynamic reconfiguration. We first developed a platform for experimentation, then took applications written in standard procedural languages and analyzed what was needed to support module participation. As a result, we have provided a structuring technique for capturing and restoring data, and provided a general technique for capturing and restoring state from anywhere in the program.

Bibliography

[1] A. Aho, R. Sethi, J. Ullman, "Run-Time Environments," **Compilers: Principles, Techniques, and Tools**, Addison-Welsey, Chapter 7, pp. 389-462, 1986.

[2] Y. Artsy, R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *IEEE Computer Magazine*, vol. 22, no. 9, pp. 47-56, September 1989.

[3] M. Bach, **The Design of the Unix Operating System**, Prentice-Hall, Chapters 6-7, pp. 146-246, 1986.

[4] A. Barak, A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software Practice and Experience*, vol. 15, no. 8, pp. 725-737, August 1985.

[5] M. Barbacci, D. Doubleday, C. Weinstock, "Application-Level Programming," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 458-465, 1990.

[6] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, R. Lichota, "Durra: A Structure Description Language for Developing Distributed Applications," *IEE Software Engineering Journal*, vol. 8, no. 2, pp. 83-94, March 1993.

[7] T. Bloom, M. Day, "Reconfiguration and Module Replacement in Argus: Theory and Practice," *IEE Software Engineering Journal*, vol. 8, no. 2, pp. 102-108, March 1993.

[8] J. Callahan, J. Purtilo, "A Packaging System for Heterogeneous Execution Environments," *IEEE Transactions on Software Engineering*, vol. 17, pp. 626-635, 1991.

[9] S. Carson, S. Setia, "Optimal Write Batch Size in Log-structured File Systems," to appear, **Computing Systems**. Currently available in *Proceedings of the USENIX Workshop on File Systems*, pp. 79-91, May 1992.

[10] K. Chandy, J. Misra, "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 4, pp. 632-646, October 1984.

[11] K. Chandy, C. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," *IEEE Transactions on Computers*, vol. C-21, no. 6, pp. 546-556, June 1972.

[12] F. Douglis, J. Ousterhout, "Process Migration in Sprite: A Status Report," *IEEE Computer Society Technical Committee on Operating Systems Newsletter*, vol. 3, no. 1, pp. 8-10, Winter 1989.

[13] F.B. Dubach, R. Rutherford, C. Shub, "Process-Oriented Migration in a Heterogeneous Environment," *Proceedings of the ACM Seventeenth Annual Computer Science Conference*, pp. 98-102, February, 1989.

[14] M.R. Eskicioglu, L.F. Carbrera, "Process Migration: An Annotated Bibliography," *IEEE Technical Committee on Operating Systems and Applications Environments Newsletter*, vol. 4, no. 4, pp. 5-16, Winter 1990.

[15] O. Frieder, M. Segal, "On Dynamically Updating a Computer Program: From Concept to Prototype," *Journal of Systems and Software*,vol. 14, pp. 111-128, 1991.

[16] M. Herlihy, B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 2, pp. 527-551, 1982.

[17] C. Hofmeister, J. Purtilo, "Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement," *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems*, pp. 101-110, May 1993.

[18] C. Hofmeister, E. White, J. Purtilo, "SURGEON: A Packager for Dynamically Reconfigurable Distributed Applications," *IEE Software Engineering Journal*, vol. 8, no. 2, pp. 95-101, March 1993.

[19] Y. Hollander, G. Silberman, "A Mechanism for the Migration of Tasks in Heterogeneous Distributed Processing Systems," *Proceedings of the International Conference on Parallel Processing and Applications*, pp. 93-98, September 1987.

[20] E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, vol. 6, no.1, pp. 109-133, February 1988.

[21] R. Koo, S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, January 1987.

[22] J. Kramer, J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 424-436, 1985.

[23] J. Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293-1306, 1990.

[24] S. Mullender, "Process Management in a Distributed Operating System," *Proceedings of the International Workshop on Experiences with Distributed Systems*, pp. 38-51, September 1987.

[25] M. Powell, B. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 110-119, November 1983.

[26] T. Pratt, **Programming Languages: Design and Implementation**, Prentice-Hall, Chapters 6-8, pp. 149-302, 1984.

[27] J. Purtilo, "The Polylith Software Toolbus," To appear, *ACM Transactions on Programming Languages and Systems*, January 1994. Also available as *University of Maryland CSD Technical Report 2469*, 1990.

[28] J. Purtilo, C. Hofmeister, "Dynamic Reconfiguration of Distributed Programs," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 560-571, 1991.

[29] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220-232, June 1975.

[30] D. Russell, "State Restoration in Systems of Communicating Processes," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 183-194, March 1980.

[31] M. Theimer, B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 18-25, 1991.

[32] M. Theimer, K. Lantz, D. Cheriton, " Preemptable Remote Execution Facilities for the V System," *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pp. 2-12, December 1985.

Appendix A

Polylith Reconfiguration Primitives

There are three categories of reconfiguration primitives: primitives for altering modules, primitives for altering bindings, and synchronization primitives. For these three categories, the approach is first to acquire a capability for making changes, then to perform one or more edits on this capability, and finally to apply the changes accumulated via this capability. In the last section of this appendix, we give operations that return information about the current configuration of the application.

A.1 Primitives for Altering Modules

Get capability to a new or existing object:

mh_obj_cap (&ocap, obj)

```
struct capability {
    int client;
    int object;
    int interface;
};

int
mh_obj_cap(cap, objname)
struct capability *cap;
char *objname;
```

The caller must declare `ocap` to be of type `capability`, and pass the address of this structure as the first parameter. The caller must preserve the value of `ocap` for the entire sequence of the initial call to `mh_obj_cap`, multiple calls to `mh_edit_...`, and the final call to `mh_chg_obj`.

For the second parameter, `obj`, the caller can pass a string containing the name of an existing object, or can set it to `NULL`. When `obj` contains `NULL`, the bus returns a capability to a new object; otherwise it returns a capability to the existing object.

The call to `mh_obj_cap` returns 0 if everything is ok, else it returns a negative value.

Edit the description of an object:

mh_edit_objattr, mh_edit_if, mh_edit_ifattr

These calls to `mh_edit_...` return 0 if everything is ok, else return a negative value.

Add or remove interfaces: mh_edit_if (&ocap, action, iface)

```
int
mh_edit_if(cap, typ, interface)
struct capability *cap;
char *typ, *interface;
```

Apply action `"add"` to create a new interface for module `&ocap`. The interface name is passed as a string in `iface`. Apply action `"del"` to remove interface `iface` from module `&ocap`.

Add, replace, or remove attributes and their values: mh_edit_objattr (&ocap, action, attrib, val)

```
int
mh_edit_objattr(cap, typ, attrib, value)
struct capability *cap;
char *typ;
char *attrib, *value;
```

Apply action `"add"` to insert or replace value of specified attribute for module `&ocap`. The parameter `attrib` is a string containing the name of the attribute, `val` is a string containing the value. Apply action `"del"` to remove specified attribute (value *and* attribute) from module `&ocap` (in this case `val` should be `NULL`).

NOTE:
Currently, when creating a new object with `mh_obj_cap (&ocap, NULL)`, the interfaces must be added with `mh_edit_if` before any object attributes can be specified with `mh_edit_objattr`.

Add, replace, or remove interface attributes: mh_edit_ifattr (&ocap, action, if, attrib, val)

```
int
mh_edit_ifattr(cap, typ, interface, attrib, value)
struct capability *cap;
char *typ, *interface;
char *attrib, *value;
```

Apply action `"add"` to insert or replace value of specified attribute for interface `if` (a string) of module `&ocap`. The parameter `attrib` is a string containing the name of the attribute, `val` is a string containing the value. Apply action `"del"` to remove specified attribute (value *and* attribute) from `if` interface of module `&ocap` (value passed should be `NULL`).

Add or remove module from the application:

mh_chg_obj (&ocap, action)

```
int
mh_chg_obj(cap, typ)
struct capability *cap;
char *typ;
```

The bus makes sure that this request was made by the same object/process that originally obtained capability to it. The `action` indicates whether the object is to be added or removed:

`"add"` Add module `&ocap` to the application. The object is started up by the bus, according to the description of `ocap` created by the preceding calls to `mh_obj_cap and mh_edit_...`. If the object is already running, no new object is started up; the request is ignored.

`"del"` Remove module `&ocap` from the application. The object is shutdown by the bus, and its description is erased. If the object description has been created but the object was never started up, the description is erased.

The call to `mh_chg_obj` returns 0 if everything is ok, else it returns a negative value.

A.2 Primitives for Altering Bindings

Get capability for rebinding interfaces:

mh_bind_cap (&bcap, applname)

```
int
mh_bind_cap(bp, applname)
int *bp;
char *applname;
```

The caller must declare `bcap` to be of type `integer`, and pass the address of this variable as the first parameter. The caller must preserve the value of `bcap` for the entire sequence of the initial call to `mh_bind_cap`, multiple calls to `mh_edit_bind`, and the final call to `mh_rebind`. Right now applications have only one application name, so if the `applname` parameter is left `NULL`, it is assumed to be the current application.

The call to `mh_bind_cap` returns 0 if everything is ok, else it returns a negative value.

Request a binding change:

mh_edit_bind (&bcap, action, &obj1, if1, &obj2, if2)

```
int
mh_edit_bind(bp, action, obj1, if1, obj2, if2)
int *bp;
struct capability *obj1, *obj2;
char *action, *if1, *if2;
```

Put a new entry in the `bcap` structure, recording action and interface(s). The actions available are:

`"add"` Add a new binding from `obj1 if1` to `obj2 if2`, overwriting any existing binding. Bi-directional bindings require an `"add"` for each direction.

`"del"` Delete binding for interface `obj1 if1` (`obj2 if2` must be specified, but may both be `NULL`).

`"cpq"` Copy messages queued for `obj1 if1` to `obj2 if2`. If no hold was applied to first iface, the whole queue gets copied. If hold was applied to first iface, then only messages that arrived after the hold get copied.

`"rmq"` Remove messages queued for `obj1 if1` (`obj2 if2` must be specified, but both may be `NULL`). If no hold was applied to the iface, the whole queue is deleted. If hold was applied, then only messages that arrived after the hold get deleted.

`"cpo"` Copy old messages queued for `obj1 if1` to `obj2 if2`. If a hold was applied to the first interface, only messages queued before the hold are copied. Thus in order to copy messages queued both before and after the hold, you must first call `mh_edit_bind(&bcap,"cpo",...)`, then call `mh_edit_bind(&bcap,"cpq",...)`. If no hold was applied, no messages are copied.

`"rmo"` Remove old messages queued for `obj1 if1` (`obj2 if2` must be specified, but both may be `NULL`). If a hold was applied to the first interface, only messages queued before the hold are deleted. If no hold was applied, no messages are deleted.

When the binding changes are applied (with `mh_rebind`), the interface names are resolved; they are not resolved with each `mh_edit_bind` command. Thus the state of the application at rebind time is not its state at the time of the `mh_edit_bind` command, but its state at the time of the `mh_rebind` command. There are three ways of specifying interfaces:

1. Put `NULL` for `&obj` and a string "*objname ifname*" for `if`.
2. Pass an object capability (from `mh_capability`) for `&obj` and a string "*ifname*" for `if`.
3. Pass an interface capability (from `mh_capability`) for `&obj` and put `NULL` for `if`.

The call to `mh_edit_bind` returns 0 if everything is ok, else it returns a negative value.

Apply the binding changes specified in `bcap`:

mh_rebind (&bcap)

```
int
mh_rebind(bp)
int *bp;
```

These take place atomically, from the application's point of view, since no new bus calls are processed during the `mh_rebind`. Although the rebinding is done atomically, acting on the application state at the time of the `mh_rebind` command, each `mh_edit_bind` command changes the application state for the subsequent `mh_edit_bind` (the edit commands are processed in the same order that they arrived).

The call to `mh_rebind` returns 0 if everything is ok, else it returns a negative value.

A.3 Synchronization Primitives

Get capability for holding objects/interfaces:

mh_hold_cap (&hcap, applname)

```
int
mh_hold_cap(hp, applname)
int *hp;
char *applname;
```

The caller must declare `hcap` to be of type `integer`, and pass the address of this variable as the first parameter. The caller must preserve the value of `hcap` for the entire sequence of the initial call to `mh_hold_cap`, multiple calls to `mh_edit_hold`, the call to `mh_hold`, and the final call to `mh_rlse`. Right now applications have only one application name, so if the `applname` parameter is left `NULL`, it is assumed to be the current application.

The call to `mh_bind_cap` returns 0 if everything is ok, else it returns a negative value.

Request hold on an object or interface:

mh_edit_hold (&hcap, action, obj, iface)

```
int
mh_edit_hold(hp, action, obj, if)
int *hp;
struct capability *obj;
char *action, *if;
```

Put a new entry in the `hcap` structure, recording action and object or interface. The symbolic name of the object or interface is stored, and this is resolved with the actual object when `mh_hold` or `mh_rlse` is applied. The `action` specifies whether the hold applies to an interface or an entire object:

`NULL` Hold interface `obj iface`. In addition to holding the interface, a `hold` token is placed in the queue, at the end of any messages that are queued (in transit) when the `mh_hold` command is applied. The `hold` token distinguishes between messages queued before the hold, and those that arrive during the hold. This is useful for module replacement, when we may want messages queued prior to the hold left at the old version, and messages arriving during the hold copied to the new version and removed from the old. (These messages can be copied to another interface or removed with `mh_edit_binding, "cpo", "cpq", "rmo", "rmq"`.) There are three ways of specifying interfaces:

1. Put `NULL` for `&obj` and a string "*objname ifname*" for `iface`.
2. Pass an object capability (from `mh_capability`) for `&obj` and a string "*ifname*" for `iface`.
3. Pass an interface capability (from `mh_capability`) for `&obj` and put `NULL` for `iface`.

`"obj"` Hold object specified by `obj` and `iface`. There are two ways of specifying objects:

1. Put `NULL` for `&obj` and a string "*objname*" for `iface`.
2. Pass an object or interface capability (from `mh_capability`) for `&obj` and put `NULL` for `iface`.

The call to `mh_edit_hold` returns 0 if everything is ok, else it returns a negative value.

Apply the holds specified in hcap:

mh_hold (&hcap)

```
int
mh_hold(hp)
int *hp;
```

These take place atomically, from the application's point of view. Once an object (interface) is held, all object-level (interface-level) requests are deferred until the object (interface) is released. The module initiating the request is blocked until the request has completed.

REQUEST TYPE	ON HELD OBJECT	ON HELD INTERFACE
mh_init	blocks	
mh_shutdown	blocks	
mh_readselect	blocks	ignores held iface
mh_readback		
mh_query_objmsgs	blocks	0 for held iface
mh_identity	blocks	
mh_query_objattr	blocks	
mh_query_objnames	blocks	
mh_capability (obj)	blocks	
mh_write (by capability)	blocks	
mh_write (by binding)		blocks
mh_read		blocks
mh_query_ifmsgs		blocks
mh_query_ifattr		blocks
mh_capability (iface)		blocks

NOTE: Currently, the `mh_write` does not block when `keepalive` is turned on. (It is on by default, and can be turned off with the `-K` run-time option.)

The call to `mh_hold` returns 0 if everything is ok, else it returns a negative value.

Release all holds specified in hcap:

mh_rlse (&hcap)

```
int
mh_rlse(hp)
int *hp;
```

Release all holds specified in `&hcap`, and resume processing of deferred requests. These take place atomically, from the application's point of view. An `mh_read` or `mh_readselect` that was not deferred but depends on a held interface is completed as the interface is released. After all objects and interfaces are released, the deferred transactions are resumed in the same order that they originally arrived. No new bus requests are accepted until these deferred transactions have been processed.

The call to `mh_rlse` returns 0 if everything is ok, else it returns a negative value.

Get and send module state:

mh_objstate_move (&ocap1, if1, &ocap2, if2)

```
int
mh_objstate_move(cap1, interface1, cap2, interface2)
struct capability *cap1, *cap2;
char *interface1, *interface2;
```

Induce module `&ocap1` to divulge its state via `if1`and forward the state to `&ocap2 if2`. Interfaces `if1` and `if2` need not be declared in the MIL program; the bus will add the interfaces to the object descriptions as necessary and bind `if1` to `if2`. Then the bus sends a `SIGHUP` signal to `ocap1`, which is responsible for providing a signal handler to accept the signal, package up its state, and send it out on `if1`. Object `ocap2` must be written so that the first thing it does upon startup is to read in its state from `if2`.

The call to `mh_objstate_move` returns 0 if everything is ok, else it returns a negative value.

A.4 Determine Current Configuration

Find all Interfaces of an Object:

mh_struct_objnames (&ocap, oname, ...)

```
int
mh_struct_objnames(obj_cap,obj_name,
                   response,rsize,if_caps,if_names,max_num_if,num_ifaces)
struct capability *obj_cap;
char *obj_name, *response;
int rsize;
struct capability *if_caps[];
char *if_names[];
int max_num_if, *num_ifaces;
```

Find all interfaces of the object specified by `obj_cap, obj_name`. The object is specified in one of two ways:

1. Put `NULL` for `&ocap` and a string "*objname*" for `oname`.
2. Pass an object or interface capability (from `mh_capability`) for `&ocap` and put `NULL` for `oname`.

The number of interfaces is returned in `num_ifaces`, and both the names of these interfaces and capabilities to them are returned: pointers to the interface names are placed in the array `if_names`, and pointers to the interface capabilities are placed in the array `if_caps`. The character buffer `response` provides space for the capabilities and names referenced by `if_names` and `if_caps`. The caller must provide the size of these two arrays in `max_num_if`, and the size of the `response` buffer in `rsize`.

The call to `mh_struct_objnames` returns 0 if everything is ok, else it returns a negative value.

Find what Interface is Bound to:

mh_struct_ifdest (&scap, sname, ...)

```
int
mh_struct_ifdest(source_cap,source_name,
                 response,rsize,if_caps,if_names,max_num_if,num_ifaces)
struct capability *source_cap;
char *source_name, *response;
int rsize;
struct capability *if_caps[];
char *if_names[];
int max_num_if, *num_ifaces;
```

Find what the source interface is bound to. The source interface is specified in one of three ways:

1. Put `NULL` for `&scap` and a string "*objname ifname*" for `sname`.
2. Pass an object capability (from `mh_capability`) for `&scap` and a string "*ifname*" for `sname`.
3. Pass an interface capability (from `mh_capability`) for `&scap` and put `NULL` for `sname`.

There is only one destination interface, so `num_ifaces` is returned with a value of 1, and both the name of this interface and capability to it are returned: a pointer to the interface name is placed in the array `if_names`, and a pointer to the interface capability is placed in the array `if_caps`. The character buffer `response` provides space for the capabilities and names referenced by `if_names` and `if_caps`. The caller must provide the size of these two arrays in `max_num_if`, and the size of the `response` buffer in `rsize`.

The call to `mh_struct_objnames` returns 0 if everything is ok, else it returns a negative value.

Find Everything Bound to an Interface:

mh_struct_ifsources (&dcap, dname, ...)

```
int
mh_struct_ifsources(dest_cap,dest_name,
                    response,rsize,if_caps,if_names,max_num_if,num_ifaces)
struct capability *dest_cap;
char *dest_name, *response;
int rsize;
struct capability *if_caps[];
char *if_names[];
int max_num_if, *num_ifaces;
```

Find everything bound to an interface (the destination interface). The destination interface is specified in one of three ways:

1. Put `NULL` for `&dcap` and a string "*objname ifname*" for `dname`.
2. Pass an object capability (from `mh_capability`) for `&dcap` and a string "*ifname*" for `dname`.
3. Pass an interface capability (from `mh_capability`) for `&dcap` and put `NULL` for `dname`.

The number of interfaces bound to the destination is returned in `num_ifaces`, and both the names of these interfaces and capabilities to them are returned: pointers to the interface names are placed in the array `if_names`, and pointers to the interface capabilities are placed in the array `if_caps`. The character buffer `response` provides space for the capabilities and names referenced by `if_names` and `if_caps`. The caller must provide the size of these two arrays in `max_num_if`, and the size of the `response` buffer in `rsize`.

The call to `mh_struct_objnames` returns 0 if everything is ok, else it returns a negative value.

Example

```
#include <stdio.h>
#define A_SIZE 10
char buf1[256], buf2[256];

struct capability {
     int client;
     int object;
     int interface;
};


main(argc,argv)
int argc;
char **argv;
{
     mh_init(&argc, &argv, NULL, NULL);

     show_binding_info("hello");
     show_binding_info("hi");
     show_binding_info("greetings");
     show_binding_info("duplicate");
     show_binding_info("print");
     show_binding_info("queries");

     mh_shutdown(0, 42, "");
}


show_binding_info(obj)
char *obj;
{    int i, j, num_ifaces, num_bindings;
     struct capability *if_caps[A_SIZE], *bind_caps[A_SIZE];
     char *if_names[A_SIZE], *bind_names[A_SIZE];
     struct capability objcap;
     char iface[80];

     if (mh_struct_objnames (NULL, obj, buf1,
           sizeof(buf1), if_caps, if_names, A_SIZE, &num_ifaces) == 0) {
          /*  Another way of achieving same result:
           *  mh_obj_cap (&objcap, obj);
           *  mh_struct_objnames (&objcap, NULL, ...
           */
          printf("%s's interfaces are:\n", obj);
          for (i=0; i<num_ifaces; i++) {
               printf("     %s\n", if_names[i]);
          }
          for (i=0; i<num_ifaces; i++) {
               if (mh_struct_ifdest (if_caps[i], NULL, buf2, sizeof(buf2),
                       bind_caps, bind_names, A_SIZE, &num_bindings) == 0) {
                    /*  Two other ways of achieving same result:
                     *  strcpy (iface, obj);
                     *  strcat (iface, " ");
```

```
             *  strcat (iface, if_names[i]);
             *  mh_struct_ifdest (NULL, iface, ...
             *          or
             *  mh_obj_cap (&objcap, obj);
             *  mh_struct_ifdest (&objcap, if_names[i], ...
             */
            printf("OUTGOING\n");
            printf("     \"%s %s\" bound to:\n", obj, if_names[i]);
            for (j=0; j<num_bindings; j++) {
                 printf("          \"%s\"\n", bind_names[j]);
            }
        }
        if (mh_struct_ifsources (if_caps[i], NULL, buf2, sizeof(buf2),
                 bind_caps, bind_names, A_SIZE, &num_bindings) == 0) {
            /*  Two other ways of achieving same result:
             *  strcpy (iface, obj);
             *  strcat (iface, " ");
             *  strcat (iface, if_names[i]);
             *  mh_struct_ifsources (NULL, iface, ...
             *          or
             *  mh_obj_cap (&objcap, obj);
             *  mh_struct_ifsources (&objcap, if_names[i], ...
             */
            printf("INCOMING\n");
            for (j=0; j<num_bindings; j++) {
                 printf("     \"%s\"\n", bind_names[j]);
            }
            printf("        bound to \"%s %s\"\n", obj, if_names[i]);
        }
    }
  }
}
```