WRITING DISTRIBUTED PROGRAMS
IN POLYLITH

Christine Hofmeister
Joanne Atlee
James Purtilo

Computer Science Department and
Institute for Advanced Computer Studies
University of Maryland

This document corresponds to Version 1.0 of the
Polylihth Software Interconnection System.
November 1990.

The Polylith effort has been supported by Office of Naval Research under contract N0014-90-J4091, and is currently part of the DARPA/ISTO Common Prototyping Language project.
Contents

1 OVERVIEW

2 BASIC FEATURES
   2.1 CREATING AN APPLICATION ........................................ 7
   2.1.1 MIL PROGRAM FOR MODULE main ............................. 9
   2.1.2 SOURCE PROGRAM FOR MODULE main ......................... 10
   2.1.3 MIL PROGRAM FOR MODULE print ............................. 11
   2.1.4 SOURCE PROGRAM FOR MODULE print ......................... 11
   2.1.5 MIL PROGRAM FOR THE APPLICATION ......................... 12
   2.1.6 COMPILING, LINKING, AND RUNNING THE APPLICATION ...... 15
   2.2 ENHANCING THE APPLICATION .................................... 15
   2.2.1 MODULE DUP .................................................... 16
   2.2.2 MIL PROGRAM FOR THE APPLICATION ......................... 17
   2.3 SENDING STRUCTURED MESSAGES ................................. 18
   2.3.1 SOURCE PROGRAM FOR MODULE main ......................... 18
   2.3.2 SOURCE PROGRAM FOR MODULE book ......................... 21
   2.3.3 MIL PROGRAM FOR THE APPLICATION ......................... 21
   2.4 SUMMARY OF BASIC POLYLITH FEATURES ....................... 23
3 ADVANCED FEATURES

3.1 MESSAGE PASSING .................................................. 29
3.2 MANIPULATING INTERFACE NAMES .............................. 32
   3.2.1 QUERYING THE BUS FOR INTERFACE NAMES ............. 32
   3.2.2 RECEIVING MESSAGES ON ANY INTERFACE .............. 34
3.3 ATTRIBUTES ...................................................... 35
   3.3.1 OBJECT ATTRIBUTES .................................. 35
   3.3.2 INTERFACE ATTRIBUTES ................................. 38
3.4 NON-BLOCKING CHECK FOR MESSAGES .......................... 39
   3.4.1 QUERYING A PARTICULAR INTERFACE .................... 39
   3.4.2 QUERYING ANY INTERFACE ............................... 41
3.5 POLYLITH BUS RUNTIME OPTIONS .............................. 41
   3.5.1 DIRECT CONNECT ......................................... 41
   3.5.2 KEEP-ALIVE ............................................. 43
   3.5.3 VERBOSE, LOGFILE .................................... 43
3.6 SUMMARY OF ADVANCED FEATURES .............................. 43
   3.6.1 MIL STATEMENTS ........................................... 43
   3.6.2 POLYLITH BUS CALLS ..................................... 45
   3.6.3 POLYLITH BUS RUNTIME OPTIONS .......................... 47

BIBLIOGRAPHY 49
Chapter 1

OVERVIEW

POLYLITH is a software interconnection system. It allows programmers to configure applications from mixed-language software components (modules), and then execute those applications in diverse environments. Communication between components can be implemented with TCP/IP or XNS protocols in a network; via shared memory between light-weight threads on a tightly-coupled multiprocessor; using custom-hardware channels between processors; or using simply a ‘branch’ instruction within the same process space.

The principle feature of POLYLITH is that the components can be implemented separately from the implementation of interfacing between those components. In turn, this provides a ‘divide and conquer’ capability for software engineers, who know that simultaneous treatment of functional and interfacing requirements within the same program makes it costly to maintain and difficult to reuse elsewhere. POLYLITH represents a software organization where interfacing decisions can be encapsulated separately, using a software bus. The bibliography lists several papers where POLYLITH has been either described or utilized in other research. A key description of the abstract result is given in [Purt90].

To date, POLYLITH has been in greatest demand by programmers who wish to use one particular software bus — the TCP/IP-based network bus. POLYLITH helps these users write applications for distribution across mixed-architecture host processors. This document is written for such users. All examples are presented in terms of distributed applications to be executed in a network. In this context, the POLYLITH bus provides message-passing primitives to handle communication between the processes, performing data transmission and any necessary coercion. This use of POLYLITH makes several assumptions:

- There is no shared memory between processes.
- Communication between processes is implemented exclusively via channels defined and controlled by POLYLITH.
- The basic communication operation provided by POLYLITH is message passing. These
messages can be used to build remote procedure call (RPC), for sending and receiving variables of any data type (structured or atomic), or for synchronization (by passing empty messages).

- Since POLYLI TH controls the communication channels, it provides any necessary data coercion between modules written in different languages, or between modules which are instantiated on different hosts.

Later forms of this manual will be written to help users who wish to implement interfacing decisions that involve shared memory or other organizations. Moreover, there are several additional tools that make using POLYLI TH much easier. These tools — such as the packager [CaPu90] and languages for manipulating interface declarations [PuAt91] — are not described here. We focus only upon POLYLI TH-ic organization. Finally, for simplicity in presentation, we give most of our examples in the C language. This may seem strange for something purporting to be a ‘mixed language programming system’ but it simplifies the preparation of a manual such as this document. Examples of how POLYLI TH interconnects components from other programming languages appear in the distribution, which will be expanded as refined language interfaces accumulate over time.

The remainder of this chapter sketches the major steps a user must perform to create a simple application in POLYLI TH. Then Chapter 2 goes through the sketch filling in the details. Chapter 3 describes how to use more ‘advanced’ features of our system. The appendices of this document contain much of the same material as the earlier chapters, except they are organized for use as a reference guide.

SKETCH OF SAMPLE APPLICATION IN POLYLI TH

A module is ‘any identifiable program unit’. For now, think of them as self-contained C programs having communication channels that can be bound to corresponding ports on other programs. Each module can be invoked many times within an application configuration — the running forms of these modules are called processes. Each process is given a unique instance name.

A note on terminology: Sometimes we refer to processes as being ‘modules’ since in the general POLYLI TH formulation an instantiated module might not be a separate process. Hopefully the context of use will make this usage unambiguous. Other times we slip and call the processes tasks, and long-time POLYLI TH users will know to call these services too. Finally, since the abstract description of this system uses a graph model of interconnection — modules correspond to nodes in the application graph, and bindings between interfaces correspond to arcs in the graph — we sometimes refer to the modules as being nodes.

Modules are said to have an ‘interface’ for each communication channel upon which the process will send or receive messages. Processes communicate through a software bus by invoking message passing routines provided by a POLYLI TH library, linked into the application. Calls to the POLYLI TH message-passing routines require the programmer to reference one of its interfaces.
Ultimately, users would not want to install these ‘bus calls’ manually, but rather they would allow an automatic packager tool to create appropriate network stubs for interfacing to the bus.

Once the program code for the modules of the application are written, the programmer describes the configuration of the application using the POLYLIUHI module interconnection language (MIL). The MIL declaration includes:

- A definition of each module in the application, describing where the program code for this module resides; where the module is to execute; and what communication interfaces the module has.

- A definition of the application itself, describing what modules are included in the application plus how those module interfaces are bound together to form a communication channel.

Figure 1.1 shows an example of two programs that are used as modules in an application. These are C programs that each call the POLYLIUHI message passing routines `mh_write` using interface `out`, and `mh_read` using interface `in`. Program a.c sends a message containing the string "msg1" and receives a message into variable `str`. Program b.c receives a message into its variable `str`, and sends a message containing the string "msg2". Although we intend to attach interface `out` in one program to interface `in` in the other, that fact is not in any way encoded in the program code.

Figure 1.2 (left) describes the information contained in the MIL definitions of the two modules. Module A is instantiated by program a.c and declares interfaces `in` and `out`, while module B uses program b.c and interfaces `in` and `out`. There is still no connection between the two modules, but now the modules can be used in a POLYLIUHI application.

Figure 1.2 (right) depicts what information the application portion of the MIL definition contains. The application has three nodes: node `foo` is instantiated with module A, and both nodes `bar` and
module A :
implementation : "a.out"
outgoing interface: "out" sends a string
incoming interface: "in" receives a string

module B :
implementation : "b.out"
outgoing interface: "out" sends a string
incoming interface: "in" receives a string

Figure 1.2: Information provided by the application’s MIL program; module definitions (left); application definition (right).

service "A" : {
implementation : { binary : "a.out" }
source "out" : { string }
sink "in" : { string }
}

service "B" : {
implementation : { binary : "b.out" }
source "out" : { string }
sink "in" : { string }
}

orchestrate "example" : {
  tool "foo" : "A"
  tool "bar" : "B"
  tool "bartoo" : "B"
  bind "foo out" "bar in"
  bind "bar out" "bartoo in"
  bind "bartoo out" "foo in"
}

Figure 1.3: The actual MIL program; module definitions (left); application definition (right).
**Figure 1.4:** Runtime instantiation of the application.

**Figure 1.4:** Runtime instantiation of the application.

**bartoo** use module B. We bind interface out of node **foo** to interface in of node **bar**, bind **bar**'s out to **bartoo**'s in, and bind **bartoo**'s out to **foo**'s in. Figure 1.3 shows the actual **POLYLITH** MIL program that corresponds to the description given in Figure 1.2. Now the application is complete.

An application runs under **POLYLITH** by invoking an appropriate implementation of the **POLYLITH** bus — in this case, the TCP/IP-based 'network bus' is what is appropriate. The bus is given a MIL program (module definition and application definition) as input, which it uses to start up the appropriate UNIX process for each of the nodes, and to set up the communication channels between these nodes.

Since each node is a separate process, each has its own thread of control and each starts executing as soon as it is created. These nodes are just concurrent processes until one of them issues a **POLYLITH** command to read a message from an interface; at that point the node blocks until a message arrives on the interface. In the application described in Figures 1.1 and 1.2, nodes **bar** and **bartoo** block immediately, waiting for a message to arrive. Node **foo** sends a message, then blocks. The messages passed in this application act to synchronize the concurrent nodes, which causes the flow of control to start in node **foo**, pass to node **bar** when **foo** sends the string "msg1", pass to node **bartoo** when **bar** sends the string "msg2", and return to node **foo** when **bartoo** sends the string "msg2".

Coordination of the message passing is also the responsibility of the **POLYLITH** bus. Messages
are not sent directly to other nodes, but are sent to the bus to be forwarded to the appropriate
node. Figure 1.4 shows the runtime instantiation of the application, with the bus coordinating
and channeling communication between nodes.
Chapter 2

BASIC FEATURES

In this chapter we explain the basic POLYLITH features by using these features in three different applications. The first example starts with a very simple program that we modify to run on a POLYLITH bus. The next example is an enhancement of the first application; we use it to demonstrate how POLYLITH makes it easy to reuse modules. The third example shows how to send complex, structured messages in POLYLITH. We end with a summary of the features that were introduced in this chapter.

2.1 CREATING AN APPLICATION

Our goal is to create an application in POLYLITH which behaves like the program shown in Figure 2.1, where the main routine passes a string to an external routine print, which writes the string to standard output. We will use POLYLITH to invoke the print routine as a remote procedure call: we put the two routines into separate modules and run the print module on a remote host, using the POLYLITH bus to communicate between the modules (Figure 2.2).

```
Basic/Hello0/main.c
extern print();

main()
{
    print ("Hello, world");
}

```

```
Basic/Hello0/print.c
#include <stdio.h>

print(s)
char *s;
{
    printf("%s\n", s);
}
```

Figure 2.1: Simple Hello0 example.
Hello, world

Figure 2.2: Hello1, the POLYLITH version of simple Hello0 example.

```
::: basic
Basic/Hello/hello.cl
::: service "main" {
  implementation: { binary: "./hello.exe" }
  client "print": {string} accepts {}
}
```

Figure 2.3: MIL program for module main in Hello1 application.

To implement a remote procedure call (RPC), the caller sends a message containing the parameters of the call, and the remote procedure returns an empty message when it has finished. Our obligations are to modify the main and print routines, and to create a POLYLITH MIL program for the application. For this application, we fulfill these obligations incrementally by:

- creating a POLYLITH MIL program to describe the main module
- modifying the main module to interact with the bus instead of directly with module print
- creating an MIL program for module print
- modifying module print to communicate with the bus
- creating a third MIL program to bind the interfaces of the two modules
- compiling, linking, and running the application

For flexibility we are writing the MIL program in three separate pieces: a description for each of the two modules, and the application description. These three will be compiled separately but linked into a single POLYLITH MIL program prior to running the application. The three pieces could be combined into a single physical file if the modules were to run on the same host, or if they were not expected to be used in any other application; this is an application design decision.
Basic/Hello/hello.c

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char **argv;
{
  mh_init (&argc, &argv, NULL, NULL);
  mh_write("print", "S", NULL, NULL, "Hello, world");
  mh_read("print", ",", NULL, NULL);
  mh_shutdown(0, 0, ",");
}
```

Figure 2.4: Source program for module main in Hello1 application.

### 2.1.1 MIL PROGRAM FOR MODULE main

The purpose of the MIL program for module main is to specify certain of its properties for the Polylith bus, including the location of the executable code and the communication interfaces the module uses. Figure 2.3 shows the complete MIL program for module main. The module is named immediately following the keyword service, and the description of the module is enclosed in braces (Figure 2.3). The implementation statement ties this abstract module description to an executable program by giving the executable's pathname from the current directory; the executable is named following the keyword binary. Any remaining statements describe the communication interfaces that the module (thus the executable program) uses; here we have just one interface.

Since module main will no longer call the print routine directly, we create a communication interface to act as a substitute for the call/return. We name this communication interface “print”, and describe the data that can be sent and received via this interface: our intention is to use this interface to communicate with module print, so we want to be able to send a string and we expect an empty message in return. This information is conveyed in the statement

```
client "print" : {string} accepts { }
```

which defines “print” as an interface that initiates communication by sending something of datatype string, and receives an empty message in return. The interface we have just defined is bidirectional; Chapter 3 describes how to define a one-way interface.
2.1.2 SOURCE PROGRAM FOR MODULE main

Figure 2.4 shows the changes we make to convert our original main routine into the source program needed for module main. The key change involves replacing the call to routine print with calls to the Polylith bus to effect a call to the module print (which we haven’t yet written).

We invoke module print by sending it a message containing the parameter “Hello, world”, but don’t send the message directly to module print. Instead we use the mh_write bus call to instruct the bus to forward our message to the module at the other end of the interface. Using mh_write, we send the bus:

```
"print"       the name of the interface we’re using
"S"           the format of the message we’re sending; here it’s string
"Hello, world" the message itself; the string parameter for print
```

Notice in Figure 2.4 that there are two NULL parameters between the message format parameter and the message string; these are placeholders for features that will appear in a future release of Polylith. The interface name “print” and the message format “S” match the interface declared in the client statement of our MIL program¹.

We intend to attach module print to the other end of the interface “print”, where it will wait for a message from its caller, then print the string it received, then return an empty message on the same interface in place of an ordinary procedure return. The mh_read bus call completes the communication with module print; we specify:

```
"print"       the name of the interface
""           the format of the message we expect to receive: an empty message
```

Notice in Figure 2.4 that the mh_read also has two NULL parameters that refer to features not available in this release of Polylith. The mh_read is a blocking call: execution will not continue past that point until a message is received on the named interface. Here our reason for blocking to wait for an empty message is to synchronize with the other module, thus completing the remote procedure call to that module.

The remaining requirements for module main are to pass the command line arguments to mh_init:

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

¹ Compare the message pattern {string} used in defining the “print” interface in Figure 2.3 with the message format "S" used in the mh_write statement in Figure 2.4. The message patterns used in the MIL are more descriptive, making the MIL programs more readable. The message formats used in Polylith bus calls are abbreviated to streamline the bus calls. Normally, a packager tool will automatically generate these bus calls, so the programmer will not see a discrepancy between message patterns and message formats.
which sets up a communication channel between the module and the bus (this is something each module must do). The main module must also call `mh_shutdown(0,0,"")` to terminate the application. When the application is running, the first `mh_shutdown(0,0,"")` encountered shuts down all nodes in the application, i.e. the entire application.

### 2.1.3 MIL PROGRAM FOR MODULE `print`

The MIL program for the `print` module is shown in Figure 2.5. Again we use the `implementation` statement to specify the executable program for this module: the file name follows the keyword `binary`, and the machine where the process will run follows the keyword `machine`. Naming a remote host for the `machine` attribute allows us to distribute the application. When the `machine` attribute is not specified, by default the module runs on the same host as the POLYLI TH bus. When running a module on a remote host, you must make sure that the `.rhosts` file on the remote machine contains the name of the machine where the bus is executing.

Now we must declare an interface so that this module can communicate with module `main`: we expect this interface to receive a string and return an empty message, so that it matches the interface declared in module `main`. The statement

```}

function “print” : {string} returns { }
```

defines “print” as an interface that receives something of datatype string, and returns an empty message. This function interface is bidirectional, and must be bound to a client interface like the one in module `main`. Although we gave this interface the same name as the interface in module `main`, we could have named it something different. The binding between the interfaces will be explicitly stated in the third MIL program, and does not depend on the names of the interfaces being identical.

### 2.1.4 SOURCE PROGRAM FOR MODULE `print`

The first difference between the original `print` routine and the new `print` module in Figure 2.6 is that the C procedure in the module is not named “print” but is named “main”. Recall
Basic/Hello/print.c

#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{
    char s[256];

    mh_init(&argc, &argv, NULL, NULL);
    mh_read("print", "S", NULL, NULL, s);
    printf("%s
", s);
    mh_write("print", ",", NULL, NULL);
}

Figure 2.6: Source program for module print in Hello1 application.

that when a POLYLITH application starts up, it creates at each node an independent process, so each module associated with a node must contain a procedure “main”. When a C program is compiled, linked, and loaded into a process, execution starts at procedure “main”. But we want this particular module to behave like a procedure that is called by another, so we put an mh_read bus call immediately following the mh_init. The mh_read blocks the module until another module sends a message, thereby initiating a “procedure call” to this print module. The mh_read parameters we use are:

- "print" interface name
- "S" message format is string
- s a variable of type string; used to receive the message

The two NULL parameters refer to features not available in this release of POLYLITH. Predictably, these parameters match the parameters of the corresponding mh_write call in module main. After printing the string to standard output, we make the mh_write bus call that the main module is waiting for. Note that we do not need an mh_shutdown call here, since this module is behaving as a remote procedure, and expects another module to do the bus shutdown.

2.1.5 MIL PROGRAM FOR THE APPLICATION

Figure 2.7 shows the last part of the MIL program for our application. First we give the application the name “one_hello” using the keyword orchestrate, then enclose the application description in brackets. The nodes comprising our application are listed one by one using the tool statement, which lists the node name and the module that will instantiate the node:
tool “node” : “module”

Our first node is named “prog” and instantiated with module main, and our second node is named “print” and instantiated with module print:

  tool “prog” : “main”
  tool “print”

The general form of the tool statement specifies both a node and a module, but if the two names are the same, we can specify just the module. The following two statements are equivalent:

  tool “print”
  tool “print” : “print”

The bind statement specifies two interfaces that are to be connected by the Polylith bus. The first interface listed is the initiating interface, the one that will issue the first mh_write bus call. Both the node name and the interface name must be specified, since interface names are unique only within a module:

  bind “node_i interface_i,j” “ node_r interface_r,k”

where node_i has a j^{th} interface interface_i,j which matches the k^{th} interface of node_r. In this application, node “prog” was instantiated with module main, which contains interface “print”; and node “print” was instantiated with module print, which contains interface “print”. So we bind “prog print” to “print print”. The compiler just assumes that such module descriptions exist, but when we later link these three MIL programs together, the linker must find a description for a module named main which has an interface named “print”, and another description for a module named print which has an interface named “print”.

13
************
Basic/Hello/Makefile
************

all: hello1 hello2

hello1: hello1.mh hello.exe print.exe
hello2: hello2.mh hello.exe print.exe dup.exe

hello1.mh: hello.co print.co hello1.co
    csl hello.co print.co hello1.co -o hello1

hello2.mh: hello.co print.co dup.co hello2.co
    csl hello.co print.co dup.co hello2.co -o hello2

hello.exe: hello.o
    cc -o hello.exe hello.o -lith

print.exe: print.o
    cc -o print.exe print.o -lith

dup.exe: dup.o
    cc -o dup.exe dup.o -lith

hello.co: hello.cl
    csc hello.cl

hello1.co: hello1.cl
    csc hello1.cl

hello2.co: hello2.cl
    csc hello2.cl

print.co: print.cl
    csc print.cl

dup.co: dup.cl
    csc dup.cl

install:
    cp print.exe /jteam/crh/print.exe

clean:
    rm -f *.o *.exe *.co hello1 hello2

Figure 2.8: Makefile for applications **Hello1** and **Hello2**.
2.1.6 COMPILING, LINKING, AND RUNNING THE APPLICATION

The Makefile in Figure 2.8 contains the commands needed to compile and link this application, called **hello1** in the Makefile. We compile and link each of our modules with the commands:

```
cc hello.c -c
cc -o hello.exe hello.o -lith
(for module main)
cc print.c -c
cc -o print.exe print.o -lith
(for module print)
```

creating the executable files **hello.exe** and **print.exe**. The `make install` copies **print.exe** to `/jteam/crh/print.exe`, because the executable files must reside in the same file directory as that specified in the implementation ... binary attribute in our MIL program. Note in the `cc` command that the **POLYLITH** library routines are linked into the modules by specifying the library `-lith`. Then we compile each of the three parts of the MIL program using the command `csc`. File **hello.cl** contains the description of module **main**, file **print.cl** contains the description of module **print**, and file **hello1.cl** contains the application description (Figures 2.3, 2.5, and 2.7). We execute:

```
csc hello.cl
ncsc print.cl
ncsc hello1.cl
```

creating the compiled versions in files **hello.co**, **print.co**, and **hello1.co**. Next we link these three files with the `csl` command:

```
csl hello.co print.co hello1.co -o hello1
```

creating the output file **hello1**; this is the file we pass to the **POLYLITH** bus. Finally, to run the application, we start up a Polylibth bus, passing it this file **hello1**:

```
bus hello1
```

2.2 ENHANCING THE APPLICATION

In this section we build a new application that is based on the application just presented. We will reuse modules **main** and **print**, and show how **POLYLITH** allows us to rebind their interfaces
without making changes to the modules themselves. The new application is shown in Figure 2.9: we have inserted a new module, called DUP, between the main and print modules. Module DUP serves as a duplicator by taking the message from module main and sending it to two print modules. This example also shows us how POLYLITH lets us use multiple instantiations of a module in an application. We are not simply invoking the print module twice; we are creating two independent nodes from the print module, each of which has its own binding to the DUP module.

2.2.1 MODULE DUP

The MIL program and source program for module DUP is shown in Figure 2.10. We continue to structure the interactions between modules as remote procedure calls: module DUP is invoked when it receives a string on its “server” interface. It initiates remote procedure calls on the “print1” and “print2” interfaces by sending the string received from the “server” interface. The \texttt{mh\_read} commands for “print1” and “print2” signal the completion of the remote procedure calls, and module DUP ends by sending an empty message on interface “server”, signaling to the caller that it has finished.

Remember that module DUP has no knowledge of where the string sent on interfaces “print1” and “print2” will end up. We intend to bind these interfaces to print modules, but we could instead bind them each to another DUP module and print out four copies of the string. Or we could bind “print1” and “print2” to different kinds of print modules, one printing to standard output and the other printing to a file. We could even bind both interfaces to the same module, provided the module had two matching interfaces.
#include <stdio.h>

main(argc, argv)

int argc;

char **argv;

{ char s[256];

mh_init(&argc, &argv, NULL, NULL);

mh_read("server", "S", NULL, NULL, s);

mh_write("print1", "S", NULL, NULL, NULL);

mh_read("print2", "S", NULL, NULL, NULL);

mh_write("print1", "", NULL, NULL, s);

mh_write("print2", "", NULL, NULL, NULL);

mh_write("server", "", NULL, NULL);

}

Figure 2.10: Module DUP for Hello2 application; MIL program (left), source program (right).

## 2.2.2 MIL PROGRAM FOR THE APPLICATION

The MIL program for this application is again composed of four parts: three module descriptions and an application description. We can reuse the module descriptions for main and print from the previous application (Figures 2.3 and 2.5); we just saw the module description for DUP (Figure 2.10), and the new application description is shown in Figure 2.11.

The four tool statements describe the nodes of the application: node “main” uses module main, node “DUP” uses the new module DUP, node “print1” uses module print, and node “print2” also uses module print. Now it’s clear why the node name can’t always be the same as the module name. The bind statements connect node “main” interface “print” to node “DUP” interface “server”, and connect the remaining interfaces of node “DUP” to the two print nodes. We were able to reuse the main and print modules without making any changes to their source or MIL programs.

Now that we’re assured that the “print1” and “print2” interfaces of module DUP are connected to separate print modules, we can see that our duplicator does not guarantee that the output from the two print modules will not be interleaved. The second print module is invoked before waiting for the return from the first. If we were to change the duplicator to wait for the return from the first before invoking the second, we could be sure that the output would not be interleaved.
2.3 SENDING STRUCTURED MESSAGES

The application in this section demonstrates how to send and receive a message containing something other than a character string. Not only can we send variables of many data types, including structures and arrays, but we can also send several variables in one message.

The simple **Phonebook** application interacts with the user to get a name, looks up the phonebook entry corresponding to the name, and then displays that entry. The application graph and Makefile in Figure 2.12 show the two modules we use, **main** and **book**. Module **book** contains the phonebook database and provides the lookup function: it waits to receive a string containing a person’s name, then it looks in the database for the corresponding phone extension and returns the database entry to the caller. Module **main** is the caller: it prompts interactively for a name, sends the name to module **book**, and receives and prints the phone extension. Typing an empty string at the prompt terminates the application.

### 2.3.1 SOURCE PROGRAM FOR MODULE main

The source program for module **main** is given in Figure 2.13. When an empty string is entered for the name prompt, the while loop ends and the **mh_shutdown** is executed, so this module controls the termination of the application. Within the loop, the **mh_write** is no different from what we’ve seen before: we’re sending a character string. But the **mh_read** incorporates new features: we’re receiving values for two variables in the message, **found** and **entry**. The message format (in the second parameter) indicates the number of and type of variables that are to be included in the message. These variables are listed in the **mh_read** call starting at the fifth parameter.

The message format for this particular **mh_read** call is "b{SI}", indicating that the first variable is a pointer to a boolean and the second is a pointer to a structure containing a string and an
integer. A complete list of message format types is provided in the next section's summary. To receive multiple variables, you concatenate their types in the message format; for example, to receive values for four boolean variables and two strings, you would use a message format of "bbbbSS". Since C does not have a boolean native type, we declare the boolean variables to be integer. The variable received for a "b" message format must be int, not char, because when the program is written in C, the bus expects a boolean to be the same size as an int. The bus will transmit the message incorrectly if the variables do not correspond to the message format.

It is important to note that all parameters corresponding to variables in an mh_read call must be pointers to the variables, and not the variables themselves. This is a constraint imposed by the C language, which allows only value parameters; other languages do not necessarily have this constraint. So we pass the address of found and the address of entry, and the mh_read command sets the values of the variables at these addresses. Also remember that the storage for these pointer variables must be allocated; if you declare

```c
struct table *p
```

and call

```c
mh_read ("lookup", "{SI}", NULL, NULL, p)
```
```
#include <stdio.h>

struct table {
    char *key;
    int value;
};
char key[256];
struct table entry;

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

    mh_init (&argc, &argv, NULL, NULL);

    printf("Name? ");
    while (strcmp(gets(key),"")) {
        mh_write("lookup", "S", NULL, NULL, key);
        mh_read("lookup","b[SI]",NULL,NULL,&found,&entry);
        if (found)
            printf("Ext. %d\n", entry.value);
        else
            printf("%s not found.\n", key);
    }
    mh_shutdown(0, 42, "");
}
```

Figure 2.13: Source code for module main in Phonebook application; POLYLiTH version (left); original (right).
you will overwrite whatever \texttt{p} points to, whether or not it was declared as \texttt{struct table}. Here we allocate storage for \texttt{entry} by declaring it to be of type \texttt{table}, which we define at the top of the file.

### 2.3.2 SOURCE PROGRAM FOR MODULE \texttt{book}

Since we expect module \texttt{book} to send a database entry to module \texttt{main}, we must use the same structure definition in both modules. Following the \texttt{struct table} definition in module \texttt{book} (Figure 2.14) is the declaration and initialization of the database \texttt{db}, an array of type \texttt{struct table}. The first entry in the database is a dummy entry for the module to pass back when no entry is found to match the key, because the \texttt{mh\_read} in module \texttt{main} must receive something of type \texttt{struct table} in all cases.

Notice that the while loop does not terminate, or rather does not terminate until the application itself is terminated by the \texttt{mh\_shutdown} in module \texttt{main}. Inside the while loop, we use \texttt{mh\_read} to get the key, then search the database for an entry to match that key. The message format "B{SI}" for the \texttt{mh\_write} indicates that the fifth parameter is a boolean variable (\texttt{int} in \texttt{C}), and the sixth is a pointer to a structure containing a string and an integer. Since here we are passing the variables \texttt{to} the bus, the value parameter restriction in \texttt{C} is not a problem; we can pass the value of variable \texttt{found}, instead of passing a pointer to the variable as we had to do with \texttt{mh\_read}. If a matching entry was not found, we send \texttt{&db[0]} to fill in the sixth parameter with something of the correct data type. If we were to send \texttt{&db[4]}, which is initialized to \texttt{NULL}, the message format would not match the variables, and the bus could not successfully deliver the message.

### 2.3.3 MIL PROGRAM FOR THE APPLICATION

The MIL program for the phonebook application, shown in Figure 2.15, combines the two module descriptions and the application description in one file. The description of interface \texttt{lookup} in module \texttt{main} contains a message pattern we haven't seen before: \{ \texttt{"boolean; \langle string; integer\rangle"} \}. This message pattern matches the message format parameter of the \texttt{mh\_read} call in module \texttt{main}, although the data type names differ. In a MIL message pattern, the type name is written out in full, pointer types are preceded by the symbol \texttt{^} (except that a pointer to a structure uses the symbols \texttt{<message\_pattern>}), and concatenation is indicated by a semicolon. The MIL message pattern types are summarized in the next section. This particular message pattern indicates that the message will contain a pointer to a boolean variable, and a pointer to a structure containing a string and an integer. Compare this pattern to its correspondent on the \texttt{lookup} interface of module \texttt{book}, which sends a boolean variable instead of a pointer to a boolean. As we just discussed, the \texttt{mh\_write} in module \texttt{book} passes the value of the boolean variable, not a pointer to the boolean variable, so the message pattern in our MIL program must reflect that.\footnote{In fact, the message patterns in the MIL program do not need to match the message actually sent. Our MIL program could have declared interface \texttt{lookup} to read and write \{ \texttt{"string"} \} everywhere, leaving the \texttt{mh\_read}}
Figure 2.14: Source code for module book in Phonebook application; POLYLIB version (left); original (right).
## 2.4 SUMMARY OF BASIC POLYLITH FEATURES

This section summarizes the Polylith features that are presented in Chapter 2. A complete summary of features is presented in the appendices.

### 2.4.1 MIL STATEMENTS

The Polylith MIL program consists of a description for each module and a description of the application. These can be in separate files, combined in a single file, or a mix of both.

```
module_description1

module_description2

...

module_description_m

application_description
```

and **mh_write** commands in both modules exactly as they are, and the bus would correctly pass what is stated in the modules. The message formats in the **mh_read** and **mh_write** calls must match the rest of their parameters; the bus uses these, not the declarations in the MIL program, to decode the parameters. In the next chapter we will see how to use the declarations from the MIL program to avoid coding the message formats explicitly in the **mh_read** and **mh_write** calls; we can instead query the bus to find out the message format stated in the MIL program.
 MODULE DESCRIPTION  A description must appear for each module used in the application. The module_name given to the module is used to identify it in the application description. This name is not referred to in the source program for the module.

```
service "module_name" : {
    implementation_statement
    interface_statement_1
    interface_statement_2
    ...
    interface_statement_k
}
```

 IMPLEMENTATION STATEMENT  The implementation statement names the program which implements this module and the host on which the module is to be created.

```
implementation : { impl_attr_name_1 : "impl_attr_value_1" ... impl_attr_name_j : "impl_attr_value_j" }
```

Table A.1 shows the attributes that can be used in the implementation statement. We’ve only discussed two implementation attributes so far: binary and machine. The value of the binary attribute is the pathname of the executable program, and the value of the machine attribute is the name of the host where the module will run. If the machine attribute is not specified, the module will run on the same host as the Polylith bus.

 INTERFACE STATEMENT  An interface statement must appear for each interface that the module expects to use; these statements declare the interface name and define what type of data will be passed. The message patterns used to describe each interface have the format:

```
msg_pattern ≡ t_1; t_2; ...; t_n
```

where \( t_i \) is the pattern type of the \( i^{th} \) variable in the message, and \( n \) is the number of variables passed in each message (\( n \) can be zero). Note that the semicolon is used to concatenate the pattern types into a message pattern. The interface pattern types are shown in Table A.2.

The client and function interface statements are used in this chapter:

```
client "interface_name" : { msg_pattern_out } accepts { msg_pattern_in }
```
where \textit{interface\_name} is the name of a bidirectional interface that initiates communication with \textit{msg\_pattern\_out}, the outgoing message pattern, and accepts a message in return with \textit{msg\_pattern\_in}, the incoming message pattern. This type of interface must be bound to a \textbf{function} interface:

\begin{verbatim}
function "interface\_name": \{ msg\_pattern\_in \} returns \{ msg\_pattern\_out \}
\end{verbatim}

where \textit{interface\_name} is the name of a bidirectional interface that accepts communication using \textit{msg\_pattern\_in}, the incoming message pattern, and returns a message with \textit{msg\_pattern\_out}, the outgoing message pattern.

**APPLICATION DESCRIPTION** The application description uses the \textit{tool\_statement} to name the nodes of the application and instantiate them with modules, then uses the \textit{bind\_statement} to bind the interfaces of these nodes.

\begin{verbatim}
orchestrate "application\_name": {
    tool\_statement\_1
    tool\_statement\_2
    ...
    tool\_statement\_n
    bind\_statement\_1
    bind\_statement\_2
    ...
    bind\_statement\_z
}
\end{verbatim}

**TOOL STATEMENT** The tool statement defines a node in the application by naming the node and specifying which module instantiates the node:

\begin{verbatim}
    tool "node\_name": "module\_name"
\end{verbatim}

Another version of the tool statement specifies that the node name is the same as the name of the module that instantiates it:

\begin{verbatim}
    tool "module\_name"
\end{verbatim}

One of these tool statements must be included for each node in the application.
BIND STATEMENT

The purpose of the bind statement is to connect the interfaces of the nodes.

\[
\text{bind } \text{"node}_i\text{ interface}_i\text{" } \text{"node}_r\text{ interface}_r\text{"}
\]

where \( \text{node}_i \) initiates the communication, and has a \( j^{th} \) interface \( \text{interface}_i \). This interface matches the \( k^{th} \) interface of \( \text{node}_r \).

2.4.2 POLYLITH BUS CALLS

The source program for each module calls routines from the POLYLITH bus library to send or receive communication on its interfaces. Before using these interfaces, the program must get the command line arguments and pass them to \text{mh}\_init:

```c
main(argc, argv)
int argc;
char **argv;
{ ... mh\_init (&argc, &argv, outfaces, infaces) ... }
```

to declare its interfaces to the bus. Parameters \( \text{outfaces} \) and \( \text{infaces} \) should be set to NULL unless you are using the direct connect (-d) bus option (see Section C.4.1).

In this chapter, we use \text{mh}\_write and \text{mh}\_read to send and receive messages. A message contains a set of variables or expressions that are passed to the bus. The bus either copies these into its memory (for sending), or uses them as addresses where data is to be put (for receiving). A message format must accompany each message to indicate the type of every variable or expression in the message:

\[
\text{msg}\_\text{format} \equiv t_1 t_2 \cdots t_n
\]

where \( t_i \) is the type of the \( i^{th} \) variable in the message, and \( n \) is the number of variables passed in each message (\( n \) can be zero). The types are concatenated to form a message format. These message format types are shown in Table B.2. Because C has only value parameters, \( \text{msg}\_\text{format}_{\text{in}} \) (the incoming message format) can contain only the pointer types, and \( \text{msg}\_\text{format}_{\text{out}} \) (the outgoing message format) can contain either the pointer or the value types.

To send a message, use:

\[
\text{mh}\_\text{write } \text{"interface}_i\text{" } \text{"msg}\_\text{format}_{\text{out}}\text{" } \text{NULL} \text{, NULL } w_1 w_2 \vdots w_n
\]
where \textit{interface\_name} is the name of an outgoing interface declared in the MIL description of the module, each \( t_i \) in \textit{msg\_format\_out} is the data type of variable (or expression) \( w_i \), and \( n \) is the number of variables in the message (can be zero). The \texttt{NULL} parameters are placeholders for features which will be available in a future release of PolyLith.

To receive a message, use:

\[
\texttt{mh\_read ("interface\_name", "msg\_format\_in", NULL, NULL, } r_1, r_2, \ldots, r_n)\
\]

where \textit{interface\_name} is the name of an interface declared in the MIL description of the module, \( r_i \) is a pointer variable or the address of a variable, each \( t_i \) in \textit{msg\_format\_in} is the pointer type of variable \( r_i \), and \( n \) is the number of variables in the message (can be zero). The \texttt{NULL} parameters are placeholders for features which will be available in a future release of PolyLith.

To terminate an application or a node, use:

\[
\texttt{mh\_shutdown( level, exit\_code, exit\_string )}\
\]

When \( level=0 \), this command notifies the bus that the application is finished. The bus terminates execution at each node and releases all the application’s communication channels. If the application is not terminated, all nodes could finish execution, but the bus would keep the application running and hold its communication channels open.

When \( level=1 \), the \texttt{mh\_shutdown} command terminates just the node issuing the command, and not the entire application. When \( level=2 \), the command acts like an \texttt{exit} command, terminating the process at the node without notifying the bus.

For shutdown \texttt{levels} of 0 or 1, the \texttt{exit\_code} and \texttt{exit\_string} parameters are written in the \texttt{logfile} when the logging option -l is turned on (see section C.4.3). The \texttt{exit\_code} is an integer value, and the \texttt{exit\_string} is a character string.

### 2.4.3 Compiling, Linking, Running

Each module must be compiled and linked using its native language compiler and including the PolyLith library (with the \texttt{-lith} flag). The executable file that is created must be named in the \texttt{binary} attribute of the \texttt{implementation} statement in that module’s MIL description. For example,

\[
\texttt{cc main.c -c} \\
\texttt{cc -o main.exe main.o -lith}\
\]
compiles `main.c` into `main.o`, and links `main.o` with the routines it uses from the POLYLITH library, creating the executable file `main.exe`.

The components of the MIL program are the module descriptions and the application description; they can be compiled in separate files or in one file.

```
csc phonebook.cl
csc main.cl
```

compiles the MIL program `phonebook.cl` into `phonebook.co`. These compiled components are linked using:

```
csl phonebook.co main.co -o phonebook
```

Here the two compiled MIL components `phonebook.co` and `main.co` are linked, creating the output file `phonebook`.

To run an application, we start up a POLYLITH bus, passing it our compiled and linked MIL program:

```
business phonebook
```

An application that does not terminate voluntarily can always be terminated with a control-C.
Chapter 3

ADVANCED FEATURES

This chapter describes the advanced POLYLITH features by presenting six variations of an application. The basic version of the application introduces one-way interfaces. The next two versions show two different ways to avoid explicitly naming the interfaces in a module’s source code; the interfaces are named only in the MIL program. The fourth application shows how the MIL is used to specify additional attributes for a module, and how to query the POLYLITH bus for attribute information instead of coding it directly in a module. The last two versions demonstrate two ways of doing a non-blocking read to receive messages. A summary of the features presented in this chapter appears at the end of the chapter.

3.1 MESSAGE PASSING

In chapter 2, we used bidirectional interfaces because each interface both sent and received messages. The application in this section, called Source_sink, introduces one-way interfaces and shows how a node can query the bus for its name. The application, shown in Figures 3.1, 3.2, and 3.3 uses a module called hello, which just sends its name on interface “send”. This module is used to instantiate three nodes of the application, hello, hi, and greetings. The print module instantiates node print; it reads a string from each of its interfaces “msg1”, “msg2”, and “msg3” and prints the strings.

First we discuss the one-way communication interfaces. The interfaces in this application either send messages or receive messages, but do not both send and receive. We could use bidirectional interfaces here, even though each interface communicates in only one direction, but one-way interfaces are sufficient. The mh_read and mh_write calls are exactly the same as for a bidirectional interface; the only difference is that a module does not call both mh_read and mh_write on a particular interface.

One-way interfaces are declared in the module description portion of the MIL program with the
Figure 3.1: Basic application structure (used in first three examples).

```
#include <stdio.h>

main(argc, argv) 
int argc;
char **argv;
{ char objname_buf[256];

   mh_init(&argc, &argv, NULL, NULL);

   mh_identity(objname_buf, sizeof(objname_buf));
   mh_write("send", "S", NULL, NULL, objname_buf);
}
```

Figure 3.2: Module `greet` (used in all six examples).
Figure 3.3: One-way interfaces (application `Source_sink`); MIL program (left), `print` module (right).
source and sink interface statements (Figure 3.3 (left)). Just as with the client and function statements, these statements declare the interface name and define what type of data will be passed. Module hello is initiating the communication by sending a string on interface “send”, so it declares its interface “send” with the source statement:

source “send” : { string }

A source interface must be bound to a sink interface; the receiving module print declares three of these, one for each of the nodes instantiated with module hello:

sink “msg1” : { string }
sink “msg2” : { string }
sink “msg3” : { string }

Next we explain how a module can query the bus for its node name. Module hello (using program greet.c) declares objname_buf as a character array and calls:

mh_identity (objname_buf, sizeof(objname_buf))

The name the bus returns in objname_buf is the node name, not the module name specified in the MII module description. So this application prints out “hello” “hi” and “greetings”, not “hello” “hello” “hello”. Note that the order in which the messages are received and printed out is statically determined by module print (Figure 3.3 (right)). The nodes hello, hi, and greetings send their messages as soon as the nodes are created, so the messages are sent in a non-deterministic order. Even if the message “hello” is the last to arrive, it will be the first to be read by module print.

### 3.2 MANIPULATING INTERFACE NAMES

The next two applications are almost identical to the previous; the three differ only in the program that implements module print. The two new versions of the print module do not name their interfaces anywhere within the module.

#### 3.2.1 QUERYING THE BUS FOR INTERFACE NAMES

The program in Figure 3.4 shows how application Query_objnames does this by querying the bus for its interface names:
Advanced/query_objnames.c

#include <stdio.h>

char s[256], interface_names[256];
char *iface[20];

main(argc, argv)
int argc;
char **argv;
{
  int i, n;
  char *p;

  mh_init(&argc, &argv, NULL, NULL);

  /* put interface names in array iface */
  mh_query_objnames(interface_names,
                    sizeof(interface_names));
  printf("interfaces are: \%s\n", interface_names);
  p = interface_names;
  n = 0;
  while (*p) {
    iface[n] = p;
    n++;
    while (((p != ',') && (*p)) p++;
    if (*p) *(p++) = NULL;
  }

  for (i=0; i<n; i++) {
    mh_read(iface[i], "S", NULL, NULL, s);
    printf("    \%s, world\n", s);
  }

  mh_shutdown(0, 42, "");
}

Figure 3.4: Querying for interface names (application Query_objnames); MII program (left), print module (right).
Figure 3.5: Application **Readselect**; MIL program (left), **print** module (right).

```c
#include <stdio.h>

char s[256], msgbuf[256], *iface_name;

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

    mh_init(&argc, &argv, NULL, NULL);

    while (1) {
        iface_name = (char *)
            mh_readselect(NULL, NULL, msgbuf, sizeof(msgbuf));
        mh_readback(msgbuf, "S", NULL, s);
        printf("%s: %s, world\n", iface_name, s);
    }
}
```

where **interface_names** is declared as a character array. The bus puts the names of all of module **print**’s interfaces into the buffer **interface_names**. The names are separated by commas in the buffer, with no intervening blanks, so the next section of code uses a **while** loop to place pointers to these names into the array **iface**. Then the program loops through the list of interface names, calling **mh_read** on each one and printing the resulting message. Once again, the **print** module reads the messages in a predetermined order, letting other messages queue up while it waits for the current one.

### 3.2.2 RECEIVING MESSAGES ON ANY INTERFACE

The program in Figure 3.5 shows how application **Readselect** avoids directly naming any interfaces in module **print**. Here we use a **POLYLITH** command that allows us to read the next message to arrive on *any* interface, instead of reading from a particular interface:

```c
iface_name = (char *)mh_readselect (NULL, NULL, msgbuf, sizeof(msgbuf))
```
where `msgbuf` is declared as a character array, and `iface_name`, a character pointer, receives a pointer to the name of the interface where the message arrived. The NULL parameters are placeholders for features which will be available in a future release of Polylith.

The module will block at the `mh_readselect` command until a message arrives on some interface, or will proceed immediately if a message is already queued. After the `mh_readselect` call is complete, `msgbuf` will contain the message. To pull the variables comprising the message from `msgbuf`, we use:

```c
mh_readback (msgbuf, "S", NULL, s)
```

where `s` is declared as a character array, and the message format is “S”. The `mh_readback` is similar to the `mh_read` in that a message format and a list of variables must be supplied. The difference between them is that `mh_read` expects an interface name, while `mh_readback` expects the name of a buffer that was filled by a prior call to `mh_readselect`. You do not need to know the interface name to do a `mh_readback` because `mh_readselect` has already pulled the message off the interface; `mh_readback` is just used to interpret the message.

In this version of module `print` we put the `mh_readselect` and `mh_readback` in an infinite loop. This time, since at each iteration we are reading a message from the next available interface, the messages are printed in an indeterminate order, probably but not necessarily in the same order in which they were sent. Because the loop does not terminate, there is no point in putting an `mh_shutdown` at the end of the `print` module. The application must be terminated by `control-C`.

### 3.3 Attributes

The fourth version of the application (Figure 3.6) is structurally somewhat different from the previous three. The application still has nodes `hello`, `hi`, and `greetings` instantiated with module `hello`, but we have added two new nodes: `number`, which is almost identical to module `hello` except that it sends an integer instead of a string, and `goodbye`, which is instantiated with module `timer`. The `timer` module sleeps for while then sends its node name to module `print`, which shuts down when it receives a message from the timer. Figures 3.7 and 3.8 show the MIL program and source code for application `Attributes`.

#### 3.3.1 Object Attributes

We do not want to hardcode the sleep time in the `timer` module, so we use an object attribute to specify the number of second to sleep. The object attribute is declared and given a value in the algebra statement of the MIL module definition:
service "timer": {
    implementation ...
    algebra: { "SECONDS=3" }
    source ...
  }

This creates an attribute named SECONDS for module timer, and gives the attribute a value of "3". This attribute value is a string containing the character ‘3’, and not the integer 3. The algebra statement accepts any number of attributes; see the summary at the end of this chapter if you want to use more than one object attribute.

Now the program instantiating the module (Figure 3.8) can query the bus at runtime for the value of the SECONDS attribute:

mh_query_objattr ("SECONDS", time_buf, sizeof(time_buf))

where time_buf, which is declared as a character array, receives the attribute value. Since we want to pass an integer to sleep, we use atoi to convert the string in time_buf to an integer.

We have just seen how to explicitly specify object attributes; there are other object attributes that are implicitly specified by your MII program. The NAME attribute is one example, and others are listed in Table B.3. The mh_identity command provides a simple way of getting the value of the NAME attribute; the following two commands are equivalent:
Advanced/attributes.cl

**************
service "hello": {
  implementation: { binary: "./greet.exe" }
  source "send": {string}
}

service "number": {
  implementation: { binary: "./number.exe" }
  source "send": {integer}
}

service "timer": {
  implementation: { binary: "./timer.exe" }
  algebra: {"SECONDS=3"}
  source "send": {string}
}

service "print": {
  implementation:
  { binary: "./pr_attributes.exe" }
  sink "msg1": {string}
  sink "msg2": {string}
  sink "msg3": {string}
  sink "number": {integer}
  sink "shutdown": {string}
}

orchestrate "attributes": {
  tool "hello"
  tool "hi": "hello"
  tool "greetings": "hello"
  tool "number"
  tool "goodbye": "timer"
  tool "print"
  bind "hello send" "print msg1"
  bind "hi send" "print msg2"
  bind "greetings send" "print msg3"
  bind "number send" "print number"
  bind "goodbye send" "print shutdown"
}

**************

Advanced/number.c

**************
#include <stdio.h>

main(argc, argv)
  int argc;
  char **argv;
{
  mh_init(&argc, &argv, NULL, NULL);
  mh_write("send", "I", NULL, NULL, 5280);
}

**************
#include <stdio.h>

Figure 3.7: Application Attributes; MIL program (left), number and print modules (right).
Advanced/timer.c

#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{ char objname_buf[256], time_buf[256];
  int seconds;

  mh_init(&argc, &argv, NULL, NULL);

  mh_identity(objname_buf, sizeof(objname_buf));

  mh_query_objattr("SECONDS", time_buf, sizeof(time_buf));
  seconds = atoi(time_buf);

  sleep(seconds);
  mh_write("send", "S", NULL, NULL, objname_buf);
}

Figure 3.8: Module timer (used in last three examples).

mh_identity (objname_buf, sizeof(objname_buf))
mh_query_objattr ("NAME", objname_buf, sizeof(objname_buf))

3.3.2 INTERFACE ATTRIBUTES

As before, module print is looping and doing an mh_readselect and mh_readback, but before doing the mh_readback, the module queries the bus for the value of an interface attribute (Figure 3.7 (right)). An interface attribute is identical to an object attribute, except that it is specific to an interface. Here we use an implicit interface attribute, PATTERN, that is specified as a result of the sink statement in the MIL module definition. The value of the PATTERN attribute is the message pattern declared in the interface statement (source, sink, client, or function).

mh_query_ifattr (iface_name, "PATTERN", msg_format, sizeof(msg_format))

puts either the string “S” or “i” into msg_format, depending on the interface named in iface_name. Then we use msg_format to decide the datatype of the variable that will receive the message, and proceed with the mh_readback.

Because the timer module eventually sends a message on interface “shutdown”, we can call mh_shutdown at that point, allowing the application to terminate normally instead of with a control-C.
3.4 NON-BLOCKING CHECK FOR MESSAGES

We change the application structure again for these last two applications. Module number is no longer used, but now the print module is sending a message to itself (Figure 3.9). The print module still loops until it receives a message from the timer module, and it sends itself one message per loop. If print tried to read its message with an mh_read, to avoid deadlock it would have to be very careful about not reading a message from itself before the message had been sent. The two versions presented in this section avoid making a blocking read (mh_read or mh_readselect) by first querying the bus to find out if any messages are queued.

3.4.1 QUERYING A PARTICULAR INTERFACE

The print module in the first version (Figure 3.10) gets its interface names using mh_query_obj_names. Note that not all interfaces are incoming any more; interface “self” is an outgoing interface, and it is included in the list of interface names. We can still query this outgoing interface for incoming messages, although it will not have any.

Within the while loop, print queries the bus for messages on each of the interfaces using:

\[
\text{mh_query_ifmsgs (iface[i])}
\]

where i loops over all the interfaces. The mh_query_ifmsgs command returns the number of messages queued, although here we read one at a time even if more are available. This command does not read any messages from the interface, so we follow it with an mh_read.
Figure 3.10: Querying an interface (application Query_ifmsgs); MIL program (left), print module (right).
### 3.4.2 Querying Any Interface

The print module in this last version (Figure 3.11) queries the bus for the number of messages queued on all of the module’s interfaces:

```c
mh_query_objmsgs()
```

The `mh_query_objmsgs` command returns the total number of messages queued on all interfaces. It does not read any of these messages, so we follow it with a `mh_readselect` if one or more messages are available. We cannot use `mh_read`, because we have no way of knowing which interface has a message queued.

Note that the value of `timer`s `SECONDS` attribute is `4` here but it was `10` in the previous example. Because the `mh_readselect` version is a more efficient approach than querying the bus at each interface, we reduced the number of seconds for the shutdown timer.

### 3.5 Polylith Bus Runtime Options

Polylith has options available that allow you to invoke different versions of the Polylith bus:

```c
bus -d -k -v -l bus_input_file
```

These options may be used in any combination. The next three sections describe `-d` (direct connect), `-k` (keep-alive), `-v` (verbose), and `-l` (logfile).

#### 3.5.1 Direct Connect

With the direct connect (`-d`) option, the bus binds interfaces directly to each other. Thus messages are not sent to the bus to be forwarded but are sent directly to another module, making communication faster. To use direct connect on an application, you must pass additional information to the `mh_init` call:

```c
mh_init(&argc, &argv, outfaces, infaces)
```

where `outfaces` and `infaces` are arrays of strings. The arrays contain the names of the outgoing and incoming interfaces respectively, and are terminated by a `NULL` string:

```c
char *outfaces[j + 1] = {"out1", "out2", ..., "outj", NULL};
char *infaces[k + 1] = {"in1", "in2", ..., "ink", NULL};
```
Figure 3.11: Querying for any message (application `Query_objmsgs`); MIL program (left), `print` module (right).
where \( j \) is the number of outgoing interfaces, and \( k \) is the number of incoming interfaces. (A bidirectional interface must be named as both an outgoing interface \textit{and} an incoming interface.)

Because the bus does not keep track of messages passed between modules that are directly connected, the \texttt{mh\_readselect}, \texttt{mh\_readback}, \texttt{mh\_query\_objmsgs}, and \texttt{mh\_query\_ifmsgs} commands are not available with direct connect.

### 3.5.2 KEEP-ALIVE

With the keep-alive (\texttt{-k}) option, the bus keeps all communication channels open between messages. (Normally, the channels are opened when a message is sent, and closed after it has been received.) The keep-alive option allows for faster communication, but can only be used when the total number of bindings in the application is small. The exact limit is determined by the number of UNIX file descriptors available, usually around ten to fifteen.

### 3.5.3 VERBOSE, LOGFILE

With the verbose (\texttt{-v}) option, the bus writes information about each bus transaction to standard output as the application executes. It can be used for debugging an application. The logfile (\texttt{-l}) option captures similar information, but writes this information to a file named \texttt{logfile} in your local directory.

### 3.6 SUMMARY OF ADVANCED FEATURES

This section summarizes the POLYLITHE features that were presented in this chapter. A complete summary of features is presented in the appendices.

#### 3.6.1 MIL STATEMENTS

**MODULE DESCRIPTION** In this chapter, we introduced the object attribute statement as the way to specify attributes and values for a module. The \texttt{obj\_attribute\_statement} belongs in your \texttt{module\_description}:
service "module_name" : {
  implementation_statement
  obj_attribute_statement
  ...
  obj_attribute_statement
  interface_statement
  ...
  interface_statement_k
}

OBJECT ATTRIBUTE STATEMENT The object attribute statement lets you specify attributes and their values in your module description. Then, using the mh_query_objattr command, a node can query the POLYBUMP bus for the value of a particular attribute. The object attribute statement is:

\[
\text{algebra} : \{ \ "obj\_attr\_name_1=\ obj\_attr\_value_1 : \ldots : \ obj\_attr\_name_j=\ obj\_attr\_value_j" \}
\]

When \(obj\_attr\_name_i\) is passed to the mh_query_objattr command, it returns \(obj\_attr\_value_i\) to the caller.

INTERFACE STATEMENT This chapter describes how to specify one-way communication interfaces with the source/sink interface statements. These statements declare the interface name and define what type of data will be passed. The message patterns used to describe each interface have the format:

\[
\text{msg_pattern} \equiv t_1 ; t_2 ; \ldots ; t_n
\]

where \(t_i\) is the pattern type of the \(i^{th}\) variable in the message, and \(n\) is the number of variables passed in each message (\(n\) can be zero). Note that the semicolon is used to concatenate the pattern types into a message pattern. The interface pattern types are shown in Table A.2.

The module that initiates communication declares its interface with the source statement:

\[
\text{source} "interface\_name" : \{ \ \text{msg_pattern}_{\text{out}} \}
\]

where \(interface\_name\) is the name of an outgoing interface, and \(msg_{\text{pattern}_{\text{out}}}\) is an interface message pattern (as described above). A source interface must be bound to a sink interface, which is declared by the receiving module:
sink “interface_name” : { msg_pattern_in }

where interface_name is the name of an incoming interface, and msg_pattern_in is an interface message pattern.

### 3.6.2 POLYLITH BUS CALLS

**RECEIVING MESSAGES ON ANY INTERFACE** A message contains a set of variables that are passed to the bus; the bus uses these variables as addresses where data is to be put. A message format must accompany each message to indicate the type of every variable in the message:

\[
msg_{\text{format_in}} \equiv t_1 t_2 \cdots t_n
\]

where \( t_i \) is the type of the \( i^{th} \) variable in the message, and \( n \) is the number of variables passed in each message (can be zero). The types are concatenated to form a message format. These message format types are shown in Table B.2. Because C has only value parameters, \( msg_{\text{format_in}} \) (the incoming message format) can contain only the pointer types.

To receive a message on any interface, use:

```c
iface_name = (char *)
mh_readselect (NULL, NULL, message_buffer, sizeof(message_buffer))
```

where message_buffer is declared as a character array, and iface_name is declared as a character pointer. The NULL parameters are placeholders for features which will be available in a future release of POLYLITH.

The module will block at the mh_readselect command until a message arrives on any interface, or will proceed immediately if a message is already queued. After the mh_readselect call is complete, message_buffer will contain the message, and iface_name will point to the name of the interface along which the message arrived. To pull the variables comprising the message from message_buffer, use:

```c
mh_readback (message_buffer, “msg_format_in”, NULL, r_1, r_2, ..., r_n)
```

where \( r_i \) is a pointer variable or the address of a variable, each \( t_i \) in \( msg_{\text{format_in}} \) is the pointer type of variable \( r_i \), and \( n \) is the number of variables in the message (can be zero). The NULL parameter is a placeholder for features which will be available in a future release of POLYLITH.
NON-BLOCKING CHECK FOR MESSAGES A module can avoid making a blocking read (\texttt{mh\_read} or \texttt{mh\_readselect}) by first querying the bus to find out if any messages are queued. To find out how many messages are queued on a particular interface, use:

\texttt{mh\_query\_ifmsgs ("interface\_name")}

where \texttt{interface\_name} is the name of an interface declared in the MIL description of the module. The \texttt{mh\_query\_ifmsgs} command returns the number of messages queued. It does not read any messages from the interface, so it is generally followed by a \texttt{mh\_read} when one or more messages are available.

To find out how many messages are queued on all of the module's interfaces, use:

\texttt{mh\_query\_objmsgs ()}

The \texttt{mh\_query\_objmsgs} command returns the total number of messages queued on all interfaces. It does not read any of these messages, so it is generally followed by a \texttt{mh\_readselect} when one or more messages are available.

ATTRIBUTES A module can query the bus for values of its attributes. The standard attributes, such as names or interface message patterns, are implicitly specified in the POLYLIB MIL program; these are listed in Table B.3. You may also query the bus for values of attributes that were explicitly declared in the MIL program.

NAME ATTRIBUTES Your POLYLIB MIL program gives name attributes to the nodes and interfaces in your application. An object can query the bus for its name using:

\texttt{mh\_identity (objname\_buffer, sizeof(objname\_buffer))}

where \texttt{objname\_buffer} is declared as a character array. The name the bus returns to the object is the node name, not the module name specified in the MIL module description. (The module name may not be unique within the application, but the node name is.)

An object can query the bus for its interface names using:

\texttt{mh\_query\_objnames (ifname\_buffer, sizeof(ifname\_buffer))}

where \texttt{ifname\_buffer} is declared as a character array. The bus puts the names of all of the module's interfaces in the buffer, whether the interfaces are strictly incoming, strictly outgoing, or bidirectional. The names are separated by commas in the buffer, with no intervening blanks.
OTHER ATTRIBUTES  An object can query the bus for the value of any of its attributes using:

\[
\text{mh\_query\_obj\_attr ("obj\_attr\_name", attr\_value\_buffer, sizeof(attr\_value\_buffer))}
\]

where \texttt{attr\_value\_buffer} is declared as a character array; the value is always a character string. The attribute specified in \texttt{obj\_attr\_name} is either one that was explicitly declared in your MIL program, or an attribute like NAME or BINARY that your MIL program implicitly specifies. Note that the following two commands are equivalent:

\[
\begin{align*}
\text{mh\_identity (objname\_buffer, sizeof(objname\_buffer))} \\
\text{mh\_query\_obj\_attr ("NAME", objname\_buffer, sizeof(objname\_buffer))}
\end{align*}
\]

The command to query the bus for the value of an interface attribute is identical to the object attribute command, except that you also specify an interface:

\[
\text{mh\_query\_if\_attr ("if\_name", "if\_attr\_name", attr\_value\_buffer, sizeof(attr\_value\_buffer))}
\]

where \texttt{if\_name} is the name of an interface declared in the MIL description of the module. Your MIL program implicitly specifies a PATTERN attribute for each interface; its value is a string containing the interface message pattern from your MIL program.

3.6.3 POLYLITH BUS RUNTIME OPTIONS

POLYLITH has options available that allow you to invoke different versions of the POLYLITH bus:

\[
\text{bus -d -k -v -l bus\_input\_file}
\]

These options may be used in any combination. The following three sections describe -d (direct connect), -k (keep-alive), -v (verbose), and -l (logfile).

DIRECT CONNECT  With the direct connect (-d) option, the bus binds interfaces directly to each other. Since the messages are not sent to the bus to be forwarded but are sent directly to another module, communication is faster. To use direct connect on an application, you must pass additional information to the \texttt{mh\_init} call:

\[
\text{mh\_init (&argc, &argv, outfaces, infaces)}
\]
where `outfaces` and `infaces` are arrays of strings. The arrays contain the names of the outgoing and incoming interfaces respectively, and are terminated by a NULL string:

```c
char *outfaces[j + 1] = { “out1”, “out2”, ..., “outj”, NULL };
char *infaces[k + 1] = { “in1”, “in2”, ..., “ink”, NULL };
```

where `j` is the number of outgoing interfaces, and `k` is the number of incoming interfaces. (A bidirectional interface must be named as both an outgoing interface and an incoming interface.)

Because the bus does not keep track of messages passed between modules, the `mh_readselect` and `mh_query_objmsgs` commands are not available with direct connect.

**KEEP-ALIVE**  With the keep-alive (`-k`) option, the bus keeps all communication channels open between messages. (Normally, the channels are opened when a message is sent, and closed after it has been received.) The keep-alive option allows for faster communication, but can only be used when the total number of bindings in the application is small. The exact limit is determined by the number of UNIX file descriptors available, usually around ten to fifteen.

**VERBOSE, LOGFILE**  With the verbose (`-v`) option, the bus writes information about each bus transaction to standard output as the application executes. It can be used for debugging an application. The logfile (`-l`) option captures similar information, but writes this information to a file named `logfile` in your local directory.
BIBLIOGRAPHY


ACKNOWLEDGEMENT

We appreciate the suggestions, editorial comments and sense of humor from Jack Callahan, Larry Herman, Elizabeth White and Anne Wilson. Their experiments with the early form of this document and distribution helped us immensely.
Appendix A

MIL SUMMARY

The POLYLIB MIL program consists of a description for each module and a description of the application. These can be in separate files, combined in a single file, or a mix of both.

\[
\text{module\_description}_1 \\
\text{module\_description}_2 \\
\vdots \\
\text{module\_description}_m \\
\text{application\_description}
\]

A.1 MODULE DESCRIPTION

A description must appear for each module used in the application. The \textit{module\_name} given to the module is used to identify it in the application description. This name is not referred to in the source program for the module.

\[
\text{service "module\_name" : \{} \\
\quad \text{implementation\_statement} \\
\quad \text{obj\_attribute\_statement}_1 \\
\quad \vdots \\
\quad \text{obj\_attribute\_statement}_a \\
\quad \text{interface\_statement}_1 \\
\quad \vdots \\
\quad \text{interface\_statement}_k \\
\}\n\]
<table>
<thead>
<tr>
<th>attribute name</th>
<th>attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>pathname of the executable program which implements this module</td>
</tr>
<tr>
<td>source</td>
<td>pathname of the source program which implements this module</td>
</tr>
<tr>
<td>machine</td>
<td>name of host where the module will run</td>
</tr>
</tbody>
</table>

Table A.1: POLYLITH MIL Module Implementation Attributes

### A.1.1 IMPLEMENTATION STATEMENT

The implementation statement names the program which implements this module and the host on which the module is to be executed.

```
implementation : { impl_attr_name_1 : "impl_attr_value_1" ... impl_attr_name_j : "impl_attr_value_j" }
```

Table A.1 shows the attributes that can be used in the implementation statement. You must specify either the `binary` or the `source` attribute, but not both. The `machine` attribute is optional; by default the module will run on the same host as the POLYLITH bus.

### A.1.2 OBJECT ATTRIBUTE STATEMENT

The object attribute statement provides a way of specifying attributes and their values in your module description. Then, using the `mh_query_objattr` command, a node can query the POLYLITH bus for the value of a particular attribute. The object attribute statement is:

```
algebra : { "obj_attr_name_1=obj_attr_value_1 ; ... : obj_attr_name_j=obj_attr_value_j" }
```

When `obj_attr_name_i` is passed to the `mh_query_objattr` command, it returns `obj_attr_value_i` to the caller.

### A.1.3 INTERFACE STATEMENT

An interface statement must appear for each interface that the module expects to use; these statements declare the interface name and define what type of data will be passed. The message patterns used to describe each interface have the format:

```
msg_pattern \equiv t_1 ; t_2 ; \ldots ; t_n
```
Table A.2: Polylith MIL Interface Pattern Types

<table>
<thead>
<tr>
<th>Pointer Types (for incoming or outgoing interfaces)</th>
<th>Value Types (for outgoing interfaces only)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pattern type</strong></td>
<td><strong>description</strong></td>
</tr>
<tr>
<td>string</td>
<td>string (pointer to char)</td>
</tr>
<tr>
<td>integer</td>
<td>pointer to integer</td>
</tr>
<tr>
<td>boolean</td>
<td>pointer to boolean (ptr to int)</td>
</tr>
<tr>
<td>float</td>
<td>pointer to float (ptr to double)</td>
</tr>
<tr>
<td>pattern_type(n)</td>
<td>array of size n, type pattern_type</td>
</tr>
</tbody>
</table>

where $t_i$ is the pattern type of the $i^{th}$ variable in the message, and $n$ is the number of variables passed in each message ($n$ can be zero). Note that the semicolon is used to concatenate the pattern types into a message pattern. The interface pattern types are shown in Table A.2.

The **source** and **sink** interface statements are used when you intend to send messages in one direction only. The initiating module declares its interface with the **source** statement:

```
source "interface_name": { msg_pattern_out }
```

where *interface_name* is the name of an outgoing interface, and *msg_pattern_out* is an interface message pattern (as described above). A **source** interface must be bound to a **sink** interface, which is declared by the receiving module:

```
sink "interface_name": { msg_pattern_in }
```

where *interface_name* is the name of an incoming interface, and *msg_pattern_in* is an interface message pattern.

The **client** and **function** interface statements are used when you intend to send messages back and forth. The module that initiates the first message declares its interface with the **client** statement:

```
client "interface_name": { msg_pattern_out } accepts { msg_pattern_in }
```

where *interface_name* names a bidirectional interface that initiates communication with *msg_pattern_out*, the outgoing message pattern, and accepts a message in return with *msg_pattern_in*, the incoming message pattern. A **client** interface must be bound to a **function** interface:

```
function "interface_name": { msg_pattern_in } returns { msg_pattern_out }
```

52
where \textit{interface\_name} is the name of a bidirectional interface that accepts communication using \textit{msg\_pattern\_in}, the incoming message pattern, and returns a message with \textit{msg\_pattern\_out}, the outgoing message pattern.

\section*{A.2 APPLICATION DESCRIPTION}

The application description uses the \textit{tool\_statement} to name the nodes of the application and instantiate them with modules, then uses the \textit{bind\_statement} to bind the interfaces of these nodes.

\begin{verbatim}
orchestrate "application\_name" : {
    tool\_statement\_1
    tool\_statement\_2
    ...
    tool\_statement\_n
    bind\_statement\_1
    bind\_statement\_2
    ...
    bind\_statement\_z
}
\end{verbatim}

\subsection*{A.2.1 TOOL STATEMENT}

The \textit{tool\_statement} defines a node in the application by naming the node and specifying which module instantiates the node:

\begin{verbatim}
tool "node\_name" : "module\_name"
\end{verbatim}

Another version of the \textit{tool\_statement} specifies that the node name is the same as the name of the module that instantiates it:

\begin{verbatim}
tool "module\_name"
\end{verbatim}

One of these \textit{tool\_statements} must be included for each node in the application.
A.2.2 BIND STATEMENT

The purpose of the bind statement is to connect the interfaces of the nodes.

\[ \text{bind } \text{"node\_name}_i \text{ interface\_name}_{i,j} \text{" } \text{"node\_name}_r \text{ interface\_name}_{r,k} \text{"} \]

where \text{node\_name}_i has a \text{j}^{th} interface \text{interface\_name}_{i,j} declared in a \text{source} or \text{client} statement. The message pattern(s) for this interface match those of the \text{k}^{th} interface of \text{node\_name}_r, which is declared in a \text{sink} or \text{function} statement.
Appendix B

BUS CALLS

The mh commands are a collection of library routines which allow a module to send and receive communication over the POLYLITH bus, and to query the bus for information about itself. The commands available from the C language are listed in Table B.1. Suitable alternative are available for other application languages as well.

B.1 GENERAL COMMANDS

The source program for each module calls routines from the POLYLITH bus library to send or receive communication on its interfaces. Before using these interfaces, the program must pass the command line arguments to mh init:

<table>
<thead>
<tr>
<th>command</th>
<th>command level</th>
<th>object</th>
<th>object-to-interface</th>
<th>interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mh_init</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mh_shutdown</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>writing</td>
<td></td>
<td></td>
<td></td>
<td>mh_write</td>
</tr>
<tr>
<td>reading</td>
<td></td>
<td></td>
<td></td>
<td>mh_read</td>
</tr>
<tr>
<td>mh_readselect</td>
<td>reading</td>
<td></td>
<td></td>
<td>mh_read</td>
</tr>
<tr>
<td>mh_readback</td>
<td></td>
<td></td>
<td></td>
<td>mh_query_ifmsgs</td>
</tr>
<tr>
<td>mh_query_objmsgs</td>
<td></td>
<td></td>
<td></td>
<td>mh_query_ifmsgs</td>
</tr>
<tr>
<td>attributes</td>
<td>attributes</td>
<td></td>
<td></td>
<td>mh_query_ifattr</td>
</tr>
<tr>
<td>mh_identity</td>
<td>attributes</td>
<td></td>
<td></td>
<td>mh_query_ifattr</td>
</tr>
<tr>
<td>mh_query_objnames</td>
<td>attributes</td>
<td></td>
<td></td>
<td>mh_query_ifattr</td>
</tr>
<tr>
<td>mh_query_objattr</td>
<td>attributes</td>
<td></td>
<td></td>
<td>mh_query_ifattr</td>
</tr>
</tbody>
</table>

Table B.1: POLYLITH Commands
main(argc, argv)
int argc;
char **argv;
{
  ...
  mh_init (&argc, &argv, outfaces, infaces) ...
}

to declare its interfaces to the bus. Parameters outfaces and infaces should be set to NULL unless
you are using the direct connect (-d) bus option (see Section C.4.1).

To terminate an application or a node, use:

  mh_shutdown( level, exit_code, exit_string )

When level=0, this command notifies the bus that the application is finished. The bus terminates
execution at each node and releases all the application’s communication channels. If the application
is not terminated, all nodes could finish execution, but the bus would keep the application
running and hold its communication channels open.

When level=1, the mh_shutdown command terminates just the node issuing the command, and
not the entire application. When level=2, the command acts like an exit command, terminating
the process at the node without notifying the bus.

For shutdown levels of 0 or 1, the exit_code and exit_string parameters are written in the logfile
when the logging option -l is turned on (see section C.4.3). The exit_code is an integer value,
and the exit_string is a character string.

B.2 MESSAGE PASSING

A message contains a set of variables or expressions that are passed to the bus. The bus either
copies these into its memory (for sending), or uses them as addresses where data is to be put
(for receiving). A message format must accompany each message to indicate the type of every
variable or expression in the message:

  msg_format \equiv t_1 t_2 \cdots t_n

where \( t_i \) is the type of the \( i^{th} \) variable in the message, and \( n \), which can be zero, is the number of
variables passed in each message. The types are concatenated to form a message format. These
message format types are shown in Table B.2. Because C has only value parameters, \( msg_{\text{format}}_{\text{in}} \)
(the incoming message format) can contain only the pointer types, and \( msg_{\text{format}}_{\text{out}} \) (the outgoing message format) can contain either the pointer or the value types.
### B.2.1 SENDING MESSAGES

There is only one way to send a message:

\[ \texttt{mh\_write ("interface\_name", "msg\_format\_out", NULL, NULL, w_1, w_2, \ldots, w_n)} \]

where \( \text{interface\_name} \) is the name of an outgoing interface declared in the MIL description of the module, each \( w_i \) in \( \text{msg\_format\_out} \) is the data type of variable (or expression) \( w_i \), and \( n \), which can be zero, is the number of variables in the message. The NULL parameters are used with the capability-based network bus; while available, they are not described in current version of this document.

### B.2.2 RECEIVING MESSAGES ON NAMED INTERFACE

To receive a message on a particular interface, use:

\[ \texttt{mh\_read ("interface\_name", "msg\_format\_in", NULL, NULL, r_1, r_2, \ldots, r_n)} \]

where \( \text{interface\_name} \) is the name of an interface declared in the MIL description of the module, \( r_i \) is a pointer variable or the address of a variable, each \( t_i \) in \( \text{msg\_format\_in} \) is the pointer type of variable \( r_i \), and \( n \) is the number of variables in the message (can be zero). The module will block at the \texttt{mh\_read} command until a message arrives on the specified interface, or will proceed immediately if a message is already queued on the interface. The NULL parameters are used with the capability-based network bus; while available, they are not described in the current version of this document.

### B.2.3 RECEIVING MESSAGES ON ANY INTERFACE

To receive a message on any interface, use:

\[ \texttt{mh\_read (NULL, NULL, NULL, \ldots)} \]
\texttt{iface\_name = (char *)}
\texttt{mh\_readselect (NULL, NULL, message\_buffer, sizeof(message\_buffer))}

where \textit{message\_buffer} is declared as a character array, and \textit{iface\_name} is declared as a character pointer. The \texttt{NULL} parameters used for are placeholders for features which will be available in a future release of Polylith.

The module will block at the \texttt{mh\_readselect} command until a message arrives on any interface, or will proceed immediately if a message is already queued. After the \texttt{mh\_readselect} call is complete, \textit{message\_buffer} will contain the message, and \textit{iface\_name} will point to the name of the interface where the message arrived. To pull the variables comprising the message from \textit{message\_buffer}, use:

\texttt{mh\_readback (message\_buffer, "msg\_format\_in", NULL, r\_1, r\_2, \ldots, r\_n)}

where \texttt{r\_i} is a pointer variable or the address of a variable, each \texttt{t\_i} in \texttt{msg\_format\_in} is the pointer type of variable \texttt{r\_i}, and \texttt{n} is the number of variables in the message (can be zero). The \texttt{NULL} parameter is a placeholder for features which will be available in a future release of Polylith.

\subsection*{B.2.4 NON-BLOCKING CHECK FOR MESSAGES}

A module can avoid making a blocking read (\texttt{mh\_read} or \texttt{mh\_readselect}) by first querying the bus to find out if any messages are queued. To find out how many messages are queued on a particular interface, use:

\texttt{mh\_query\_ifmsgs ("interface\_name")}

where \textit{interface\_name} is the name of an interface declared in the MIL description of the module. The \texttt{mh\_query\_ifmsgs} command returns the number of messages queued. It does not read any messages from the interface, so it is generally followed by an \texttt{mh\_read} when one or more messages are available.

To find out how many messages are queued on \textit{all} of the module’s interfaces, use:

\texttt{mh\_query\_objmsgs ()}

The \texttt{mh\_query\_objmsgs} command returns the total number of messages queued on all interfaces. It does not read any of these messages, so it is generally followed by an \texttt{mh\_readselect} when one or more messages are available.
### Attributes Implicitly Specified by MIL Program

<table>
<thead>
<tr>
<th>Attribute of object of interface</th>
<th>Specified in</th>
<th>Attribute description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>tool_statement</td>
<td>node name</td>
</tr>
<tr>
<td>SOURCE, BINARY, MACHINE</td>
<td>implementation_statement</td>
<td>module implementation attribute</td>
</tr>
<tr>
<td>PATTERN, RETURN</td>
<td>interface_statement</td>
<td>interface message pattern, return pattern for bidirectional interface</td>
</tr>
</tbody>
</table>

#### Table B.3: Attributes Implicitly Specified by MIL Program

### B.3 Attributes

A module can query the bus for values of its attributes. The standard attributes, such as names or interface message patterns, are implicitly specified in the POLYLIB MIL program; these are listed in Table B.3. You may also query the bus for values of attributes that were explicitly declared in the MIL program.

#### B.3.1 Name Attributes

Your POLYLIB MIL program gives name attributes to the nodes and interfaces in your application. An object can query the bus for its name using:

\[
\text{mh_identity} \ (\text{objname_buffer, sizeof(objname_buffer)})
\]

where `objname_buffer` is declared as a character array. The name the bus returns to the object is the node name, not the module name. (The module name may not be unique within the application, but the node name is.)

An object can query the bus for its interface names using:

\[
\text{mh_query_objnames} \ (\text{ifname_buffer, sizeof(ifname_buffer)})
\]

where `ifname_buffer` is declared as a character array. The bus puts the names of all of the module’s interfaces in the buffer, whether the interfaces are strictly incoming, strictly outgoing, or bidirectional. The names are separated by commas in the buffer, with no intervening blanks.

#### B.3.2 Other Attributes

An object can query the bus for the value of any of its attributes using:
mh_query_objattr ("obj_attr_name", attr_value_buffer, sizeof(attr_value_buffer))

where attr_value_buffer is declared as a character array; the value is always a character string. The attribute specified in obj_attr_name is either one that was explicitly declared in your MIL program, or an attribute like NAME or BINARY that your MIL program implicitly specifies. Note that the following two commands are equivalent:

mh_identity (objname_buffer, sizeof(objname_buffer))

mh_query_objattr ("NAME", objname_buffer, sizeof(objname_buffer))

The command to query the bus for the value of an interface attribute is identical to the object attribute command, except that you also specify an interface:

mh_query_ifattr ("if_name", "if_attr_name", attr_value_buffer, sizeof(attr_value_buffer))

where if_name is the name of an interface declared in the MIL description of the module. Your MIL program implicitly specifies a PATTERN attribute for each interface; its value is a string containing the interface message pattern from your MIL program.
Appendix C

USING POLYLITH TOOLS

C.1 COMPILING MODULES

Each module must be compiled and linked using its native language compiler and including the Polylith library (with the \texttt{-lith} flag). The executable file that is created must be named in the \texttt{binary} attribute of the \texttt{implementation} statement in that module’s MIL description. For example,

\begin{verbatim}
cc main.c -c main.o
cc -o main.exe main.o -lith
\end{verbatim}

compiles \texttt{main.c} into \texttt{main.o}, and links \texttt{main.o} with the routines is uses from the Polylith library, creating the executable file \texttt{main.exe}.

C.2 COMPILING THE MIL DECLARATION

The components of the MIL program are the module descriptions and the application description; they can be compiled in separate files or in one file.

\begin{verbatim}
csc phonebook.cl
\end{verbatim}

compiles the MIL program \texttt{phonebook.cl} into \texttt{phonebook.co}. These compiled components are linked using:
Here the two compiled MIL components `phonebook.co` and `main.co` are linked, creating the output file `phonebook`. This output is a text file that contains all the information the POLYLITH bus needs to run the application. You may want to look at this file to see what your MIL program produced: lines starting with `O` contain object attributes; lines starting with `I` list an object’s interfaces; lines starting with `B` contain binding information; and lines starting with `A` contain interface attributes.

C.3 RUNNING THE APPLICATION

To run an application, we start up a POLYLITH bus, passing it our compiled and linked MIL program:

```
bus bus_input_file
```

An application that does not terminate voluntarily can always be terminated with a `control-C`.

C.4 BUS OPTIONS

POLYLITH has options available that allow you to invoke different versions of the POLYLITH bus:

```
bus -d -k -v -l bus_input_file
```

These options may be used in any combination. The next three sections describe `-d` (direct connect), `-k` (keep-alive), `-v` (verbose), and `-l` (logfile).

C.4.1 DIRECT CONNECT

With the direct connect (`-d`) option, the bus binds interfaces directly to each other. Since the messages are not sent to the bus to be forwarded but are sent directly to another module, communication is faster. To use direct connect on an application, you must pass additional information to the `mh_init` call:

```
mh_init(&argc, &argv, outfaces, infaces)
```
where `outfaces` and `infaces` are arrays of strings. The arrays contain the names of the outgoing and incoming interfaces respectively, and are terminated by a `NULL` string:

```c
char *outfaces[j + 1] = { "out1", "out2", ..., "out_j", NULL };
char *infaces[k + 1] = { "in1", "in2", ..., "in_k", NULL };
```

where \( j \) is the number of outgoing interfaces, and \( k \) is the number of incoming interfaces. (A bidirectional interface must be named as both an outgoing interface and an incoming interface.)

Because the bus does not keep track of messages passed between modules, the `mh_readselect` and `mh_query_objmsgs` commands are not available with direct connect.

### C.4.2 KEEP-ALIVE

With the keep-alive (-k) option, the bus keeps all communication channels open between messages. (Normally, the channels are opened when a message is sent, and closed after it has been received.) The keep-alive option allows for faster communication, but can only be used when the total number of bindings in the application is small. The exact limit is determined by the number of UNIX file descriptors available, usually around ten to fifteen.

### C.4.3 VERBOSITY AND LOGGING

With the verbose (-v) option, the bus writes information about each bus transaction to standard output as the application executes. It can be used for debugging an application. The log file (-l) option captures similar information, but writes this information to a file named `logfile` in your local directory.
Appendix D

SYSTEM NOTES

This chapter of the manual is the most volatile, as it is the repository of system notes ... scraps of information that describe the current state of our distribution system. However, whereas this chapter is also the least organized, we hope its inclusion will also prove to be the most useful to those of you who are known to be building upon POLYLITH as a base.

1. **Distribution:** In case you did not receive this document via a standard Polylith distribution tape, you can find it on Internet by anonymous ftp from 
   \[\text{flubber.cs.umd.edu}\]
   There are README files therein that should guide you to what you need. Plenty of other software is available at the same site — take your fill! Tar images are typically supplied with makefiles that ‘do the right thing.’ (They also typically have `make install` and `make clean` features too.)

   This distribution is suitable for use upon Sun 3 workstations, with SunOS versions 3.4 through 4.0.3 (and probably more); DEC Vaxes with BSD-derivative implementations of Unix; and Decstation (and MIPS) workstations. The system ‘mostly’ works on all Encore multiprocessors, except there is a continuing bug in their Unix implementation having to do with how interrupted system calls are treated (if you pause your application then resume it, then you’re likely to find the bus will complain about system calls returning in indeterminant states). The system works on Sun 4 and other sparcstations as long as you don’t compile your applications with extensive optimizations enabled. If you don’t have our packager to generate exactly the correct stubs, then you must limp along with a hacked treatment of varargs in our `mh` calls; this hack fails in the case that you turn on extensive optimization typical for sparc architectures, since all the assumptions about location of parameters within an activation record (or register window) then break.

   All examples used in this document are packaged as-is with the distribution. Follow along the manual as you try the programs!

2. **Need help?** Send questions, suggestions and editorials to polylith@cs.umd.edu
3. **include/endian.h** If your site is closely tracking BSD source modifications, then you will find some of the network structures and macros have been reorganized. In particular, some compilations will fail for lack of having the correct definitions. This should only affect construction of the *bus* — if it fails for these reasons, then check whether you have a system include file called *endian.h*. If so, then you can just change the bus configuration file called *config.h* — go ahead and define the symbol called *MARYLAND* (you'll find the line already there, commented out ... just uncomment it).

4. **Floating data:** The automata that are responsible for coercion of *floating* representation has been gutted — we could not bear to inflict a slapstick piece of code to the world. This means that for the short run, vaxes can only transact floats with other vaxes, suns with suns, etc. This will change once we complete a *robust* version of our converter. Those of you who read source for recreation will see from where it has been removed. Of course, this is not a trivial component to build, as not everything is representable across all machines — the code must know to step around the vax’s terrible treatment of exponents (generating the ‘right’ exceptions when trying to transmit a value too large); it must know how to address the IEEE floating representation for values like NAN; and it must certainly not crap out at extreme values.

5. **Mixed-language examples:** To date this distribution contains relatively few mixed-language examples. This will change with time as we gradually refine examples to the point where it would not be criminal to inflict them upon the world. We have Pascal, Ada, Franz Lisp, Common Lisp and many other examples, each in various degrees of refinement. If you have very pressing needs for a particular language, then contact us directly to learn what to do.

6. **TCP_NODELAY:** We have discovered some dialects of BSD Unix (such as earlier Sequent releases) do not support all of the network socket options we originally assumed. One of these is the TCP_NODELAY option. Right now this is compiled in to our bus code — you’ll see the bus complain about this on each operation when messages are sent. It is only an annoyance (and performance loss), not a fault. Edit the messages out and you’re on your way. The next release will have these conditionally compiled, and you can fix it with just a fix to the config file.

7. **Volatility of Polylith syntax:** The current syntax represents a six year old engineering decision, balancing the expressiveness of interconnection structures against the need to get rapid experience with bus organization. With the advent of CPL/CPS funding, we are finally improving the language. When this occurs we will provide an upgrade path for most applications written in the old (current) MIL syntax. We *know* the current notation is awkward, especially for associating object attributes with particular instances of modules.

8. **Trivia:** Where did the names of our tools come from? Originally we followed the time-honored tradition of making up brand new names to describe otherwise normal CS objects. One of these objects is a program graph, that we called a “cluster”. The names for our MIL-processing tools were therefore “cluster specification compiler” (or *esc*) and “cluster specification linker” (or *ctl*). The tool names have stayed even though we know refer to the
MIL structures as just MIL structures. Similarly, our first implementation of the TCP/IP-based bus (earlier called “toolbus”) was referred to as a “message handler” hence all the mh prefixes and suffixes.

9. **Bus configuration options:** For simplicity of design in this experimental platform we have chosen to compile in some statically-fixed table sizes. These include such things as the maximum size of any given message (measured in ‘flat’ number of bytes), the maximum number of messages that can be queued for other tools within the bus, and so forth. You can examine and control these from within the bus config.h file. Probably our release has some of these turned down fairly small for performance of the demo problems. If you find yourself limited, then you need only change the declarations and recompile. If you do, then be sure to rebuild the Polylith library and relink your binaries.

10. **Remote process startup:** It is difficult to give one release of software that can demonstrate how remote startup of tasks could be done on all sites — everyone has different protection domains. The most efficient way is to add your own reexec-like capability to inetd and distribute some bus responsibilities across all named hosts. However, we don’t think many site managers viewing our distribution will look upon such changes kindly! Therefore, for this release we have contrived a ‘more portable’ way of starting up remote tasks, which uses the fairly-robust BSD tool rsh. But while common to most sites, rsh is also fairly dumb about the finer-grained needs of clients like Polylith: in some cases you will need to worry about ensuring that remotely-invoked tasks are correctly terminated (since the bus cannot always find the right remote pid’s through rsh); remote printf’s will not always get flushed to your local stdout as your intuition might like; and remote reads are definitely not sequenced correctly with the read-ahead of your local tty. We anticipate installing a bus design change that will use rsh to start up a remote copy of the bus to spawn all tasks just for that site; this will allow both IO and process cleanup to be handled much more neatly. Related to the startup problem is the task of ensuring you have the right binaries on the right host to be started up. Again, there is great variety in how this can be accomplished (you might have NFS, you might not ... you might have compilers that know how to generate code for your target machine, but you might need to remotely execute a make instead ... and so on.) All the overhead needed to ensure binaries are where you want them points to the need for a CCM system that is knowledgeable about the diversity — exactly as our Honeywell colleagues on this effort are working on.

11. **Writing coercion routines:** When trying to interface a new language to the Polylith bus, you need to show how control structures from your language correspond to the abstract Polylith bus calls. (Or rather, correspond to this particular bus’s functionality ... after all, the general Polylith result is that you can define an abstract interconnection media once, then separately define how particular application domains map into the abstraction. The network bus defined here is an implementation of only one of many possible interconnection abstractions.) An important part of this task is showing how your data correspond to Polylith-support primitive data types. This correspondence for C is implemented in a file called fac.c, whose compiled form is stored in the library libith.a. When you need to create these maps, you might consider following our heuristic — first figure out how to map
your new language to a C-level at all, then figure out how to adapt your control structure to suit the interfaces in our existing C fa\.c library. This lets you avoid having to wade through the obligations of matching the bus protocol directly! We have planned a toolkit to assist in this activity should it be needed, but until it is completed the bus protocol for interoperation is cryptic at best.

12. **A classic fault for new users:** Once creation of concurrent processes is made trivial, a standard surprise encountered by our users is when they build a simple reader and writer toy (one process simply spins sending out a message, the other process spins in a loop reading those messages). Simple? Seems so until you run it and watch the communication media — including the bus — complain. Users learn about such toys in OS classes that cover timesharing in a different chapter of the course. What happens on Unix machines with a bus that supports buffering of messages is that the writer will get a timeslice and pump out messages unchecked. Thousands of messages later — perhaps hundreds of thousands, depending on the host — the reader might finally get its slice. Meanwhile, the communication media is stuck trying to buffer a deluge of information. With our automatic packaging tool, you could have an easy option of declaring certain interfaces as being synchronous, and all stubs between components would be created for you appropriately, eliminating this problem. Until we distribute this tool, however, the user must know to build in ‘acks’ manually.

13. **Another classic fault:** Often users will build demo applications that are heavy on communication and light on processing demands. Depending on your usage and host architecture, you may occasionally see messages (displayed by individual processes) that notify you of `Client retry...`). Narrowly, this means that one of your underlying Unix hosts may have run out of free IP ports (or that your kernel is so slow in processing TCP requests that one of the requests for connections within the Polylith protocol failed to succeed within a reasonable amount of time). By default, this bus implementation opens and closes each socket as it is needed, in order to minimize the number of open file descriptors for each process. Remember that Unix imposes a small upper bound on fd’s, so the size and complexity of applications would be limited if open connections had to be maintained. The tradeoff is that better than 95% of your network communication costs will be spent in open, close and connect. If you know that the maximum number of interfaces on each process (including the bus) is less than the maximum number of fd’s available to each process, then you can warrant that to the bus when you invoke it (the -k option ... “keep alive”), and your performance will improve significantly. In general, Polylith beats on Unix in many ways it was never expected to be used, and frequency with which ports are acquired and then discarded is one of these ways.

14. **Stuff we should have written in this manual but didn’t:** Based upon internal reviews of this report, with comments on the drafts by several CPL sites, we are aware of several oversights:

- In the current Polylith syntax, comments are expressed using the pound sign ‘#’. This can occur anywhere, and all text from that point to the end of the line is consid-
ered comment text. CSC is rumored to behave unsociably if given C-style comment delimiters.

- We have implemented many busses for evaluation and testing. Recently, an bus based upon *capabilities* — thought of as pointers to objects or specific interfaces to objects — was completed and found to be efficient. This bus is a superset of the original bus intended for this manual, and hence is what you find in the current distribution. We have updated the syntax of all examples and all text in this manual to match the accessors to the new bus, but we have not yet written a chapter on how to utilize the added functionality. You will find some of this in the examples, but we recognize the need for another chapter or three in the manual.

- The current class of network busses have a poor protocol for ‘direct connection’ — an option where, for purposes of increased performance, the bus invokes all application processes, introduces them to one another, and then allows all processes to communicate directly with one another. At this time, processes that intend to participate in such an application must have additional data structures provided to the bus initialization call. We know this is unnecessary, and will be improving it.

- In response to popular demand — yes, we plan a data dictionary of all bus structures, plus a manual for how to write new presentations of an abstract bus to particular language implementations.

As the saying goes, “Fixed in version two ...”