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 reconfiguring applications dynamically, where developers may alter the application without loss of service. After presenting the overall framework within which reconfiguration is possible, we describe our formal approach for programmers to capture the state of a process abstractly. An environment to support experimentation with dynamic reconfiguration is then described in detail.

This research was initiated with funding from Office of Naval Research, contract N0014-90-J4091.

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. The components of this application structure may be distributed across a heterogeneous network. Within this 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.
main(argc,argv)
int argc;
char **argv;
{
    char word[256], *w;
    int len,i;
    char s[256] = "";

    printf("Enter a word: ");
    while (strcmp(gets(word),"") != 0)
    {
        create(s);
        len = strlen(word);
        w = word;
        for (i=0; i<len; i++) {
            push(s, w);
            w++;
        }
        printf("Reversed: ");
        for (i=0; i<len; i++) {
            char item; istack t = *s;
            item = t->item;
            *s = (*s)->next;
            printf("%c", item);
        }
        printf("\n\nEnter a word: ");
    }
}

Figure 1: Simple stack application: reverser module.

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. While a trivial application itself, this example 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.

Figures 1 and 2 contain C routines for an application that reverses a string using stack operations. The reverser routine requests a string as input, which is pushed on the stack, a character at a time. Then the reverser pops characters from the stack, printing them as they come off the stack. The routines for the stack operations are contained in one file, isolated from the reverser routine.

We will illustrate this problem in terms of an existing distributed programming system, POLYLITH [14]. In order to run this example on two hosts in 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
service "stack" : 
  implementation : { 
    binary : "/jteam/crh/stackl" 
    machine : "konky.cs.umd.edu" 
  } 
  function "create" : { } returns () 
  function "push" : { string } returns () 
  function "pop" : { } returns ( string ) 
  function "empty" : { } returns ( boolean ) 
  function "full" : { } returns ( boolean ) 
}

service "client" : 
  implementation : { 
    binary : "/jteam/crh/reverser" 
    machine : "flubber.cs.umd.edu" 
  } 
  client "create" : { } accepts () 
  client "push" : { string } accepts () 
  client "pop" : { } accepts ( string ) 
}

orchestrate "reverse" : 
  tool "stack" 
  tool "client" 
  bind "client create" "stack create" 
  bind "client push" "stack push" 
  bind "client pop" "stack pop" 

---

Figure 3: MIL declaration for stack application.

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 [14]). The user could then directly execute this application, as POLYLITH is responsible for invoking the executables (in this case on each of the hosts named konky and flubber, respectively), 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 would be to replace the reverser module with a new version that perhaps had a different user interface. If this change is requested while the reverser module is within the while loop waiting for the user to enter a word, the reconfiguration is easy to effect: the old process can be terminated, a new one initiated, and the stack module need never know the difference (although the user would see a duplicate prompt).

A more difficult module reconfiguration would be to replace the implementation of the stack module, which uses a linked list to represent the stack, with an implementation that uses an array representation. This module replacement is complicated by the fact that the stack data structure resides in the stack module, and we need the stack data to be preserved. If we restrict the replacement to occur only when the stack module is between operations, then the reconfiguration can occur without any effect on the reverser module, but we must somehow transmit the stack data to the new implementation. In general, process state information is not limited to a single data structure; the complete process state must be identified and captured. Moreover, in some cases, the process state information may need to be adapted during transmission in order for it to be suitable for the new implementation; these issues are considered in Section 4.2.

2. **Structure.** An example of how the structure of this application could be changed is if we decided to introduce a new module to transform the data as it is passed from the reverser to the stack (Figure 4). If the developer decides to change the running application so that it reverses and capitalizes the input string, we can define a new upcase module which receives a character, capitalizes it, and sends it out. The push interface of the reverser is bound to the input interface of the upcase module, and the output interface of upcase is bound to the push interface of the stack. As we describe in Section 4.1, the reconfig module initiates this reconfiguration, although the request could come from any module in the application, from a user interface, or even from the POLYLITH bus itself.

3. **Geometry.** An application’s geometry is the set of decisions concerning how modular structure is mapped onto available hosts for execution, plus the decisions concerning communication media. Therefore, an example of geometric reconfiguration would be to relocate the stack module from its original host to another host. In the case that both hosts are of like architecture and operating system, then the migration is a straightforward engineering operation. However, heterogeneity will defeat existing migration techniques.
To deal with this problem, we feel that the same technique proposed to deal with change of module implementation will also help us capture process state for the task’s relocation.

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 [lo]. 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 do programmers replace a node? How can persistent state contained within the process be exposed for transmission to a new version of the 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 [16].)

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 any of 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 some 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 replace 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 [14]. 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, 4], 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 [7]. 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 runtime 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 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 5. 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_held (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 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 \texttt{mh-obj-state-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.

\textbf{Structural Changes.} First we describe reconfiguration that is purely structural: no process state need be transferred from one module to another. Binding changes are described with a series of \texttt{mh-edit-bind} commands, and applied atomically with the \texttt{mh-rebind} command. Applying the binding changes atomically is important because if only a portion of the application has been held, messages could continue to arrive on bindings that are being deleted and rebound.

After making any necessary binding changes, we delete a module by invoking two POLYLITH primitives. We access a module by acquiring a capability to the module with the primitive \texttt{mh-obj-cap}. Then we invoke \texttt{mh-chg-obj}, passing the acquired capability and the option "del". We use a capability to access the module instead of naming it explicitly because during reconfiguration the module name may not be unique. In the case of module replacement, there may temporarily be two modules with the same name, the old version and the new.

Adding a module to an application is similar: we first acquire a capability to a new module, then edit that capability to describe the new module, then install the new module by invoking \texttt{mh-chg-obj} with the option "add". There are three primitives for editing a mod-
The initial structure of the application is shown in Figure 4: the reverser client and stack server are bound over the push, pop, and create interfaces, the stack's encode and decode interfaces are unbound, and there is a reconfig module floating in the application, with no interfaces or bindings. We want to leave the structure of the application the same, but replace the stack module with another implementation of stack, one that uses an array for its internal representation of the stack data.

Figure 8 shows the sequence of commands that bring about this module replacement. The reconfig module gets two capabilities to the stack module, old and new. Module old will be deleted, and module new is instantiated with “stack.array” before it is added. Upon receiving the old.object.move command, the bus sends a message to the encode interface of module old, causing it to begin executing its encode operation. When old has captured and packaged up its process state, it passes the state to the bus and suspends itself. The bus has meanwhile created the process for new, which starts by executing its decode operation. The bus passes the encoded state to the waiting decode operation, which decodes the process state and returns from its decode operation. Bindings between the reverser client and old are replaced by bindings between the client and module new. Finally, the old module is deleted, and all held interfaces are released. Although this module re-
/* internal representation of stack */
struct inode {
    char item;
    struct inode *next;
};
typedef struct inode *istack;

/* external representation of stack */
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;
    }
}

decode(i,s,x-s)
istack *i-s;
struct xstack *x-s;
{
    int i;
    create (i-s);
    for (i = x-s->size-1; i >= 0; i--)
        push (i-s, x-s->item[i]);
}

/* Italicized pseudo-code replaces source details that are not important for this example */
/* buffer for internal stack rep */
char i[256];
/* buffer for external stack rep */
char xstack_buf[256];
struct xstack xstk = { 0, xstack_buf };

main()
{
    if (replacement) {
        read xstk on "decode" interface
        decode(s, &xstk);
    }
    while (1){
        perform requested operation
        if (message at "encode" interface)
        {
            read on "encode" interface
            encode(s, &xstk);
            write xstk on "encode" interface
            read on "encode" interface
        }
    }
}

Figure 7: Stack encode and decode (left); POLYLITH interface for stack module (right).

placement required a long list of commands to remove and replace bindings, clearly they could be generated automatically in a case like this, where bindings are remapped by name.

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 state capture and restore does not include capture of 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 requires an explicit capture of the program counter, restoring the thread of execution entails not just updating the program counter, but restoring the activation record stack, so that procedure/function returns and non-local data references can be handled correctly in the resumed process.

Our initial implementation of reconfiguration uses the application's semantic information to simplify the re-
configuration obligations. To reconfigure the stack application described in the previous section, we wrote our own `encode` and `decode` operations. These operations capture and restore only the stack data structure; loop counters and other variables are not needed because we have carefully limited our reconfigurable states. Reconfiguration can occur only when the module is not servicing a `push`, `pop`, or other request for the client, so the stack module can be resumed without restoring the program counter and without reconstructing the activation record stack. With the stack module controlling when `encode` can be invoked, the reconfiguration is not asynchronous; the module accepts a reconfiguration signal only when it is in a reconfigurable state.

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 [10]. 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.

Because our `encode` and `decode` operations are capturing and restoring only the stack data structure, they look remarkably like the operations we would use to transmit an abstract data type according to the Herlihy and Liskov scheme. However, there are fundamental differences first in the functionality of the `encode` and `decode` operations, and second in the capturing and restoring of the execution thread.

The functionality of our `encode` and `decode` is to capture and restore the process state, not just the state of an ADT. In this case our process state happened to be equivalent to an ADT state, because the data in the stack ADT was sufficient to restore the process state. Other non-ADT modules (like the `reverser` client) will not have a process state that looks like an ADT, regardless of how the execution thread is captured.

In our `encode` and `decode` operations the execution thread is captured implicitly by controlling when the process state is captured. The program counter is not included in the process state because, whenever the `encode` operation is invoked, the value we want to restore to the program counter is 0 (restart at the beginning of the program). 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 every reconfigurable state; this is equivalent to allowing the encode messages to arrive asynchronously, with the stack module immediately moving to a reconfigurable state by finishing its current operation.

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. 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. [4, 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 [10].

Our approach is based upon the software bus abstraction as currently implemented in the POLYLITH system [14]. 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 [6]. More abstractly, transmission of abstract data types (ADTs) is presented in [7]. 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 [8], 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 [11] and MESA [18]. 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 [9] 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 [4] implements a distributed- and parallel-programming resource appropriate for a homogeneous set of hosts. The HCS project [13] shows one way to provide a heterogeneous RPC capability in a distributed environment. Concert [19] and Marionette [17] are more variations on a theme. Several early projects emphasized a network filesystem approach (such as Locus [15].) An interesting approach to cross-architecture procedure call using a common backing-store is given by Essick [5]. 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 [12].

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 exposes our overall approach, and describes our workbench for evaluating diverse, reconfigurable applications. In future papers we plan to provide a summary of more realistic applications built within our environment, which will include a characterization of the attainable performance. Our research is continuing, as we move towards automatic techniques for conditioning software to be reconfigurable within this framework.

REFERENCES


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.