A PACKAGING SYSTEM FOR HETEROGENEOUS EXECUTION ENVIRONMENTS

John R. Callahan James M. Purtilo
Computer Science Department
University of Maryland, College Park, Maryland 20742

ABSTRACT

In many execution environments software components cannot interoperate easily because of differences in their interfaces and implementations. Additional software is often required to integrate such components and implement the interfacing decisions between them. For example, a procedure call across architectures may require extensive software to relocate data and coerce parameters. Even when powerful integration facilities are available, application programs need some additional software — often called 'stubs' — so they can access the available communication media. Interface software can be more expensive to program that other software, since its creation requires knowledge of the machine architectures and communication mechanisms. Moreover, it must be rewritten whenever components are reused in different configurations.

This paper describes a way to automatically generate custom interface software for heterogeneous configurations. Whereas previous research focused on 'stub generation' alone, our approach generates stubs as well as the configuration methods needed to integrate an application. Using this approach, developers may build support tools that hide the details of how software configurations are 'packaged' into executables. This approach is implemented within the Unix environment in a system called Polygen, which we have used for evaluation and demonstration.

Keywords: Stub generation, heterogeneous systems, configuration management, module interconnection language, interface extraction.

This research is sponsored by Office of Naval Research under contract N0014-90-J4091. The results are currently being applied to prototyping problems in a project is supported by the DARPA/ISTO Common Prototyping Language initiative, in collaboration with Honeywell Systems and Research Center.

1 INTRODUCTION

In an era of escalating software costs, reuse of software components is an economic necessity. Equally acute is the need to integrate components in the presence of heterogeneity, whether in source languages, architectures, or communication media. Unfortunately, additional software must be developed to implement interfacing decisions for each heterogeneous configuration. Application programs must be adapted to use the desired architecture and communication media, or they must be extended to do so. Interface software can be expensive to create, and must be rewritten whenever components are reused in different configurations.

One way to increase the potential for software reuse is to limit the growth of dependencies between components. For example, module interconnection languages (MILs) have been effective in managing *structural* dependencies, i.e., those concerning visibility or compatibility of variables and interface names [5]. However, the availability of heterogeneous systems increases the likelihood for *geometric* coupling, which are dependencies due to the relation between components and where they execute in the underlying architecture. To minimize these dependencies, programmers typically organize their programs so that calls to an underlying communication system are as isolated as possible. They hope that stubs will localize the impact of subsequent changes in the communication system, therefore reduce the cost of reusing the component. This approach is generally successful, but there is still the manual tasks of identifying the remote interfaces, creating the stub code, and determining how the stub should be integrated with the application.

When programmers must adapt their components for each new application, some economic benefits of reuse are lost. In order to regain them, we must turn to automatic techniques, from which several questions arise. How do we generate the interface software needed for one application component to interoperate with another? How are the components and interface software packaged together into executable objects? How do we analyze source programs to discover potential dependencies in the first place? These are problems we focus on in this paper.

We will describe a method for automating the generation of custom interface software for heterogeneous configurations. Whereas previous research has focused on 'stub generation' alone, our approach generates stubs as well as the configuration methods needed to integrate an application. Using this approach, developers may build support tools that hide the details of how software configurations are 'packaged' into executables.

This method is implemented in a system called Polygen within the Unix environment. Polygen integrates heterogeneous components by generating the interface software needed to integrate and transform (for instance, compile and link) a configuration into a set of executable objects. Programmers provide only the source code for application components plus an abstract characterization of their decisions concerning the desired geometry of a configuration. Polygen then acts as a 'linker' to analyze the information and generate the necessary interface software. The process is guided by a set of abstract composition rules that characterize the integration capabilities of an environment. In this manner, components can be composed and reused in different applications without being modified explicitly by the software developer.

2 MOTIVATION

Software is partitioned for many reasons into *modules* — identifiable and homogeneous (but possibly divisible) units of computation. Consider a simple application implemented in Figure 1. The two modules are implemented in different programming languages (C and Common Lisp) and may be configured to execute either on a single processor, on two processors in a shared-memory multiprocessor, or on two different machines in a distributed system. In each case, they must be modified to use a common interface mechanism. As comparison, the integration task is trivial if both modules were implemented in the same programming language and configured to run on a single processor — both modules would be integrated into a single executable object without the use of interface software except as implemented by the native linker/loader (e.g., using stack and jump instructions). In many cases, however, the integration task is not as trivial.

The way we adapt these modules for integration in more general cases depends in part on their source languages, their execution location in the available hardware configuration, and the interprocess communication (IPC) facilities available in an environment. Figure 2 gives an example of module adaptations for one execution environment. The extra code in both cases is necessary because the modules reside in separate executable objects that are integrated by communication mechanisms. A connection between them is established at run time by the environment through `RENDEZVOUS` functions. The call to a `BYTESWAP` function may be necessary if the processes runs on two hosts with differing representations of an `integer`. For the lisp function, a `dispatcher` routine becomes the new 'main' program — the typical 'read, eval and print' operation of lisp must be replaced in order for the process to act as as server in support of other modules. In general, we might place the extra code in a separate module to gain procedural transparency. Such a module, called a stub, would be linked in or loaded separately.

Source language, execution location, and IPC properties as shown in this example are some of the implementation differences that increase coupling and decrease reuse by requiring programmers to adapt modules through the use of interface software. We would like to reuse modules in as many different configurations and environments as possible *without manually creating or modifying source code.* To achieve this goal, we need to adapt modules automatically by generating interface software from abstract specifications of applications. Such a capability would reduce coupling and increase the possibilities for reuse by isolating architectural and communication dependencies.

Programmers also need assistance with more than just the generation of stubs themselves. Each desired configuration requires different communication mechanisms and integration strategies. Once the stubs have been generated and the commands needed to integrate the new files have been enumerated, tools such as `makefiles` [7] can help programmers obtain executables reliably — the problem is identifying the program units and generating the appropriate commands (e.g., generating the makefile for mixed language programs) in the first place. This can be a tedious task that no programmer is interested in performing manually. Existing stub generation systems often replace the manual adaptation of source programs with the manual task of establishing configuration control over the application.

```c
#include ``client.h''

main()
{
   char key[256];
   int retval;

   printf("Name? ");
   while(gets(key) != NULL) {
      if((retval = lookup(key)) != 0)
         printf("%s at ext. %d", key, retval);
      else
         printf("%s not found.",key);

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

```lisp
(setq TABLE '(
   (Jim 2706)
   (Dave 1234)
   (John 5678)
))

(defun lookup (name)
   (table-lookup name TABLE)
)

(defun table-lookup (name list)
   (cond
      ((null list) 0)
      ((equal name (caar list)) (cadar list))
      (t (table-lookup name (cdr list)))
   )
)
```

Figure 1: Module 'main' (left) and function 'lookup' (right) for a simple application.

The client and server modules (above) can be integrated in our execution environment through the use of interprocess communication primitives. Each source module, however, must be modified (below) to use these facilities. Integration methods are different for each environment, but most will require such adaptations depending on the capabilities of local compilers, linkers, and interpreters.

```c
#include ``client.h''

main()
{
   char key[256], buffer[1024];
   int retval;
   int fd, ip;
   if ((fd = RENDEZVOUS(LOOKUP)) < 0)
         exit(1);
   printf("Name? ");
   while(gets(key) != NULL) {
      if (SEND(fd, key) < 0) exit(2);
      if (RECEIVE(fd, buffer) < 0) exit(3);
      if(buffer[0] != (char)0) {
         ip = &(buffer[strlen(buffer)+2]);
         retval = BYTESWAP(*ip);
         printf("%s at ext. %d", key,retval);
      } else {
         printf("%s not found.",key);
      }
      printf("Name? ");
   }
}
```

```lisp
(setq TABLE '(
   (Jim 2706)
   (Dave 1234)
   (John 5678)
))

(defun lookup (name)
   (table-lookup name TABLE)
)

(defun table-lookup (name list)
   (cond
      ((null list) 0)
      ((equal name (caar list)) (cadar list))
      (t (table-lookup name (cdr list)))
   )
)

(defun dispatcher (h)
   (do* ((desc (RECEIVE h) (RECEIVE h)))
        (SEND (sender desc) (invoke desc))
   )
)
```

Figure 2: How the two modules must be modified for integration.

Remote procedure call specification compilers have existed for many years, but few have been coupled with configuration management tools. The Matchmaker [9], Courier [21], SunRPC [19], and XDR [18] RPC compilers, for example, are stub generators. Such compilers must be ported manually in some cases to handle environment-specific details. The HRPC [4] and HORUS RPC compilers [8] are notable exceptions. The HORUS stub generator is parameterized with system and language schema files, while the HRPC project extends this parameterization to include RPC protocols. The Interface Description Language (IDL) [15] project also implements a stub generator. In all cases, integration of stubs, source components, and existing servers is left to the designer.

As an alternative to static generation of stubs, some projects have designed efficient remote evaluation mechanisms for heterogeneous applications. Distributed applications gain substantial performance improvements through the use of customized interface mechanisms like RPC or REV stubs [3, 14, 17, 6]. Stubs in these projects are often handwritten or rewritten from those generated automatically because their performance is critical in many systems and their design if often dependent upon the context of use in a configuration.

In comparison, Polygen accommodates many of these approaches by further parameterizing the stub generation process, as will be shown. Stubs between components can be customized for software configurations as well as environments. If modules are implemented in the same language and execute on the same host, stubs may not be necessary. Efficient interface mechanisms can be created without sacrificing interconnection abstractions. This is the advantage of coordinating stub production with configuration management tools.

Polygen also relies upon technology from the module interconnection language community. In the past, MIL projects have focused primarily upon issues of interface and module compatibility. More recently, the Polylith system showed how MILs could be employed to control communication issues in a distributed network [13]. Polylith introduced a software bus organization to encapsulate interfacing decisions, and, in this way, software components that do not interface directly can interoperate efficiently. This is the particular communication system we have chosen to generate stubs for within Polygen, since Polylith simplifies many of the data coercion and relocation requirements.

The Inscape project [11] is an alternate MIL approach, which primarily focuses on the semantics of module composition processes. Also, the Conic [10] and Durra [2] projects have recently addressed the same problems as Polylith with a similar "toolkit" approach, but without the aid of composition abstractions like the software bus. A number of rule-based software composition models have been constructed for specific programming languages and environments [1]. They establish rules for composing objects and producing the infrastructure needed to construct applications in particular environments. The XCON-in-RIME project [16], for example, also addresses software reuse problems by describing components and composition methods using assertions and rules. Our method, however, unites this approach with methods for handling heterogeneity and the application of local tools — compilers, linkers, stub generators, and configuration management programs.

3 THE PACKAGING PROCESS

We have created a packaging system to meet the integration needs motivated in the last section. Our system, called Polygen, allows designers to reuse modules in many different environments by separating the logical design of an application from the geometry implementing that design. The system is a collection of tools used to build executable objects from source modules. The tools transform MIL specifications of modules into the interfacing mechanisms and integration methods needed to interconnect modules in a particular environment. The interfacing mechanisms (sometimes called 'wrappers', usually just thought of as stubs) come in many forms — they may specify macro substitutions, perform source transformations, or implement stub procedures. In general, interfacing mechanisms must consist of a set of modules along with all commands necessary to prepare them for interoperation with other resources in a configuration.

3.1 DEFINITIONS

A *module specification* is an abstract description of a software component. Its most important role is to describe the interfaces defined by the module, as well as interfaces to resources that the module uses. A module specification also describes the properties of a component. In this project, the Polylith MIL is used to support integration activities, so properties are organized as name-value pairs. A name may be a keyword or a user-defined identifier. A value is either a primitive type (integer, string, boolean, float), a constructed type (structure, array), a predicate expression, a sequence of values, or a set of values. The interfaces of a module are defined similarly with the interface name and its properties separated by ":" characters. Module specifications that simply describe the properties and interfaces of a component are called *primitive modules*. Figure 3 contains the module specifications for the components shown in Figure 1. In this example, the `pattern` property is used to describe the 'interface pattern', that is, the order and type of parameters on the interface.

A module specification can also be used to compose other modules into an application. A *composite specification* describes a collection of modules and the bindings between their interfaces. (An example composite specification that uses the components of Figure 3 is shown in Figure 5, as will be described.) In this sense, an application can be represented by a directed graph. Each node of the graph corresponds to an instance of a module. Each edge of the graph corresponds to a connected pair of interfaces — a use to a definition.

Module specifications may have many possible implementations, written in a variety of source languages and associated with the specification as another form of module property. Once the user develops a composite specification, the source programs must be united with appropriate program stubs, then transformed into executables (which may span several files, due to heterogeneity in the system.) This collection of source programs, generated stubs and commands to build executables is called a *package*. *Packaging* is the activity of analyzing source program interfaces, determining compatibility of source with available communication media, generating stubs and creating all the necessary configuration commands.

```
module client {
   language = 'C'
   source   = '/u/callahan/client.c'
   extract  = '/u/callahan/client.x'
   }
   use interface printf
       :  pattern = { str }
       :  pattern = { str str }
       :  pattern = { str str int }
       :  accepts = { int }
   use interface gets
       :  pattern = { str }
       :  accepts = { int }
   use interface lookup
       :  pattern = { str }
       :  accepts = { int }
}
```

```
module server {
   language = 'KCL'
   source   = '/u/callahan/server.lsp'
   extract  = '/u/callahan/server.x'
   define interface lookup
          :  pattern = { str }
          :  returns = { int }
   define interface table-lookup
          :  pattern = { str list }
          :  returns = { int }
}
```

Figure 3: Module specification for client component (left) and server component (right).

3.2 POLYGEN

The distinct phases of packaging are shown in Figure 4. Polygen's utility in this process is based upon having primitive module specifications for the available source programs. (In the case that no specifications are available for the source, then the system provides users with a tool to extract the interface descriptions in the first place; this is done by techniques discussed later in Section 3.2.4.)

3.2.1 COMPOSITION. The first phase is a design activity is *composition*. Developers create a composite specification of their application in terms of primitive module specifications. This expression of the application's modular structure should represent a design for meeting functional requirements. Ideally, it should not contain extensive information concerning geometry (that is, which architectures each module should execute on and what communication mechanisms would bind modules together) since this information should be added separately. Of course, having the ability to separately annotate a logically-correct design with geometric information is where the leverage of our approach is found, since programmers are then free to vary those annotations quickly in order to experiment with different configurations rapidly. Geometric properties are attached to both modules and bindings, and their values can either guide the packager to build executables for some desired architecture, or they can represent constraints concerning what configurations are *not* feasible. Polygen provides an editing tool for programmers to add annotations without changing the design specifications.

The composite specification in Figure 5 describes an application composed of two unique modules and a description of the bindings between their interfaces. In this case, the interfaces are bound implicitly by name. A sample annotated design is shown in Figure 6, where desired `LOCATION`s for execution of each module are declared, and a the method for all inter-module communication is constrained to be TCP/IP via a Polylith network bus.

3.2.2 GENERATION.

This is the key phase in packaging. A composite specification may be realized in many ways, depending upon both the application and available execution environments. To implement an application in a given environment, Polygen must analyze the constraints affecting compatibility, select interconnection options between interfaces, generate all necessary stubs and enumerate the configuration commands needed to integrate all components. This resulting package is specific to the application and a target execution environment.

Polygen creates a package by partitioning configurations into sets of components that are compatible for integration into the same executable image according to the constraints of the given environment. (We often refer to these sets as *partitions*.) Once partitions are determined, the commands for creating executables from for are generated. For example, multiple source files written in the same programming language (call it X) may be compiled using the X compiler and linked into a single executable using the X link editor. Such a compatibility is encoded as a composition rule and used by Polygen to partition configurations. If a partitioning is possible for a configuration, then Polygen generates the package needed to create the application's executables. Otherwise, the target environment does not support the desired integration, and Polygen reports the error. A detailed description of the partitioning algorithm is given in Section 4.

In order to reason about modules and their compatibility, Polygen requires a characterization of the interconnection capabilities of target execution environments, plus an abstract description of the compatibility of various programming languages within those environments. (This information is given in terms of rules created by a site administrator, as discussed in Section 4.) Polygen applies these rules to produce the interface software needed to implement a composition in a particular environment. The package includes source programs and stubs, plus all the configuration commands needed to compile, link, invoke and interconnect the programs at execution time. The configuration commands are given in terms of a Unix `makefile`. For example, the source programs in Figure 2 would be part of the package for our demonstration problem, as would be the `makefile` necessary to translate and integrate the modified source programs.

3.2.3 CONSTRUCTION.

Finally, the construction phase consists of applying the configuration commands to actually obtain all executables. In Polygen, this means the makefile is executed. Most of the construction tools are provided by the host execution environments in the form of compilers, linkers, loaders, and other configuration management tools.

A number of executable objects may be produced from a generated implementation. Executable objects come in many forms: binary images, scripts, and other types of interpreted programs. They may execute as processes on separate hosts and interoperate via a run-time instance of the software bus — the stubs and the interprocess communication facilities available in an execution environment.

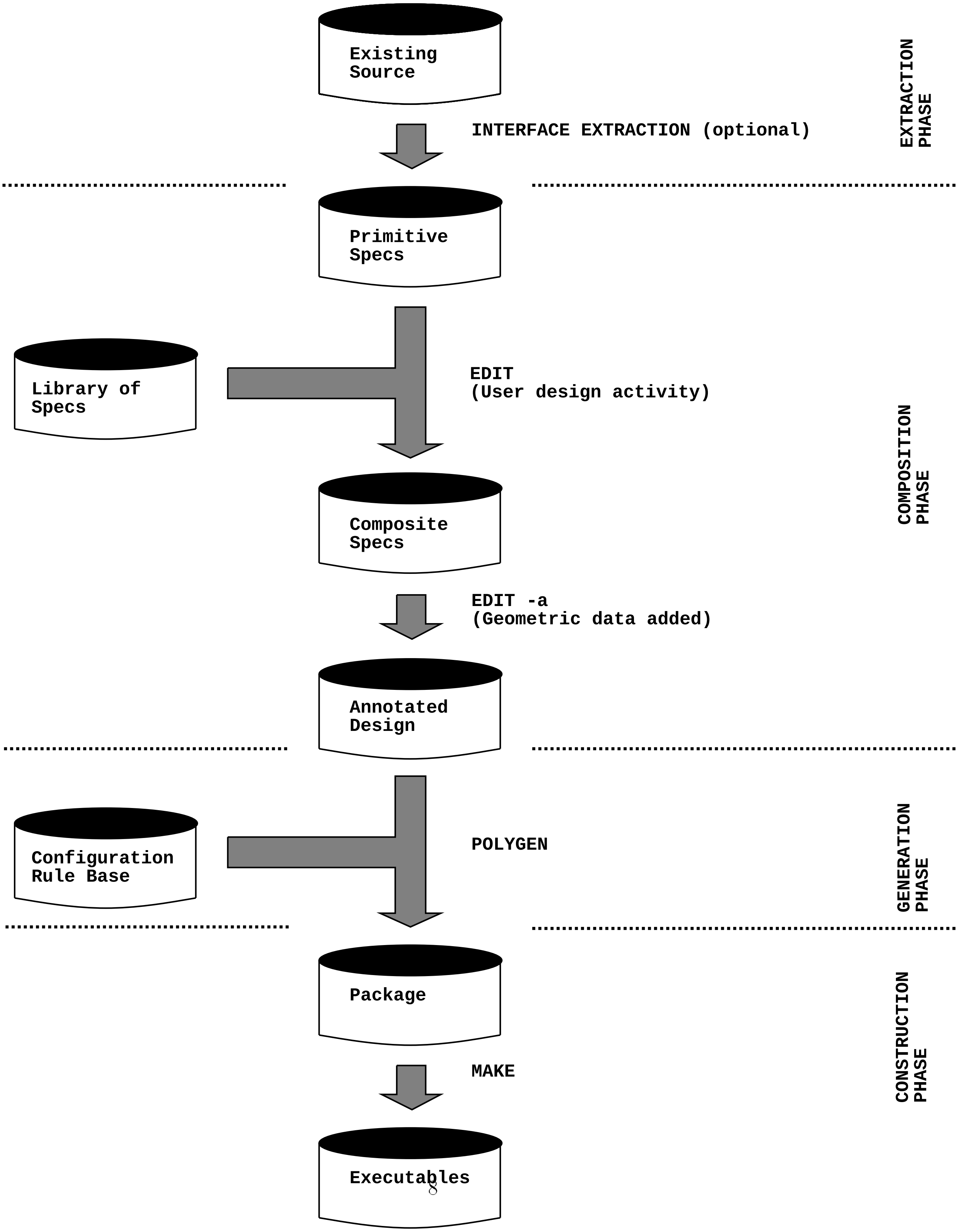

Figure 4: The Polygen packaging process.

```
module test {
   bus :  'polylith_tcpip' ::
   tool client ::
   tool server ::
   bindall
}
```

Figure 5: A design for the phone application.

```
module test {
   tool client :  LOCATION = ``flubber.cs.umd.edu'' ::
   tool server :  LOCATION = ``konky.cs.umd.edu'' ::
   bindall
}
```

Figure 6: An annotated design for the phone application.

3.2.4 EXTRACTION. Extraction tools are used when an existing module has no MIL specification in the first place, or when we wish to verify that an implementation is consistent with its specification. An extraction tool transforms source code into abstract structural descriptions, a collection of assertions about the properties and interfaces of a module. This signature can be created automatically by compilers, parsing tools [12], or manually in cases where extraction tools are unavailable.

In Polygen, the *scan* tool is used to extract such information from source modules. The results contain declarations about the properties and interfaces of a module, and specifications based on them are useful when integrating old implementations into new environments or managing the consistency of implementations and their specifications during development. Thus, we can create MIL representatives for existing libraries and gain the leverage of reusing them in as many configurations as permitted by their implementations. While extraction is a straightforward task, it is a necessary phase of our system and is one of the ways that our research is distinct from other efforts that only provide stub generation from user-defined interface descriptions. Currently, extraction tools are provided for C, Pascal, Ada and Lisp.

3.3 AN EXAMPLE

An implementation is characterized by a set of modules, their stubs, and a configuration program. In our execution environment (a UNIX environment on a local area network), these correspond to the source files, stubs, and commands needed to build executable objects. The commands are part of a configuration program — a UNIX *makefile* [7]. It is produced by the package tool along with the interface descriptions.

We wish to create an executable application given the code shown in Figure 1. The script of the entire process, which includes both user commands and the execution of the configuration

	Initially the user only has source code. So ...
`% scan -o client.cl client.c`	*create a module specification for the* `client` *component*
`% scan -o server.cl server.lsp`	*create a module specification for the* `server` *component*
`% edit test.cl client.cl server.cl`	*Next the user creates a design from his specs (see Figure 5)*
`% polygen -m test.cl`	*Finally, the user has polygen create the package. The -m option asks that the executables be constructed too. Hence, the following output is from commands called from the makefile created automatically by polygen.*
`csc test.cl`	*compiles the application specification into test.co*
`csc client.cl`	*compiles the* `client` *specification into client.co*
`csc server.cl`	*compiles the* `server` *specification into server.co*
`csl test.co client.co server.co -o test`	*creates a root executable that executes a1 and a2*
`wrapgen a1.w`	*creates the client.h and a1.c wrappers*
`cc -c client.c`	*compiles the* `client` *component into client.o*
`cc -c a1.c`	*compiles the a1.c wrapper into a1.o*
`cc -o a1 client.o a1.o -lith`	*creates the first executable object (a binary image)*
`wrapgen a2.w`	*creates the a2.lsp wrapper*
`echo ''#!/bin/csh -f'' > a2`	*creates the second executable object (a shell script)*
`echo ''kcl server.lsp a2.lsp'' >> a2.out`	
`chmod +x a2`	
`% test`	*The user may run the application.*

Figure 7: Script for the design (user commands prefixed by a "%" prompt).

program, is shown in Figure 7. Using the extraction tools, we first create the specifications shown in Figure 3. Next, we construct the system specification shown in Figure 5. We then invoke the package tool to create an implementation — a UNIX `makefile` and the interface description files `a1.w` and `a2.w`. The `makefile` contains the configuration program needed to create the necessary stubs and integrate them with modules into executable objects. Finally, the `makefile` is itself executed, which (according to rules specific to our environment) creates two separate executable objects because the modules cannot be linked together by a conventional linker — i.e., they are incompatible in this execution environment. The generated stubs (`client.h`, `a1.c`, and `a2.lsp`) are shown in Figure 8.

This example involves a difference between the source languages of the two modules. The generated implementation is designed to integrate the two modules despite this difference. The packaging system determines whether or not the two modules can be loaded into a single executable object or the methods by which they can be linked together. In this way, a developer can ignore the details of a composition and concentrate on the description of the interconnections and geometry of an application. The packager system uses this description as a set of constraints to produce an appropriate implementation.

3.4 INTERCONNECTION SUBSYSTEM

Polygen does not replace existing forms of communication and interconnection systems, but rather assists users in utilizing those resources. Polygen currently relies upon the Polylith software in-

client.h

```
========================
#define main client_main
extern int lookup();
```

a1.c

```
========================
#include <polylith.h>
main(argc,argv)
{
   int r = OK;
   mh_init(&argc,&argv,NULL,NULL);
   r = client_main(argc,argv);
   mh_shutdown(ALL,r,NULL);
}

lookup(arg1)
char *arg1;
{
   int r;
   mh_write("lookup", "S", NULL, NULL, arg1);
   mh_read("lookup", "i", NULL, NULL, &r);
   return r;
}
```

a2.lsp

```
========================
(defun mh-dispatcher ()
   (do* (
          (message (mh-readselect) (mh-readselect))
          (interface (car message) (car message))
        )
        (nil 'neverreturned)
        (cond
              ((equal interface ``lookup'')
                (mh-write ``lookup''
                      (lookup (car (cadr message)))
                )
              )
              (t (mh-error message))
        )
   )
)
(mh-initialize)
(mh-dispatcher)
```

Figure 8: Wrappers for the client (left) and server (right) instances.

terconnection system for its communication and mixed-language programming requirements [13]. However, our inference capability is not limited to only Polylith. Polygen can generate packages for other execution environments if the compatibility rules and methods for them are expressed to our inference engine. Polylith and Polygen have an important difference in responsibilities: the former provides interconnection services to programmers, and the latter helps them reason about how to access those services.

Our experience is that any pair of modules can be declared compatible only with respect to a cited execution environment and communication system. Two modules might be 'load-compatible' in Polylith (that is, a Polylith link editor can build the two modules' object code into the same process image), but they might not be able to interoperate if an alternate communication system is used. As Polygen is enriched with information on how modules can be interconnected using new communication resources, users will gain flexibility in how their applications can be configured.

Key properties that affect compatibility of program units are the type and the representation of data in an interface. Polygen does not attempt to infer coercion mechanisms for heterogeneous data directly, but rather inherits transformation capabilities from whatever communication system has been characterized in its rule base (that is described in Section 4.) Our choice of Polylith as the principle interconnection subsystem within Polygen greatly simplified these issues for our initial experiments with the system, since the software bus organization elides from the programmer all representation issues for primitive and record data types. Hence, Polygen users may base their design decisions (in the composition phase) upon a single type system (the Polylith

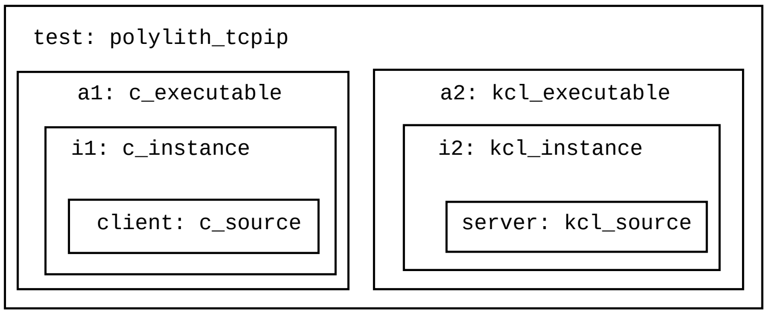

Figure 9: A partitioning for the complete example.

MIL), using techniques described in [13]. An ability to transmit abstract data types across Polygen-created interfaces is similarly dependent upon what interconnection rules to characterize compatibility have been installed in the underlying inference engine.

4 THE PARTITIONING METHOD

A package depends upon the integration capabilities of an environment. In the earlier example, two modules were configured into separate executable objects because they were incompatible to some degree — they could be composed only if the appropriate stubs were generated. In other environments, it might be possible to load and execute heterogeneous components within a single address space. Polygen must have enough inference capability to distinguish between the cases, and then, having determined a target configuration, it must generate both the stub source programs and the methods for integrating the final application. This section describes a method for determining the contents of a package.

An environment's interconnection capabilities constrain what types of integrations are possible. Given such constraints, one can determine whether it is possible to describe an application in terms of sets of compatible components. In Polygen, these sets are called *partitions* and are represented by directed, acyclic graphs whose internal nodes represent integration methods and terminal nodes represent source components. The partitioning for the example in Section 3.3 is shown in Figure 9. In this case, the partitioning is a tree because all modules are based on distinct source components and there exists a single composition method at the root (i.e., the `polylith_tcpip` bus). A partitioning with a single root is called a *valid partitioning*, and a package may be created for a configuration if and only if the configuration has a valid partitioning.

Depending on the integration capabilities available in an environment, several valid partitionings may be constructed for a single configuration. A valid partition is created in a bottom-up fashion. First, we identify the types of components in the configuration. Next, we determine the methods for integrating them into larger objects (i.e., partitions). The integration activity continues

```
module client {
/* see Figure 3 */ }

module server {
/* see Figure 3 */ }

module test {
   bus :  'polylith_tcpip' ::
   tool client ::
   tool server ::
   bindall
}
```

$\Longrightarrow$

```
?- ['/usr/polygen/package.pl'].
pmodule(client).
language(client,'C').
source(client,'client.c').
include(client,'client.h').
main(client).
import(client,lookup,[int],[str]).
pmodule(server).
language(server,'KCL').
source(server,'server.lsp').
export(server,lookup,[int],[str]).
instanceof(i1,client,test).
location(i1,'flubber.cs.umd.edu').
instanceof(i2,server,test).
location(i2,'flubber.cs.umd.edu').
bind(i1,lookup,i2,lookup,test).
?- package(test,polylith_tcpip).
```

Figure 10: Module specifications converted to Prolog assertions.

iteratively until a single method is found that integrates all of the objects of the previous iteration. The final method represents the root of a valid partitioning and includes all the components of a configuration.

Polygen implements this method in Prolog. A site administrator — *not* each programmer — describes the interconnection capabilities of an environment in terms of rules that constrain the satisfaction of partitioning goals. To construct a package, Polygen first reads in the given module specifications for an application. These are converted into a set of Prolog assertions (Figure 10) that encode facts about the modules and bindings in a configuration. After reading the assertions in Figure 10, Polygen attempts to satisfy the goal

$$?-\texttt{package}(\texttt{test}, \texttt{polylith\_tcpip}). \tag{1}$$

which asks the question, "Is it possible to create at least one package for the configuration named `test` using the Polylith TCP/IP-based bus in this environment?" If this goal can be satisfied, at least one package will be created for the given configuration. The Prolog inferencing mechanism searches the rule base and attempts to satisfy the package rule

```
package(N,T) :-
       modules(N,M),
       instances(N,I),
       partition(I,[[N,T,P]]),
       createpackage(M,I,[N,T,P]).
```

Prolog assigns the variable `N` the name "test" and the variable `T` the name `polylith_tcpip` and attempts to satisfy the sub-goals. The first two subgoals determine that the modules `M =`

[client,server] and module instances I = [i1,i2] are used in the "test" configuration. Next, a possible partitioning on the instances in I is found and placed in the list P. In this case, the partition goal is of the form

$$?-\mathtt{partition}([\mathtt{i1},\mathtt{i2}],[[\mathtt{test},\mathtt{polylith\_tcpip},\mathtt{P}]]). \tag{2}$$

The partition predicate asks "Does there exist a partitioning of the modules in this execution environment?" A partitioning is a list of the forms

$$[label,method,[t_1,\ldots,t_n]]$$

where $t_1,\ldots,t_n$ is a list of module instances (i.e., implementations) and partitionings. At the leaves of a partitioning are the instances in the list I. The method is the name of the composition method used to integrate the objects in $t_1,\ldots,t_n$. The label is the symbolic name assigned to a partition. For example, goal (2) asks "Does there exist a valid partitioning on the instances i1 and i2 such that they can be integrated on a TCP/IP-based bus?" In our environment, this goal would be satisfied if

$$\mathtt{P}=[[\mathtt{a1},\mathtt{c\_executable},[\mathtt{i1}]],[\mathtt{a2},\mathtt{kcl\_executable},[\mathtt{i2}]]]. \tag{3}$$

This partitioning states that a composition can be created using a TCP/IP-based bus if two separate objects are created for each instance.

The Prolog inference engine attempts to satisfy partition goals using the composition rules for the current environment. These rules are authored by system administrators and kept in a read-only system file. The composition rules shown in Figure 11, for example, are used to determine whether or not a set of instances can be composed into a single executable object. According to the rules in Figure 11, a set of instances are composed using the c_executable method if all the components are instances of modules written in C, execute on the same host, and as a set have no more than one main entry point.

Once a set of valid partitionings is determined for the instances in a configuration, the goal called createpackage acts as a code generator. The composition methods in (3) — c_executable and kcl_executable — are invoked while satisfying this goal. These methods generate the package. Since a partitioning is simply a tree whose leaves are module instances, the interface software can be generated by traversing the partition from the root and invoking the composition method at each node.

The current Polygen driver code is implemented by 50 Prolog rules. Each composition method consists of about 50 rules including the compatibility rules and the rules for generating stubs and makefile rules. To add a new composition method, one must write new compatibility rules and code generation rules. The compatibility predicate is of the form

```
type(X,c_instance) :- instanceof(X,M,_), language(M,'C').

samelocation(_,[]).
samelocation(X,L) :-
       location(X,S),
       applist(location,[S],L).

compatible([X],c_executable) :- type(X,c_instance).
compatible([X|L],c_executable) :-
       type(X,c_instance),
       countmain([X|L],M),
       M =< 1,
       samelocation(X,L),
       compatible(L,c_executable).
```

Figure 11: Some compatibility rules used in the package tool.

$$\texttt{compatible}(\texttt{L},\texttt{T})$$

where L is a list of partitions or instances and T is the name of the composition method. There are two types of code generation rules for each composition method in an environment: makefile rules and stub rules. Both types are invoked while satisfying the `createpackage` sub-goal of a `package` goal on a partitioning.

The root composition method need not be given explicitly in the `package` goal — it may be determined from the composition rules using the deductive capabilities of Prolog. The composite specification in Figure 6, for example, does not require the use of a particular bus implementation. In this case, the package goal would be of the form

$$?-\texttt{package}(\texttt{test},\texttt{B}). \tag{4}$$

The subsequent partition goal would be of the form

$$?-\texttt{partition}([\texttt{i1},\texttt{i2}],[[\texttt{test},\texttt{B},\texttt{P}]]). \tag{5}$$

The bus implementation `B` is determined during the attempt to satisfy the partition goal. In our environment, goal (5) is satisfied by the variable assignments

$$\texttt{B} = \texttt{polylith\_xns}, \texttt{P} = [[\texttt{a1},\texttt{c\_executable},[\texttt{i1}]],[\texttt{a2},\texttt{kcl\_executable},[\texttt{i2}]]].$$

based on the properties of instances `i1` and `i2`. As specified in Figure 6, the instance `i2` (the server) is located on the host machine `konky.cs.umd.edu`. In our environment, this host does not implement the Polylith TCP/IP-based bus, but it does implement an XNS-based version. Luckily, the host `flubber.cs.umd.edu` on which `i1` resides implements both bus versions. The developer is unaware that the generated stubs and configuration file in this case look very different as a result of location annotations in the composite specification.

5 CONCLUSION

We have described a packaging system that allows diverse software components to be easily interconnected within heterogeneous programming environments. Interface software and stubs can be generated for programmers automatically once they express their application's geometry in a few simple rules and MIL attributes. By generating custom interface code for each application (based on analysis and extraction of interfacing requirements), our approach is able to produce executables whose run-time performance is comparable to manually-integrated applications.

An important feature of this approach is how easy it is for system managers to add a new language or execution environment to Polygen. Each time a new rule declaring compatibility is added, so too does the manager provide a set of possible commands that would make two components compatible according to that rule. An example of this (that if done manually would entail writing more source code than can be reasonably given here in its entirety — certainly more source than a programmer would want to generate manually without need) is the ease by which one can tailor application programs for use on workstations. The addition of a small set of rules yields a packager that can layer network RPC stubs underneath wrapper codes needed to integrate the application into a powerful window system. As a result, not only can the simple example used throughout this paper be run as a distributed application, but each component can be set up to interact with users via its own window on the local host workstation. Should the user elect to run under a different window system, then, once again, all the source components can be adapted automatically for use on the new platform.

The availability of early stub generation systems relieved programmers having to create interface software manually. Polygen also relieves programmers of having to identify and extract the interfaces in the first place, and of having to tell their configuration management system how the stub programs should incorporated into the application.

References

[1] B. Allen and S. Lee. A Knowledge-Based Environment for the Development of Software Parts Composition Systems. *Proceedings of the IEEE 11th International Conference on Software Engineering* (May 1989), pp. 104-112.

[2] M. Barbacci, D. Doubleday, C. Weinstock and J. Wing. Developing Applications for Heterogeneous Mechina Networks: The Durra Environment. *USENIX Computing Systems*, vol. 2, (1989), pp. 7-35.

[3] B. Bershad. High Performance Cross-Address Space Communication. Ph.D. Dissertation, University of Washington, Seattle, Technical Report 06-02, (1990).

[4] B. Bershad, D. Ching, E. Lazowska, J. Sanislo and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, vol. 13, (August 1987), pp. 880-894.

[5] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Trans on Software Engineering*, vol. 2, (June 1976), pp. 80-86.

[6] J. Falcone. A Programmable Interface Language for Heterogeneous Distributed Systems, *ACM Transactions on Computer Systems*, vol. 5, (November 1987), pp. 330-351.

[7] S. Feldman. Make: A Program for Maintaining Computer Programs, *UNIX Programmer's Manual*, USENIX, (1984).

[8] P. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering*, vol. 13, (January 1987), pp. 77-87.

[9] J. Jones, R. Rashid, and M. Thompson. Matchmaker: An Interface Specification Language for distributed processing. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, (January 1985), pp. 225-235.

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

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

[12] J. Purtilo and J. Callahan. Parse Tree Annotations, *Communications of the ACM*, vol. 32, (December 1989), pp. 1467-1477.

[13] J. Purtilo. The Polylith Software Bus. *University of Maryland CSD Technical Report 2469*, (1990). Submitted for publication.

[14] M. Schroeder and M. Burrows. Performance of the Firefly RPC, *ACM Transactions on Computer Systems*, vol. 16, (February 1990), pp. 1-17.

[15] R. Snodgrass. **The Interface Description Language**, Computer Science Press, (1989).

[16] E. Soloway, J. Bachant and K. Jensen. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a Very Large Rule Base. *Proceedings of the National Conference on Artificial Intelligence*, (July 1987).

[17] J. Stamos and D. Gifford. Implementing Remote Evaluation, *IEEE Transactions on Software Engineering*, vol. 16, (July 1990), pp. 710-722.

[18] Sun Microsystems, *External Data Representation Reference Manual*, Sun Microsystems, (January 1985).

[19] Sun Microsystems, *Remote Procedure Call Protocol Specification,* Sun Microsystems, (January 1985).

[20] M. Tiemann. Solving the RPC Problem in GNU C++, *Proceedings of the 1988 USENIX C++ Conference*, (1988), pp. 343-361.

[21] Xerox Special Information Systems, Courier: The Remote Procedure Call Protocol, Xerox Corporation, (1981).

ACKNOWLEDGEMENTS

We are grateful to John Gannon, Dewayne Perry and Marv Zelkowitz for their many helpful comments and suggestions concerning both our packaging system itself and this paper concerning it. Thanks to the hearty users — in particular, Jorma Taramaa — who bravely tried predecessors of Polygen and, through their suffering, inspired us to construct a packager that really works.