THE POLYLITH SOFTWARE BUS

James M. Purtilo
Computer Science Department and Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

ABSTRACT

We describe a system called POLYLITH that helps programmers prepare and interconnect mixed-language software components for execution in heterogeneous environments. POLYLITH's principal benefit is that programmers are free to implement functional requirements separately from their treatment of interfacing requirements; this means that once an application has been developed for use in one execution environment (such as a distributed network) it can be adapted for reuse in other environments (such as a shared-memory multiprocessor) by automatic techniques. This flexibility is provided without loss of performance.

We accomplish this by creating a new run-time organization for software. An abstract decoupling agent, called the *software bus*, is introduced between the system components. Heterogeneity in language and architecture is accommodated since program units are prepared to interface directly to the bus, not to other program units. Programmers specify application structure in terms of a module interconnection language (MIL); POLYLITH uses this specification to guide *packaging* (static interfacing activities such as stub generation, source program adaptation, compilation and linking). At run time, an implementation of the bus abstraction may assist in message delivery, name service or system reconfiguration.

Classification: D.2.2, D.3.3.

1 INTRODUCTION

Software for distributed systems is often more complex and less portable than other software. Developers of such software should reasonably expect to cope with some additional complexity when their application necessarily employs multiple threads of control. But the cost of developing distributed programs remains high even when the application is fundamentally serial in nature (for instance, when the programmer only needs to make a remote procedure call to access a non-local resource.) Moreover, once a program has been developed for one type of network, then it is not easily ported for use in other execution environments.

Higher development costs result from programmers implementing many diverse requirements simultaneously in the same program unit. Not only must the programmer achieve the desired functionality, but he must also anticipate the program unit context of use: system calls must be installed for communicating with other program units, parameters must be marshaled for transmission, data representations may need to be coerced, and so on. Considering these issues simultaneously is a complex task that is expensive to perform.

Once a programmer pays the expense of constructing a distributed program unit, it is not widely reusable. To introduce network interfacing code is to introduce dependencies upon an environment that subsequently narrow the range of applications able to employ that program. Later use of that software, even within the same general application, requires adaptation — the system calls must be changed, the data structures may need to be marshaled differently, and name conflicts may need to be resolved. In other words, the cost (as measured in programmer time, testing obligations, and configuration management complexity) is increased.

The problem is clear: current software development techniques for diverse execution environments result in programs that are intricately coupled with their environment, containing components that are not cohesive (they must contain code to serve many different functional and non-functional requirements at the same time.)

Our research has produced a new software organization that encapsulates communication and data transformation activity, so that treatment of *interfacing* requirements is decoupled from that of *functional* requirements. As a result, programmers are able to code without having to pay constant attention to constraints imposed by the underlying architectures, language processors or communication media. Such constraints cannot be completely ignored, but they can be isolated for independent treatment. Further, once the application has been devised for one execution environment, then tailoring the interface needs for execution in other environments should be a separate and automated activity.

Section 2 contains a concrete example to highlight the interfacing problems we intend to solve; this example is used throughout the paper for illustration. Our approach to the problem, as implemented in a system called POLYLITH, is described in Section 3, after which we discuss experiences with use of our system and related work.

2 MOTIVATION

In order to study the types of coupling that arise in diverse systems, consider the simple 'phonebook' example shown in Figures 1 and 3 (respectively, a user interface and a table lookup function.) One way these two C routines could be integrated is to compile them separately, then link the object codes into an executable binary. The function names, types, and data representations must match for program to run.

To run these routines as distinct processes on diverse architectures, separated by a network, many implementation obligations must be discharged:

- An additional `main()` program must be generated for the implementation of `lookup()` to function as a separate process. Similarly, `local`'s call to `lookup()` must be replaced by IO system calls.
- The user must ensure that the parameters are marshaled correctly. This entails transforming the data structure in such a way that it can be transmitted in a stream to the other task. In `local`, little would need to be done with the transmission of one string (the variable called `key`); however, the response from `remote` requires dereferencing the pointer to the desired record, and extracting each of the two fields to be packed into a stream response.
- In conjunction with the data marshaling, one of the two tasks must coerce the low-level representation of primitive data types to match the representation of the other task's underlying architecture.
- Now that more than one process is involved, the user needs to decide how all processes would be started up on the appropriate machines, and likewise must decide how synchronization between the threads of control would be handled.

The simple routine from Figure 1 could easily grow to the more complex routine shown in Figure 2. Unfortunately, once a programmer has manually adapted the programs for use in a network, then they cease to function back in the original (single process) context of use. Similarly, manual adaptation would be needed to execute in other environments, such as running as separate threads on a shared memory multiprocessor; with one of the processes implemented in a different language than C; or with one of the processes replaced by a high level simulator (to help capture of requirements early in a development life cycle.)

Fabricating the desired application out of existing components might be even more difficult, however, in the case that alternate languages are used. For example, the `remote` module might equally well have been implemented by either the Lisp code shown in Figure 4 or the Ada program shown in Figure 5, neither of which has control behavior that directly matches the C counterpart.

Good programmers will minimize the amount of manual adaptation that would be required for subsequent porting of their programs. For instance, the network startup code from the example could be isolated into a single initialization procedure, returning an IO descriptor for use in subsequent communication with the other task. Further, the OS-specific calls could be relocated into a user-defined network stub. However, such adaptation activity is awkward at best, is prone to error when performed manually, and still does not result in an application component that is transparent to the context of use. Furthermore, these conventions are of little use for programs written in different languages or running across a network of hosts with different operating systems.

This example motivates a development environment where programmers may completely avoid manual introduction of elaborate interfacing codes into their applications. Moreover, they should have a capability to quickly specify the *geometry* of their application (that is, the mapping of their program structure onto the available system architecture) separately from the implementation of their individual program units. These capabilities are what the POLYLITH software interconnection system provides.

```c
#include <stdio.h>
struct table {
    char *key;
    int value;
};
main()
{
    char key[256];
    struct table *retval;



    printf("Name?  ");
    while(gets(key) != NULL) {
        if(retval = lookup(key))




            printf("%s at ext.  %d",
                key, retval->value);
        else
            printf("%s not found.",key);

        printf("Name?  ");
    }
}
```

```c
#include <stdio.h>
struct table {
    char *key;
    int value;
};
main()
{
    char key[256], buffer[1024];
    struct table *retval;
    int fd, i, ip;
    if ((fd = RENDEVOUS("remote.lookup")) < 0)
            exit(1);
    printf("Name?  ");
    while(gets(key) != NULL) {
        if (SEND(fd, key) < 0) exit(2);
        if (RECEIVE(fd, buffer) < 0) exit(3);
        if(buffer[0] != (char)0) {
            ip = &(buffer[strlen(buffer)+2]);
            i = BYTESWAP(*ip);
            printf("%s at ext.  %d",
                key,i);
        } else {
            printf("%s not found.",key);
        }
        printf("Name?  ");
    }
}
```

Figure 1: (Left) Simple `local` program unit.

Figure 2: (Right) Adapted `local` program unit.

The text on the left in Figure 1 illustrates a simple C program which repeatedly accepts a string entered by the user, passes the string to a `lookup` command (assumed to be bound to a "phone book" routine, to look up the string 'name'), and accepts back from that call an integer value (taken to be the phone number associated with the input string, or, if a zero value, indication that no entry exists for the name.) In contrast, the text on the right in Figure 2 illustrates what can become of the trivial user interface when it must be adapted in order to meet non-functional requirements imposed by the underlying interconnection system, say, for remote procedure call to a host of different architecture elsewhere in a network.

```
#include <stdio.h>
struct table {
    char *key;
    int value;
};
static struct table db[] = {     ...
            { "Jack", 1732 },
            { "Christine", 1566 },
            { "Jim", 1832 },     ...
            { "Elizabeth", 1566 },
            NULL
};
struct table *lookup(key)
char *key; {
    int i=0;
    while(db[i].key != NULL){
        if(strcmp(key,db[i].key) == 0) return (&db[i]);
        i++;
    }
    return NULL;
}
```

Figure 3: Simple `remote` program unit.

This is a simple implementation of the `lookup` operation, as called from the components in Figures 1 and 2. The C function simply performs a linear search in the data structure (probably the world's worst way to implement a phonebook program, but adequate for our illustration.) Once an entry matching the parameter is found, the record is passed back to the caller; if no match is found, a NULL value is returned to signal as such.

```
(setq mynames '((Jack 1732)(Christine 1566)(Jim 1832)(Elizabeth 1566)))
(defun lookup (name)
    (let
        ((x (assoc name mynames)))
        (cond
             ((null x) 0)
             (t (cadr x))
        )
    )
)
```

Figure 4: An alternative implementation for `remote`.

```
Package Phonebook is
MAX: constant integer:= 5; Subtype keyarray is string(1..12);
Type TableEntry is record key : keyarray; value: integer; end  record;
Type Table is array (0..MAX) of TableEntry;

Procedure  lookup( key: in keyarray;  f : out boolean;  e: out TableEntry);
end Phonebook;

Package body  Phonebook is

db: Table := Table'(0 => (key => "dummy entry ", value => 0),
                    1 => (key => "Jack        ", value => 1732),
                    2 => (key => "Christine   ", value => 1566),
                    3 => (key => "Jim         ", value => 1832),
                    4 => (key => "Elizabeth   ", value => 15660),
                    5 => (key => "            ", value =>0) );

Procedure  lookup( key: in keyarray;  f : out boolean;  e: out TableEntry) is
  found : boolean;       i     : integer;
begin
  found := false; i := 0;
  while (db(i).key /= "            ") and (not found) loop
     if key = db(i).key then
         found := true;
     else i := i + 1;
     end if;
  end loop;
  if not found then  e := db(0); else e := db(i); end if;
end lookup;
end Phonebook;
```

Figure 5: Yet another alternative implementation for `remote`.

3 POLYLITH ARCHITECTURE

Section 2 exposes a number of difficult interfacing problems for individual program units. These problems generalize, and we now raise the discussion to interfacing issues of *modules*, collections of individual program units and data declarations. Project Polylith has sought to overcome the difficult problems that arise when modules implemented in different languages are to execute on heterogeneous architectures, supported by varying communication media.

In order to address these problems, we have sought to meet the following requirements:

1. Specification of an application's structure (i.e., its "design") and the implementation of individual components should be independent from each other. Programmers should be free to declare *which* modules should appear in a design without regard to any constraints placed on the modules' implementation; conversely, the implementation of functionally-correct modules should be independent of interfacing considerations particular to any context of their use.
2. Similarly, the application's geometry should be strongly separated from the implementation of individual modules. Programmers should be able to specify *where* modules execute without having to adapt any module source code.
3. Programmers should be able to specify *how* components communicate without having to adapt programs at the source level. This requirement includes coercion and marshaling of data that is transmitted between components.

These requirements are clearly a synthesis of systems and software engineering concerns. Distributed programming environments have been constructed to hide some interfacing concerns from programmers, and similarly module interconnection languages are available to separate application design from implementation in a homogeneous execution environment (c.f., Section 4.3). But none of the previous approaches meet all three major requirements simultaneously; each approach still contains some form of coupling between the design and implementation levels or to a particular environment.

Our approach has been implemented and tested in a system called Polylith. To address the first major requirement, Polylith provides a module interconnection language (MIL) for expressing an application's structure. This MIL is declarative, and independent of any particular application language — the choice of language for implementing individual software components is left to the designer. The MIL is based on a simple graph model of interconnection, where nodes in a program graph correspond to modules, and arcs in the graph represent bindings between module interfaces. It is described in Section 3.1.

To control application geometry, the Polylith MIL allows attributes to be associated with nodes in the graph structure. These attributes allow designers to abstractly declare a desired locality for execution of the named program unit. If the user elects to give no guidance to Polylith concerning geometry, then the system constructs binaries for components according to default guidelines. Regardless, for both this and the previous requirement, the graph approach ensures that structure is exposed to the designer for manipulation and analysis. This activity is described in Section 3.3.

The programmer's decisions concerning placement of modules onto processors still leaves flexibility in how the modules are to interoperate. For instance, an RPC between two interoperating processes can be made by a variety of communication channels or protocols. Our third major requirement is that programmers must be able to alter these interfacing mechanisms without adapting source programs or the structural specification — this is accomplished in Polylith by varying the choice of *software bus*. A software bus is any agent that encapsulates, and hence isolates, all run-time interfacing concerns for an application. An abstract software bus establishes the domain of discourse for programmers seeking to interconnect diverse software components; an implementation of that bus lets programmers leverage their decisions about interconnection abstractions. To change interfacing properties, one changes the bus, not the application modules. The bus abstract interface is described in Section 3.2.

3.1 A MODULE INTERCONNECTION LANGUAGE

Since modules will ultimately be implemented in other existing languages, the sole requirement for POLYLITH's MIL is to express and organize system structure. The basic language construct is therefore of the form

`module` *name* { *declarations* }

where *name* gives a name for the module being defined, and *declarations* represents a sequence of declarations of the structural properties of this module: interfaces, nesting of additional modules, and bindings between interfaces within the scope. The *name* is optional, since there are many situations where we will need to build a module but never reference it externally.

In the *declarations* of a module, interfaces on that module are declared by either

`use interface` *name*

or

`define interface` *name*

The `define` key word declares the interface *name* is an available resource; `use` declares that a non-local resource is required by this module.

The C programs used for motivation in Section 2 would have generic descriptions as shown in Figure 6. In order to create and run an application, a complete design must be created: from the set of all modules defined in a given scope, those that the programmer wants for the application must be selected, then the bindings between interfaces must be established. The `tool` and `bind` constructs shown in Figure 7 perform this instantiation and interconnection respectively[1]. While there are many external interfaces in the implementation of these modules (`printf`, `main` and so on), only the `lookup` interface is of interest to the designer, therefore it is the only name to appear at the design level. Omission of other names from the `module` signature allows the packager system (responsible for creating executables, Section 3.3) wide latitude in how those names should be bound, typically from a library.

The POLYLITH graph model is attributed; symbols with values can be associated with each element of an application graph. Attributes are expressed using the syntax "*symbol=value*". Any number of such definitions can appear after a module *declarations* statement (and hence associate a valued-symbol with that interface), or can appear after the `module` definition itself (hence associating attributes with the entire module.) Attributes are used in several ways, for instance, to organize *interface patterns* — type information concerning data or a procedure's parameters — a `PATTERN` attribute is associated with an interface, as in Figure 6. The combination of `module` declarations with `PATTERN` attributes is directly analogous to the definition of an abstract data type's signature and sorts, respectively.

The text in Figure 7 represents only the most basic composition of an application program graph. In general, the POLYLITH MIL provides several features to simplify a binding task that can often be a tedious manual activity. For example, our demonstration problem defines and uses the desired interface under the same name, `lookup`; hence, the programmer could include a "`bind $all`" directive inside a given

[1] "Instantiation" is indeed the appropriate term, as in general many copies of the generically-defined object may be incorporated within an application system.

```
module main {
    use interface lookup :  PATTERN=string
}

module demo {
    define interface lookup :  PATTERN=string
        :  RETURNS=↑{ string ; integer }
}
```

Figure 6: MIL definitions for example modules.

```
module {
    tool main
    tool demo
    bind main.lookup demo.lookup
}
```

Figure 7: Sample use of the abstract modules.

scope, which would declare that all such obvious interface matches should be bound, provided there are not multiple definitions or type mismatches.

3.2 THE SOFTWARE BUS

A bus is any agent that encapsulates interfacing details for a software application. Programmers first make decisions about how to implement an application's functional requirements, then select desirable interfacing mechanisms by their choice of bus. Subsequent changes in interfacing behavior are obtained by changing busses, not the application.

When someone defines an abstract software bus, he or she establishes the domain of discourse for developers to reason about the compatibility of components they want to integrate. The criteria for establishing what is a module, the range of valid data types on module interfaces, and the control mechanisms available for modules to interoperate must all be defined. When that abstract bus is implemented, the developer has a basis for relating each new language and platform to the abstraction operationally. For the purposes of clarity, we will present the remainder of this section as if there is only one abstract bus, but in fact the methodology by which various abstract busses are established is the topic of a separate work [PuSW91].

An abstract bus can be implemented in many ways. Fundamentally, implementation of a bus entails making two types of decisions:

- **Dynamic:** How do processes started by this bus communicate?
- **Static:** How do requests for non-local processing (e.g., procedure calls) for a given language (on a given architecture) correspond to accessors in the bus abstraction?

Decisions concerning dynamic interfacing properties are embodied in "the bus," which, in all but the most trivial scenarios, is present in the form of an additional process (or processes) within the execution environment. Decisions concerning preparation of binaries for use by a given bus are embodied in a corresponding *packaging system* (by which we refer to all tools needed to adapt, translate and link an application program into an executable unit.) Clearly, a bus and its packager must work in concert. These decisions are generally made once by a site manager, who must be knowledgeable about the available languages and architectures[2]. Ordinary programmers do not normally implement their own bus mechanisms, though the system is distributed with all the resources necessary for them to do so if they wish. The bus accessors to a simple Polylith bus is described in Appendix A, for illustration.

Information concerning an application's overall structure cannot appear within source programs without violating our requirement for minimal coupling between components. Yet many interfacing scenarios — notably parallel and distributed programs — require that structure and geometry be known absolutely at the time the application is started. Therefore it is inescapable that any bus, as agent responsible for starting tasks, must have some MIL-like notation to guide its actions. This is exactly the relationship between the Polylith MIL and each bus: the MIL specification is read by the bus, which then is able to invoke tasks, acquire communication channels and initiate interoperation between application objects. During execution, the bus may assist in communication, instrumentation or reconfiguration of the application.

Details concerning how an application's binaries are generated for a particular bus are deferred until Section 3.3, so that we may now examine concrete bus execution scenarios. In terms of the problem from Section 2, a sequence of such scenarios is traced in Figure 8. The first two, all modules are packaged for execution in the same process space. In scenario *(a)* the binary prepared by the packager is indistinguishable from the binaries created by existing compiler tools. On the other hand, if the application is implemented in different languages, then some parameters' representation may need to be coerced, and a configuration like scenario *(b)* would be created by the packager. Either way, the bus simply spawns the single process and terminates.

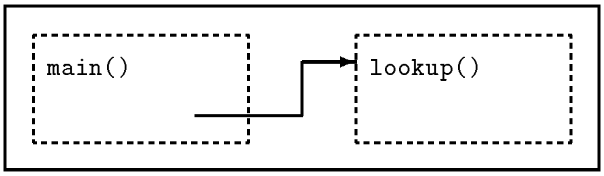
Scenario *(c)* is representative of applications having multiple processes that must be managed. In the case that processes reside on different processors in a distributed network, then — as guided by the MIL specification — the bus must start processes on the appropriate hosts (e.g., our network busses currently use generic `rexec` resources for Unix sites), and establish communication links among these tasks. These are typically implemented by available underlying protocols, such as TCP/IP and XNS.

If each task is to execute on a different processor in a multiprocessor environment, then the bus has the same responsibilities, but fulfills them differently. For example, on the multiprocessors currently available to us (which all run Unix) the packager will have prepared a single binary, which when invoked will spawn all additional processes via `fork()` calls; this is necessary since the principal method of communication supported by the multiprocessor bus is shared memory, which is not preserved between Unix processes separated by a call to `exec()`. The bus is simply another task started up to synchronize the multiple threads. Nonetheless, this is a typical interfacing scenario that no programmer would wish to implement manually, yet which is useful for a variety of applications.

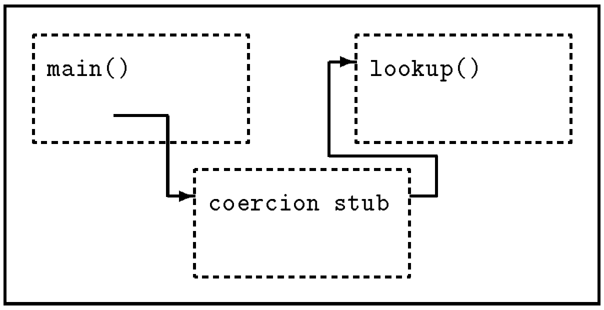
Even for a fixed architectural configuration and interconnection medium, there are still good reasons to have variety in the available busses — users still need variety in their interfacing properties. Focusing on network-based configurations, one of our bus implementations is built for performance: once all distributed processes have been invoked, each task is set up to establish its own IP ports for network traffic, then send this information back to the bus. Once all modules have been 'registered' in this fashion, the bus passes the information to the appropriate tools so they can initiate direct connections. In short, the bus introduces all tasks to one another, then steps out of the way. The resulting use of ports in the IP domain is then indistinguishable from hand-crafted programs, except that no tasks's source program contains absolute

[2]We will continue to use the title 'site manager' to describe anyone who is in charge of maintaining interfaces in an organization. Someone has to know about a new language in order for it to be incorporated into a polylithic system.

(*a*) The call to `lookup` from `main` is bound directly to the object code in the same process space. No additional data coercion or relocation is incorporated.



(*b*) If different implementation languages are used, then the initial call is bound to a translation filter, which performs the coercion and then makes the actual call.



(*c*) In this scenario, the non-local reference from the user's `main` requires both translation and relocation of parameters. The initial call goes to the coercion stub, which then requests that the bus transmit the parameters to the non-local resource. Since the remote resource may also be on a different architecture, the bus provides the parameters to another stub appropriate for that host, which in turn finally makes the actual call.

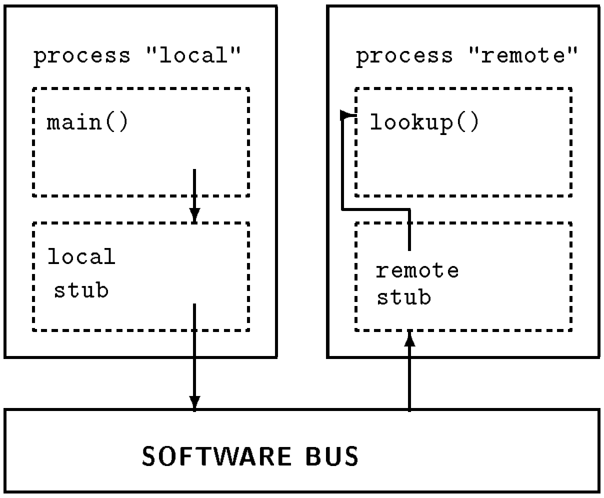


Figure 8: Three execution scenarios.

network or host identifiers (names that limit its range of application.)

Alternately, a network-based bus may retain control of all communication ports, forcing processes to contact the bus to route all traffic. While serializing messages in this way degrades performance, it has the advantage of exposing all network traffic to debugging or visualization tools. Again, this is transparent to any application source program. Regardless of how the bus is implemented, the resulting scenario is the same. The bus abstraction removes interfacing obligations from both the program structure and the individual module implementations.

The ultimate flexibility of this approach is apparent when a new host and its operating system must be incorporated. The site manager must only make decisions about how to implement a bus that can communicate with the new host. Thereafter programmers are free to construct programs that span the enhanced system architecture, without ever having to adapt application source code.

3.3 PACKAGING OF EXECUTABLES

Each bus must have a packager to prepare appropriate binaries for it. In general, the task of adapting programs so they can interact directly with other software is difficult. However, the idea in POLYLITH organization is to introduce a fixed target — the bus abstraction — as an intermediary. Therefore, if the developer can defer packaging until the MIL specification has been created, then all that is left to do with the source code is to map the internal procedure into one of a small set of known bus accessors — a much easier task.

There are four major aspects to packaging:

1. Control of the packaging process itself.
2. Control of how a language domain maps non-local references into the POLYLITH bus abstraction. This includes stub generation and linkage decisions.
3. Control of the correspondence between the POLYLITH representation of data and that found in a particular implementation language and architecture.
4. Control of how modules in the logical program structure are assigned to execute in particular processes (i.e., control of the geometry.)

The first aspect reflects a traditional need: packaging requires many compiler and linker steps that no user wants to perform manually. The POLYLITH packaging process is straightforward adaptation of such tools as the `makefile` system [Feld78]. Its only embellishment is that it is parameterized to accept the name of the bus for the target execution environment.

The second aspect of packaging pertains to the correspondence between the bus abstraction and an implementation language's mechanisms for requesting nonlocal service (such as procedure or function calls, message transmission and so on). Among all language constructs that request some form of non-local processing, a site manager must decide which to expose to the bus, and then how to map them into the bus interfaces. Often a packager is set up to produce *procedure call* semantics. However, there are many alternatives to procedure call semantics, and MIL attributes provide developers with additional variety. For example, an attribute called `MESSAGE` can be associated with interfaces in the design, which directs the packager to implement message passing semantics in the stub; bindings on such interfaces are directional, with the definition of the interface viewed as the 'source' of the message, and the use of the interface viewed

```
module main {
    use interface lookup :  PATTERN=string
} :  SOURCE=local.c

module demo {
    define interface lookup :  PATTERN=string
        :  RETURNS=↑{ string ; integer }
} :  SOURCE=remote.c

module {
    tool main :  LOCATION=brillig.cs.umd.edu
    tool demo :  LOCATION=slithy.cs.umd.edu
    bind main.lookup demo.lookup
}
```

Figure 9: Sample design illustrating use of attributes for geometric guidance.

as the message 'destination.' Other attributes can direct whether the communication is to be synchronous, whether queuing of messages is to be supported, and so on. Users direct the initiation of multiple control threads by assigning appropriate values to attributes at the design level [PuRG88].

The third major aspect to packaging is for site managers to decide how *data* correspond between POLYLITH's standard representation and that of a particular language and architecture. Our approach here is quite pragmatic: we ask the manager to provide an operational specification of the correspondence between types in their new language and the types supported by POLYLITH. For each primitive type and combinator of interest, managers should implement a representation map (and its inverse) between a datum in the new language and how it would appear in POLYLITH standard representation. This representation map is called by the bus protocol (as guided by high level type information from the MIL specification), and is typically linked into the application component stubs. Examples of such decisions made by managers (decisions that are made once, not requiring extensive consideration by application developers later on) are illustrated from our example in Section 2. The manager here has decided that the C type definition `int` corresponds to the POLYLITH type `integer`, and the method of transforming an `int` into POLYLITH `integer` might require the byte swapping shown in Figure 2.

Coercion in POLYLITH currently employs simple notation to describe the structure of interface parameters. A 'regular expression' over the language of primitive data types supported by the given bus defines the range of values that `PATTERN` attributes can attain, such as shown in Figure 6. This is not nearly as expressive as other languages that are devoted to definition of interface structures, as discussed in Section 4.3. Nonetheless, our experience has been that this is a surprisingly useful notation. In general, the way this coercion activity is guided by the MIL declarations is described in detail in [Purt86].

The final aspect of packaging is to determine how modules are assigned to processes. In the absence of any other guidance, this assignment is largely up to each packager, which will build executables by straightforward techniques. This activity is directed by the MIL specification, and each `tool` declaration is interpreted by the packager as being a potential process assignment. For packagers targeting single-process configurations, `tool` is ignored; packagers for distributed architectures generate stubs for network access based on `tool` declarations; packagers for multi-processor architectures establish a binary that immediately spawns a separate thread for each `tool`; and so on. To over-ride the default packaging conventions, users may vary attributes in the design to suggest which modules should be co-located in a given process.

Figure 9 specifies a configuration having the same logical structure as shown in Figures 6 and 7, except we assign attributes that give extra directions. A network-based packager would use this guidance to establish an application that would run on exactly these two hosts at our site. More realistically, a programmer would place another variable name as the value of each `LOCATION` attribute; the bus would prompt the user for desired locality upon invocation of the system.

4 CONSIDERATIONS

POLYLITH was first implemented in 1985, and has been used in a variety of applications and experiments. Initially, most users to date were drawn to POLYLITH for help in scientific computing applications: they use the MIL as a language for coarse-grained parallel programming, such as originally described in [PuRG88]. But the system serves software engineering activities as well [LuHa86].

Flexibility in how the MIL itself can be manipulated is demonstrated in [PuJa90]. This project makes simple extensions to the MIL in order to support software fault tolerance. An application's structure is decorated with information concerning multiple implementations, voting programs and recovery blocks; the specification is then transformed into a standard POLYLITH interconnection graph for execution in a network environment.

Whereas our initial implementation of the bus focussed on binding transparency at the source code level, an experimental use of a different presentation — one *exposing* the network in ways that are intended to be useful to C programmers in distributed applications — is described in [PuJa89]. This project created a different map of control and data representations between C and the abstract bus.

The most novel bus implementation to date is Minion, a *visual* bus [Purt89]. At run-time, this bus exposes the communication protocol to outside display and debugging tools. As with any bus, these interfacing details are encapsulated, and hence transparent to the application program, except for timing constraints. We are using this framework to control reconfiguration of running applications.

Most recently, the heaviest use of POLYLITH is found within the DARPA prototyping community; the simplicity of interconnecting diverse program units expands the range of existing programs from which developers can draw easily and quickly when fabricating a prototype for experiments [PuLC91].

Based upon these experiences such as these, the remainder of this section addresses issues raised most frequently concerning POLYLITH.

4.1 PERFORMANCE

The flexibility of polylithic organization can be obtained without loss of execution performance. Transparency is at the source program level, so for a given program unit, our packaging system does not produce just one binary for use in all circumstances, but rather produces one of many possible executables as guided by the structure, declarations of geometry and target bus. As a result, our differences from other systems are not so much in what executables are produced and executed, but rather in how those executables were derived.

Consequently, what is of more interest is the extent to which use of novel interfacing properties might affect performance. Consider the problem used for motivation in Section 2. For purposes of benchmarking, we provided a long stream of `lookup` requests as the "standard input" to the main program; this application was packaged exactly as described in Section 3, targeted for execution on a pair of Decstation 3100 workstations at our site. No special precautions were taken to limit outside network traffic. The basic unit of measure is the number of procedure calls per second that can be sustained to `lookup` from `main`. The two-node program graph runs at approximately 400 calls per second when initiated by a TCP/IP-based bus, with an interoperation protocol optimized for performance. This also corresponds to the performance of a non-POLYLITH-based version of the program that we hand-crafted for comparison. (When both processes are executed on the same Decstation, the performance approaches 1000 calls per second. We attribute this gain to loopback in the Unix kernel's network device driver — this is a great property for prototyping.) When executed with a different network bus, one designed to log all traffic for inspection or debugging, the application runs with just over 200 calls per second. Since there is a bus agent 'between' the two application tasks, each logical procedure call requires twice the number of actual network operations. (The figure reported is not exactly half the earlier value, due to opportunity for some overlap in processing provided by the interconnection protocol.)

4.2 TRANSPARENCY

Although POLYLITH-managed interfaces yield performance comparable to hand-generated interfacing for the same application structure, there is a broader question, which is whether programmers who build interfaces manually would be basing their activity *on the same application structure*. Many application structures cannot remain transparent to the architecture and still achieve high performance requirements. There are many opportunities for programmers to implement their own software caches and communication protocols — they may take advantage of knowledge concerning the application's behavior (in particular its use of the communication system) to improve overall performance. POLYLITH does not attempt to discover any of these opportunities for improving performance; polylithic organization does give a framework for incorporating any application-specific interfacing code that programmers develop for themselves.

This issue has many manifestations. Our experience with POLYLITH in distributed configurations is that users can become very distant from the underlying architecture — they rely upon a packager to distribute the components. However, automatic packaging tools can separate modules that might reasonably have been assigned to execute in the same process, and, depending upon use, the resulting communication costs can be high. The intent of our research has never been to blind the programmer to the architecture, only to let the programmer to deal with interfacing separately from functionality. *Knowledgeable* guidance in packaging is always preferred. Polylithic organization provides a framework for user's to express that guidance, and within which communication costs can be exposed; once default packaging decisions are shown to be undesirable, the application can repackaged, easily and transparently. Programs that have been hand-crafted do not have the luxury of such inexpensive reconfiguration, should users discover usage patterns are different than they expected.

A related experience is that programmers of heterogeneous systems will occasionally assign a module to execute on a host for which it is unsuitable. For example, assigning our example's `lookup` function to execute on our Connection Machine, as we are able to do, does not yield a significant increase in performance. Programmers in POLYLITH can forget that while their code can port between machines, the algorithms to best exploit some architecture might not relocate so easily. Again, the purpose of our research is not to enforce homogeneity, as to do so would eliminate the best use of hardware available to us. We wish instead to indulge in heterogeneity without the high costs of interfacing.

Finally, in any multi-tasking environment, there is a question of whether referencing semantics are transparent in the case of pointer data. Ideally, behavior of the centralized and distributed configurations of an application would correspond exactly. This is not currently the case in POLYLITH. As with many systems, *by-reference* parameters are typically implemented as *value-result* parameters instead. (How this is done precisely is a decision made within each packager.) However, we are currently working on a source-to-source transformation approach to address this problem; we hypothesize that programs containing pointer data can be conditioned at a high level to contain appropriate bus interfaces so that the intended nonlocal value can be accessed. Preliminary experiments in this approach are promising.

4.3 RELATED WORK

This work is a synthesis of results from many research areas, and there are correspondingly many other projects to which we must be compared. The contributing areas range from data representation and distributed languages to configuration management systems.

Much work has been done in primitive data representation in the presence of heterogeneity. The POLYLITH approach benefited from review of previous experiences with Courier [Xero81]. Sun Microsystem's XDR [SunM88] is a similar approach, as is UTS, a 'universal type system' internal to the MLP (Mixed Language Programming) system [HaMS88].

More abstractly, transmission of abstract data types (ADTs) is presented in [HeLi82]. Two new interface accessors, *transmit* and *receive*, are added to the ADT's signature, and the developer provides a suitable implementation of these routines for each host. When the ADT is to be transmitted, these new accessors are used by the system to relocate the essential state information in a suitable form. While developed independently, POLYLITH's algorithm for marshaling parameters for transmission bears a strong similarity. The essential difference is that POLYLITH's algorithm focuses on primitive types and simple aggregates for which the transformation can be generated automatically, based on knowledge of the interface provided by the MIL specification. Though Herlihy's ADT method requires the developer to create additional routines, it certainly applies to a much wider class of parameters; it would be very reasonable to consider installing this capability within POLYLITH packagers in the future.

Another alternative for strengthing POLYLITH's interface type system is IDL, the interface definition language. IDL provides a rich method for expressing the semantics of shared data structures [Lamb87, Snod89]; the code for accessing data on common interfaces is then generated automatically, thereby guaranteeing consistent treatment by all parts of the system. Recently, a mapping activity involving IDL (and very similar to the decisions given a site manager as discussed in Section 3.3) was presented in [ShSn89]. It would be natural to encapsulate IDL-generated source into a standard POLYLITH module, and then use the MIL to bind interfaces appropriately. The systems are quite complementary — code generated by the IDL system could be made available to the packager system, as an additional configuration item managed by the MIL.

POLYLITH's focus to date has been 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 MIL would bind the instance's accessors into those modules using it, and thereafter those modules would transact *capability to* that instance (rather than 'flattening' it for transmission.) This approach is very similar to that shown in [JLHB88], where a *call by object-reference* method is described in detail.

Early work on MILs was performed by DeRemer and Kron [DeKr75]. Shortly after, project Gandalf focused on the software development environment itself [HaNo86, Notk85], implementing a MIL known as Intercol for describing the structure of an application [Tich80], and permitting mixed language programming (subject to the restriction that the language processors be created to conform to Gandalf interfacing structures.) Most recently, environments and languages such as Inscape and SLI are appearing [Perr89, WWRT91], in which developers can express not only the desired structure, but also assertions to guide use and interconnection of the configuration items. Inscape addresses many of the same sorts of packaging issues as POLYLITH. The AdaPIC tool set also focuses on stronger analysis of how interfaces are used, an activity blended with support for other development activities [WoCW89]. Our approach to simple stub generation is very similar to that of Gibbons [Gibb87], although we can generate stubs from automatically extracted interface information, whereas Gibbons' Horus system requires users to explicitly declare interface structure.

Structure-oriented languages (containing some MIL features) were used to control a distributed programming environment in several earlier projects, notably CLU [LiAt81] and MESA [Swee85]. Both support distributed programming by coupling their notation with their supporting systems (Argus [Lisk88] and Pilot, respectively). 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 [JoRT85] 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.

Especially appropriate for multiprocessor configurations are Camelot [Bloc89] (a transaction facility built on top of Mach) and Avalon (a language resource constructed using Camelot.) The V Kernel [Cher88] implements a distributed- and parallel-programming resource appropriate for a homogeneous set of hosts. The HCS project [NoBL88] shows one way for providing a heterogeneous RPC capability in a distributed environment; more recently, a 'lightweight' remote procedure call was demonstrated [BALL90]. Concert [YGSW89] and Marionette [SuAn89] are more variations on a theme. Several early projects emphasized a network filesystem approach (such as Locus [PWCE81].) An interesting approach to cross-architecture procedure call using a common backing-store is given by Essick [Essi87]. The ISO OSI framework for network interconnections appears similar to our approach [OSI81], but is more oriented to encapsulating decisions related to network concerns, as opposed to our focus on data and control integration.

Most recently, there appears to be a trend towards even stronger use of MIL-like languages for control of distributed applications, with the emergence of Conic's environment and toolset [MaKS89], and of Durra [BDWW89]. The Mercury system supports heterogeneity in applications by managing a networked object repository [LiSh88]. All of these systems focus only on distributed execution environments.

5 CONCLUSION

A fundamental 'divide and conquer' principle in software is that an application's structure can be designed separately from the construction of its components. Designers are free to manipulate the abstract structural specification, while programmers have freedom in how they implement the modules. Our bus-based approach introduces a third degree of freedom by encapsulating properties of the execution environment: location of data, distribution of programs, and media for interoperation. These details are hidden from the implementor of a module in just the same way that implementation decisions are separated from specifications.

This new freedom to vary separately the structural specifications, functional implementations, and interfacing properties provides many benefits. An obvious by-product is a facility for mixed-language programming: our approach encapsulates the choice of source language within a packager, and hence programmers are free to access other procedures without regard to possible conflicts in data representation or calling conventions. A farther-reaching implication is that we can now design our application structures independent of how they are mapped onto host configurations, then have a formal basis for refining the designs to meet performance demands on specific architectures.

There are more steps to producing an executable using bus organization, but a greater proportion of the packaging activities can be performed automatically, rather than with programmer intervention. Hence there is a net savings in effort invested by the programmer, with greater flexibility in how the application can subsequently be executed. Our research on software interconnection is continuing, using the POLYLITH model as a basis. Currently we are focusing on two related activities, techniques to control dynamic program reconfiguration and process migration in heterogeneous environments.

Acknowledgement: The author wishes to thank John Gannon, Udaya Shankar, Marvin Zelkowitz, Ralph Wachter, Bill Gear, Paul Saylor, Jack Callahan, Christine Hofmeister and Elizabeth White for their many helpful comments and suggestions concerning this research and document. We are grateful to the Office of Naval Research for their continuing support.

REFERENCES

[BDWW89] Barbacci, M., D. Doubleday, C. Weinstock and J. Wing. Developing applications for heterogeneous machine networks: The Durra environment. Usenix *Computing Systems*, vol. 2, (1989), pp. 7-35.

[BALL88] Bershad, B., T. Anderson, E. Lazowska and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, vol. 8, no. 1, (February 1990), pp. 37-55.

[Bloc89] Bloch, J. The Camelot library: C language extension for programming general purpose distributed transaction system. *Proc of 9th Conf on Distributed Computing Systems*, (June 1989), pp. 172-180.

[Cher88] Cheriton, D. The V distributed system. *CACM*, vol. 31, (1988), pp. 314-333.

[DeKr76] DeRemer, F. and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, vol. 2, no. 2, (June 1976), pp. 80-86.

[Essi87] Essick, R. The cross-architecture procedure call. *Doctoral dissertation.* UIUC Dept of Computer Science UIUC-R-87-1340, (1987).

[Feld78] Feldman, S. I. Make – A program for maintaining computer programs. Bell Laboratories Report, (August 1978).

[FSAC89] Finkel, R., M. Scott, et al. Experience with Charlotte: simplicity and function in a distributed operating system. *IEEE Trans Software Engineering*, vol. 15, (June 1989), pp. 676-685.

[Gibb87] Gibbons, P. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, vol. 13, no. 1, (January 1987), pp.77-87.

[HaMS88] Hayes, R., S. Manweiler, and R. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software Practice and Experience*, vol. 18, no. 7, (July 1988), pp. 641-600.

[HaNo86] Habermann, N., D. Notkin. Gandalf: Software Development Environments. *IEEE Transactions on Software Engineering*, vol. 12, no. 12, (December 1986), pp. 1117-1127.

[HeLi82] Herlihy, M., and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 4, (October 1982), pp. 527-551.

[JLHB88] Jul, E., H. Levy, N. Hutchinson and A. Black. Fine-grained mobility in the Emerald system. *Transactions on Computer Systems*, vol. 6, no. 1, (February 1988), pp. 109-133.

[JoRT85] Jones, M., R. Rashid and M. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. *Proce of 12th ACM Symposium of Principles of Programming Languages*, (1985).

[Lamb87] Lamb, D. A. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, (July 1987), pp. 297-318.

[LiAt81] Liskov, B. and R. Atkinson. CLU Reference Manual. *Springer-Verlag LNCS 114*, (1981).

[LiSh88] Liskov, B., and L. Shira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *Proc of SIGPLAN Language Design and Implementation*, (June 1988).

[Lisk88] Liskov, B. Distributed programming in Argus. *CACM*, vol. 31, (1988), pp. 300-313.

[LuHa86] Lubars, M. and M. Harandi. Intelligent support for software specification and design. *IEEE Expert*, vol. 1, no. 4, (1986), pp. 33-41.

[MaKS89] Magee, J., J. Kramer and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, vol. 15, (June 1989), pp. 663-675.

[NoBL88] Notkin, D., A. Black, E. Lazowska, et al. Interconnecting heterogeneous computer systems. *CACM*, vol. 31, (1988), pp. 258-273.

[Notk85] Notkin, David. The GANDALF Project. *Journal of Systems and Software*, vol. 5, no. 2, (May 1985), pp. 91-104.

[OSI81] ISO Open Systems interconnection – Basic Reference Model. ISO/TC 97/SC 16 N 719, International Organization for Standardization, (August 1981).

[Perr89] Perry, Dewayne. The Inscape Environment. *Proceedings of 11th International Conference on Software Engineering*, (1989), pp. 2-12.

[PuJa90] Purtilo, J., and P. Jalote. An environment for developing fault tolerant software. *IEEE Transactions on Software Engineering*, vol. 17, (1991), pp. 153-159.

[PuJa89] Purtilo, J., and P. Jalote. An environment for prototyping distributed applications. *Proceedings of the Ninth International Conference on Distributed Computing Systems*, (June 1989), pp. 588-594.

[PuLC91] Purtilo, J., A. Larson and J. Clark. A methodology for prototyping in the large. *IEEE* 13^{th} *International Conference on Software Engineering*, (May 1991), pp. 2-12.

[PuRG88] Purtilo, J., D. Reed and D. Grunwald. Environments for prototyping parallel algorithms. *Journal of Parallel and Distributed Computing*, vol. 5, (1988), pp. 421-437.

[Purt86] Purtilo, J. A software interconnection technology to support specification of computational environments. Doctoral dissertation. *UIUC Dept of Computer Science UIUC-R-86-1269*, (1986).

[PuSW91] Purtilo, J., R. Snodgrass and A. Wolf. Software bus organization: reference model and comparison of existing systems. MIFWG Technical Report 8, *University of Arizona Computer Science Department*, (1991).

[Purt89] Purtilo, J. MINION: An environment to organize mathematical problem solving. *Proceedings of the 1989 International Symposium on Symbolic and Algebraic Computation*, (July 1989), pp. 147-154.

[PWCE81] Popek, G., B. Walker, J. Chow, et al. LOCUS: A network transparent, high reliability distributed system. *Proc of 9th Symp on Operating Systems Principles*, (December 1981), pp. 169-177.

[ShSn89] Shannon, K., and R. Snodgrass. Mapping the Interface Description Language Type Model into C. *IEEE Transactions on Software Engineering*, vol. 16, no. 11, (November 1989), pp. 1333-1346.

[Snod89] Snodgrass, R. *The Interface Description Language: Definition and Use.* Computer Science Press, (1989).

[SuAn89] Sullivan, M., and D. Anderson. Marionette: A system for parallel distributed programming using a master/slave model. *Proc of 9th Conf on Distributed Computing Systems*, (June 1989), pp. 181-189.

[SunM88] XDR: External Data Representation Standard. Sun Microsystems Reference Manual, (1988).

[Swee85] Sweet, Richard E. The Mesa Programming Environment. *Proceedings of the ACM SIGPLAN Symposium on Programming Issues in Programming Environments*, (June 1985), pp. 216-229.

[Tich80] Tichy, Walter F. Software Development Control Based on System Structure Description. *Carnegie-Mellon University Dept. of Computer Science Report CMU-CS-80-120*, (January 1980).

[WoCW89] Wolf, A., L. Clark and J. Wileden. The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE TSE*, vol. 15, (1989), pp. 250-263.

[WWRT91] Wileden, J., A. Wolf, W. Rosenblatt and P. Tarr. Specification Level Interoperability, *CACM*, (May 1991), pp. 72-87.

[Xero81] Courier: the remote procedure call protocol. *Xerox Corporation Xerox System Integration Standard XSIS 038112*, (December 1981).

[YGSW89] Yemini, S., G. Goldszmidt, et al. CONCERT: A high-level language approach to heterogeneous distributed systems, *Proc of 9th Conf on Distributed Computing Systems*, (June 1989), pp. 162-171.

APPENDIX A — BUS INTERFACE

This appendix describes some of the interface accessors to a simple POLYLITH bus prototype. All *interface* parameters in the discussion below name an external port from the current module's point of view; it does not directly name another component, but rather the name will be bound to other components later on by the MIL. The *tape* parameters are to specify the type structure of other parameters in the interface; it is given in terms of a simple 'regular expression' notation, and, if needed, is typically available by querying the bus with mh_query_* operations.

mh_read(*interface* , *tape* , *param1* , *param2* , ...)
This performs a read operation on the named interface. Values, placed into the named parameters, will already be transformed into a representation suitable for the current process and application language to use. This is a blocking operation.

mh_write(*interface* , *tape* , *param1* , *param2* , ...)
The given parameters are written to the named interface. This is a blocking operation.

mh_readany(*buffer*)
mh_readback(*tape* , *packet* , *param1* , *param2* , ...)
A read is performed on any interface to this process which has a pending message. If more than one is ready, then one is selected non-deterministically. This call will block until at least one interface has a message to be returned. No type or actual parameters are provided to receive the unwound data — the data are received in Polylith standard representation with the name of the successful interface packed into the data structure. A subsequent call to mh_readback can unwind the representation into the actual parameters once the interface is known. The mh_readback behaves like mh_read except the data are taken from a buffer rather than an actual IO operation.

mh_identity(*buffer*)
The absolute name by which the current bus knows this module is packed into the given buffer. This is useful in applications having many tasks instantiated from the same abstract module, where each may need to identify itself uniquely in logging or error messages.

mh_shutdown()
Any tool in the application can request that the bus be shut down. All tasks started by the bus are tracked down in an orderly fashion and terminated.

mh_query_ifattr(*interface* ,*attribute* , *value*)
mh_query_ifmsgs(*interface*)
mh_query_objattr(*attribute* , *value*)
mh_query_objmsgs()
mh_query_objnames(*value*)
The bus protocol requires that it function as a repository for information specific to a running application. These mh_query_* interfaces are the means to request information from the repository. Tasks can request the number of pending messages on a given interface, or the total number of pending messages across all of its registered interfaces. It can request the value for an attribute associated with either a specific interface or the entire module instance. Finally, it can ask the bus to divulge just *which* interface names it believes the module has — this is excellent for implementing tools that may need to simulate some module, for which it otherwise has no information until run-time.

mh_initialize(*parameter list*)
The bus protocol may require that separate processes have their own communication ports (in the case that the application may wish to have processes directly communicate with one

another for reasons of performance.) This call allows such resources to be acquired, after which the contact information is passed back to the bus for transmission to other processes.

As discussed in the body of this paper, each implementation of the above abstract interface constitutes a new bus. Similarly, a site manager would need to implement how each language (and its architecture) maps into the above abstraction. These obligations are easy to fulfill for procedural languages such as Pascal, Ada, C and Fortran. They are more challenging to fulfill for interactive languages such as Lisp: the language environment also includes user interface support which must be handled. This gives the site manager additional flexibility in the types of bus maps that can be constructed. In the case that users wish to utilize Lisp's "read, eval and print" evaluator as their interactive front-end, then construction of a bus interface entails adding a straightforward implementation of the operations listed above; if, however, the user wishes to make calls from across the bus to a Lisp in the back end, then the normal Lisp evaluator must be replaced by an internal "listen on the bus, eval and return to the bus" loop, that is, the module must perform its own internal dispatching. This has been successful approach for incorporating several Lisp and Prolog implementations into the POLYLITH environment.