A FRAMEWORK FOR DYNAMIC RECONFIGURATION OF DISTRIBUTED PROGRAMS

Christine R. Hofmeister   James M. Purtilo
Computer Science Department
University of Maryland, College Park, MD 20742

ABSTRACT

Current techniques for a software engineer to change a computer program are limited to static activities — once the application begins executing, there are few reliable ways to reconfigure it. We have developed a general framework for reconfigurating application software dynamically. A sound method for managing changes in a running program allows developers to perform maintenance activities without loss of the overall system's service. The same methods also support some forms of load balancing in a distributed system, and research in software fault tolerance. Our goal has been to create an environment for organizing and effecting software reconfiguration activities dynamically. First we present the overall framework within which reconfiguration is possible, then we describe our formal approach for programmers to capture the state of a process abstractly. Next, we describe our implementation of this method within an environment for experimenting with program reconfiguration. We conclude with a summary of the key research problems that we are continuing to pursue in this area.

This research was supported by a grant from the National Science Foundation, contract NSF CCR-9021222. An earlier and shorter version of this paper appeared as "Dynamic Reconfiguration of Distributed Programs," Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 560-571, 1991.

1 OVERVIEW

Capabilities for managing dynamic software reconfiguration — changes to the implementation of a running program — are increasingly in demand. Users of highly-available systems must perform maintenance on software components in-place; managers may discover the need to instrument some application only after it has been placed in operation; and both users and managers alike may desire to relocate parts of a running program in order to improve its performance (e.g., the task could be relocated from a local workstation to a remote supercomputer when executing the computationally demanding portions of a program.) Whereas techniques for *static* control of application programs have been available for years — under the software engineering label *configuration management* — dynamic techniques have not been widely addressed.

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 to manage three general types of changes:

1. **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 at run time; 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.

2. **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.

3. **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.

Our research provides a coherent framework for considering all three forms of reconfiguration in the presence of heterogeneity, as would be required for the sample applications cited above. First, we motivate the various forms of dynamic reconfiguration that programmers need, and describe other work towards providing such capabilities. Then we describe our approach to solving a key subproblem, that of capturing the state of an executing task so that it may be re-established elsewhere or in other forms. Our prototype environment for demonstrating and experimenting with dynamic reconfiguration is then described, after which we conclude with a summary of the additional research problems that we continue to pursue.

2 MOTIVATION

This section presents a concrete example to motivate the reconfiguration problem. The example, a distributed version of the well-known dining philosophers problem, will help us describe the requirements for a dynamic reconfiguration system, and also describe the many scientific problems that must be solved in order to obtain the benefits of reconfiguration.

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, 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 [4]. The details of this algorithm are not critical to our purpose here, so we show only the pseudo-code for a diner in Figure 2. Our original example has four diners, each a separate process, passing forks and requests for forks on bindings between two diners (Figure 1, right). Because the algorithm is decentralized, the protocol for sharing forks is contained in each diner, and is based entirely on its own local state.

We illustrate this problem in terms of an existing distributed programming system, POLYLITH [15]. In order to run this example on a heterogeneous network using POLYLITH, the user needs to provide a simple description of the application's modular structure, in terms of a module interconnection language (MIL). Once that is done, POLYLITH is responsible for packaging and invoking processes, and for coercing data representation, synchronization, and marshalling of data during communication.

Figure 3 shows the MIL declaration necessary for the user to implement this distributed application. By providing this text to the POLYLITH packaging system, the user's C source files would be accessed and compiled, then linked with automatically generated network stubs (i.e., procedures that intercept the call in the local process and perform a remote procedure call through the network; this activity is described in detail in [15].) The user could

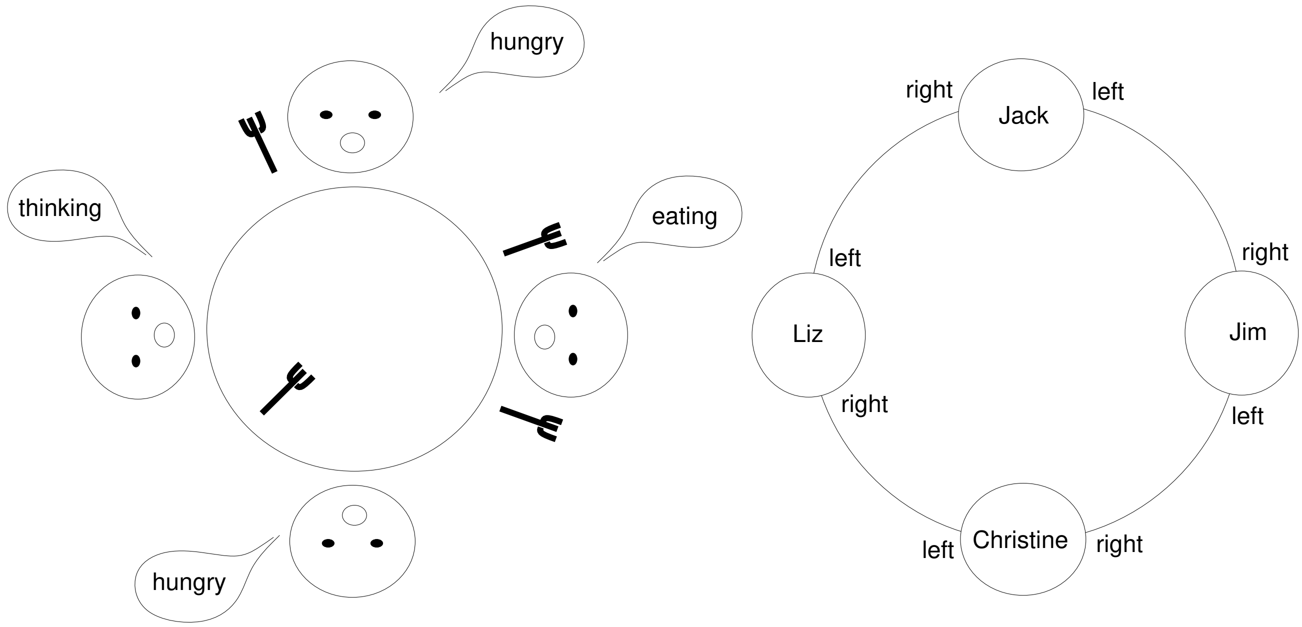

Figure 1: The Dining Philosopher problem.

then directly execute this application, as POLYLITH is responsible for invoking the executables and establishing a communication channel between the tasks. All the user sees is that the application works 'as expected.'

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

1. **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.

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

3. **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.

Throughout these changes, the user should see no interruption of service — the program must continue to 'meet spec' except possibly for timing constraints, which we do not address in our research at this time.

Kramer and Magee describe a formalism that characterizes precisely when a distributed program may be reconfigured, and in what way; furthermore, they describe an experimental implementation in Conic [11]. Their approach focuses upon changes that are primarily either `creation` or `deletion` of nodes, plus connection establishment and removal between those nodes. Their work is compelling, and our research is influenced by it as we focus upon the next questions: how can persistent state contained within the process be exposed for transmission to another

```
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 2: Pseudo-code for `diner.c`.

```
service "diner" : {
    implementation : {  binary : "/world/Users/crh/diner.out" }
    algebra : { "STATUS=($S)" }
    client "left" : { string } returns { 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 3: MIL declaration for Dining Philosophers.

process? At what points during a program's execution can its state be reliably captured for later restoration? And how can this all proceed transparent to source programs, written in arbitrary languages? Section 3 will begin to address these questions.

3 RECONFIGURATION FRAMEWORK

Our objective is to provide a robust framework for dynamically reconfiguring a distributed application, even when the execution environment is itself diverse and heterogeneous. We focus on mechanisms that are *external* to the application program, not internal; that is, we are interested in changing the application based on requests from outside the currently-executing program, whether initiated by the user or another program. Focusing on internal mechanisms would be too restrictive, in that all possible future configurations would have to be anticipated and represented in the initial software source, hence denying us from incorporating new software components that did not exist at initiation time. (A good example of a system that provides only internal reconfiguration is NIL, with its later implementation called Hermes [17].)

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:

- Users need an easy way to configure and invoke a (possibly distributed) application.
- Users must have a notation for identifying the program components or attributes that they wish to reconfigure. They must be able to name both individual modules and aggregates of modules composed into a structure.
- Users must be able to visualize the current state and geometry of a running program. There can be no reliable way for users to reconfigure a program if they do not understand what processes are currently being employed and where they are running.
- 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.
- The execution environment must ensure programmers 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.
- Similarly, any reconfiguration mechanism in the execution environment must ensure that *all* information characterizing a process is captured and represented. This includes 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. The ideal behavior would be for all such kernel-based information to be adapted during migration, transparent to the application's execution (except for possible differences in performance). For homogeneous distributed systems, then there is strong evidence from other projects (such as Charlotte [1]) this ideal can be achieved. However, this objective is unlikely to be met in highly diverse distributed systems, especially when the developer is not given the freedom to adapt the operating system — our objective is to provide reconfiguration *without* requiring modification of the underlying operating systems.
- The execution environment needs a way to mark some of the processes as *non-relocatable*, recognizing that some modules must necessarily act as guards to private resources. For example, access to the file system would most reasonably be handled by incorporating one non-relocatable process. It will still be possible to re*place* such a guard, but only when the developer is able to design the module so that it can later be updated; only the designer can make decisions about how to re-establish, say, access to a file that might have been changed externally during the reconfiguration step.

Our approach to meeting the above requirements is to build upon the existing POLYLITH software interconnection system [15]. POLYLITH already 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 starting point for investigating how applications might later be reconfigured.

The POLYLITH bus organization satisfies our requirements concerning coercion of data's representation in a heterogeneous system. The bus already manages data transformation during normal communication; therefore, by showing how to capture the state of an executing process into a reasonable data structure (by techniques to be discussed), then this same coercion mechanism serves equally well in the relocation of process state to other hosts.

The bus abstraction also helps us 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 incarnation of some module has been invoked, the bus will simply direct subsequent communication to the new version, abandoning the old version. It is possible for programmers to devise an application that defeats this principle, but one must try very hard to do so.

All requirements for a reconfiguration environment that have been discussed so far can be met by extending the POLYLITH interconnection system. However, our remaining requirement is by no means the least: how to characterize the state of an executing process so that it may be either altered or relocated? Moreover, how can we provide this capability at the minimum cost to programmers? Can it even be provided completely transparent to the application source code? Can it be provided without loss of run-time performance? The first of these questions is addressed in Section 4, where we describe the method abstractly. The latter questions can only be addressed experimentally, which is why we have constructed a set of extensions the POLYLITH software interconnection system. These extensions provide a workbench for us to build and study the reconfiguration of distributed applications.

4 ADT FORMULATION OF PROCESS

Our approach to reconfiguration of individual processes is based on formulating them in terms of abstract data types (ADTs): reconfiguration of a software process is performed using an abstract characterization of the component, to be captured at run time. This idea contrasts with previous approaches to migration in homogeneous systems [1, 5], because those methods relocate a process by moving its actual representation in the operating system, not an abstraction. The actual representation is architecture-dependent, and for this reason these approaches do not directly apply to a heterogeneous computing system. The only object of study in previous work is the binary representation of a process; moreover, there is no framework available for users to even *name* a component that they wish to be reconfigured. In contrast, our approach is based on having a way to extract the abstract state of a process independent of its host architecture. This abstraction can then guide the subsequent invocation of a comparable implementation of that task.

The problem then is to find how to characterize the process state abstractly at run-time. To accomplish this we use a generalization of the approach to transmission of ADTs that was presented in [8]. In Herlihy's work, two new operations, *encode* and *decode*, are added to the ADT, and the developer provides a suitable implementation of these operations for each host. When the ADT is to be transmitted, these new accessors are used by the system to extract the internal state of the data type into an external representation that can be shared among all valid implementations of the data type.

Such a transmission scheme is effective for the usual formulations of ADTs found in most applications. However, it alone is not sufficient for our use in reconfiguration. Each instance of an ADT that we wish to transmit is not a passive datum to be operated on by an application at its leisure — the process has a thread of control and will change its state rapidly. Worse yet, the necessary state information is not contained just within the executable image, but rather is cached on its behalf of the process within the CPU registers, program counters and many OS data structures. The ADT transmission scheme must be generalized to account for this dispersed process state.

In our approach to modeling processes as instances of a process ADT, each source module defines an 'abstract type,' and each executing process corresponds to an implementation of that type. The process run-time structures characterize the value of that instance, and therefore the state can be extracted at execution time via a suitable representation function. Reconfiguration begins when some agent within the application framework stops normal activity and causes a process to invoke its representation function, divulging a characterization of its state in an external format. This can be used by an *inverse* representation function to parameterize the invocation of any other valid implementation of that same ADT.

For purposes of this paper, programmers must provide representation functions for modules manually, and all of our examples of the use of our enhanced POLYLITH system are presented as such. We now present details concerning the environment we have constructed for experimenting with reconfiguration. First, Section 4.1 describes

PRIMITIVES FOR SYNCHRONIZING RECONFIGURATION

`mh_hold_cap (&hcap,applname)`	Get capability for holding interfaces and/or objects in application `applname`
`mh_edit_hold (&hcap,NULL,obj,iface)`	Hold interface `iface` of module `obj`
`mh_edit_hold (&hcap,NULL,obj,NULL)`	Hold module `obj`
`mh_hold (&hcap)`	Apply all holds specified in `&hcap`
`mh_rlse (&hcap)`	Release all holds specified in `&hcap`

PRIMITIVES FOR ALTERING MODULES

`mh_obj_cap (&ocap,obj)`	Get capability to module `obj`
`mh_obj_cap (&ocap,NULL)`	Get capability to a new module
`mh_edit_objattr (&ocap,"add",attrib,val)`	Insert or replace value of specified attribute for module `&ocap`
`mh_edit_objattr (&ocap,"del",attrib,NULL)`	Remove specified attribute from module `&ocap`
`mh_edit_if (&ocap,"add",iface)`	Add specified interface to module `&ocap`
`mh_edit_if (&ocap,"del",iface)`	Remove specified interface from module `&ocap`
`mh_edit_ifattr (&ocap,"add",if,attrib,val)`	Add or replace value of specified attribute for interface `if` of module `&ocap`
`mh_edit_ifattr (&ocap,"del",if,attrib,NULL)`	Remove specified attribute from interface `if` of module `&ocap`
`mh_chg_obj (&ocap,"add")`	Add module `&ocap`
`mh_chg_obj (&ocap,"del")`	Remove module `&ocap`
`mh_objstate_move (&ocap1,if1,&ocap2,if2)`	Induce module `&ocap1` to divulge its state via `if1`; forward it to `&ocap2 if2`

PRIMITIVES FOR ALTERING BINDINGS

`mh_bind_cap (&bcap,applname)`	Get capability for altering bindings in application `applname`
`mh_edit_bind (&bcap,"add",obj1,if1,obj2,if2)`	Add a new binding between interfaces `obj1 if1` and `obj2 if2`
`mh_edit_bind (&bcap,"del",obj1,if1,obj2,if2)`	Delete binding between interfaces `obj1 if1` and `obj2 if2`
`mh_edit_bind (&bcap,"cpo",obj1,if1,obj2,if2)`	Copy messages queued for interface `obj1 if1` to interface `obj2 if2`
`mh_rebind (&bcap)`	Apply all binding changes specified in `&bcap`

Figure 4: Polylith Reconfiguration Primitives

the extensions to POLYLITH needed to support our experimental activities. Then Section 4.2 describes the use of these extensions for the ADT framework portrayed above.

4.1 RECONFIGURATION PRIMITIVES

The extensions to POLYLITH were intended to support experimentation with reconfiguration tasks. They allow us to suspend communication between modules during reconfiguration, alter the structure of the application, and transfer state information from one module to another. The reconfiguration can be initiated by any module of the application, or by a third party. All reconfiguration changes are accomplished by invoking a series of POLYLITH primitives; these are described in Figure 4. The three groups of reconfiguration primitives use the same approach to applying changes: first get a *capability* for applying the change (`mh_hold_cap`, for example), next make a series of edits to describe the change (`mh_edit_hold`), then apply the change atomically (`mh_hold`).

The first group of primitives 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. An additional parameter to `mh_edit_hold` (which we do not describe here) indicates whether unread messages will be moved to another interface.

Purely structural changes (adding or deleting modules, and changing bindings) can be done without any support

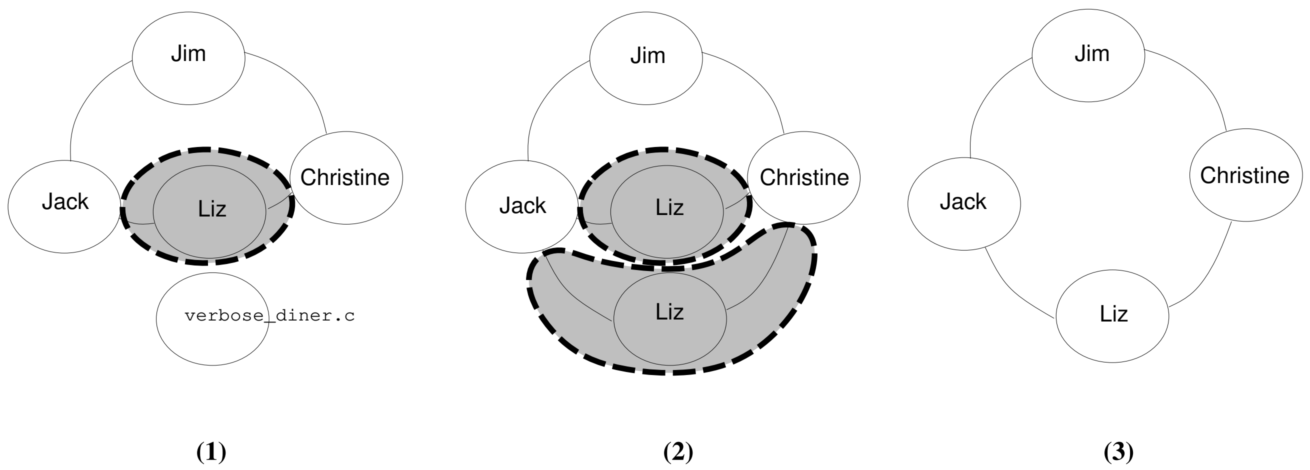

Figure 5: Replacing a diner with a verbose diner.

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. The `mh_objstate_move` command induces the old module to encode its state, then manages the transfer of state from old to new. Because POLYLITH controls the application configuration, it manages the application-level changes of creating, moving, or removing modules and adjusting bindings between them. Thus the modules need only local knowledge of their own behavior, and have no global knowledge of any other module in the application.

Module Implementations. In Section 2 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 5).

The reconfiguration events, shown in Figure 6, begin with acquiring access to the old diner and creating a new diner. The `BINARY` attribute specifies the implementation of the new diner as a verbose diner, and the `STATUS` attribute indicates how the new diner should initialize its state.

Next the old diner is told to divulge its state on interface `encode`. It complies 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. The binding changes are described with a series of `mh_edit_bind` commands, and applied atomically with the `mh_rebind` command.

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.

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 up. The sequence of events for this example may look daunting, but the task of replacing a module by one with the same interfaces can be standardized, and we have written a generic replacement routine that takes care of all these details, requiring only the names of the module and the new implementation. We have

```
mh_obj_cap (&old,diner);                                 /* get access to old diner and create new */
mh_obj_cap (&new,diner);
mh_edit_objattr (&new,"add","BINARY","verbose_diner.out");
mh_edit_objattr (&new,"add","STATUS","clone");
                                                      /* get state from old diner and send it to new */
mh_objstate_move (&old,"encode",&new,"decode");
                                                               /* remove bindings for old diner */
mh_bind_cap (&bcap,NULL);
mh_edit_bind (&bcap,"del",&old,"right",  right_neighbor,"left");
mh_edit_bind (&bcap,"del",right_neighbor,"left",&old,"right");
mh_edit_bind (&bcap,"del",&old,"left",   left_neighbor,"right");
mh_edit_bind (&bcap,"del",left_neighbor, "right",&old,"left");
                                                                  /* add bindings for new diner */
mh_edit_bind (&bcap,"add",&new,"right",  right_neighbor,"left");
mh_edit_bind (&bcap,"add",right_neighbor,"left",&new,"right");
mh_edit_bind (&bcap,"add",&new,"left",   left_neighbor,"right");
mh_edit_bind (&bcap,"add",left_neighbor, "right",&new,"left");
                                                                    /* copy messages in transit */
mh_edit_bind (&bcap,"cpo",&old,"left",   &new,"left");
mh_edit_bind (&bcap,"cpo",&old,"right",  &new,"right");
mh_rebind (&bcap);
                                                         /* start up new diner and remove old */
mh_chg_obj (&new,"add");
mh_chg_obj (&old,"del");
```

Figure 6: Reconfiguration events for replacing a diner.

not yet discussed the old and new diners' participation in this replacement scenario; the details of capturing and restoring their process state are given in Section 4.2.

Structure. The example we gave in Section 2 of a structural change was to add a diner to the 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 7. 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 immediately, without waiting for the application to reach some predetermined state.

The sequence of events in this reconfiguration scenario are shown in Figure 8. The new module looks just like the other diners, except for its `NAME` and `STATUS` attributes. In this example 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. The `mh_objstate_move` primitive must be invoked for each neighbor, and there is no guarantee that the two diners will divulge their state at the same time. Thus we must freeze that portion of the application by holding the affected interfaces as specified in the two `mh_edit_hold` commands, which are applied atomically when the `mh_hold` is invoked.

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. Whereas in our prior example applying the binding changes atomically was critical, in this example, because the interfaces are being held, the atomicity is not important. 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 all fork states, and after initialization it follows the same protocol rules as all other diners.

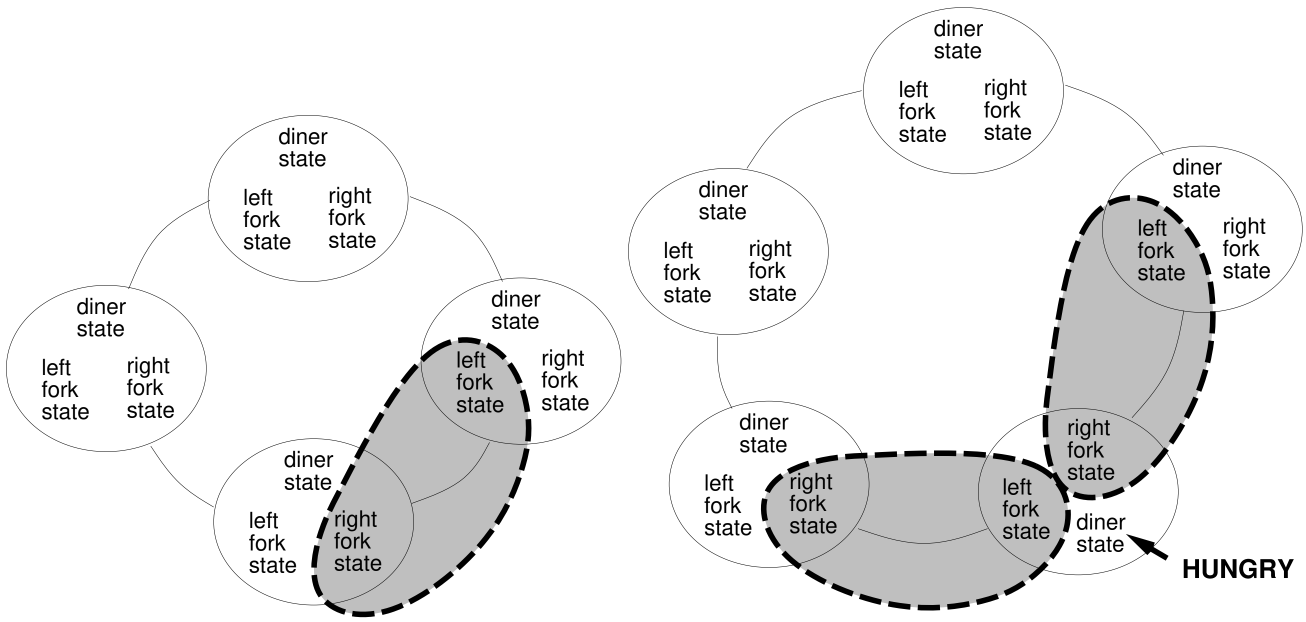

Figure 7: Adding a new dining philosopher.

```
mh_obj_cap (&new,left_neighbor);                                        /* create the new diner */
mh_edit_objattr (&new,"add","NAME",newname);
mh_edit_objattr (&new,"add","STATUS","composite");
                                       /* hold right side of left neighbor and left side of right neighbor */
mh_hold_cap (&hcap,NULL);
mh_edit_hold (&hcap,NULL,left_neighbor,"right");
mh_edit_hold (&hcap,NULL,right_neighbor,"left");
mh_hold (&hcap);
                                                         /* remove binding between left and right neighbors */
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");
                                                                          /* bind new diner to neighbors */
mh_edit_bind (&bcap,"add",left_neighbor,"right",newname,"left");
mh_edit_bind (&bcap,"add",newname,"left",left_neighbor,"right");
mh_edit_bind (&bcap,"add",newname,"right",right_neighbor,"left");
mh_edit_bind (&bcap,"add",right_neighbor,"left",newname,"right");
                                                         /* copy messages in transit when hold was applied */
mh_edit_bind (&bcap,"cpo",right_neighbor,"left",newname,"left");
mh_edit_bind (&bcap,"cpo",left_neighbor,"right",newname,"right");
mh_rebind (&bcap);
                                                       /* get state from neighbors and send it to new diner */
mh_objstate_move (left_neighbor,"right_fork_state",newname,"right_fork_state");
mh_objstate_move (right_neighbor,"left_fork_state",newname,"left_fork_state");
                                                                 /* start up new diner and release neighbors */
mh_chg_obj(&new,"add");
mh_rlse (&hcap);
```

Figure 8: Reconfiguration events for adding a diner.

Geometry. The third and final scenario described in Section 2 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 accomodate heterogeneity, handles all underlying details.

These first two reconfiguration scenarios demonstrate two different ways of synchronizing reconfiguration activities. In the first example (as well as the third), synchronization is accomplished by blocking the old module after it has divulged its state, then altering the binding and copying messages in transit atomically. In the second scenario, synchronization is achieved by holding the interfaces of the two modules that will divulge state information. This keeps the modules and the binding between them in the same state: each module is free to execute until it tries to communicate on the held interface, and any messages in transit when the hold occurred are copied to the new bindings.

4.2 CAPTURE/RESTORE PROCESS STATE

To support reconfiguration we must be able to characterize the state of an executing process and capture that state. Our ultimate goal of automatic capture of process state requires that we fill in the abstract representation of the process state without explicit help from the process. If we do not use semantic information about the application to selectively preserve only data that is relevant to the process state, then all data must be captured. This includes static variables from the data area, dynamic variables from the stack, programmer-allocated data from the heap, file descriptor and signal handler information stored by the operating system, and such things as process priority and cumulative cpu time.

The other major aspect of process state capture and restore deals with the execution thread. The first issue is determining when during execution we can capture sufficient state information to allow the process to restart; when is the process in a *reconfigurable state*. If the abstract state capture does not explicitly include the program counter, then the only reconfigurable states are program states where execution could safely resume at the beginning of the program. In this case the execution thread is captured implicitly, with an implicit value of 0 (the beginning of the program, for purposes of discussion here). A second issue is that when the process state includes the program counter, capturing and restoring the thread of execution may entail capturing and restoring the activation record stack, so that procedure/function returns and non-local data references can be handled correctly in the resumed process.

Kramer and Magee define a reconfigurable state as one in which all modules involved in the change are *quiescent*: they will not initiate any new communication, and have provided all services needed for other modules to reach their quiescent state [11]. They prove that this quiescent state is reachable for all modules involved in a reconfiguration. However, the communication between modules is limited to certain types of interactions, primarily rpc-type interactions. Because we do not restrict the types of interactions between modules, we cannot guarantee that in any application all modules will be able to reach a reconfigurable state. It is possible to write an application where a module would be prevented from reaching its reconfigurable state because it depended on interaction with another module already blocked in its reconfigurable state.

Our first two reconfiguration scenarios present distinctly different approaches to capturing and restoring state. In the first example, where the module is being replaced, we capture and restore the full state of the module, including the program counter. In the second example, we capture the partial state of two different modules, and do not capture the program counter. But in intializing the new module, we use the two partial states and appropriate default values to create a composite state.

Figure 9 shows what we add to `diner.c` and `verbose_diner.c` in order to support state capture and restoration for both reconfiguration scenarios. (When comparing this to Figure 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, our approach is to have the module provide *encode* and *decode* operations to capture and restore its own process state. Ultimately, these could be generated automatically when the module is compiled, but for now we rely on *encode* and *decode* operations provided by the programmer.

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

catch_reconfig() {
   if (left_fork_state is requested) send left fork state on interface left_fork_state;
   if (right_fork_state is requested) send right fork state on interface right_fork_state;
   if (encode is requested) reconfig_requested = 1;
}

main() {
   if (status is special) set initial values so that graph is acyclic;
   else if (status is composite) {
      receive left fork state on interface left_fork_state;
      receive right fork state on interface right_fork_state;
   }
   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 9: Reconfigurable version of `diner.c`.

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 has been prepared to receive that signal with procedure `catch_reconfig()` (Figure 9), and because the *encode* operation was requested, it turns on a 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. 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. 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, we don't bother with an explicit `goto` the end of the loop, we just allow execution to resume at the beginning of the loop.

In the second reconfiguration scenario, the `mh_objstate_move(&left_neighbor,"right_fork_state",newname, "right_fork_state")` command 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 `composite`, 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.

Our experiences to date are that use of the POLYLITH bus organization does not necessarily result in performance loss compared to a manually constructed version of the same distributed application. Using the POLYLITH reconfiguration techniques described here, the cost of replacing bindings is insignificant, and the cost of creating or deleting modules reduces to the cost of creating or deleting processes in the underlying operating system. For replacing a module, in addition to the creation/deletion cost incurred, there is a cost in capturing/transmitting/restoring process state, which is heavily dependent on the size and complexity of that state. In the reconfiguration scenarios presented in this paper, the abstract process state is fully described by a few boolean variables, so the cost of capturing, transmitting, and restoring the process state is negligible. It is important to note that the entire application need not be suspended for a reconfiguration; we can hold just the affected portion of the application, allowing the rest to proceed with its normal processing.

5 RELATED WORK

Only parts of this spectrum of capabilities have been addressed in the past. Geometric reconfiguration (but only between processors of like architecture and operating systems) has been considered in the form of process migration, e.g. [5, 1]. More recent research provides some reconfiguration of system structure, e.g. [3]. The most important previous work in this area is the formalism exposed within the Conic system [11].

Our approach is based upon the software bus abstraction as currently implemented in the POLYLITH system [15]. This project is related to a large body of previous technologies. Much work has been done in primitive data representation in the presence of heterogeneity. For example, our approach benefited from review of previous experiences with Courier. Sun Microsystem's XDR is a similar approach, as is UTS, a 'universal type system' internal to the MLP (Mixed Language Programming) system [7]. More abstractly, transmission of abstract data types (ADTs) is presented in [8]; Herlihy's ADT transmission mechanism inspired our work on capturing and transmitting the state of an executing process.

POLYLITH's previous focus was on simple data structures for interfaces. This stems from a design principle established early in the project, that any instance of a sufficiently rich data type deserves to be given its own module (and hence can be packaged in its own process space in appropriate environments). The POLYLITH language binds the instance's accessors into those modules using it, and thereafter those modules transact *capability to* that instance, rather than 'flattening' it for transmission. This approach is very similar to that shown in [9], where a *call by object-reference* method is described in detail.

Structure-oriented languages were used to control a distributed programming environment in several earlier projects, notably CLU [12] and MESA [19]. Both support distributed programming by coupling their notation with their supporting systems. Each of these systems represent a significant step forward in the area's ability to realize the vast potential of distributing a computation. Subsequently, Matchmaker [10] provided a transformational approach to the problem of integrating distributed components: an application would be written in a synthesis of, say, Pascal and a higher-level 'specification language.' This source would be transformed into ordinary Pascal code having accessors to the host communication system inserted explicitly, again for static control of distribution.

Especially appropriate for multiprocessor configurations are Camelot [2] (a transaction facility built on top of Mach) and Avalon (a language resource constructed using Camelot.) The V Kernel [5] implements a distributed- and parallel-programming resource appropriate for a homogeneous set of hosts. The HCS project [14] shows one way to provide a heterogeneous RPC capability in a distributed environment. Concert [20] and Marionette [18] are more variations on a theme. Several early projects emphasized a network filesystem approach (such as Locus [16].) An interesting approach to cross-architecture procedure call using a common backing-store is given by Essick [6]. Finally, the Durra system allows for some forms of dynamic reconfiguration within the Ada environment [3], while the Mercury system supports heterogeneity in applications by managing a networked object repository [13].

6 CONCLUSION

We have described a broad framework that organizes software reconfiguration activities, specifically within a distributed programming environment. In order to run experiments within this framework, we have constructed an execution environment containing a few, fundamental reconfiguration capabilities. This paper has exposed our overall approach; described our workbench for evaluating diverse, reconfigurable applications; and demonstrated its utility in sample programs. The collection of primitives given to programmers for utilizing our system represents a type of 'assembly language' for dynamic reconfiguration; like an assembly language, these primitives are quite flexible, but also perhaps best employed through automatic generation from more abstract declarations. Therefore, and as a result of our experiences with this system, we are continuing our research by investigating abstractions to better help programmers direct dynamic reconfigurations within our framework. In addition, we are studying techniques for automatically identifying and introducing the representation functions to extract process state during reconfiguration operations.

REFERENCES

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

[2] J. Bloch, "The Camelot Library: C Language Extension for Programming General Purpose Distributed Transaction System," *Proc of 9th Conf on Distributed Computing Systems*, pp. 172-180, 1989.

[3] M. Barbacci, D. Doubleday, C. Weinstock, J. Wing, "A Status Report on Durra: A Tool for PMS-level Programming," *Proceedings of 3rd Workshop on Large-grained Parallelism*, 1989.

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

[5] D. Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, pp. 314-333, 1988.

[6] R. Essick, *The Cross-architecture Procedure Call.* Ph.D. Thesis, UIUC Dept of Computer Science UIUC-R-87-1340, 1987.

[7] R. Hayes, S. Manweiler, R. Schlichting, "A Simple System for Constructing Distributed, Mixed-language Programs," *Software Practice and Experience*, vol. 18, no. 7, pp. 641-600, 1988.

[8] 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.

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

[10] M. Jones, R. Rashid, M. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Proc of 12th Symp on Principles of Prog Languages*, 1985.

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

[12] B. Liskov, R. Atkinson, *CLU Reference Manual*, Springer-Verlag LNCS 114, 1981.

[13] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, W. Weihl, "Communication in the Mercury System," *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, pp. 178-187, 1988.

[14] D. Notkin, A. Black, E. Lazowska et alia, "Interconnecting Heterogeneous Computer Systems," *Communications of the ACM*, vol. 31, no. 3, pp. 258-273, 1988.

[15] J. Purtilo, "The Polylith Software Toolbus," to appear, *ACM Transactions on Programming Languages and Systems*, currently available as *University of Maryland CSD Technical Report 2469*, 1990.

[16] G. Popek, B. Walker, J. Chow et alia, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proc of 9th Symp on Operating Systems Principles*, pp. 169-177, 1981.

[17] R. Strom, D. Bacon, "Hermes: A High-level Process-based Language for Reliable Distributed Computing," *Proceedings of 3rd Workshop on Large-grain Parallelism*, 1989.

[18] M. Sullivan, D. Anderson, "Marionette: A system for Parallel Distributed Programming using a Master/slave Model," *Proc of 9th Conf on Distributed Computing Systems*, pp. 181-189, 1989.

[19] R. Sweet, "The Mesa Programming Environment," *Proceedings of the ACM SIGPLAN Symposium on Programming Issues in Programming Environments*, pp. 216-229, 1985.

[20] S. Yemini, G. Goldszmidt et alia, "CONCERT: A High-level Language Approach to Heterogeneous Distributed Systems," *Proc of 9th Conf on Distributed Computing Systems*, pp. 162-171, 1989.

ACKNOWLEDGEMENT

We appreciate the guidance given to us by Rich LeBlanc. Also, we are grateful to Jeff Kramer and Jeff Magee, both for their helpful comments concerning this research and for cheerfully sharing their experiences with Conic.