SOFTWARE PACKAGING

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

Many computer programs cannot be easily integrated because their components are distributed and heterogeneous, i.e., they are implemented in diverse programming languages, use different data representation formats, or their runtime environments are incompatible. In many cases, programs are integrated by modifying their components or interposing mechanisms that handle communication and conversion tasks. For example, remote procedure call (RPC) helps integrate heterogeneous, distributed programs. When configuring such programs, however, mechanisms like RPC must be used explicitly by software developers in order to integrate collections of diverse components. Each collection may require a unique integration solution. This thesis describes a process called *software packaging* that automatically determines how to integrate a diverse collection of computer programs based on the types of components involved and the capabilities of available translators and adapters in an environment. Whereas previous efforts focused solely on integration mechanisms, software packaging provides a context that relates such mechanisms to software integration processes. We demonstrate the value of this approach by reducing the cost of configuring applications whose components are distributed and implemented in different programming languages. Our software packaging tool subsumes traditional integration tools like UNIX MAKE by providing a rule-based approach to software integration that is independent of execution environments.

Chapter 1

Introduction

Most high-level programming languages provide function and procedure call abstractions in order that software developers can define their own operations and reuse libraries of functions written by other programmers. A function or procedure is *seamless* to use because a "call" is an abstraction that is independent of any runtime environment, i.e., the use and definition of a function does not change between environments. Such abstractions make programs more reusable and portable to many types of environments regardless of their operating system and hardware characteristics. Even though functions may be defined in separate components such as files or libraries, tools like compilers and link editors handle the tasks of translating and combining these components into programs designed to execute in a specific environment. The developer must invoke the proper tools in their proper sequence to build the application, but the components are the same regardless of the environment.

Integration is the step-by-step process of translating and combining software components into new components. For example, a piece of source code can be translated into object code by a compiler and then combined with libraries by a link editor to produce an executable program. The resulting executable program defines a runtime implementation of the application for a specific machine and operating system. The cost of constructing the compiler and link editor is offset by the ability to recompile that source code across many environments. Furthermore, the cost of the tools is amortized over all the programs written in the programming language.

The integration process is more difficult, however, if functions are implemented in different programming languages or as remote services in a distributed system. While function and procedure call abstractions can be implemented by several mechanisms in such situations (e.g., pragmas, pipes, remote procedure call (RPC)), developers must often construct additional software that provides a "bridge" between components. This additional software, such as remote procedure call stubs, is expensive to develop and unlikely to be reused in other applications. Code generators, such as stub compilers, can be used to produce the additional software automatically, but developers must provide interface specifications in such cases. As in homogeneous environments, developers must invoke the proper tools in order to integrate an application into an executable program, but this process is much more complex in heterogeneous applications.

The problem of integrating heterogeneous programs becomes critical as the need for software reuse grows. By reusing existing programs in new designs, we can significantly reduce development costs, but many data representations and runtime environments are incompatible without the use of "bridge" code. For instance, the United States Department of Defense estimates that most of its 1.4 billion lines of code is used to pass data back and forth between incompatible and inconsistent applications[Stra92]. Much of this "bridge" code is redundant and highly-dependent on each system or execution environment.

This thesis presents a method for seamlessly integrating computer programs in heterogeneous, distributed execution environments. We have developed a tool called a *software packager* that determines which tools can use to integrate collections of programs. The software packager allows computer programmers to connect programs together abstractly without explicit concern for reconciling implementation differences. The software packager determines whether or not programs can be integrated based on the types of components involved and the available integration tools (e.g., compilers, linkers, stub generators). If it is possible to integrate the components, then the packager determines which tools are needed, how to apply them, and the proper sequence of their application.

Component types are based on characteristics like programming language, entry points, and control properties. Current integration methods require that developers know what types of components are compatible, their interconnections, how to implement the interconnections, and how to integrate the components in each execution environment. Software packaging requires only that the developer know about component types and their interconnections. The software packager relies on a set of production rules in each environment to determine how to implement the interconnections and integrate the components. Like a compiler, the cost of constructing the production rules is balanced by the ability to port the application to other environments and amortized over all applications packaged in the execution environment.

Software packaging reduces the cost of integrating software systems when compared to existing configuration methods such as UNIX MAKE, remote procedure call and similar tools. Using these traditional tools, developers must explicitly specify the process of integrating computer programs. If a program is reconfigured, then the integration process may be altered depending on the available integration tools meaning that the developer must respecify the integration for each change. Reconfiguring a system includes such tasks as moving a system to another computing environment, distributing components on processing elements, and implementing components in different programming languages. Such changes strongly impact how a system is integrated. The software packager reduces the impact of reconfigurations by providing a high-level approach to integration for a set of programs and processors, much like a compiler does for a single program and machine.

Previous integration approaches focus solely only on integration tools, like distributed agents and remote procedure call stub generators, instead of the integration processes that require these tools. With tools alone, the developer must still specify how to integrate an application explicitly. The developer must alternate between abstraction and implementation: connecting components together and implementing those connections. Software packaging leverages integration tools implicitly. This results in faster development since programmers can deal with connections in a seamless fashion and integration processes are determined automatically.

1.1 Integrating a Heterogeneous Application

Suppose our task is to develop a factorial application by integrating existing source components written in different programming languages. The application is modularized into two components: a *client* for dealing with input/output and a *server* that implements a `factorial` function. One solution based on this design is shown in Figure 1.1. It consists of a client component implemented

```
(defun compute-facs ()
    (if (<= (si:argc) 1) (format t "usage:  factorial num1 num2 ... %")
        (do ((i 1 (+ i 1))) ((>= i (si:argc)))
            (let ((n (read-from-string (si:argv i))))
            (format t "The factorial of  D is  D. %" n (factorial n))))))
```

(A) Lisp implementation of client component

```
factorial(x)
    int x;
{
    if(x <= 1) return 1;
    else return(x * factorial(x-1));
}
```

(B) C implementation of server component

Figure 1.1: Source implementations of client (A) and server (B) components.

in the Lisp[1] programming language and a server component implemented in the C programming language. The client component relies on a `factorial` function that is external to its definition. The server component implements the `factorial` function required by the client. The client invocation of `factorial` sends an integer as an argument and expects an integer in return. The server provides a complementary interface.

While the two programs are compatible relative to their abstract interfaces, their implementations are incompatible. The major problem is that the Lisp and C runtime environments are different in several ways, e.g., they represent strings and numbers differently. Such incompatibilities can be reconciled by introducing additional software to bridge the gap between the two implementations and their runtime environments.

There may be several ways to integrate heterogeneous programs in an execution environment. We integrate the Lisp and C implementations via a wrapper — additional code that is used to transform, convert, and bridge runtime differences and data representations. In our environment it is possible to mix Lisp and C code in the same runtime environment via wrappers. The final system is integrated into a single executable object. The function call to `factorial` is implemented by a procedure call mechanism that operates through the wrapper – additional Lisp code needed to describe the factorial function and dynamically load the object code into the Lisp runtime environment.

Figure 1.2 illustrates the steps necessary to integrate the components of our factorial program.

[1]Austin-Kyoto Common Lisp (AKCL).

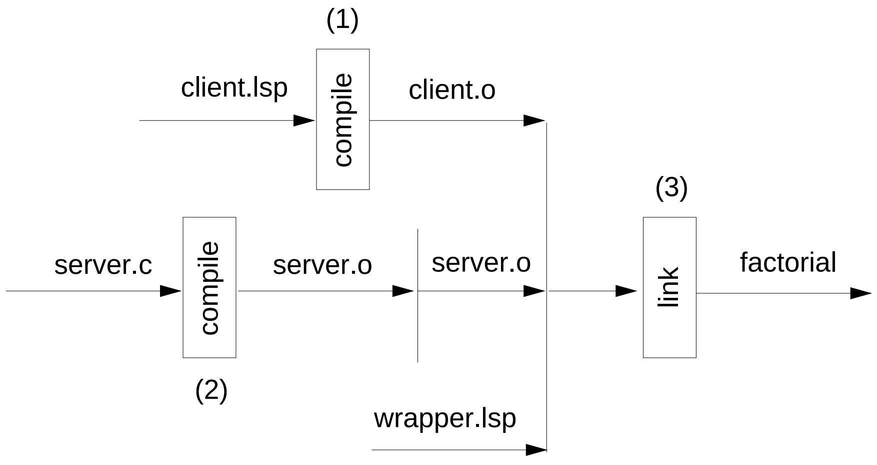

Figure 1.2: Integrating Lisp and C code.

The process is shown as a directed acyclic graph to reflect the dependencies between steps. Assume the Lisp and C code shown above are stored in the files `client.lsp` and `server.c` respectively. These are compiled by the Lisp and C compilers respectively to produce the object files `client.o` and `server.o` (steps 1 and 2). Although the files have identical extensions (.o), they are organized differently and cannot be integrated by the standard link editor. Additional code is needed to integrate them. The wrapper code is in the file `wrapper.lsp` which contains the single line

```
(defentry factorial (int) (int factorial))
```

that describes an entry point to an external C function named factorial. Next, we use the Lisp interpreter as a link editor (step 3) to produce the factorial executable using the following commands

```
(load "client.o")
(si:faslink "wrapper" "server.o")
(si:init-hook '((compute-facs) (bye)))
(si:save-system "factorial")
```

The `load` command loads the compiled client Lisp code (`client.o`), `si:faslink` loads the wrapper file (`wrapper.lsp`) and the server object code (`server.o`), `si:init-hook` specifies the entry point of the program, and `si:save-system` produces the executable file called `factorial`. The user can then run the program to produce the desired output:

```
% factorial 5 6 7
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
%
```

1.2 Software Packaging

The behavior of any solution to the factorial problem should be independent of how we implement each component. The integration process, however, is highly dependent on what implementations

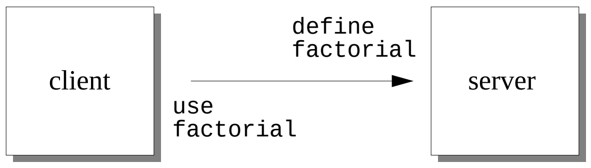

Figure 1.3: Abstract client and server structure.

are available. While each component relies on functional abstraction to isolate itself from such decisions, different implementations of each component will influence how the function call abstraction is implemented. In homogeneous programming environments, call abstractions permit seamless integration between components because the abstractions are an integral part of the language and runtime environment. In heterogeneous environments, the software packager provides the same transparency for multiple-language and distributed applications by determining how to integrate diverse components based on the types of components and the available integration tools.

The software packager accepts modular descriptions of applications as input and generates a *package* that implements the given system in an execution environment. For example, the module structure of the factorial program is shown in Figure 1.3. This structure includes the components and their interconnections. Given a description of the modular structure of an application and the implementations for each component, the packager generates a package that includes the code, wrappers, and integration steps necessary to build an implementation of the application. The software packager adapts components, chooses compatible implementations, and selects the appropriate tools needed to integrate a system of components.

The software packager determines the steps necessary to integrate an application in an execution environment. Each environment provides *production rules* that characterize the abstract integrations the are possible in that environment. These rules are reused by many applications in an environment. The developer supplies only the modular description of a program and implementations of the components. The packager uses the production rules in each environment to determine how to integrate these implementations based on their types and interconnections.

The major advantage of software packaging lies in its ability to automatically determine the integration steps for software products after reconfigurations. Existing methods require extensive changes to configuration programs (e.g., MAKEFILES) after application components are added, reimplemented, or distributed. In Chapter 5, we will compare software packaging with existing methods, UNIX MAKE and remote procedure call, to show that reconfigurations are more easily accommodated through the use of package specifications.

1.3 Problems

There are many reasons why differences exist between software components that make integration difficult. In the next sections, we outline the reasons why interconnections between heterogeneous systems are desirable and complex. Our solution is motivated by the economic need to reuse existing software in new systems despite their implementation differences. A high-level, modular software design can be reused in different contexts regardless of how its components and their connections are implemented. With software packaging, an application can be integrated using several different technologies. We are not limited to a single integration system, but use many systems to implement interconnection abstractions in order that developers can focus on how to structure their application instead of how to interconnect the components in a variety of environments.

1.3.1 Legacy Code and Systems

Old systems must coexist with new systems because it would cost too much in many cases to upgrade all components in a collection simultaneously. An existing program cannot be discarded simply because another program is installed. New programs must be compatible with existing systems or the new methods must be adopted incrementally by adapting the old system or gradually converting it. Furthermore, existing databases and files may not be compatible with new programs. For this reason, many programs are designed to be backward compatible to avoid isolating existing users. Even when a new system is introduced, old systems may remain for long periods of time until they are upgraded. In many cases, service must be provided continuously even while upgrades are in progress.

1.3.2 Coupling between Software Components

Besides data representation conflicts, computer programs also differ because they depend on different execution contexts. For example, a program that uses one set of interrupt signals to control its execution cannot be combined in the same address space with a program that uses the same signals for other reasons. In this case, the programs must execute in separate address spaces with their own interrupt vectors. Such programs are coupled to specific execution contexts external to their implementation, i.e., they depend on specific runtime environments. By definition, coupling reduces software reuse in other contexts. Coupled programs can be adapted to new contexts (e.g., SunView programs can run under X windows with minor massaging and the XView library), but this is rare and expensive to implement. It is, however, an alternative to reimplementation.

1.3.3 Specialization of Languages and Systems

Computers, languages, and protocols are specialized for problem domains. Numeric problems may best be solved in FORTRAN rather than LISP. We must recognize that computer systems, languages, and protocols will continue to be specialized. Specialization is necessary because it allows developers to construct solutions in terms convenient to a specific problem-domain. However, problems decompose into subproblems in several domains. Designing an airfoil, for example, may require a computer-aided design (CAD) program as well as a finite-element processing program. We must allow developers to use the tools that are most appropriate to their problem domains and find ways to integrate the diverse solutions to their subproblems.

1.3.4 Efficiency of Execution

Certain representations of information may be accessed faster than others. A list of names and phone numbers may be alphabetized for quick access by humans, but an operator may need a list ordered by phone numbers. People and operators require different "views" of the same information. Many programs store information in different formats to access their view of the data, but this limits exchanging the information with other programs that have different views. The information can exist as copies in separate views, but keeping the separate copies consistent if changes occur is a difficult problem. Developers must balance the tradeoffs between efficiency and consistency control when designing a system.

1.3.5 Distribution of Components

The trend in modern computer systems is toward decentralization. Users now have powerful processing capabilities at their personal disposal at reasonable prices. The lack of central control over computing resources, however, has resulted in the development of divergent systems with their own languages and protocols. The existence of legacy systems and the need for efficiency and specialization has created enormous differences between systems. Distributed computer systems can be connected via many technologies and this choice impacts the performance and reliability of the final system. The location and access to information services in such systems are important decisions for designers who must accommodate these configuration constraints.

1.4 Outline

Software packaging allows developers to combine software components with different implementations. Although such differences must be reconciled in order to integrate the components, the software packager determines what bridges are needed based on the types of components. For each execution environment, a set of production rules describe the types of integrations possible. Thus, it is possible to integrate components only if the proper tools exist. Given the proper tools, the details of integrations are hidden from the developer.

Chapter 2 describes the principles of software packaging in terms of the relation between program structure and the integration process. Chapter 3 describes the packaging language used to describe application structure and components. Chapter 4 describes the rule language used to describe the integration processes available in an environment. Finally, Chapter 5 presents examples of software packaging applied to significant programming problems. We compare software packaging to existing integration tools in terms of convenience and reduced complexity of specifications of heterogeneous programs.

Chapter 2

Concepts

Despite the differences between software systems, many programs can be integrated if the proper tools are available. In Chapter 1, we outlined the steps necessary for integrating software components in an example with different implementations. Each step of the integration process involved the use of tools like compilers, linkers, converters, wrapper and stub generators. Such tools are used to integrate a given set of software components (e.g. source files) into a final product (e.g., an executable file). The tools in an execution environment define the integrations possible in that environment. Software packaging allows developers to determine the integration processes automatically based on the types of components in an application.

2.1 Software Packaging

Like the factorial program, many software applications have a logical structure of components that is independent of how each component is implemented and how the application is integrated. For example, Figure 2.1 depicts a *production graph* for the factorial program given the Lisp and C implementations. The left side of the graph depicts the application as a composition of client and server components. This side is called the *software structure graph.* Implementations are shown as descendants of a component (shown as rectangles). This is true for the application itself as well as its components. At the leaf nodes of the structure graph are primitive implementations (e.g., source code). These represent implementations that cannot be further subdivided.

The right side of the graph depicts the integration of the implementations into an executable program. This side is called the *software manufacture graph.* Each node in the manufacture graph represents a translation or combination of components using available tools in an environment. The manufacture graph also determines a partial order in which components can be integrated independently.

Given the software structure graph, the software packager automatically constructs the software manufacture graph. The software structure graph is specified in a textual specification language called the PACKAGE specification language (described in Chapter 3). The developer only specifies a "package" and inputs this to the software packager. The output is a program that builds the application from selected primitive implementations of program components. For example, the output is actually a UNIX MAKEFILE in our prototype.

The packager determines the software manufacture graph based on production rules that characterize the *abstract* software manufacture graphs in an environment. The approach is much like having a grammar for a programming language. The packager uses inferencing algorithm to resolve

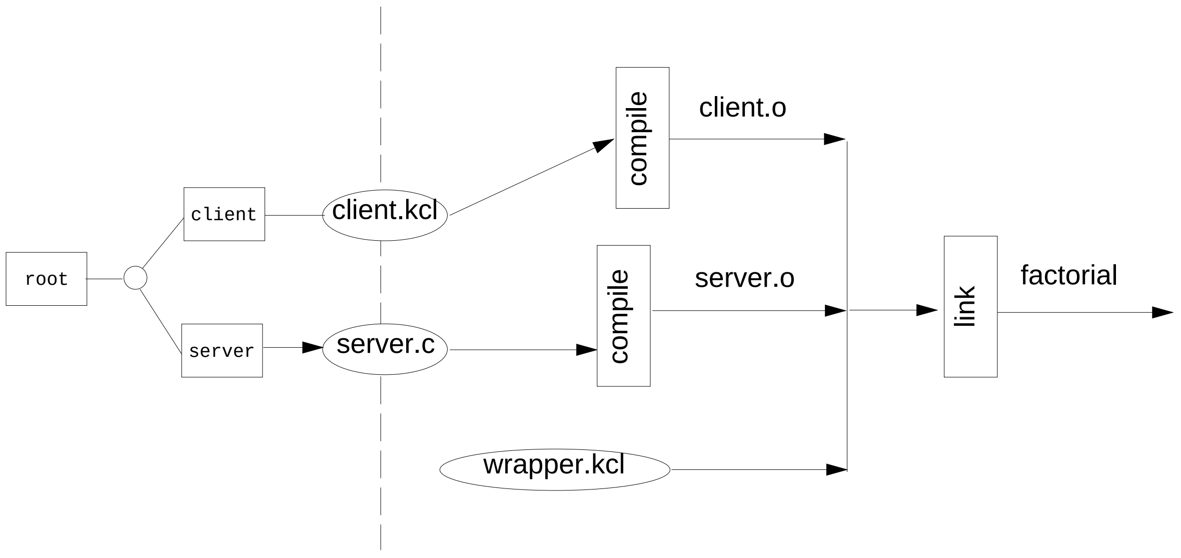

Figure 2.1: Production graph for the factorial application.

which implementations are compatible and how to build the target object based on the production rules. Although the rules are complex to construct, they are used by many applications in an environment.

The next sections describe software structure graphs, concrete and abstract software manufacture graphs, and the details of the software packaging process. This chapter ends with a discussion of related work including an overview of tools like UNIX MAKE that employ software manufacture graphs. We compare and contrast software packaging to these existing methods.

2.2 Software Structures

Regardless of how the factorial application is integrated, its abstract structure remains the same as shown in Figure 1.3. Developers often refer to this structure as the "architecture" of an application. Programmers who must maintain software products implicitly use this structure to orient themselves with the layout of a product. This section describes a technique for specifying such structures which are explicit and independent of particular execution environments.

One can specify a computer program in many forms. Most programs are comprised of files, statements, functions, variables, and other components. These components depend on each other and their execution is sequenced in some fashion so that the overall program has the desired behavior. Each component relies on resources defined by other components. The structure of an application includes a description of each component and the dependencies between them.

One major problem is how to describe a component in a manner that is independent of any implementation. We present an approach to describing software structures, called *structure graphs*, that can be used to describe many levels of design: from gross structures to statement-level constructs. Software structure graphs are based on MILs [DeKr76], but differ from previous efforts in that our structures are hierarchical and components may have multiple implementations. This allows for selection of compatible implementations based on their types and other properties.

We view any software artifact as a module — a black-box characterized only by the specified behavior of its interface. An interface is a collection of ports or channels on which messages are received or generated. Ports represent resources implemented by a module: function calls, events, or input-output streams. Within a module, but hidden from the outside, is an implementation. A module may have several implementations but only one may be "inside" the module at a time. For example, a program with the same input-output behavior can be written in two different programming languages. From the outside, it does not matter which implementation is chosen, but its behavior should be consistent with its interface specification.

2.2.1 Choice

Portable software products often have multiple implementations of their components to handle special cases, i.e., different device drivers may be configured depending on the target platform. In the worst case, different implementations of the entire program exist for each target platform. Choosing the appropriate implementations depends on the target platform. Differences between component implementations can be large or small. A single source file may represent multiple implementations because it may be compiled differently depending on the target platform. For example, the `#ifdef` macro in C is used frequently to compile alternative parts of source code depending on a configuration context.

Choice is a fundamental constructor in large software structures. Even a simple data abstraction may have multiple implementations. For example, the Map data type (i.e., associative arrays) in the GNU C++ library has the following implementations:

AVLMap	implement maps via threaded AVL trees
RAVLMap	implement as AVL trees with ranking
SplayMap	implement maps via splay trees
VHMap	implement maps via hash tables
CHMap	implement maps via chained hash tables

These implementations are subclasses of the class `Map`, but their interfaces are identical. Such relationships between abstract classes like `Map` and subclasses that are specialized based only on implementation differences occur in object-oriented systems that do not separate subtyping and subclassing [HaWe91]. A subtype refines an interface whereas a subclass represents an alternative implementation. Module-oriented programming distinguishes between the two concepts by separating interfaces and implementations and providing for implementation choices within software structures.

The choice of an implementation for any module is based on a variety of factors including the required performance of operations, storage overhead, and data representation strategy. An implementation chosen for one part of a system may constrain implementation choices in other parts of the system. Implementations are compatible relative to such constraints. For example, choice of an implementation using dynamically allocation may mean that all components must use the same memory management style.

2.2.2 Composition

Another fundamental constructor in software structures is composition. Groups of modules are connected together because they define and use shared resources. For example, the factorial solution in Figure 1.3 is a composition of two module instances: a client and server. Composition is an implicit operation in most programming systems. For example, most programming languages bind uses and definitions of resources together if their names are the same (*by-name* binding). Use of a resource implies that the component defining the resource must be integrated into the product at some point. Linker/loaders assimilate components by matching uses to definitions. Our use of explicit module interfaces and bindings is necessary in cases where integration requires more complex bridges between components. This is particularly true in heterogeneous, distributed systems where remote procedure call (RPC) stubs or other types of links must be generated to integrate components at runtime. Strict encapsulation permits our packaging system to wrap components in new contexts as needed.

2.2.3 Software Structure Graphs

We combine choice and composition within a framework for constructing descriptions of software products called *software structure graphs* or simply "structure graphs." A structure graph is a directed (possibly cyclic) graph whose root represents a software product and its alternative implementations at many levels. There are two types of module implementations within a structure graph: composite implementations and primitive implementations. Within a graph, alternatives represent subsystem implementations. At the leaves of the graph are primitive implementations (e.g., source code, programs, services, etc.). A structure graph is similar to an AND-OR graph. The children of the root module represent implementation alternatives (OR nodes). Each alternative is either a composite implementation (AND node) or a primitive implementation (P node). Thus, a structure graph is a hierarchical description of an application, its subsystems, and alternate implementations. A structure graph is not a tree because there may be sharing at levels of the graph and cycles involving recursive implementations (i.e., a subsystem implementing a `factorial` module may itself include an instance of a factorial module).

The leaf nodes of a structure graph are called "primitive" because they correspond to native implementations of modules in an environment that cannot be broken down further into subsystems. Typically, primitive implementations correspond to source code files, but may also represent services, tools, data, or any software artifact or collection of artifacts. Software structure graphs do not limit the developer to one-to-one mapping to files, rather multiple files could be associated with a single primitive implementation or a single file could be associated with multiple primitive implementations.

Figure 2.2 represents a structure graph for the factorial example in which the server has an additional implementation: a remote service. Rectangles represent module instances, open circles represent compositions and ellipses represent primitive implementations. In this case, each module has only one instance in the structure graph. Wherever a module instance occurs in a graph, its associated subgraph is copied. The structure graph represents all alternatives for components within the application. In the factorial example, the client module has one primitive implementation — the Lisp implementation. The server module has two possible implementations: the C implementation and the remote implementation.

The structure graph provides developers with an explicit model for constructing, viewing, and

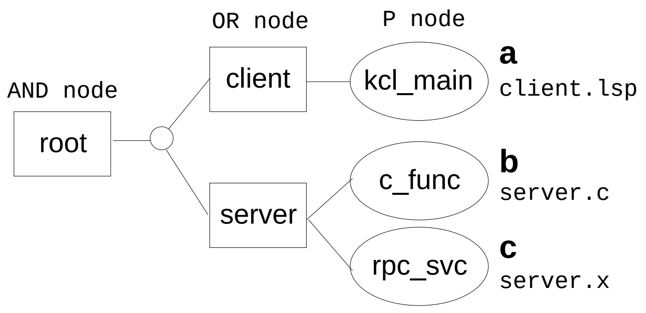

Figure 2.2: Structure graph for the factorial product.

maintaining alternative versions of software systems and their implementations. All software components are "black boxes" that may have one or more implementations. The software packager uses environment-specific rules to choose compatible implementations for components within a structure graph.

The software packaging specification language is a *module interconnection language* (MIL) that allows programmers to describe software structures in terms of choices and compositions of software modules. A module specification describes the resources provided and used by a software component. This is more general that an object-oriented approach that describes software components only in terms of resources they provide. For example, the interface of a stack *object* is typically described as providing three basic functions: *push, pop, top.* Figure 2.3 depicts a generic stack *module* with the same interface, but we associate two implementations with the stack module. These implementations are composite implementations consisting of other module instances. The external ports of the stack module are connected to ports of internal modules. This is known as *aliasing* and represents the bridges between higher and lower level abstractions. For example, one composite implementation includes instances of modules `ArrayStack` and `Array` that implements stacks using arrays while the other implementation includes instances of module `ListStack` and `List` that implements stacks using linked lists. The structure graph for the stack module and its implementations are shown in Figure 2.4.

2.3 The Software Manufacture

The process of integrating software components is known as a *software manufacture* [Bori89]. A software manufacture is the step-by-step process of synthesizing new software artifacts from existing ones by applying tools available in an execution environment. The available tools include any language translators and integration mechanisms installed in an environment. A software manufacture for an application specifies how to build a product from given set of components. These components include source files, executables, data files, or even executing services.

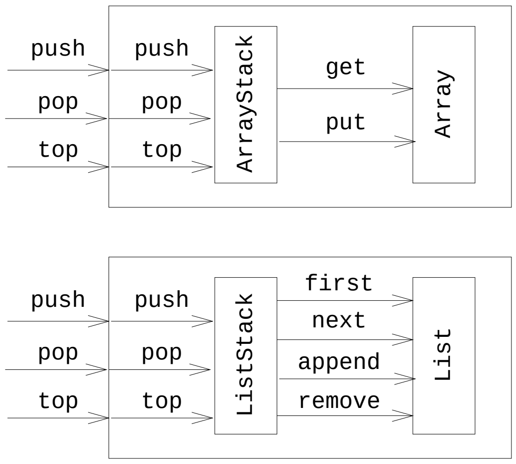

Figure 2.3: Two alternate composite implementations of a stack module

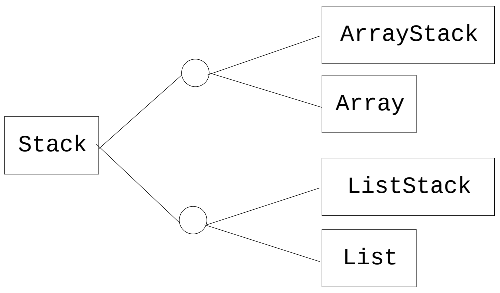

Figure 2.4: Structure graph showing two alternate composite implementations of a stack module

2.3.1 Software Manufacture Graphs

In our first solution to the factorial problem, we proposed that the Lisp and C components could be integrated via Lisp wrappers. Based on the directed acyclic graph in Figure 1.2, a process to perform the integration could be

(1) compile Lisp code into object code
(2) compile C code into object code
(3) link object code files

since these steps obey the partial order specified by the graph. Any ordering of integration steps that obeys the partial order is valid. Many configuration management tools exploit this partial order that exists within integration processes.

A graph that depicts the partial order of steps necessary to integrate software components into a product is called a *software manufacture graph.* The graph specifies the steps necessary to build an application from a given set of components as a partial order. Each node in a graph corresponds to an action that performs a single step in the integration process. A manufacture graph proceeds from left to right with the raw components as input on the left and the final product(s) as output.

2.3.2 Abstract Software Manufacture Graphs

We introduce the concept of the *abstract software manuafcture graph* to characterize the integration processes available in the environment. In any environment, the available tools dictate the types of software manufacture graphs that are *legal*, i.e., those that represent valid integration processes. We characterize the general form of these graphs as abstract software manufacture graphs.

We characterize the abstract form of legal manufacture graphs through the use of production rules. Figure 2.5 depicts an abstract form of the manufacture graph in Figure 1.2. In Figure 2.5, we relabel the nodes in Figure 1.2 with production rule numbers and the transitions with *object types.* The leaf nodes on the left side of the graph represent primitive object types in the environment. These may or may not correspond to source files and can be associated with other artifacts in the system, e.g., ports, sockets, memory addresses, and services. Object types have associated attributes that identify such properties. For example, the `kcl_main` is an object type that represents a Lisp source file with an entry point. The `kcl_main` object type has a `FILE` attribute associated with it that specifies the source file in the environment. The `kcl_main` object also specifies an `ENTRY` attribute for the program entry point. Object types and attributes are covered in detail in Chapter 3.

Nodes within the graph are labeled with production rules that correspond to procedures that utilize environment tools. For example, the rule

```
c_func_obj          <= c_func
```

specifies a method for producing a `c_func_obj` object from a single `c_func` object. This corresponds to a special case of the "`.c.o`" suffix rule. Production rules in abstract software manufacture graphs are similar to those found in attribute grammars: the left side of a rule represents the target while the right side is a list of the components from which the target is constructed. Like symbols within attribute grammars, objects in production rules also have attributes and actions that manipulate these attributes to produce an integration.

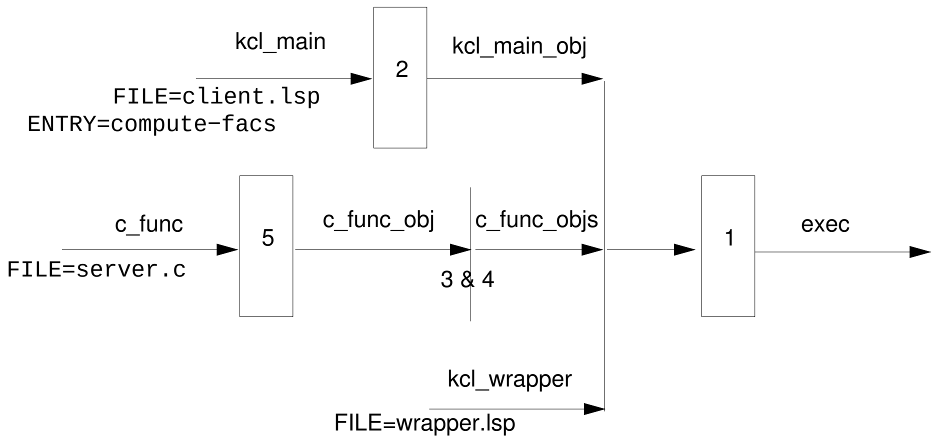

Figure 2.5: Abstract software manufacture graph for client-server RPC

Each environment specifies its own unique software integration processes in terms of a set of production rules. The production rules used in Figure 2.5 are

(1)	exec	←	kcl_main_obj c_func_objs kcl_wrapper
(2)	kcl_main_obj	←	kcl_main
(3)	c_func_objs	←	c_func_obj c_func_objs
(4)	c_func_objs	←	
(5)	c_func_obj	←	c_func

These rules form a "grammar" for legal software manufacture graphs in an environment. For example, the graph in Figure 1.2 is a legal software manufacture graph according to the production rules above. Unlike suffix rules or IMAKE procedures, production rules relate the tools available in an environment to integration processes (i.e., sets of related rules). Every environment can characterize its legal manufacture graphs via production rules. New tools are leveraged by adding new rules.

We can derive a *concrete software manufacture graph* given a collection of primitive objects and a set of production rules. This is the basic approach of software packaging: determine a means of integrating compatible components based on available integration processes. Developers specify the objects, but they do not specify the production rules. These are written and installed in an environment by system administrators. They are accessed and shared by all developers in an environment. They change when tools are added or removed from the system. Developers must be aware of object types (i.e., leaf node types in the manufacture graph), but this is an improvement over having to remember platform-specific methods as in MAKE or procedures as in IMAKE. In the next section, we explore the specification of objects within application structures that are independent of programming environments.

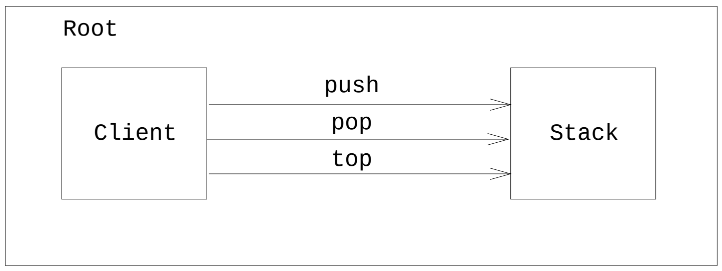

Figure 2.6: Composite implementation of client and stack application.

2.4 The Software Packaging Process

A client program that uses functions defined by a stack module need not be aware of how stack is implemented but some constraint may dictate which stack implementation is chosen. The problem of determining the choices of implementations based on constraints between components is part of the larger problem of *software scalability*, i.e., composing software components into larger programs and choosing compatible implementations.

Software packaging is ideally suited to handle software scalability problems. In Figure 2.6, a client module relies on resources provided by the stack module, i.e., it calls the stack functions to perform some computation. The entire application can be viewed as the graph in Figure 2.7 that include all implementation choices for the client and stack modules. Depending on constraints imposed either higher in the structure graph or by the target environment, one implementation will be chosen over the other. In the absence of any constraints, the choice of implementation can be made arbitrarily.

If we choose a particular set of implementations for all modules in a structure graph, the selected leaf nodes comprise *an* implementation of the entire application called a *rendering*. Figure 2.8 depicts the two possible renderings of a client and stack application. The internal nodes of a graph serve to organize the application choices, place constraints, and connect ports of modules together. The leaf nodes of the structure graph represent actual native code, files, services, and tools that comprise an application rendering.

If we collapse the structure graph and map all aliases and connections into connections between leaf nodes, the connected graph of primitive modules that remains is called an *application graph*. An application graph contains only the primitive implementations and their direct interconnections. Figure 2.7 shows the two possible application graphs for a client and stack application.

2.4.1 Algorithm

Software packaging translates a structure graph description of an application into an integration process. Specifically, it involves constructing a software manufacture graph from selected leaf nodes of a software structure graph. Figure 2.9 depicts a combined graph for the factorial example. Selected leaf nodes of the structure graph are also leaf nodes of the manufacture graph. The combined graph is called a *production graph*. In this case, selected leaf nodes of the structure graph are leaf nodes in the manufacture part of the graph. This production graph represents

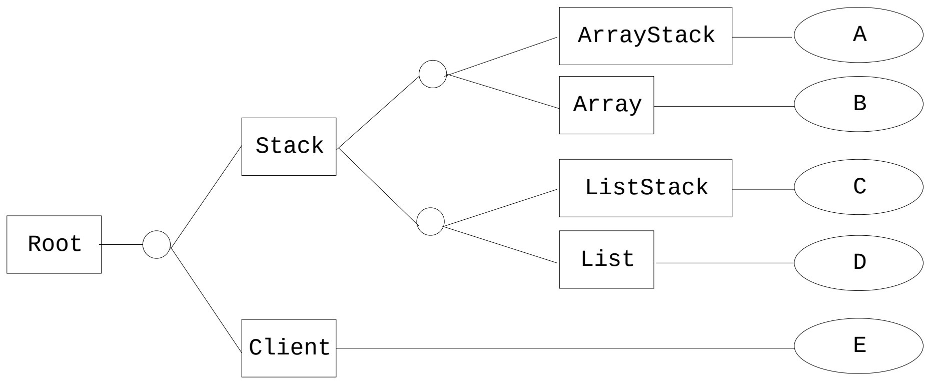

Figure 2.7: Structure graph showing implementations of client and stack application.

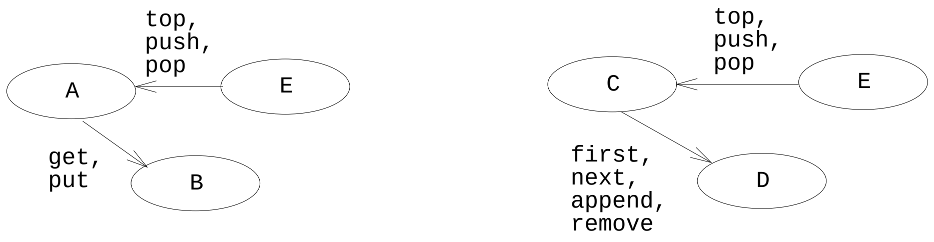

Figure 2.8: Two application graphs represent selected primitive implementations.

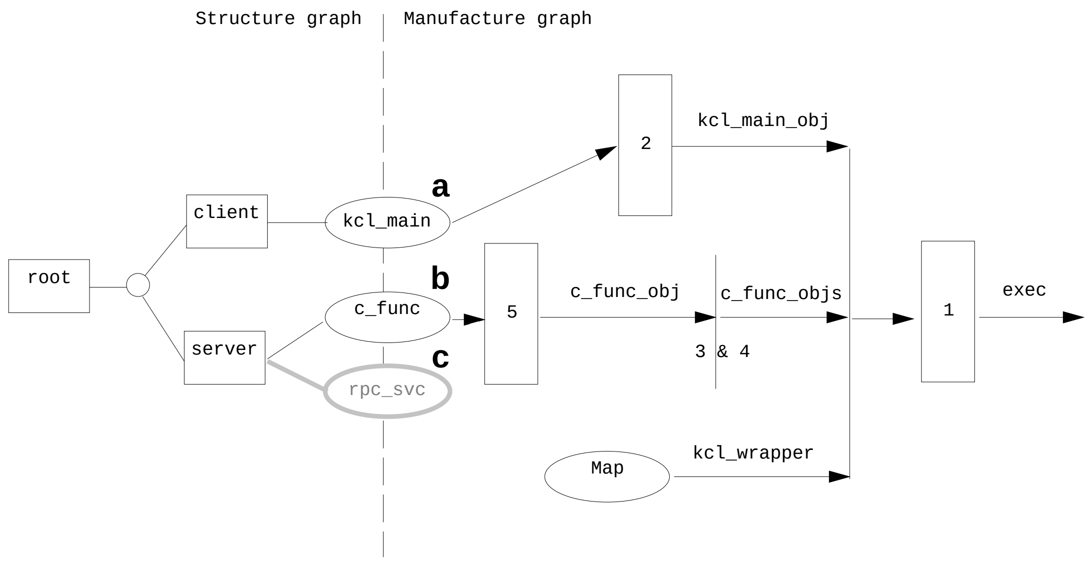

Figure 2.9: A complete production graph.

the solution comprised of the C and Lisp implementations. An alternate implementation of the server is eliminated because the C implementation for the server was chosen by the packager instead. We denote the elimination of an alternative and its subgraph by dotted lines. Solid lines denote successful alternatives within the structure part of a production graph. All lines within the manufacture part of a production graph are solid since it represents the derived integration process.

The packager uses a depth-first, backtracking search algorithm on a set of production rules to derive the manufacture graph for a given structure graph. Starting with the collection of all leaf nodes of a structure graph, the packager tries to find a compatible set of leaf nodes relative to a set of production rules. The process is similar to that used by inferencing algorithms in logic programming languages like Prolog. First, the leaf objects are placed in a collection known as the *pool*. The packaging algorithm then searches each rule from the top-most rule down trying to cover all objects in the pool. At some point in the search, the software packager backtracks if it cannot find an object required by a production rule. During backtracking, objects are returned to the pool and the packager tries alternative production rules. The packaging process fails if no production process exists or some components cannot be covered. During the packaging process, the packager eliminates paths to alternative objects within the pool using the structure graph. Nodes that represent alternative implementations are known as "cousins." The packaging algorithm removes "cousins" of each candidate leaf node within the pool. The elimination of the rpc_svc implementation in Figure 2.9, for example, was the result of including the c_func implementation in the manufacture graph. In more complex examples, the packager eliminates alternatives at all levels in the structure graph.

The packaging algorithm is shown in Figure 2.10. Following this algorithm, we trace the step-by-step execution of the packager in Table 2.1 as it determines the production graph in Figure 2.9 from the structure graph given in Figure 2.2 and the rules

```
procedure package(in object,inout pool,in remainder,out cover) is
begin
   if(object == Map) then succeed
      if(object in pool) then
         remove object from pool
         remove sibling and cousin objects from pool
         if objects in pool are members of legal remainder set
            then succeed
            else fail
      else
         for all rules such that object -> object1 object2 ...  objectN
            temppool[0] = pool
            tempremainder = pool
            loop through objectX (X=1...N) until exhausted or failure
               if X in 1..N-1 then
                  node = package(objectX,temppool[X],temppool[X],cover)
               else
                  node = package(objectX,temppool[X],tempremainder,cover)
               if package failed (node == NULL) then
                  if X == 0 then return NULL;
                     X = X - 1;
                     remove cover from temppool[X]
                  endif
                  temppool[X+1] = temppool[X];
            end loop
            if objects in pool are members of remainder set
               then return node;
      end
   return NULL;
end

pool = all leaf nodes of the structure graph
package("exec",pool,{},cover)
```

Figure 2.10: The packaging algorithm.

Step	Action	Rule	Pool	Remainder
1	add all leaf nodes to the pool		a,b,c	empty
2	call package(exec)		a,b,c	empty
3	try rule #1	1	a,b,c	empty
4	call package(kcl_main_obj)		a,b,c	a,b,c
5	try rule #2	2	a,b,c	a,b,c
6	call package(kcl_main)	2	a,b,c	a,b,c
7	remove kcl_main from pool	2	b,c	a,b,c
8	succeed rule #2	2	b,c	a,b,c
9	call package(c_func_objs)	3	b,c	b,c
10	try rule #3	3	b,c	b,c
11	call package(c_func_obj)	3	b,c	b,c
12	try rule #5	5	b,c	b,c
13	call package(c_func)	5	b,c	b,c
14	remove c_func from pool (and cousin rpc_svc)	5	empty	b,c
15	succeed rule #5	5	empty	b,c
16	call package(c_func_objs)	3	empty	empty
17	try rule #3 (recursively)	3	empty	empty
18	try rule #5	5	empty	empty
19	call package(c_func)	5	empty	empty
20	fail (c_func not in pool and no rules)	5	empty	empty
21	fail rule #5	5	empty	empty
22	fail rule #3	3	empty	empty
23	try rule #4	4	empty	empty
24	succeed rule #4	4	empty	empty
25	succeed rule #3	3	empty	empty
26	call package(kcl_wrapper)	1	empty	empty
27	try rule #6	6	empty	empty
28	call package(Map)	6	empty	empty
29	succeed rule #6	6	empty	empty
30	succeed rule #1	1	empty	empty

Table 2.1: Trace of the packaging algorithm.

1	exec	$\leftarrow$	kcl_main_obj c_func_objs kcl_wrapper
2	kcl_main_obj	$\leftarrow$	kcl_main
3	c_func_objs	$\leftarrow$	c_func_obj c_func_objs
4	c_func_objs	$\leftarrow$	
5	c_func_obj	$\leftarrow$	c_func
6	kcl_wrapper	$\leftarrow$	Map

beginning with the `exec` node as the desired target object type. This trace shows each rule as it executes in the search and the contents of the pool at each step where **a** represents the `kcl_main` components, **b** represents the `c_func` components, and **c** represents the `rpc_svc` component. At step (1), all leaf nodes of the structure graph are placed in the pool as candidates. The packager first searches the pool for an object of type `exec`. Since no such object is in the pool, the packager then calls the rule `exec` $\leftarrow$ `kcl_main_obj c_func_objs kcl_wrapper` in step (2). The packager subsequently calls the package algorithm recursively on the items on the right-hand side of this rule (steps 4,9,26). For the first item, `kcl_main_obj`, the packager first searches the pool for an object of this type. This is unsuccessful and the packager proceeds to use the rule `kcl_main_obj` $\leftarrow$ `kcl_main` in step (5). The packager calls the package algorithm recursively on the object type `kcl_main` in step (6) and finds it (object a) in the pool. The packager removes this object from the pool in step (7). It does not have any cousins (i.e., alternate implementations). We can see an example of cousin removal in step (14) when the c_func implementation is removed from the pool along with the remote implementation rpc_svc. The call to the rule kcl_main_obj ← kcl_main succeeds in step (15) and in step (16) the algorithm recursively searches for a c_func_objs object because of the rule c_func_objs ← c_func c_func_objs called in step (10). The packager will look for another c_func object in the pool in step (20) and fail because all primitive implementations have been included in the package. The packager then tries another c_func_objs rule, namely the rule "c_func_objs ←" that always succeeds as in step (24). This cause the rule c_func_objs ← c_func c_func_objs to succeed. The packager then searches in step (26) for the last object on the right-hand side of the top level rule — a kcl_wrapper object. This triggers a search for a Map object in step (28). A Map is a distinguished object that is always available. Thus, the search is complete because the kcl_wrapper rule succeeds which then causes the exec rule to succeed. Maps are described in a later section.

There is one more step to ensure that a package has been found — all implementations must be examined by the search. A structure graph is covered if all active leaf nodes within the structure graph are leaf nodes of the derived manufacture graph. Therefore, no valid path must exist from the root of the structure graph to a leaf node. The remainder argument serves to check whether or not all leaf nodes have been covered. Computing the coverage of a derived manufacture graph is necessary because many production rules can succeed if they denote collections of objects. Determining coverage is done by maintaining a legal remainder list during the search process. The remainder set is used by the packaging algorithm to ensure that all elements of the pool have been accounted for in the search. This set is equivalent to the objects in the pool, except fot the right-most part of the derivation. The right-most rules in the constructed manufacture graph has a remainder == empty which means that at the end of the search there must be no members remaining in the pool. For instance, if the server module has no implementations, then we could still construct a manufacture graph for the application as shown in Figure 2.11. This is because the "c_func_objs ←" rule always succeeds, but a package is not created because coverage is not achieved since the production graph contains nodes with valid paths in the structure graph that are not included in the manufacture graph.

The packager will succeed with the first viable package that it finds based on the order of the production rules. This is similar to the way in which most logic programming languages (e.g., Prolog) order the application of rules. If a package is not found, an incomplete production graph is produced and the developer is notified of the module instances that did not have valid implementations.

2.4.2 Maps

During the integration process, not the packaging process, bridges may need to be built between diverse components. Such bridges are similar to backpatching actions taken by link editors: they connect the *uses* of functions to their *definitions*. In a distributed, heterogeneous application, various tools may need information about implementations such as entry points, ports, and other resources used by a component in order to generate wrapper and stub code. Thus, the packager produces an external cross reference file so that such tools can readily access such information. For example, a stub generator can determine the properties of entry points to a component for which it needs to generate stubs during integration.

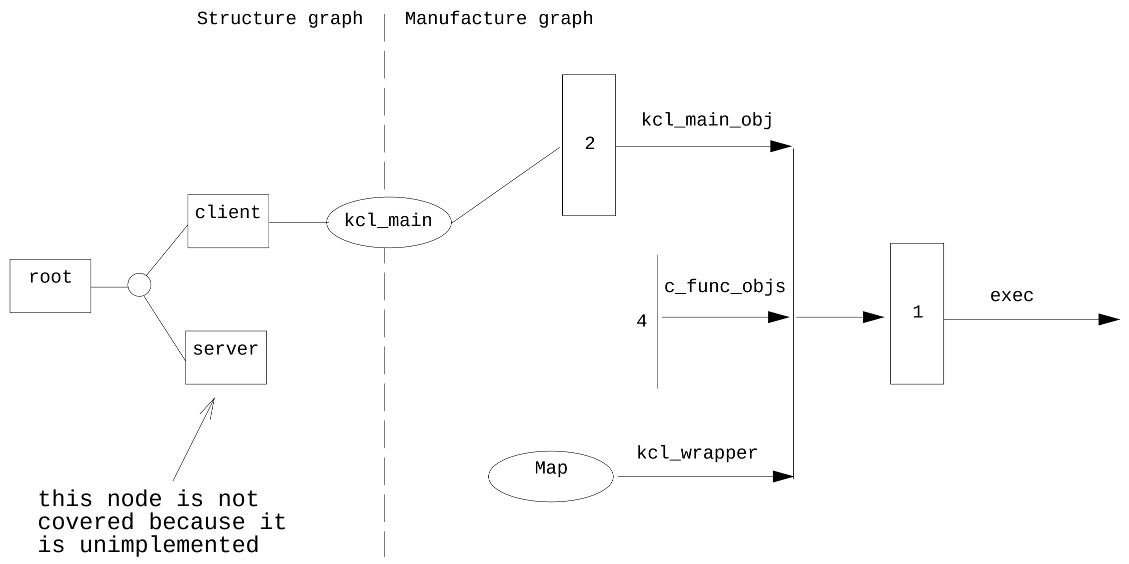

Figure 2.11: A incomplete production graph.

Production graphs do not explicitly describe the interconnections between components, but at the boundary between the structure and manufacture graph we can derive the application graph (section 2.4) that contains all the relevant direct connections. The bindings between interface ports of modules within the structure graph creates a mapping between the ports of primitive nodes at the leaves, i.e. the connections of the application graph. If the packager is able to create a valid manufacture graph, it also produces a database called a "Map" that contains the interconnections between all packaged components. A map is a cross-reference list that can be used by integration tools to construct bridges, like stubs, between components. The map in the factorial example consists of a single connection from the client's factorial port to the server's factorial port. This map can be used by a code generator to produce the `kcl_wrapper` object. Since stub and wrapper generators rely on the map, it may be included as an object in the production rules that is always available.

A map enables tools like stub generators to access information about and implement the interconnections between components much like a link editor backpatches object code in homogeneous applications. Our approach is extensible because future integration tools can read the maps to determine how to implement interconnections. In our environment, we have built several bridge tools that rely on maps to generate stub specifications from maps for several protocols including Sun RPC [Sun85b], Polylith [Purt85], and NIDL [Apol83].

2.4.3 Actions

The integration rules describe the abstract form of the legal manufacture graphs in an environment, but they do not perform the actual integration. With each rule, we associate actions that contain commands that invoke the proper integration tools. Once the packager determines a valid manufacture graph, the resulting graph is traversed and the actions associated with each node in the graph are executed. The traversal process is similar to the second-pass of a compiler traversing a parse tree built by the first pass.

In the rule specification language (described fully in Chapter 4), actions are contained within braces and may be interleaved with the objects on the right-hand side of a production rule. An action in the packager is similar to a semantic action in an attribute grammar specification language like YACC. For instance, the rule for producing a c_func_obj object from a c_func object is augmented with an action

```
c_func_obj        :  c_func
                  :{
                  !cc -c $1.FILE
                  }
                  ;
```

that invokes the C compiler on the value of the FILE attribute of the c_func object. Chapter 4 expands on the types of commands that can be used within actions. Actions that occur between items on the right-hand side of a production are executed *during* the packaging search process. The last action of a production rule, however, is special: it is only executed if the rule succeeds and is included in the software manufacture graph, i.e. during the traversal of the constructed manufacture graph. When the whole graph is built, the packager traverses the graph and executes these final actions.

2.5 Related Work

We have presented the basic concepts of software packaging that build on prior concepts including software manufacturing and module interconnection languages (MIL). The next sections discuss related work in software manufacturing, heterogeneous and distributed systems, and configuration languages in general.

2.5.1 MAKE

The most well-known tool that automates the software integration process based on software manufacture graphs is the UNIX MAKE program [Feld79]. Given a description of the dependencies between files in an application, MAKE invokes the tools needed to build an application. It determines the sequence in which tools are used to build an application. The software developer is responsible for specifying the dependencies between files and the steps needed to rebuild a file if one of its dependents is changed. These specification are stored in MAKEFILES. The partial order in a MAKEFILE specification is based on course-grain changes to files, i.e., if a file is updated, then dependent targets must be updated as well in order to maintain consistency. Other systems exist that base updates on more fine-grain changes, but the basic principle of such tools is to maintain an invariant condition on the program configuration.

The UNIX MAKE program allows designers to specify software manufacture graphs in order to automate the rebuilding of products should one or more of its components change. Dependencies in a MAKEFILE are specified as relationships between *targets* and *dependents*. Both targets and dependents correspond to files. Targets may be dependents of other targets. This establishes the partial order between components. For example, a MAKEFILE and its partial order are shown in

```
file1:  file2 file3
        command1

file2:  file4
        command2
```

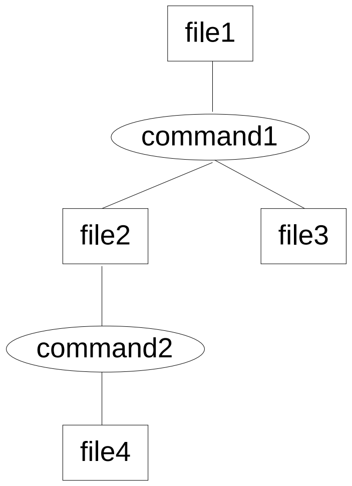

Figure 2.12: A MAKEFILE and its partial order

Figure 2.12. Each *command* in a MAKEFILE is a list of commands (one per line) for rebuilding the target from the dependents. If one or more of the dependents change (i.e., its file date and time is later than the target), then the commands are executed. If the target is a dependent of another target, then execution of commands continues according to the partial order specified by the entire MAKEFILE.

The MAKEFILE in Figure 2.13 specifies the integration steps for the Lisp and C components via wrappers. The MAKEFILE dependencies correspond to the manufacture graph in Figure 1.2. In this MAKEFILE, the lines

```
.c.o:
        cc -c $*.c
```

specify a *suffix rule* that specifies that any file named with the extension ".c" can be translated into a file with the same name with the extension ".o" by using the `cc` tool (the C compiler). Although we have included this rule in the MAKEFILE above, such rules are often implicitly defined by each execution environment. A suffix rule is abstract in the sense that it applies to all files of a particular type specified by a file extension, e.g., `.c`. Such rules are limited in their ability to express dependencies and the integration capabilities within an environment.

The MAKEFILE in Figure 2.13 assumes that the wrapper (`wrapper.lsp`) already exists. The developer must supply this wrapper that describes the interface of the server component. Such a wrapper could be generated automatically by a wrapper generator program based on interface specifications of the server component, but the programmer must supply this specification as well. Such interface specifications are not a part of the integration process in existing methods.

While this MAKEFILE works in one environment, it may not work in other environments. One of the major problems with MAKE is portability. MAKE was originally designed to maintain computer programs, but it has been extensively used to port and build programs across execution environments. Errors are commonplace when porting software manually via MAKEFILES. There are many

```
all:  factorial

.c.o:
        cc -c $*.c

factorial:  client.o wrapper.lsp server.o
        echo "(load \"client\")" > init.lsp
        echo "(si:faslink \"wrapper\" \"server.o\")" >> init.lsp
        echo "(si:init-hook '((compute-facs) (bye)))" >> init.lsp
        echo "(si:save-system \"factorial\")" >> init.lsp
        kcl
        rm -f init.lsp

client.o:  client.lsp
        lc client.lsp

server.o:  server.c
        cc -c server.c
```

Figure 2.13: Makefile for the Lisp and C factorial program.

differences between programming environments: compilers, IPC mechanisms, file paths, and installation options. Developers are forced to modify MAKEFILES directly because of such differences and include implementation alternatives based on platform-specific features. Macros alleviate some portability problems, but they are statically declared and globally scoped on the MAKEFILE and complex to use in large applications. Suffix rules also help because they are defined by the local environment, but such rules are limited to simple dependencies.

2.5.2 IMAKE

A better approach to portability is promoted by the IMAKE tool [McNu91]. The IMAKE utility is a tool that handles portability problems by leaving it to the execution environment to define integration procedures. An IMAKEFILE is a portable configuration "program" that invokes these procedures. IMAKE is implemented using the C preprocessor that expands the procedure calls into MAKE production rules. For instance, the following is an IMAKEFILE that integrates our Lisp and C components via a Lisp wrapper:

```
#include <local.imake>
lispcomponent(client)
ccomponent(server)
lispcprogram(factorial)
```

Each execution environment defines its own procedures, i.e., in the *local.imake* file shown in Figure 2.14. This file contains implementations for the procedures `lispcomponent`, `ccomponent`, and `lispcprogram`. Given these procedures and an IMAKEFILE, the IMAKE tool produces a MAKEFILE customized for the target environment. Although the implementation of IMAKE is crude (i.e., via the C preprocessor), it allows integrations to be specified independent of target execution environments. It has been used successfully in the distribution of the X window system [Wall87] across many execution environments. While MAKE dependency rules may be specified in any order,

```
#define lispcomponent(x)                                                    \\@
LISPNAME= x                                                                 \\@
LISPWRAP= x ## _wrap                                                        \\@
                                                                            \\@
x.o:  x.lsp                                                                 \\@
        lc x.lsp                                                            \\@
                                                                            \\@
$(LISPWRAP).o:  $(LISPWRAP).lsp                                             \\@
        lc $(LISPWRAP).lsp

#define ccomponent(x)                                                       \\@
COBJ = server.o                                                             \\@
x.o:  x.c                                                                   \\@
        cc -c x.c

#define lispcprogram(x)                                                     \\@
all:  x                                                                     \\@
                                                                            \\@
.c.o:                                                                       \\@
        cc -c $*.c                                                          \\@
                                                                            \\@
x:  $(LISPNAME).o $(LISPWRAP).o $(COBJ)                                     \\@
        echo "(load \"$(LISPNAME)\")" > init.lsp                            \\@
        echo "(si:faslink \"$(LISPWRAP)\" \"$(COBJ)\")" >> init.lsp         \\@
        echo "(si:init-hook '((compute-facs) (bye)))" >> init.lsp           \\@
        echo "(si:save-system \"x\")" >> init.lsp                           \\@
        kcl                                                                 \\@
        rm -f init.lsp
```

Figure 2.14: IMAKE procedure definitions in `local.imake` file.

IMAKE procedures must be invoked in some sequence because some procedures depend on macros or rules declared by other procedures. For example, the lispcprogram procedure above depends on the `ccomponent` procedure to construct the `$(COBJ)` macro.

File inclusion mechanisms within MAKEFILES and IMAKEFILES provide a means to specify platform-independent configurations. By assigning targets to predefined macros, a developer can include a common set of rules as defined by the system. In the case of MAKE, standard macros are assigned values that parameterize predefined rules. This approach, however, is limited because multiple rules based on lists of targets are not possible or limited. This is due to the fact that macro expansion is static and prevents the use of more complicated constructs such a list iterations and complex conditionals.

Like IMAKE, the software packager also relies on system-dependent integration "rules" but the major difference lies in the ability of the packager to infer integration steps rather than stating the procedures explicitly. IMAKE was developed for porting homogeneous software products between hardware platforms and does not easily handle heterogeneous integrations. It cannot infer the intergration steps based on the types of components, but relies on the developer to invoke the proper integration procedures.

2.5.3 Program Changes

A major problem with using tools like MAKE and IMAKE is handling program evolution. If a component is changed, then MAKE can rebuild the target product. On the other hand, if the interface of a component changes or a component is added or removed, then the dependency relationships between components may change. In many cases, this means that the MAKEFILE must also be altered. MAKE and IMAKE were designed to maintain *static* dependency structures. They do not handle updating dependencies themselves that result from changing gross software structures. High-level configuration decisions can drastically change the nature of an integration process. For example, if we decide to reimplement the client component in C, then we can rewrite the MAKEFILE as

```
factorial:  client.o server.o
        cc-o $@ client.o server.o

.c.o:
        cc -c $*.c
```

Similarly, new tools effect the types of possible integrations in an environment. If we introduce an Lisp-to-C translator, we can configure the application to take advantage of this new capability. A new interprocess communications facility may also impact integration processes. Such changes to configurations and environments occur frequently in many environments because applications and their supporting platforms evolve with the introduction of new technologies.

One of the major advantages of software packaging is the ability to accommodate software reconfigurations without having to respecify the integration process for a product. This is a significant gain over existing methods where reconfigurations require drastic changes to integrations. In Chapter 5, we compare integrations using software packaging to integrations using UNIX MAKE to demonstrate this advantage. We will use software packaging to produce MAKEFILE specifications automatically and show how small reconfiguration changes produce large changes to MAKEFILE specifications.

2.5.4 Other Manufacture Graph-Based Tools

Other tools that employ dependency-based methods for building software applications include NMAKE [Fowl85], GNUMAKE [Smit91], and SHAPE [MaLa89]. Some provide more sophisticated manipulation of macros and all employ file inclusion mechanisms or platform-dependent rules. All of these tools, however, require the explicit use of integration procedures that depend on the configuration characteristics of an application. The developer must specify the software manufacture graph explicitly in order to integrate an application. Mechanisms like suffix rules and IMAKE procedures automate the building process, but there is no relationship between rules and procedures in these systems.

We have seen that changes to the structure of an application or differences between execution environments can determine the integration process for an application. This means that the manufacture graph for an application can change from environment to environment. It can also change during program development. Our approach relies on the use of an inference engine to derive manufacture graphs automatically. Like IMAKE, the software packager relies on each environment to specify its available integration processes. Our approach, however, differs because the rules describe the abstract integration processes not just the disjoint integration procedures available in an execution environment.

Suffix rules in MAKE, for example, are a simple forms of this abstraction. For example, the suffix rule

```
.c.o:
        cc -c $*.c
```

states that any file written in the C programming language can be compiled into an object file. The extension "`.c`" is a convention that identifies the file to be a certain *type* of component: a source file written in C. We have extended this notion to include complex relationships between types of components and the integration processes in execution environments.

2.5.5 Module Interconnection vs. Object-Oriented

The PACKAGE specification language is a module interconnection language with some unique features, namely, the ability to depict choices of implementations. Many configuration management tools present similar organizations of software structures using hierarchical file systems with enhancements for handling alternatives. For example, NMAKE [Fowl85] depends on a standard directory structure for organizing product implementation alternatives. INTERCOL [Tich80] presents a similar structure with implementation choices within the configuration language. Our approach is similar, but the choice of implementation for a component is not specified explicitly in the structure, rather it is left to the packager.

Many programming systems support the separation of interface and implementation to reduce coupling within programs. This separation allows designers to concentrate on distinct subparts of a problem. As long as the interface of a component remains fixed, its associated implementation is irrelevant to another developer using the interface. This allows programmers to work independently of one another and isolate changes to implementations. The separation greatly reduces coupling within programs. Coupling increases the likelihood that small changes will propagate extensively within a program. This increases the chance for errors and inconsistencies if done manually.

Object-oriented programming promotes the separation of interface and implementation, but it presents a single-implementation model. An interface specification for an object lists methods defined by the object. An interface is relatively independent of its implementation. Exceptions are usually for pragmatic reasons like performance (e.g., member function implementations and private variables defined within C++ class definitions). Most object-oriented systems are homogeneous; all components are implemented in the same language and executables are designed to execute within a single address space. Furthermore, there are no "choices" because each interface has a single associated implementation. This is sufficient in a homogeneous environment, but lacks extensibility to heterogeneous configurations.

Another major difference between object-oriented and module-oriented programming involves the use of indirection. Traditionally, an interface defines a set of resources (e.g., methods) defined by an object. References to other objects are embedded within object implementations. This is sufficient because all objects have similar implementations (i.e., programming language, runtime support). A module interface, however, describes the resources defined *and used* by a module. References to other modules are always indirect [Wegn90]. This strictly encapsulates a software component. No implicit form of coupling is possible because all interactions are explicitly specified through the module interface. A module describes a software component as a self-contained entity [Tich80]. Module-oriented programming subsumes object-oriented programming because the correspondence between a module interface and its implementations is one-to-many. Module-oriented programming allows developers to explicitly address interconnections between providers and users of resources. This additional level of indirection permits the rebinding of clients to services that were never intended to be used together thus enhancing software reuse.

2.5.6 Remote Procedure Call

Several projects have attempted to solve the problem of integrating heterogeneous, distributed applications through the use of remote procedure call (RPC). This technology is important to solve the mechanics of the integration problem, but it does not solve the larger problem of simplifying the integration process. Indeed, the software packager relies on stub generator tools to bridge applications.

The HORUS system [Gibb87] helps generate RPC stubs for many programming languages and comunication protocols without reimplementing the stub generator in each execution environment. HORUS consists of a driver program that employs two schema files to achieve system independence: a machine-dependent schema and a language schema. The stub specification is given as input and HORUS produces stubs based on the features of the target machine and programming language. HORUS is a generic stub generator but the developer is responsible for using it and writing the stub specifications.

The HRPC project [NoBL88, Notk90] takes a similar approach to stub generation as HORUS, but also employs runtime mechanisms to resolve differences between RPC protocols. Applications in a distributed system may employ different RPC protocols (i.e., Sun, NIDL, Courier). HRPC applications may connect to any of these services by dynamically determining which protocol to use at runtime. The stubs are not statically generated, but dynamically configured. This has performance implications, but once connections are established, the HRPC system is fixed until some change in the service occurs. Once again, the developer must write the stub specifications in the HRPC language, invoke stub generation tools, and link the HRPC library into the application.

Another project related to RPC is the Common Object Request Broker Architecture (CORBA) [OMG90] under the direction of the Object Management Group (OMG) — a consortium of vendors trying to standardize software components and the design of bridges between them to enable easier integration. CORBA provides for more complex interactions between programs than procedure calls, but like HORUS it defines an Interface Description Language (IDL) that is language and system independent. In general, the runtime design of CORBA is closely related to the notion of a *software bus* as discussed in the next section.

In general, RPC tools are necessary to bridge heterogeneous, distributed applications, but they do not make programming such applications easier. Developers must determine what tools to use. If the configuration of an application changes, the intrgation may change drastically. The developer must reintegrate the application by applying different tools. The software packager eliminates this step by determining the integration process automatically based on the types of components. It may employ RPC mechanisms as described in this section. If a configuration changes, the developer simply repackages the application.

2.5.7 The Software Bus

A improvement on remote procedure call involves adding a level of indirection between components in a heterogeneous, distributed system. Instead of components being directly connected to one another as client and server, a third-party process routes messages from one process to another. If a process produces a message on a port, the router directs the message to receiver(s) according to some mapping that may be statically or dynamically specified. This module-based approach is more flexible because participants in the system may come and go. For instance, in the middle of a session, the server process may be replaced with no affect on other processes. The router might queue impending messages to the server while a new server enrolls in the overall configuration.

This model of integration, known as the *software bus* model, views software components as pluggable modules into a communications backplane. It provides a great deal of flexibility in distributed, heterogeneous environments by adding a level of indirection to transactions between runtime components. The bus routes messages between participants and queues undelivered messages for future delivery. This approach can implement remote procedure call as well as asynchronous interactions styles.

If there exists a runtime environment in which all the primitive components can operate, then we can build and execute the application. If we view each connection as a message channel, then the common runtime environment would include a communication mechanism to realize these channels. The task of the software packager is to choose compatible implementations of modules and derive a viable runtime environment that supports the execution of all constituent modules and their interconnections. The software packager determines whether or not an appropriate software bus exists based on the types of components and the integration tools in the target environment. In the process of integrating the components, they may need to be adapted and additional components such as wrappers and stubs may be introduced.

Several projects including the Portable Common Toolkit Environment (PCTE) [Vera89], Polylith [Purt85], PVM [Begu90], CONIC [MaKS89], CORBA [OMG90], and the Portable Common Runtime (PCR) environment [Weis90] are based on the software bus approach to software integration as a means of encapsulating software and promoting reuse.

2.6 Summary

In many environments, it is difficult to integrate heterogeneous, distributed software not because we lack the technology to do so but because the integration process is complex. An application may be a patchwork of connections between different systems. If one component changes (i.e., is reimplemented or moved to another hardware platform), this has profound impact on the runtime organization of the entire system.

The software packager is independent of the particular technology used to integrate applications. It does, however, coordinate the use of these technologies and associated tools. The packager determines which tools are necessary based on a description of their integration characteristics. In the next two chapters, we explore the PACKAGE specification language and the rule specification language details for expressing software designs and production rules.

Chapter 3

Package Specifications

This chapter is designed to be reference manual that describes the syntactic units for the PACKAGE specification language. The PACKAGE language is used to describe software structure graphs for applications. The PACKAGE language is a module interconnection language (MIL) in which software components are described as units (either processes or static code) that provide and use resources. An application contains instantiations of modules and connections between resources uses and definitions.

3.1 Overview

A PACKAGE specification describes the software structure graph for an application. A specification describes a directed, rooted graph (possibly cyclic) whose root node represents the entire application. Each PACKAGE specification *must* describe at least one implementation for the application either as of a collection of components (a composite implementation) or a single object (a primitive implementation). The name Root is a distinguished lexical identifier within a PACKAGE specification. At least one implementation for the Root must be expressed in a PACKAGE specification. Each direct descendent of the root node represents an alternative implementation of the application itself. The structure graph is elaborated by describing the subcomponents, their implementations, and connections within an application.

Figure 3.1 is a PACKAGE specification that describes an application consisting of a client program and a server program. The corresponding software structure graph is shown in Figure 3.2. The specification is comprised of six syntactic units declared at a global level: a composite implementation for the Root module, an interface description for a Client module, an interface description for a Server module, a single primitive implementation for the Client module and two primitive implementations for the Server module. PACKAGE specifications typically consist of a series of declarations of modules and their implementations.

Modules may have multiple associated implementations that are either composite or primitive. Unlike INTERCOL [Tich80] and the previous version of the software packager [CaPu91], instances of modules within composite implementations do not need to specify which implementation should be used. The choice is determined by the packager tool. The PACKAGE specification enumerates *all* possible implementations of modules within an application as a subgraph of a module instance. Tools like NMAKE rely on the UNIX file system to specify choices of implementations in a similar fashion, but a PACKAGE specification explicitly describes this structure without attaching choices to particular file systems.

```
include stdpkg.pkg

implement root as {
               Client:  c;
               Server:  s;
               bind c'factorial to s'factorial;
}
module Client {
               use factorial(int)(int);
}
module Server {
               def factorial(int)(int);
}
implement Client with kcl_main {
               FILE          =client.kcl
}
implement Server with c_func {
               FILE          =server.c
}
implement Server with rpc_svc {
               PROGNUM       =407888
               LOCATION      =thumper.cs.umd.edu
}
```

Figure 3.1: PACKAGE specification for a client and server application.

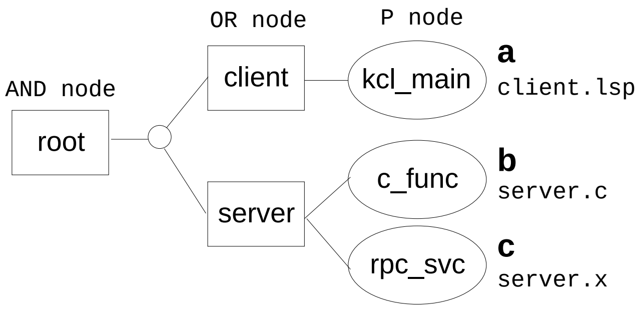

Figure 3.2: Software structure graph for a client-server example.

3.2 Modules

The `Client` and `Server` module interfaces are declared after the `Root` composite implementation in Figure 3.1. The packager uses a two-pass approach to build the software structure graph so that modules can be instantiated before their declaration. The `Client` module interface consists of a single `use` port representing a function call to an external resource named `factorial`. The `Server` module interface consists of a single `def` port providing a function resource named `factorial`.

While a module may have multiple implementations, it may only have one interface description. Modules are uniquely identified by their name and parameter types. A module declaration is of the form

```
module identifier1 ( parameters ) :  ancestors {
            ports
            attributes
}
```

where *parameters* is a list of variable names and *ancestors* is a comma-separated list of module names. Module parameters may be referenced within the body of the module specification. They expand to string or numeric values in the same manner as attribute references. Module ancestors refer to modules from which a module inherits attributes and ports in a fashion similar to inheritance in object-oriented languages except this style of inheritance is involves only the interface not the implementations of the ancestor modules.

Ports within a module are distinguish by their names and parameter types. Port names may be overloading as in C++ with different parameter types. Ports and attributes are explained in a later section. If a module has no ports or attributes, it may be declared as

```
module identifier1 ( parameters ) ;
```

The `Root` is the only module not specified in a package. It has the implicit declaration

```
module Root;
```

that is predeclared in the standard package header which is included via the `include` directive. The `Root` module may have default ports and attributes, but these are usually specific to an execution environment.

3.3 Ports

Ports are associated with modules by instantiating them within a module declaration. In our example, the Client has a single "`use`" port and the Server has a single "`def`" port. A `use` port represents a function call that expects a single value in return. A `def` port represents a function implementation. There may be many types of ports including sources (`src`), sinks (`snk`), and errors (`err`). A port instance within a module declaration is of the form

```
( min, max ) porttype identifier1 ( path1 ) ... ( pathn )
            :  attributes
;
```

where *porttype* is a `use`, `def`, `src`, `snk`, `err` or user-defined port type. *min* and *max* are the minimum and maximum number of connectors that may be attached to this port. The minimum and maximum values are used to place constraints on connections to and from a port. For example, a `use` port can only be connected once since multiple connections would imply a broadcast procedure call. Ports of type `def`, however, may have an unlimited number of incoming connections corresponding to procedure invocations. The $path_i$ specifications of a port declaration describe the types of messages on the resource. A path is a sequence of message types that include primitive types (integers, strings, floats), other module names (references), or static data structures known as classes. Classes are described in a later section of this Chapter.

Port types are declared at the global level. A port type is a pattern specifying a legal port type. Default port types are defined in the standard package header (`stdpkg.pkg`). In general a port type declaration is of the form

```
port identifier ( argument_type ) ... ( argument_type ) {
          attributes
}
```

For example, the `use` and `def` port types are declared in the standard package header as

```
port use(...)(?);
port def(...)(?);
```

where the ellipses imply that both port types take any number and types of arguments. The question mark means that both return only one value of any type as a result. Both ports associated with the `Client` and `Server` modules in our example are legal instances of their port types.

3.4 Composite Implementations

Modules may have multiple associated implementations. There are two types of implementations: composites and primitives. A composite implementation describes a collection of module instances and their connections. The implementation of `Root` in our example is a composite implementation. A composite implementation is a circuit-board diagram of connected modules: it describes the subcomponents and the "wiring" between their ports that comprise an implementation for a module. It represents a subsystem because the instances within a composite also have associated implementations, but these are not visible at this level of abstraction. Each instance is simply a black box. In general, a composite implementation is of the form

```
implement identifier ( formals ) as {
          instances
          connections
}
```

where an *instance* is a declaration of a module instance and a *connection* is a link between a single port or groups of ports. The connections within a composite implementation describe how to wire the ports of the module instances together. The designer is not constrained to wire ports one-by-one. There are constructs for performing connections by pattern and port type.

Instances within composite implementations are specified with or without an instance name and may have attached attributes. The form of an instance description is

```
modulename :   identifier2 dimensions, ..., identifiern dimensions
               attributes
;
```

where *modulename* is the name of a module and *identifier*$_i$ is the name of the instance. All module names must be defined, even if a module has a null interface (e.g., Root). Arrays of modules may be declared with multiple dimensions. Attributes can be associated with individual instances. These attribute assignments are scoped on the subgraph below the instance, not the entire composite subgraph.

In our example specification, the `Root` implementation is the only composite implementation. It contains two instances: one instance of the `Client` module and another instance of the `Server` module. Within the `Root` implementation, the instances of the `Client` and `Server` modules are assigned the names `c` and `s` respectively. The `bind` statement

```
bind c'factorial s'factorial;
```

specifies that the `factorial` port of the `Client` instance `c` is connected to the `factorial` port of the `Server` instance `s`. Composite implementations for modules that have ports (unlike Root in this case) may use the `alias` directive to connect ports of internal instances to the external ports. For example, the specification

```
implement Stack as {
          ArrayStack astack;
          Array a;
          alias 'top to astack'top;
          alias 'push to astack'push;
          alias 'pop to astack'pop;
          bind astack'get to a'get;
          bind astack'put to a'put;
}
```

describes a stack module composite implementation comprised of an ArrayStack instance and an array instance. The alias directives connect inner ports to outer ports within a composite. The specification

```
implement Stack as {
          ArrayStack astack;
          Array a;
          alias '* to *'$1;
          bind *'* to *'$2;
}
```

is equivalent to the previous specification but employs a shorthand notation for connecting groups of ports instead of individually. The connecting phrase

```
bind *'* to *'$2;
```

is a "cliche" for binding ports by name. This phrase is equivalent to name-binding that is employed by many link editors. Connections within packaging specifications are checked to ensure that the connection constraints on ports are within their limits.

Composite implementations of modules represent subclass implementations of embedded module instance. For example, a stack of integers that can be queried for its height could be described by the interface

```
module cstack {
        def top()(int);
        def push(int)();
        def pop()();
        def height()(int);
}
```

and implemented as

```
implement cstack as {
        Stack s;
        Counter c;
        alias '* to *'$1;

        alias 'push to c'increment;
        alias 'pop to c'decrement;
        alias 'height to c'current;
}
```

where the embedded stack is augmented with a counter to form a new subclass implementation of a stack called a `cstack` (counting stack). The module `cstack` is also a subtype of stack because the `cstack` interface ports are a superset of the stack ports. Unlike object-oriented languages like C++ [Stro86], the subtyping and subclassing in PACKAGE specifications are separate. While this has some disadvantages, such as performance, it totally encapsulates software modules and their implementations to promote their reuse in many contexts. The CONIC system [MaKS89] takes a similar approach to separate subtyping and subclassing but does not employ multiple implementations. The RESOLVE programming language [HaWe91] employs the same separation and multiple implementations, but implementations are all coded in RESOLVE and distinguished by performance and space characteristics.

3.5 Primitive Implementations

Modules may also have associated primitive implementations. Primitive implementations are different from composite implementations because they refer to native objects that implement a module, not a subsystem of module instances and bindings. In our example, the `Client` module is implemented by a `kcl_main` object and the `Server` is implemented by `c_func` and `rpc_svc` objects. The specification of a primitive implementation has the form

`implement` *identifier*$_1$ `[ (` *parameters* `) ] with` *identifier*$_2$ `{`
    attributes
`}`

where *identifier*$_1$ is a module name and *identifier*$_2$ is an implementation object type. The attributes assign string or numeric values to named variables. Object types are declared in the standard header as

`object` *identifier* `:` *ancestors* `{`
    attributes
`}`

where the attributes set default values within primitive implementations of the object type. Object types may also have ancestors that define additional attributes. Object type attributes describe component parameters such as source file names, tools, data files, etc. Details on object types and their attributes are described in a later section.

3.6 Arrays

Aggregate instances of modules can be declared within composite implementations. The semantics for each element a module instance array is equivalent to those for instances that are individually declared. The name of an instance in an array includes its index. For example, we can create multiple instances of the Client module within the Root implementation

```
implement Root as {
            Client c[2];
            Server s;
            bind c[*]'factorial to s'factorial;
}
```

This creates an array of two Client module instances accessed as c[0] and c[1]. The c[*] in the bind connector specifies a wildcard match on all elements of the array of Client instances. This bind operation is equivalent to

```
bind c[0]'factorial to s'factorial;
bind c[1]'factorial to s'factorial;
```

Modules can declare aggregate ports as well. For example, the Server module could be declared as

```
module Server {
            def factorial[2](integer)(integer);
}
```

Within the Root composite, connections can be made from two different Client instances to the two different ports on a single server instance

```
implement Root as {
            Client c[2];
            Server s;
            bind c[*]'factorial to s'factorial[$1];
}
```

where \$i matches the ith wildcard in an operation. Thus, this is equivalent to the explicit specification

```
implement Root as {
            Client c[2];
            Server s;
            bind c[0]'factorial to s'factorial[0];
            bind c[1]'factorial to s'factorial[1];
}
```

Wildcards are used extensively within packaging specifications. They are convenient for specifying bindings by names as well as more complex binding relationships.

3.7 Connectors

The `bind-to` operation is not a primitive in the PACKAGE language, rather it is an instance of a *connector* type. A connector instantiates an object similar to a module instance. The `bind-to` connector is declared in the standard header as

```
connector bind(use)to(def);
```

Connectors may be primitive or composite. The bind-to connector is an example of a primitive connector.

```
connector identifier1 ( name1 ) identifier2 ( name2 ) {
            instances
            connections
}
```

where the *name* items are valid port types. A composite connector is similar to a composite implementation and can contain embedded module instances and connections. The instances and connectors are expanded inline into the composite implementation calling the connector. This means that connectors may only have single implementations and do not form subgraphs in the structure graph.

3.8 Attributes

All syntactic units in PACKAGE specifications, including modules, ports, implementations, and connectors, may have associated attributes. An attribute may be assigned a value that is a string or numeric value, a set or list. For example, the `FILE` attribute of the Client primitive implementation in the client-server specification has the string value "`client.kcl`" that specifies the file name of the component. The `FILE` attribute illustrates the fact that object types need not be associate one-to-one with files, but may have attributes that reference multiple files.

String attributes are assigned using the = operator while numeric attributes are declared using the := operator. An attribute may also be a list (i.e., sequence) or set of values. For example, the object type `driver` is described as

```
object driver {
            WEIGHTS [
                        :=4
                        :=5
            ]
            ARCHS (
                        =sun
                        =mips
            )
}
```

includes a list attribute, `WEIGHTS`, that contains two numeric values and a set attribute, `ARCHS`, that contains two string values. List and set attributes can be nested. Nested attributes may be unnamed or named. For example, a list with named attributes

```
EMPLOYEE [
                NAME          = Mary Smith
                PHONE         := 5436
]
```

emulates a record data type. Attributes within sets must be unnamed or have unique names. String and numeric attributes must be specified on separate lines with a "\" used to specify a string across multiple lines.

Attributes are dereferenced using the $(*name*) construct similar to that used in tools like MAKE. Unlike MAKE, however, whose attributes are visible at a single lexical level, package attributes are scoped on the software structure graph not on the lexical structure of the PACKAGE specification. For example, in the specification

```
implement Root as {
              DIR          = /src
              Client:  c;
              Server:  s;
              bind *'* to *'$2;
}

implement Client with c_main {
              FILE         = $(DIR)/client.c
}
```

the value of the FILE attribute in the context of the Root implementation is /src/client.c. Leaf nodes of a structure graph may be shared if their module names and attributes are identical. For example, the specification

```
implement Root as {
             Client:  c1;
             Client:  c2;
             ...
}

implement Client with c_main {
             FILE     = client.c
}
```

yields the structure graph shown in Figure 3.3. Most object types are subtypes of the object type basic specified as

```
object basic {
                NUM          = $@
}
```

where $@ is an attribute that assigns a unique number (NUM) to each instance of a primitive implementation. The $@ attribute reference yields this unique number and makes the corresponding node in the structure graph distinct from other nodes. Other special attribute references include $@ for the unique node number, $# for the module instance name, and $* for the index if the instance is part of an instance array.

Attribute values can be reassigned or changed. By prefixing the assignment of an attribute with a "+" or "-" the value of an attribute can be altered or queried according to Table 3.1. The right-hand side of an attribute assigned with the := operator is an expression that can evaluate to a numeric result.

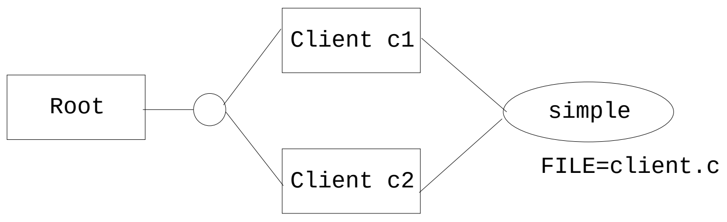

Figure 3.3: Software structure graph with shared implementation.

+(...)	set union
+[...]	list append
+:=	numeric addition
+=	string concatenation
-()	set difference
-[]	list difference
-:=	numeric subtraction
-=	substring elimination
@	member of (lists and sets)

Table 3.1: Attribute assignment operators

3.9 Constraints

Syntactic units within PACKAGE specifications may also contain constraints. Constraints resemble attributes: they are scoped on the structure graph. If an attribute assignment violates a constraint at some higher level in the structure graph, that unit is not expanded. For example, in the PACKAGE specification

```
implement Root as {
                MACHINE        == sun
                Client:  c;
                Server:  s;
                bind *'* to *'$2;
}

implement Client with c_main {
                MACHINE        = sun
                FILE           = client_sun.c
}

implement Client with c_main {
                MACHINE        = mips
                FILE           = client_mips.c
}
```

the second primitive `Client` implementation is not expanded as a candidate implementation of the `Client` instance in the Root composite because it violates the higher constraint in the structure graph. The form of a constraint is *label* : : *identifier relop expression*

where the *label* is optional, *identifier* is an attribute name and the *expression* or yields a string, numeric, set, or list value. The following relational operators may be used to express constraints

`==`	equal
`!=`	not equal
`<=`	less than or equal/subset of
`>=`	greater than or equal/superset of
`>`	less than/proper superset of
`<`	greater than/proper subset of
`<<`	member of
`!<`	not member of
`>>`	contains
`!>`	does not contain

and expressions are the same as those for attributes. Constraints may be labeled or unlabeled. Only labeled constraints can be removed. To remove a constraint, the `-NAME` construct removes the last constraint labeled `NAME`.

3.10 Classes

Ports describe the resources used and defined by a module. Many ports are the descriptions of functions that take arguments and return results. These port descriptions are known as paths that define the types of these arguments. A path may include the names of primitive data types such as integers, strings and floats, or the names of modules, or static data structures. The static data structures may be defined by *class* descriptions. A class description is of two forms

```
class identifier1 = identifier2 dimensions ;

class identifier1 {
              signatures
}
```

where *identifier*$_1$ is the class name, *identifier*$_2$ is an alias, and a *signature* is an embedded field declaration. The module Position in the following PACKAGE specification passes two integers via a structure described by a point class on its `putpoint` port

```
class point {
              int x;
              int y;
}

module Position {
              use putport(point)();
}
```

Classes can be embedded by using the class name in a signature. For instance, a points class may be defined as a record of two points

```
class points {
            point one;
            point &two;
}
```

In this case, the second point is a *reference* to a point instead of an actual point record. The packager will determine how to reconcile references either through the use of pointers within the same address space or more elaborate means like shared memory during the packaging process. Likewise, module names can also be used within path descriptions instead of classes. For instance, if there is a Point module, the Position module could pass a Point module instance

```
module Point {
            def x()(int);
            def y()(int);
}

module Position {
            use putport(Point)();
}
```

on its putpoint port. The packager will determine how to transfer the state of the Point instance within a constructed runtime environment either through memory pointers if the application is packaged in the same address space or more elaborate means if the connection between a Position instance and another module instance is between addresses spaces as in a distributed application. For example, if an implementation of the Point is able to transmit its value over a persistent media [HeLi82] then this implementation will be chosen in the appropriate context. If no implementation is capable of persistent representation, then this limits the types of possible integrations.

Names in paths that do not correspond to class or module definitions are deferred to the packager. These types are ultimately handled by integration tools themselves. For instance, the C++ link editor handles type casting between connected ports of C++ module implementations. Similarly, in the absence of type definitions on items in path expressions in module ports, the packager passes the job of handling type checking to the integration tools.

3.11 Summary

The PACKAGE specification language is used to express the form of software structure graphs of software applications. Such graphs represent the implementation alternatives and modular structure of the application. Attributes are visible on the graph and can be used as parameters for lower-level components in the graph. Likewise, constraints can restrict the selection of components included as subgraphs of module instances within embedded composite implementations. Other systems use file system structures to organize software applications and their implementations, but our specification language explicitly states and manipulates this structure.

Chapter 4

Rule Specifications

This chapter is designed to be reference manual that describes the rule specification language. The rule language is used to describe the abstract integration processes in an execution environment. The rules represent the abstract form of legal manufacture graphs in an execution environment. The rule language is based on an approach similar to attribute grammars that employs both synthesized and inherited attributes. It is also similar to a production system that uses backtracking search to unify components. The packager starts at the target rule for the desired object type (e.g., executable) and proceeds down the production rules searching for object types that match the leaf nodes of the structure graph. If an object type is found or a rule matches, the packager includes a node in a concrete manufacture graph corresponding to that rule in the same manner a multi-pass parser constructs nodes in a parse tree.

Once the manufacture graph is constructed, it is traversed by executing the actions associated with each node. Actions are sequences of commands that perform the actual integration steps. For example, in our implementation, our rule actions produce MAKEFILE specifications.

4.1 Overview

Rule are specified by system administrators and shared by all developers in an environment. Individual programmers need not specify any integration rules, just the package specifications. The rules describe the available integration processes in terms of how tools like compilers, linkers, and stub generators are used to build applications from software components. The production rules form a grammar that defines the abstract form of software manufacture graphs in an environment. For example, the rules

```
executable      <= main_function functions
functions       <= function functions
functions       <=
```

describe a simple set of productions for programs comprised of a main function (i.e., a program entry point) and zero or more functions that are called from the main function or each other. The third rule has an empty right-hand side that acts as a closure on the right-recursive second rule. The main_function and function object types are terminal because there are no rules for them. If we associate files with each terminal item, the graph in Figure 4.1 is a concrete manufacture that could be produced from this set of rules.

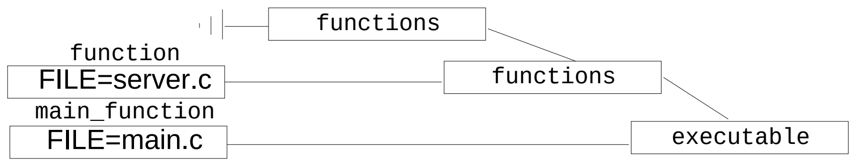

Figure 4.1: A concrete manufacture graph.

```
executable        <= main_function functions
                  :{
                        ARCH         = sparc
                        $2
                        !cc -c -target $(ARCH) $1.FILE
                        !cc -target $(ARCH) -o a.out $1.FILE:r.o $2(OBJECTS)
                  }
                  ;

functions         <= function functions
                  [ (OBJECTS) $1.FILE:r ``.o '' $2(OBJECTS) ]
                  :{
                        !cc -c -target $(ARCH) $1.FILE
                        $2
                  }
                  |
                  ;
```

Figure 4.2: Sample rule productions with actions.

Each node in the resulting manufacture graph corresponds to an integration step in a partial order that integrates the application. Each production rule contains items and actions. Actions associate integration steps with each production rule. For example, we can attach actions to rules as in Figure 4.2[1]. The `:{ ... }` construct is an action that specifies a sequence of commands. The different types of commands are described in later sections of this Chapter. Commands define inherited attributes like `ARCH` and synthesized attributes like `FILE` and `OBJECTS`. Attributes within manufacture graphs are similar to those in attribute grammars, but some inherited attributes are implemented using constructs known as *translations*. Translations are denoted in square brackets (i.e., "[...]") attached to production rules after the items but before the last action. Attributes like `FILE` on terminal items are accessed directly from the properties of associated leaf nodes of the structure graph.

Given the leaf nodes of the structure graph, the packager constructs a manufacture graph and traverses the resulting graph by executing the actions associated with each node as specified by the corresponding rule. In this case, the commands

[1]Attribute references may be suffixed by a ":" followed by a single character (e.g., "r") that computes a string function. In this case, the ":r" operator returns the root of the file name without its extension (e.g., main.c:r $\rightarrow$ main)

```
cc -c -target sparc server.c
cc -c -target sparc main.c
cc -target sparc -o a.out main.o server.o
```

are executed as a result of traversing the graph in Figure 4.1. Actions that are between items on the right-hand side of a production rules are executed *during* the packaging process, i.e., during the search. Actions may test attributes of objects, perform explicit "cutoffs" of searches, and set attribute values. The last action of any production rule is special. If the packaging process is successful and a rule is included in the derived manufacture graph, then the last action of each production is executed while traversing the resulting abstract graph from the root.

Rules are specific to each execution environment, but they are shared by all developers within the environment. Specifying rules is not an easy task, but because because they are shared by all developers this increases the value of the rules. Previously, individual developers within an environment had to trade MAKEFILE "cliches" that allow them to perform integrations or reinvent such processes for new applications. The shared production rules eliminate the need to share such integration programs informally. Developers may leverage integration tools without knowing the details of how the integration is accomplished. Again, this is similar to the level of transparency provided by link editing tools within homogeneous systems.

4.2 Rules

The packager production rules are similar to rules within attribute grammars in that they employ synthesized and inherited attributes and have associated actions. A packager rule is of the form

```
lhs            :  rhs
               :{
                      commands
               }
               ;
```

where *lhs* is a single object type name and *rhs* is a sequence of identifiers and actions. Name of items that do not appear on the left-hand sides of any production rules are terminal object types. It is possible that the leaf nodes of structure graph could contain object types that correspond to rules. The packager first searches the pool for an object type and then any rules associated with that name. For example, a library object type can be composite implementation or primitive object type.

4.3 Actions

Actions that are embedded within the right-hand side of a production are executed during the packaging search process. The last action is prefixed by a ":" and if it exists it is executed only if the rule is included as a node in the derived manufacture graph. An action specifies a set of *commands* that are executed in sequence. Commands may be used to print out information, redirect output, execute shell commands, set attributes, set constraints, and execute actions attached to subtrees of the manufacture graph. The next sections explain each command type: attributes, outputs, calls, subactions, and cutoff.

4.3.1 Attributes

Like structure graphs, rules may assign attributes to nodes of the resulting manufacture graph. Attributes are scoped on the manufacture graph rather than lexically on the rule specifications. An attribute may be assigned a value that is a string or numeric value, a set or list. For example, the ARCH attribute of the executive rule above has the string value "sparc" that specifies the machine architecture of the target execution component.

String attributes are assigned using the = operator while numeric attributes are declared using the := operator. An attribute may also be a list (i.e., sequence) or set of values. For example, the object type `driver` is described as

```
object driver {
          WEIGHTS [
                      :=4
                      :=5
          ]
          ARCHS (
                      =sun
                      =mips
          )
}
```

includes a list attribute, WEIGHTS, that contains two numeric values and a set attribute, ARCHS, that contains two string values. List and set attributes can be nested. Nested attributes may be unnamed or named. For example, a list with named attributes

```
EMPLOYEE [
               NAME          = Mary Smith
               PHONE         := 5436
]
```

emulates a record data type. Attributes within sets must be unnamed or have unique names. String and numeric attributes must be specified on separate lines with a "\" used to specify a string across multiple lines.

Attributes are dereferenced using the $(*name*) construct similar to that used in tools like MAKE. Unlike MAKE, however, whose attributes are scoped at a single lexical level, rule attributes are scoped on the software manufacture graph not on the lexical structure of the rule specifications. Attribute values can be reassigned or changed. By prefixing the assignment of an attribute with a "+" or "-" the value of an attribute can be altered according to Table 4.1. The right-hand side of an attribute assigned with the := operator is an expression that can evaluate to a numeric result.

4.3.2 Output

A command in an action prefixed with a # character directs the line to the current output stream. If a line is prefixed with "?(*name*)", then the line is output to the error stream if the attribute *name* is defined. The command "`> filename`" redirects ouput to a file. The command "`>> filename`" redirects and appends output to a file.

4.3.3 Calls

A command line prefixed with a "`!`" invokes a system call to the UNIX shell. The command line may contain references to synthesized and inherited attributes.

+(...)	set union
+[...]	list append
+:=	numeric addition
+=	string concatenation
-()	set difference
-[]	list difference
-:=	numeric subtraction
-=	substring elimination
@	member of (lists and sets)

Table 4.1: Attribute assignment operators

4.3.4 Subactions

After the manufacture graph is constructed, the traversal and execution begins at the root, but is directed further down the manufacture graph by the actions. The command

```
$2
```

in the action

```
:{
                $2
                !cc -c $1.FILE
                !cc -o a.out $1.FILE:r.o $2(OBJECTS)
}
;
```

executes the actions associated with the nodes constructed by the functions productions in the manufacture graph. After the traversal of the subgraph is complete, the two system calls are executed.

Actions may also be named. The default action has no name, but named actions are invoked as `$i{name}`. For instance, the `$2{sparc}` command in the first rule will execute the action labeled `(sparc)` in the `functions` ← `function functions` rule show here

```
executive        <= main_function functions
                 :{
                        $2{sparc}
                        !cc -c $1.FILE
                        !cc -o a.out $1.FILE:r.o $2(OBJECTS)
                 }
                 ;

functions        <= function functions
                 :{ (sparc)
                        $2{sparc}
                        !cc -target sparc -c $1.FILE
                 }
                 { (mips)
                        $2{mips}
                        !cc -target mips -c $1.FILE
                 }
                 |
                 ;
```

will execute the action labeled (sparc) associated with the functions rule. If the subaction specifier is simply $i, then the unnamed action corresponding to the ith subgraph of the current node is executed. This allows multiple actions to be associated with a single rule to permit selective traversals of the derived manufacture graph.

4.3.5 Cutoff

The command line containing a single * specifies that the search should stop (i.e., fail) at this point. The cutoff operator can only be used in embedded actions on the right side of a production rule.

4.4 Translations

Synthesized attributes are implemented using translations. A translation follows the last item on the right-hand side of a production rule and is placed before the last action. An translation is contained in square brackets in the form

`[ (` *action_name* `)` *items* `]`

where the items in a translation are attributes and strings. The concatenation of these items produces a string that is used as the value of an attribute labeled name. Examples of translations are found in the next section. For example, the `OBJS` translation in Figure 4.3 is used to collect names of object files. The behavior of the translation is similar to that of a synthesized value in an attribute grammar.

4.5 Constraints

Like package specifications, rule actions may contain constraints. Like attributes, constraints are scoped on the manufacture graph. If setting an attribute within the scope of a constraint violates that constraint, then the search is cutoff. Furthermore, object types within the software structure graph that violate constraints are not considered as candidates to be included in building a manufacture graph as terminal items within the scope of constraints. For example, the rules in Figure 4.3 are used in the case of integrating a main program written in C (a `c_main` object type) and zero or more objects written in C that contain functions (`c_func` object types). Given the package specification in Figure 4.4, the package tool would succeed using the rules in Figure 4.3 and yield the output shown in Figure 4.5. This output is a MAKEFILE for integrating the application. The packager selected the second primitive implementation of the Server module because the `ARCH` attribute of the first implementation violates the `ARCH` == `$1.ARCH` constraint in the `exec` object rule. The item `$1.ARCH` is a reference to a synthesized attribute of the `c_main` item. The primitive implementation for the `Client` module is selected for the `c_main` slot in the manufacture graph. An embedded action sets the constraint and restricts subsequent selection of components to those modules that either do not have an `ARCH` attribute or satisfy the constraint. The implementation for the `Util` module, for instance, does not specify an `ARCH` attribute and therefore satisfies the constraint.

```
exec                <= c_main
                    {
                           ARCH         ==$1.ARCH
                    }
                    c_funcs
                    :{
                           #all:  a.out
                           #
                           #a.out:  $1.FILE:r.o $2(OBJS)
                           # cc -o a.out $1.FILE:r.o $2(OBJS)
                           #
                           #$1.FILE:r.o:  $1.FILE
                           # cc -c $1.FILE
                           #
                           $2
                    }
                    ;

c_funcs             <= c_func c_funcs
                    [ (OBJS) $1.FILE:r ".o" ]
                    :{
                           #$1.FILE:r.o:  $1.FILE
                           #            cc -c $1.FILE
                           #
                           $2
                    }
                    |
                    ;
```

Figure 4.3: Sample rule productions with actions for C programs.

```
include stdpkg.pkg

implement Root as {
      Client;
      Util;
      Server;
}
implement Client with c_main {
      FILE          =client.c
      ARCH          =sparc
}
implement Util with c_func {
      FILE          =util.c
}
implement Server with c_func {
      FILE          =server_1.c
      ARCH          =mips
}
implement Server with c_func {
      FILE          =server_2.c
      ARCH          =sparc
}
```

Figure 4.4: Sample package specification of a C program with alternative Server implementations.

```
all:  a.out

a.out:  client.o util.o server_2.o
              cc -o a.out client.o util.o server_2.o

client.o:  client.c
              cc -c client.c

util.o:  util.o
              cc -c util.c

server_2.o:  server_2.c
              cc -c server_2.c
```

Figure 4.5: MAKEFILE output as result of successful packaging of C program

4.6 Summary

Production rules are highly dependent on the available tools in an execution environment. Furthermore, they are complex and difficult to express, but reusable across many applications programs. The packager uses the rules to "compile" a PACKAGE specification for a specific platform. The same PACKAGE specification can be compiled on other platforms. Being able to port the application in this manner amortizes the cost of constructing the production rules so long as they are reused between applications.

Chapter 5

Results

To build a computer program, one must integrate many different types of software components: functions, data files, resources, libraries, and services. The process of integrating these components is as complex as programming the components themselves, especially if the components are distributed or implemented in different programming languages. The integration process involves many different tools including compilers, link editors, and stub generators. The programmer must explicitly invoke these tools in their proper sequence to perform the integration and build the application.

The integration process for an application will vary between execution environments because each environment will provide a different set of integration tools. The programmer must use tools and components that may still be unique to each environment in order to build an application. Even if an application claims to be "portable" to many environments, this may only mean that the source code is insensitive to changes between execution environments, but the integration processes are unique for each target environment.

The integration process for an application will also change if an application is reconfigured in some fashion, e.g., its components are distributed or implemented in different programming languages. Even if the logical structure of an application remains unchanged, a small change to its configuration can dramatically impact the integration process. Each change requires respecifying the integration steps and rebuilding the application. If an application's components are added, removed, or modified, the integration process must be changed as well.

Software packaging simplifies the task of integrating computer programs because the packager automatically determines the steps necessary to integrate an application. Each environment provides a set of integration rules that characterize the types of integrations possible in that environment. If application components are distributed or implemented in different programming languages, the packager uses the integration rules to determine whether or not it is possible to integrate an application in that environment. The rules are based on the capabilities of the integration tools available in the execution environment. While it is incumbant on the environment to provide the integration rules, the cost of their construction is amortized over all applications packaged in the environment.

An integration process is determined by the packager based on the types of components in an application. With tools like MAKE, developers must know about the types of components in an application, e.g., what language they are written in and whether or not one component provides a starting entry point. They must also know how to integrate those components to create new artifacts. Developers "reinvent the wheel" by needlessly rewriting MAKE rules for new applications. The knowledge of how to use the tools is passed between developers in an ad-hoc fashion often using existing integrations as examples. Software packaging describes the capabilities of tools and reuses this knowledge across applications and developers. Developers must still be aware of the characteristics of their components and what components are compatible, but this knowledge is independent of any execution environment. We demonstrate the savings over existing integration methods by comparing PACKAGE specifications with MAKEFILE specifications. We show that MAKEFILE specifications require extensive changes when an application is reconfigured while PACKAGE specifications are relatively insensitive to reconfigurations across environments and components in comparison to existing integration methods.

The software packager relies on each execution environment to provide integration rules but extends this approach to heterogeneous, distributed environments. Existing integration tools, like NMAKE, IMAKE, and MAKE, also rely on environment-specific integration rules but they are used primarily in homogeneous contexts to provide portability between hardware platforms. We show that software packaging also provides portability within homogeneous applications and extends the rule-based approach to heterogeneous, distributed applications by including other integration tools, like stub generators, into the integration rules. NMAKE and IMAKE suffer the same problems as MAKE when program components are distributed or implemented in multiple languages because they do not address interconnections between components. Furthermore, existing RPC tools and interface descriptions languages (IDLs) are inadequate because they do not address configuration issues. Such tools supply the bridges between components, but the developer is responsible for integrating the application. Our approach relies on existing tools and combines wrapper generation and configuration management to promote transparent integration in heterogeneous, distributed execution environments.

5.1 Approach

Software packaging allows the programmer to deal with modules and bindings in an abstract manner without concern for (1) the differences in languages between modules (2) the location of their execution and (3) the binding mechanism used between runtime components. If the execution environment provides the proper integration tools, the software packager determines which tools are necessary to integrate the application components. While software packaging relies on rules specific to each environment, the cost of constructing these rules is amortized over the total number of integrated applications.

In the next sections we present three case studies in which we compare the effects of reconfiguration changes on PACKAGE specifications and MAKEFILES. Application components may be implemented in different programming languages and distributed across multiple processors. We examine the effects reconfiguration changes to application specifications. By comparing the extent of changes on PACKAGE and MAKEFILE specifications, we evaluate the usefulness of our approach.

Our results show that PACKAGE specifications are less expensive to build and alter than MAKEFILE specifications for the same application. These results show that MAKEFILE specifications require extensive rewriting when components are reconfigured.

It is difficult to show that a programming language is abstract insofar that convenience to the programmer is increased. Convenience is defined loosely in terms of how "terse" it is for the programmer to specify a solution. The best we can hope to do is show that a new approach is more convenient that existing methods. We measure the convenience to the programmer in terms of the lines of code changed in a specification before and after the reconfiguration. The comparison does not include the cost of reimplementing any components. In the case of PACKAGE specifications, we do not include the cost of writing the integration rules for each environment because this cost is not imposed on the developer. In the case of MAKE, however, the developer must specify the integration steps explicitly for each application. We do not, however, include the cost of implementing any additional software in either case because this can be handled by stub generators.

5.1.1 An Example

In Chapter 1, we specified the module-based structure of a factorial application and specified a solution in Chapter 3 via the PACKAGE language. In Chapter 2, we introduced a second possible implementation for the server component, a remote service, in order to demonstrate the use of choice in software designs. In Figure 5.1 and Figure 5.3 we show two separate PACKAGE specifications where the only difference is in the implementations of the `Server` module. Although we can specify both implementations of the server in the same specification, we separate them in order to force the packager to choose the only available implementation. In a joined specification, the packager would arbitrarily choose one or the other if there are no relevant constraints. We produce two separate MAKEFILES by packaging each specification as shown in Figure 5.2 and Figure 5.4. The corresponding production graphs are shown in Figure 5.5 and Figure 5.6. Recall that the difference between the two solutions is the result of reimplementing the server component but the logical structure of the application remains unchanged. The PACKAGE specifications are nearly identical with the exception of the change to the implementation for the server component. In comparison, the MAKEFILES are very different from one another.

Currently, writing MAKEFILES by hand is a tedious task. Each developer must know about the type of component and what tools to use in order to build an executable program. If there is a change in the configuration, the developer must respecify the integration by rewriting the MAKEFILE. Software packaging eliminates this step. Although the developer must know about the types of components and which as compatible, the details of integration are left to the packager tool. The effort to change the PACKAGE specification when the server component is reimplemented is considerably less than the effort needed to rewrite the MAKEFILE. The cost of writing the production rules can be amortized over all packaged applications since they are reused. The cost of reimplementing the component is not counted in either case.

Even though the PACKAGE approach relies on the environment to provide the integration rules, they are used by all developers to configure all types of applications. In our examples, we rely on a fixed set of rules with the exception of the portability example which we present for illustrative purposes. Even though our use of PACKAGE specifications for these examples does not justify amortizing the cost of these rules, our experience so far indicates that the component types are very general and can be applied in a wide variety of applications.

PACKAGE specifications allow developers to deal with abstract interconnections between components that would otherwise be difficult to implement. For example, when connecting heterogeneous applications, a developer must implement connections between components. This may involve use of separate interface and integration specifications. For example, Sun RPC is useful for describing interface specifications, but the developer must organize the overall application in a MAKEFILE. PACKAGE specifications centralize component descriptions and interconnections in a single specification.

```
include stdpkg.pkg

module Client {
        use factorial(int)(int);
}

module Server {
        def factorial(int)(int);
}

implement Root as {
        Client:  c;
        Server:  s;
        bind c'factroial to s'factorial;
}
implement Client with kcl_main {
        FILE    = client.kcl
}
# C implementation of the server component
implement Server with c_func {
        FILE    = server.c
}
```

Figure 5.1: A Package specification for the Factorial solution (Lisp-C)

```
all:  factorial

factorial:  client.o client_wrap.o server.o
        echo "(load \"client\")" > init.lsp
        echo -n "(si:faslink \"client_wrap\" >> init.lsp
        echo "\"server.o \")" >> init.lsp
        echo "(si:save-system \"factorial\")" >> init.lsp
        kcl
        echo "(compute-facs)" > init.lsp
        echo "(bye)" >> init.lsp

client.o:  client.lsp
        lc client.lsp

server.o:  server.c
        cc -c server.c

client_wrap.o:  client_wrap.lsp
        lc client_wrap.lsp

client_wrap.lsp:  Map
        kclwrap 1 > client_wrap.lsp
```

Figure 5.2: Makefile specification for the Lisp-C factorial solution

```
include stdpkg.pkg

module Client {
        use factorial(int)(int);
}
module Server {
        def factorial(int)(int);
}
implement Root as {
        Client:  c;
        Server:  s;
        bind c'factroial to s'factorial;
}
implement Client with c_main {
        FILE    = client.c
}
# Remote implementation of the server component
implement Server with rpc_svc {
        PROGNUM = 407888
        LOC     = thumper.cs.umd.edu
}
```

Figure 5.3: A PACKAGE specification for the Factorial solution (Lisp-RPC)

5.1.2 Portability

We examine the issue of software portability at the end of this Chapter. When an application is ported to a new execution environment, the integration process is different and some components of the application must be reimplemented. Software packaging allows developers to specify alternative implementations and facilitate integrations across multiple environments.

For example, if we include both implementations of the `server` component in the factorial example in a single PACKAGE specification, the packager will choose a compatible set of implementations from the set of possible implementations. Figure 5.7 shows a production graph for such a specification in which the packager has chosen the C implementation of the `server` component. The remote implementation (shown in a diamond with a dotted arrow from the server module instance) is not selected and therefore not included in the manufacture graph. A different environment may have selected the remote service if tools for integrating the C implementation do not exist in that environment.

The use of alternative implementations allows PACKAGE specifications to be used to "port" applications between execution environments by structuring software designs such that large subsystems of code are insensitive to environment changes. Developers often call this part of the application code the "kernel" of a system. Other subsystems are more dependent on specific environmental factors such as display type, processor type, and operating system. In a poorly designed solution, most modules will be system dependent. A well-structured design, however, will minimize the impact of porting the application.

At the end of this Chaper, we examine a highly portable and complex software package, the X11 Window System server, as a case study to demonstrate portability of PACKAGE specifications. The X11 server contains both device-dependent and device-independent modules. Multiple implemen-

```
all:  factorial

factorial:  client.o client_wrap.o server_clnt.o
        echo "(load \"client\")" > init.lsp
        echo -n "(si:faslink \"client_wrap\" >> init.lsp
        echo "\"client_bind.o server_clnt.o -lc\")" >> init.lsp
        echo "(si:save-system \"factorial\")" >> init.lsp
        kcl
        echo "(rpcinit)" > init.lsp
        echo "(compute-facs)" >> init.lsp
        echo "(bye)" >> init.lsp

client.o:  client.lsp
        lc client.lsp

client_bind.o:  client_bind.c server.h
        cc -c client_bind.c

client_bind.c:  Map
        kclstubgen 1 > client_bind.c

server_clnt.o:  server_clnt.c server.h
        cc -c server_clnt.c

server_clnt.o server.h:  server.x
        rpcgen server.x

#
# Programmer must know how to invoke the sunrpcgen tool,
# with the Sun RPC program number and the host machine.
#
server.x:  Map
        sunrpcgen -s 407888 -h thumper.cs.umd.edu > server.x

client_wrap.o:  client_wrap.lsp
        lc client_wrap.lsp

client_wrap.lsp:  Map
        kclwrap -r 1 > client_wrap.lsp
```

Figure 5.4: Makefile specification for the Lisp-RPC factorial solution

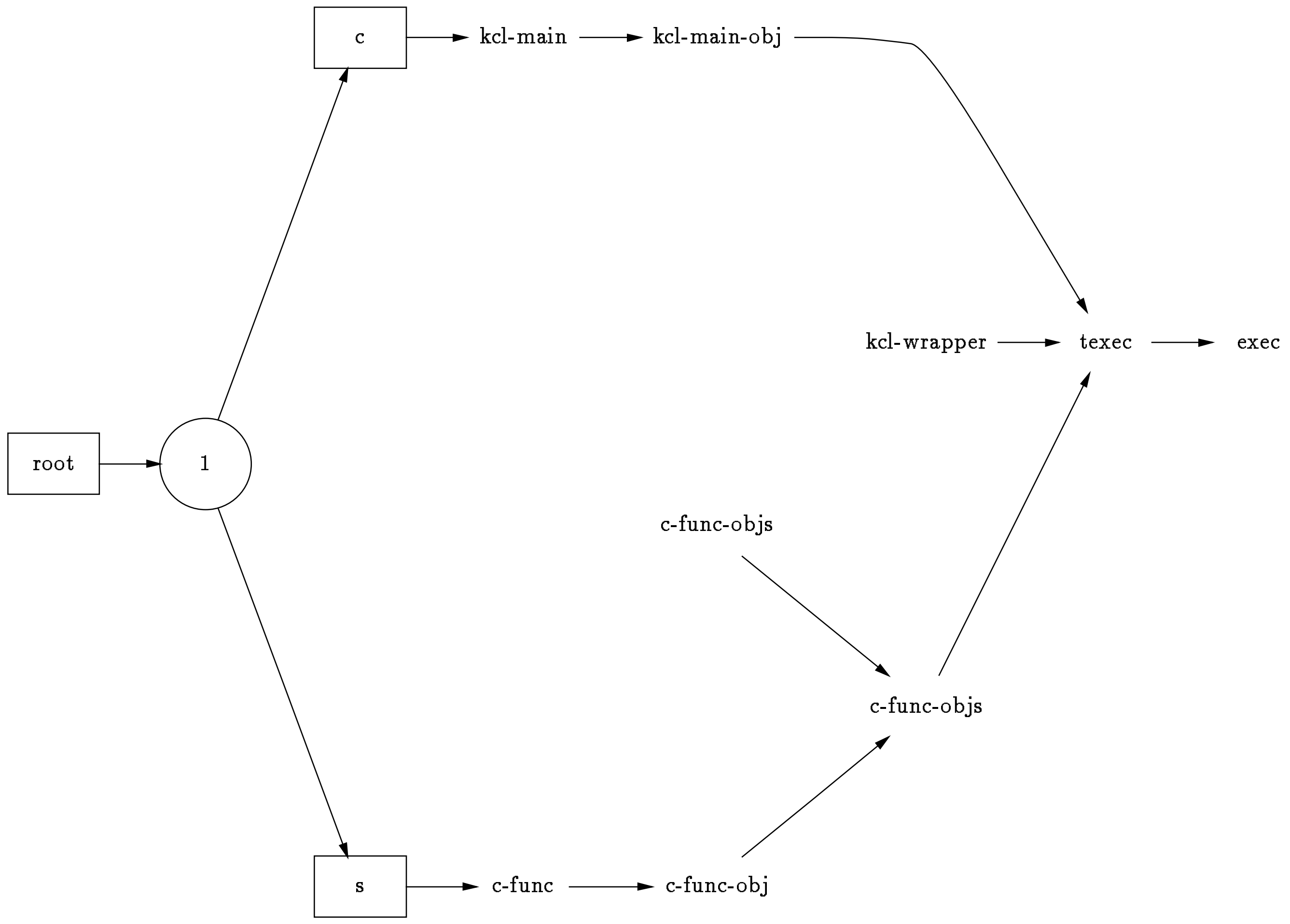

Figure 5.5: Production graph for the Lisp-C factorial solution.

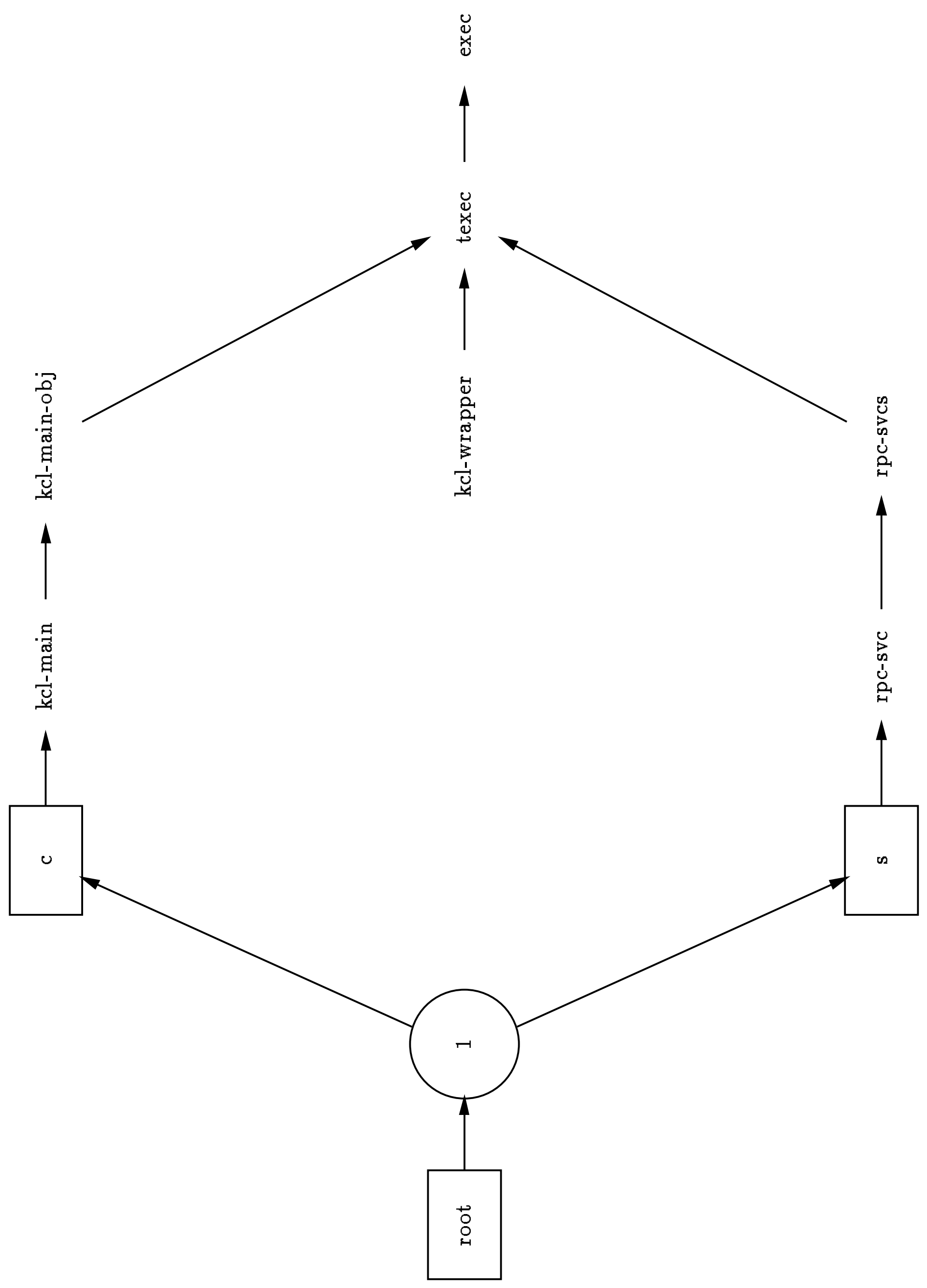

Figure 5.6: Production graph for the Lisp-RPC factorial solution.

tations exist for the device-dependent modules while the device-independent components are source code portable. The X11 server is typically configured using IMAKE which relies on the C preprocessor and the UNIX directory structure to organize alternate implementations. We respecify the X11 server as a software package to demonstrate that our approach is robust, backward-compatible and elucidates the organization of the X11 server implementation better than the existing approach.

5.2 Heterogeneity

When constructing a new application, it is cost-effective to reuse existing components where possible because reuse dramatically reduces the cost of implementation. The UNIX shell is a good example: programs can be connected together by pipes that transfer character data. Programs can be integrated easily with other programs so long as the other programs produce or consume character streams on the appropriate interfaces, i.e. `stdin, stdout, stderr`.

Many programs, however, cannot be integrated because of incompatibilities between their data representation formats. For example, both solutions to the factorial problem rely on stubs to bridge the differences between integer representations. The Lisp environment has the ability to do this through the `defentry` function. The PACKAGE specifications in Figure 5.1 and Figure 5.3 state that the `Client` and `Server` components have compatible ports and bind the `use` and `def` ports together. The difference between their representations of integers is handled by the packager and tools that generate the wrappers needed to integrate application components.

Software packaging allows software developers to reuse components and ignore differences between data representations if there exist appropriate integration tools such as wrapper generators. For example, a solution to Parnas' Keyword-In-Context (KWIC) program [Parn72] can be built from a variety of existing components: a line reader, a line shifter, and line writer. The purpose of the KWIC program is to read a set of input lines such as

```
the quick brown fox
jumped over the lazy dog
```

and produce a corresponding ouput file such as

```
the quick brown fox
quick brown fox the
brown fox the quick
fox the quick brown
jumped over the lazy dog
over the lazy dog jumped
the lazy dog jumped over
lazy dog jumped over the
dog jumped over the lazy
```

in which all lines are versions of the input lines shifted by one word. This is done to create a list of lines in which the first word can be indexed and thus show all words in their context of use.

A PACKAGE specification for a solution to the KWIC problem is shown in Figure 5.9. All modules have single implementations written in the C++ programming language. For instance, the source code for the shifter implementation is shown in Figure 5.8. The MAKEFILE for this solution is shown in Figure 5.11 and the corresponding production graph is shown in Figure 5.10.

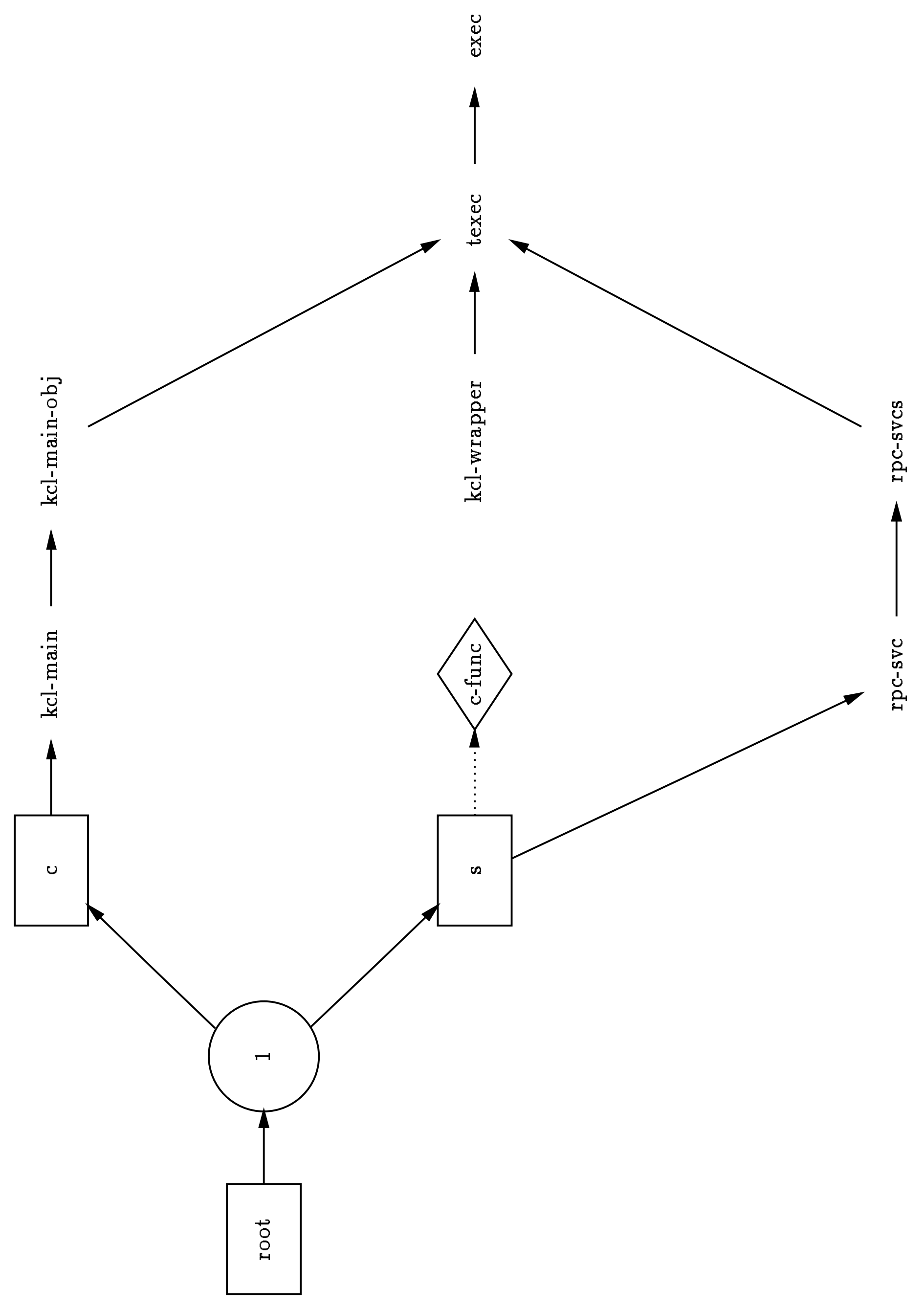

Figure 5.7: Joint production graph for the factorial solution.

```
#include <String.h>
#include <OrderedCltn.h>

void skipblanks(String& s,int& x) {
        while((x < s.length()) && (s[x] == ' ')) x++;
}

void skipword(String& s,int& x) {
        while((x < s.length()) && (s[x] != ' ')) x++;
}

void shiftlines(OrderedCltn& inlines,OrderedCltn& outlines) {
        String *s,*t;

        for(int i=0;i < inlines.size();i++) {
                s = (String*)inlines[i];
                int length = s->length();
                for(int j=0;j < length-1;) {
                        skipblanks(*s,j);
                        t = new String(*s);
                        int k = 0;
                        int x = j;
                        while(k < length) {
                                (*t)[k++] = (*s)[x % length];
                                x++;
                        }
                        outlines.add(*t);
                        skipword(*s,j);
                }
        }
}
```

Figure 5.8: Source code for the shifter component of the KWIC program.

The integration process is straightforward. All component implementations are compiled and linked using the C++ compiler and linker respectively.

This first solution does not sort the output lines or eliminate identical lines. In Figure 5.13, we introduce a sort module into the PACKAGE specification. The sort module has two interface ports: an `instream` port and an `outstream` port. The statement

```
bind main'sortlines to sorter'inlines return sorter'outlines;
```

connects the `sortlines` port (a `use` port) to the `inlines` port (an `instream` port) and `outlines` port (an `outstream` port) of the sorter module. This is possible because there is a connector definition in the `stdpkg.pkg` file

```
connector bind(use(x)(y)) to(instream) return(outstream) {
        Bridge b($x,$y);
        bind $1 to b'input;
        bind b'outcall to $2;
        bind $3 to b'incall;
}
```

that enable ports to be connected in this fashion. The `bind-to-return` connection between the `use` port and the two `stream` ports is translated into a subsystem including a `Bridge` module instance and three connections using the `bind-to` connector. The existence of the `bind-to-return` connector does not imply that the connection is implemented, but a compatible implementation for the `Bridge` module must exist as well. The `Bridge` module is specified as

```
module Bridge(p,q) {
        def input($p)($q);
        outstream outcall;
        instream incall;
}
```

where `p` and `q` refer to any type names. Many different implementations of the `Bridge` module might exist in an environment. For example, the implementation

```
implement Bridge(p,q) with strmbridge($p,$q);
```

associates a `strmbridge` object as an implementation that converts an object of type (`p`) into a stream and returns an object of type (`q`) constructed from a stream. A property of all `cc_main` and `cc_func` objects is that they pass either primitive data types (integers, strings, floating point numbers, and characters) or complex data types implemented by NIH classes [Gorl90]. NIH classes have the property that they can be rendered onto persistent media such as files or byte streams. This capability allows object state to be transported *between* components executing in separate address spaces. Without this capability, we would be limited in the types of integrations that are possible only *within* the same address space.

A MAKEFILE for the second solution to the KWIC problem is shown in Figure 5.14 and the corresponding production graph is shown in Figure 5.12. Although the differences are not dramatic, this case demonstrates the usefulness of the connection abstractions in PACKAGE specifications. In line 24 of the MAKEFILE in Figure 5.14, the directive `-DPROCNAME=sortlines` is a parameter of the `strmbridge` component. This configuration information relies on knowledge of the interconnections

```
include stdpkg.pkg
module Main {
            use readlines()(Lines)();
            use shiftlines(Lines)(Lines)();
            use writelines(Lines)()();
}
module Reader {
            def readlines()(Lines)();
}
module Shifter {
            def shiftlines(Lines)(Lines)();
}
module Writer {
            def writelines(Lines)()();
}
implement Root as {
            APPNAME=shifter
            Main:  main;
            Reader:  reader;
            Shifter:  shifter;
            Writer:  writer;
            bind main'readlines to reader'readlines;
            bind main'shiftlines to shifter'shiftlines;
            bind main'writelines to writer'writelines;
}
implement Main with cc_main {
            FILE=main1.cc
}
implement Reader with cc_func {
            FILE=reader.cc
}
implement Shifter with cc_func {
            FILE=shifter.cc
}
implement Writer with cc_func {
            FILE=writer.cc
}
```

Figure 5.9: PACKAGE specifications for the first KWIC solution

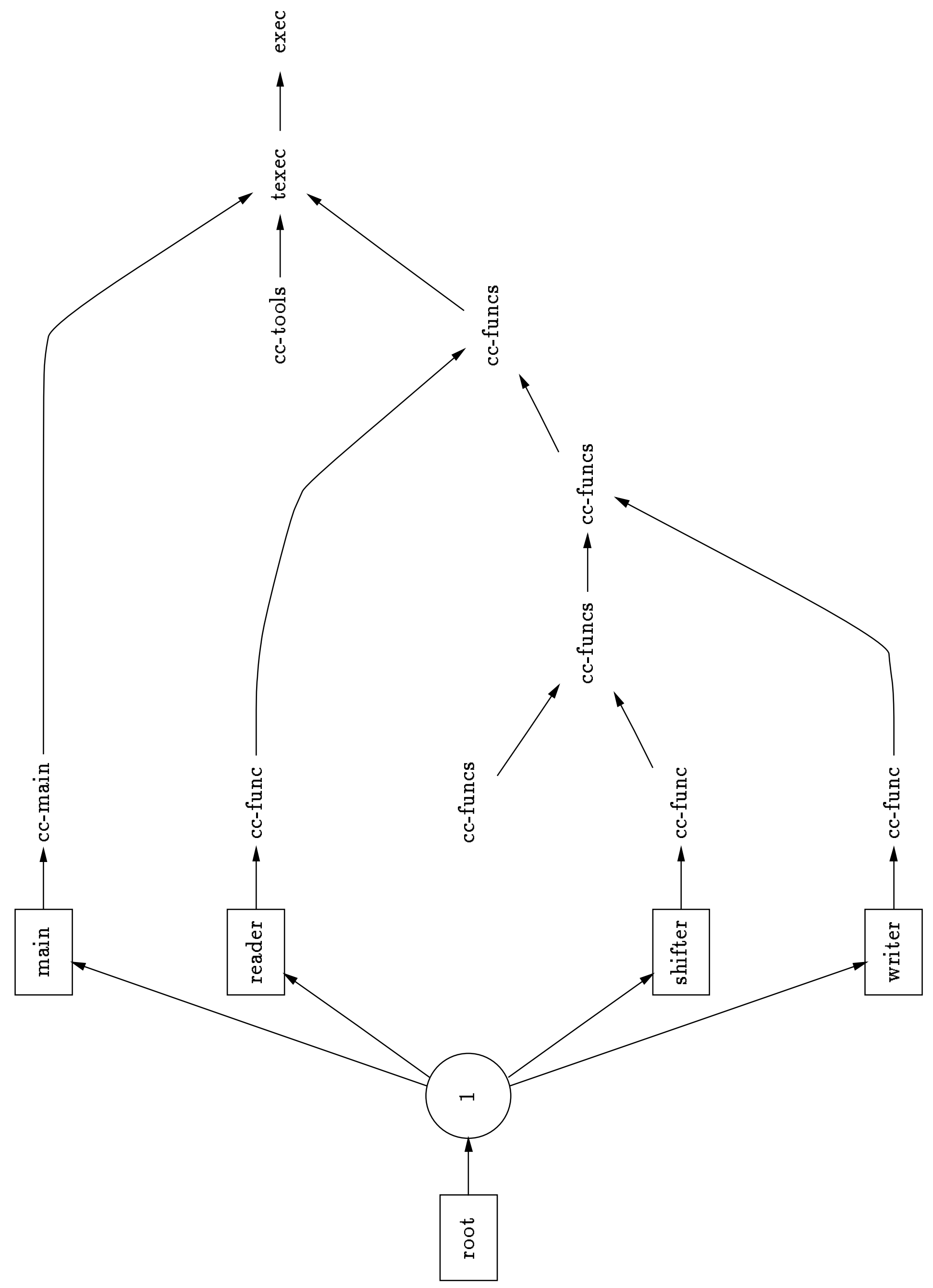

Figure 5.10: Production graph for the first KWIC solution.

```
all:  shifter

shifter:  main1.o reader.o writer.o shifter.o
      CC -g -o shifter -I/thumper/include main1.o reader.o \
            writer.o shifter.o -L/thumper/lib/sun4os4 -lnihcl -lniherr

main1.o:  main1.cc
      CC -g -I/thumper/include -c main1.cc

reader.o:  reader.cc
      CC -g -I/thumper/include -c reader.cc

writer.o:  writer.cc
      CC -g -I/thumper/include -c writer.cc

shifter.o:  shifter.cc
      CC -g -I/thumper/include -c shifter.cc
```

Figure 5.11: Makefile specification for the first KWIC solution

between components independent of the integration process. PACKAGE specifications eliminate the need for explicit specification of such information in separate interface specifications. The developer deals only with modules, implementations, and abstract connections.

PACKAGE specifications are no worse than MAKEFILES in many simple cases. In complex integration cases, however, PACKAGE specifications are more abstract because the developer need not deal with integration details. This has the advantage that developers need not remember the details of using integration tools. Instead, a developer is only concerned with what object types are available. Software packaging offers no specific integration solution or conversion protocol, rather it leverages existing tools. If such tools do not exist in an environment, then it will not be possible to integrate an application.

5.3 Distribution

In many applications, program components are distributed to take advantage of parallelism in order to improve program performance. Ray tracing is one application for which we can write parallel solutions that substantially outperform sequential solutions. The purpose of ray tracing is to produce a photo-realistic image of a scene containing objects and a light source. The scene is computed by a program that accepts a geometric description of objects as input. For example, the data in Table 5.1 renders the image in Figure 5.16 of a single mirrored sphere at a specified location in 3-space of radius 50 against a checkered background. The ray tracing program follows the path of each light ray backwards from the camera position through the pixel plane, to all incidental objects, and back to the light source as shown in Figure 5.15.

The ray tracing algorithm lends itself easily to parallelism because each pixel in the pixel plane can be computed independently. We construct a solution with slightly less granularity: it computes each horizontal scan line independently, but pixels sequentially with each line. We construct two solutions to the ray tracing problem. The first configuration implements ray tracing sequentially

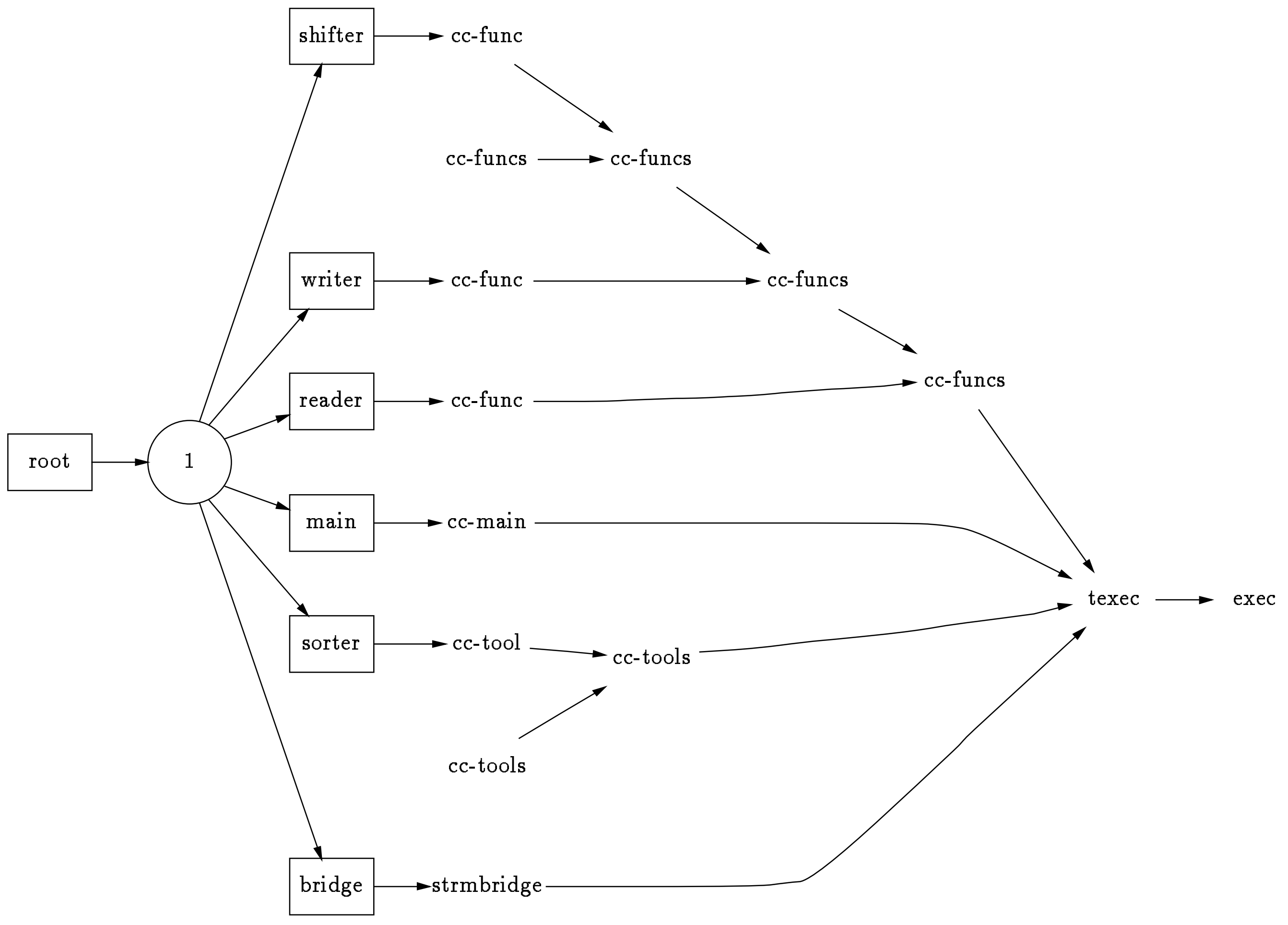

Figure 5.12: Production graph for the second KWIC solution.

```
include stdpkg.pkg

module Main {
            use readlines()(Lines)();
            use shiftlines(Lines)(Lines)();
            use sortlines(Lines)(Lines)();
            use writelines(Lines)()();
}
module Sorter {
            instream inlines(Lines);
            outstream outlines(Lines);
}
implement Root as {
            APPNAME=shifter
            Main:  main;
            Reader:  reader;
            Shifter:  shifter;
            Sorter:  sorter;
            Writer:  writer;
            bind main'readlines to reader'readlines;
            bind main'shiftlines to shifter'shiftlines;
            # bind a use port to in and out streams
            bind main'sortlines to sorter'inlines return sorter'outlines;
            bind main'writelines to writer'writelines;
}
implement Main with cc_main {
            FILE=main2.cc
}
implement Sorter with cc_tool {
            TOOL=sort
}
```

Figure 5.13: PACKAGE specifications for the second KWIC solution (with sorting)

```
all:  shifter

shifter:  main2.o reader.o writer.o shifter.o wraptool10.o
      CC -g -o shifter -I/thumper/include main2.o reader.o \
writer.o shifter.o wraptool10.o -L/thumper/lib/sun4os4 -lnihcl -lniherr

main2.o:  main2.cc
      CC -g -I/thumper/include -c main2.cc

reader.o:  reader.cc
      CC -g -I/thumper/include -c reader.cc

writer.o:  writer.cc
      CC -g -I/thumper/include -c writer.cc

shifter.o:  shifter.cc
      CC -g -I/thumper/include -c shifter.cc

#
# Here a converter is used to implement the bridge between
# the use port of the shifter and the in and out
# stream ports of the sorter component
#
wraptool10.o:  wraptool.cc Map
      CC -g -o wraptool10.o -I/thumper/include -c \
            -DPROCNAME=`../toolscan/toolscan -p def 10` \
            -DTOOL="\"sort\"" wraptool.cc
```

Figure 5.14: Makefile specification for the second KWIC solution (with sorting)

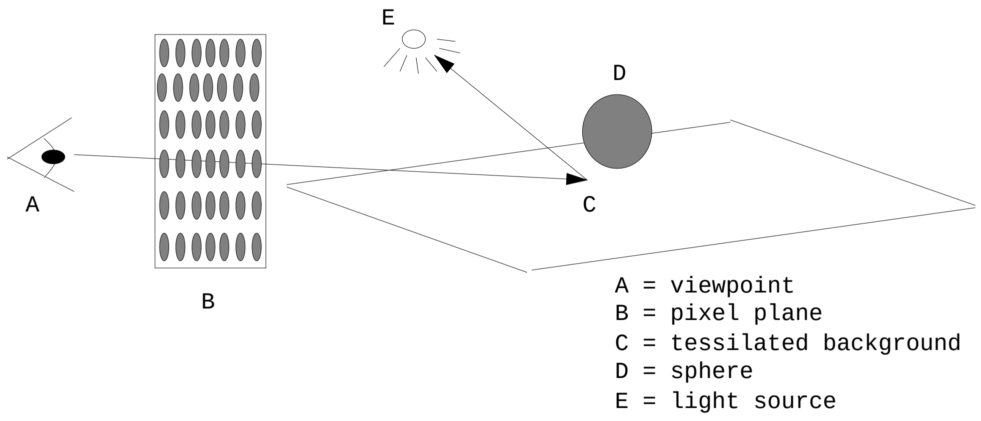

Figure 5.15: Conceptual diagram of the ray tracing algorithm.

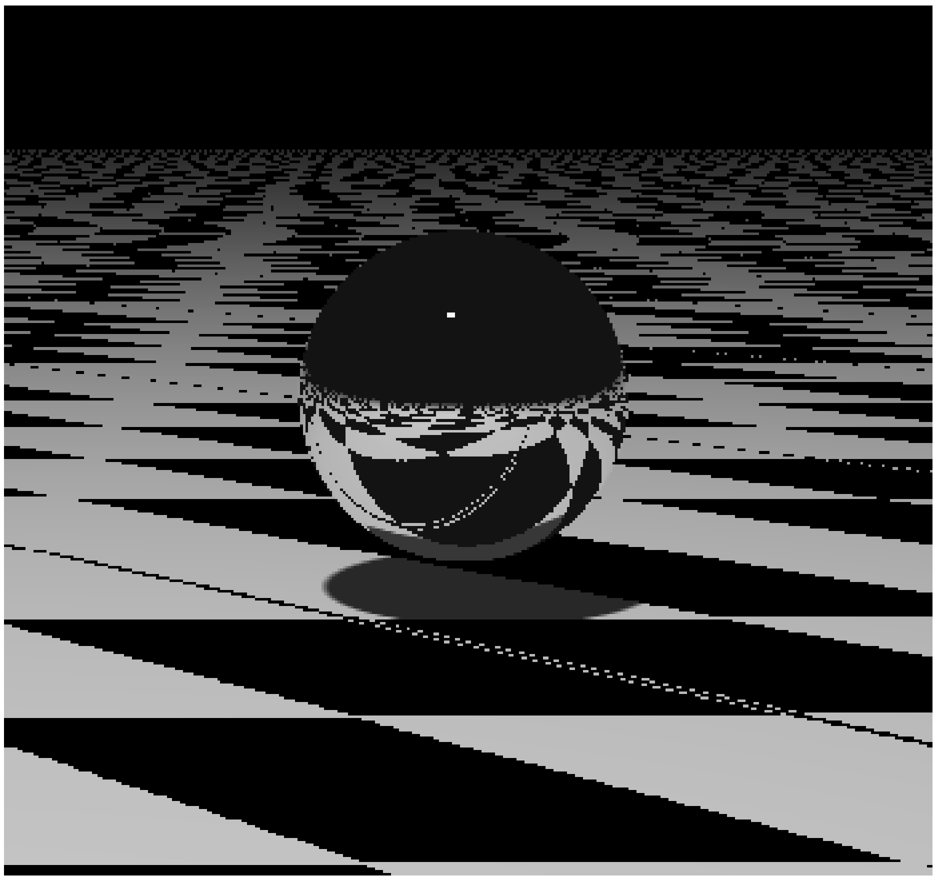

Figure 5.16: The resulting ray traced image.

x	y	z	radius	reflect	refract	opaque	density	ambience
140.0	65.0	140.0	50.0	0.0	0.0	0.0	0.9	0.1

Table 5.1: Description of a sphere for the ray tracing program.

— a single server computes each scan line. In a second configuration, three servers are utilized to compute scan lines in parallel. Both client and server are implemented in C. The first solution relies on standard link editing to configure the application. The second solution relies on the Polylith software bus to integrate the application. The servers are distributed to separate processors on a local area network.

The PACKAGE specification for the sequential solution is shown in Figure 5.17. In this case, the application consists of 3 components: the main program, the tracer, and a collector component that writes the resulting image to a file. The data types exchanged between components are included from an external specification in the file `raytypes.pkg` shown in Figure 5.18. The data types are either primitive (i.e., int, double, string) or structures specified in terms of class definitions shown in Figure 5.18. This approach is similar to other interface description languages such as Scorpion/IDL [Snod89], OMG/IDL [OMG90], and Sun RPCL [Sun85b]. The integration tools will use these definitions to generate the appropriate stub code.

The MAKEFILE for integrating the sequential solution is shown in Figure 5.19 and the corresponding production graph is shown in Figure 5.22. The components are integrating using the standard C compiler and link editor because all the components are programmed in C and should execute on the same physical machine.

The distributed solution to the ray tracing problem is shown in Figure 5.20 and Figure 5.21. In this case, we distribute the tracer component across three machines in a local area network. We introduce a `Splitter` module to multiplex the `ray_trace` call to compute the scan lines across the processors. Three implementations of the `Tracer` module are instantiated with different `LOC` attributes that specify machine addresses. In the first solution, all components resided in the same address space. Their location attributes were unspecified and therefore the identical by default. In the second solution, different production rules are triggered as a result of the differences in `LOC` attributes.

The corresponding MAKEFILE shown in Figure 5.24 through Figure 5.25 for this solution specifies the use an interprocess communication mechanism called Polylith [Purt85] to implement the connections between components at runtime. The production graph in Figure 5.23 gives a bird's-eye view of the entire integration better than the MAKEFILE. The components are wrapped with stubs that permit communication across processors using Polylith. The Polylith system can wrap and integrate components of `c_main` and `c_func` types. Polylith stub generators access the packager Map to determine which ports are connected and the representations of parameter data types. Polylith relies on TCP/IP sockets to implement interconnection between ports of module implementations.

The resulting MAKEFILES for the two solutions to the ray tracing problem are drastically different from each other. Even though the PACKAGE specification is slightly different, the integration processes are drastically different due to the distribution of components. The source code for the components is unchanged in each case. The wrappers allow the same code to be used in different contexts.

Software packaging addresses the problem of adapting code to new contexts. This allows component implementors to use remote and local procedures in the same fashion. The developer is

```
include stdpkg.pkg
include raytypes.pkg
module Client {
            use load_orb(sphere;double;double;double;double;double)(int);
            use load_bkgnd(str)(int);
            use set_params(params)(int);
            use openoutfile(str)(int);
            use ray_trace(lines)(int);
            use collect(lines;int)(int);
}
module Scan {
            def load_orb(sphere;double;double;double;double;double)(int);
            def load_bkgnd(str)(int);
            def set_params(params)(int);
            def ray_trace(lines)(int);
            use gather(int;int;raw[640])(int);
}
module Collect {
            def openoutfile(str)(int);
            def gather(int;int;raw[640])(int);
            def collect(lines;int)(int);
}
implement Root as {
            APPNAME=rtrace
            #
            # Assign default execution location. Since there
            # is only a single scanner and all components have
            # the same LOC attribute value, then the packager
            # should produce a single executable solution.
            #
            LOC=crosshare.cs.umd.edu    # default execution location
            Client:  c;
            Scan:  s;
            Collect:  x;
            bind c'load_orb to s'load_orb;
            bind c'load_bkgnd to s'load_bkgnd;
            bind c'set_params to s'set_params;
            bind c'openoutfile to x'openoutfile;
            bind c'ray_trace to s'ray_trace;
            bind s'gather to x'gather;
bind c'collect to x'collect;
}
implement Client with c_main {
            FILE =rclient.c
}
implement Scan with c_func {
            FILE =rserver.c
}
implement Collect with c_func {
            FILE =rcollect.c
}
```

Figure 5.17: PACKAGE specification for the sequential ray tracer.

```
class vector {
          double x;
          double y;
          double z;
          double l;
          double xzl;
}
class sphere {
          vector cent;
          double rad;
}
class params {
          sphere &ls;
          vector &vp;
          double &bkcon;
}
class lines {
          double ymin;
          double ymax;
}
```

Figure 5.18: PACKAGE class specifications for the ray tracing solutions (`raytypes.pkg` file).

```
all:  rtrace

rtrace:  rclient.o rcollect.o rserver.o
      cc -o rtrace rclient.o rcollect.o rserver.o -L. -lm

rclient.o:  rclient.c
      cc -c rclient.c

rcollect.o:  rcollect.c
      cc -c rcollect.c

rserver.o:  rserver.c
      cc -c rserver.c
```

Figure 5.19: Makefile specification for the sequential ray tracer.

```
include stdpkg.pkg
include raytypes.pkg
module OrbSplit {
            def in(sphere;double;double;double;double;double)(int);
            use out[3](sphere;double;double;double;double;double)(int);
}
module BkgndSplit {
            def in(str)(int);
            use out[3](str)(int);
}
module ParamSplit {
            def in(params)(int);
            use out[3](params)(int);
}
module RayMux {
            def in(lines)(int);
            use out[3](lines)(int);
}
module DeMux {
            def in[3](int;int;raw[640])(int);
            use out(int;int;raw[640])(int);
}
implement Root as {
            APPNAME =rtrace
            LOC=crosshare.cs.umd.edu  # default execution location
            Client:  c;
            OrbSplit:  os;
            BkgndSplit:  bs;
            ParamSplit:  ps;
            RayMux:  mux;
            #
            # Assign specific execution locations to parallel
            # scanners. The location differences will trigger
            # the use of an RPC-based integration mechanism.
            #
            Scan:  s0 :  LOC=thumper.cs.umd.edu
            Scan:  s1 :  LOC=harvey.cs.umd.edu
            Scan:  s2 :  LOC=xring.cs.umd.edu
            DeMux:  demux;
            Collect:  x;
            bind c'load_orb to os'in;
            bind os'out[*] to s$1'load_orb;
            bind c'load_bkgnd to bs'in;
            bind bs'out[*] to s$1'load_bkgnd;
            bind c'set_params to ps'in;
            bind ps'out[*] to s$1'set_params;
            bind c'openoutfile to x'openoutfile;
            bind c'ray_trace to mux'in;
            bind mux'out[*] to s$1'ray_trace;
            bind s*'gather to demux'in[$1];
            bind demux'out to x'gather;
            bind c'collect to x'collect;
}
```

Figure 5.20: PACKAGE specification for the distributed ray tracer.

```
implement OrbSplit with p_raw {
           FILE =splitorb.c
           COPTS =-DFANOUT=3
}
implement BkgndSplit with p_raw {
           FILE =splitbkgnd.c
           COPTS =-DFANOUT=3
}
implement ParamSplit with p_raw {
           FILE =splitparams.c
           COPTS =-DFANOUT=3
}
implement RayMux with p_raw {
           FILE =raymux.c
           COPTS =-DFANOUT=3
}
implement DeMux with p_raw {
           FILE =demux.c
           COPTS =-DFANOUT=3
}
```

Figure 5.21: PACKAGE specification for the distributed ray tracer (con't).

only concerned with implementation types for modules. The packager determines whether or not any implementations can be integrated. This is a significant improvement over existing tools that require knowledge of component types and integration methods.

5.4 Portability

The X Windows server is one of the most portable software applications in the commercial and public domains. It runs in a plethora of execution environments: almost all UNIX platforms, MacIntosh, and IBM PCs. To accomplish this task, the developers employ a technique similar to the choice mechanism in software packaging: many components in the X server have multiple, alternative implementations. The program is highly modular because application components are classified into two categories: device-dependent and device-independent. Device-dependent components have multiple implementations but a standard interface. Device-independent components are source code portable between systems and implement the core of the server.

This organization is well suited to software packaging. We specify the X11 server as a package to demonstrate that software packaging can be used to configure existing software. Existing configuration tools like NMAKE and IMAKE rely on the structure of the file system to differentiate alternate implementations of device-dependent components. This is similar to the approach in software packaging but instead of file and directory names used to distinguish alternatives, the packager uses attributes and constraints to choose appropriate implementations. This allows the developer to infer different configurations rather than specifying them explicitly.

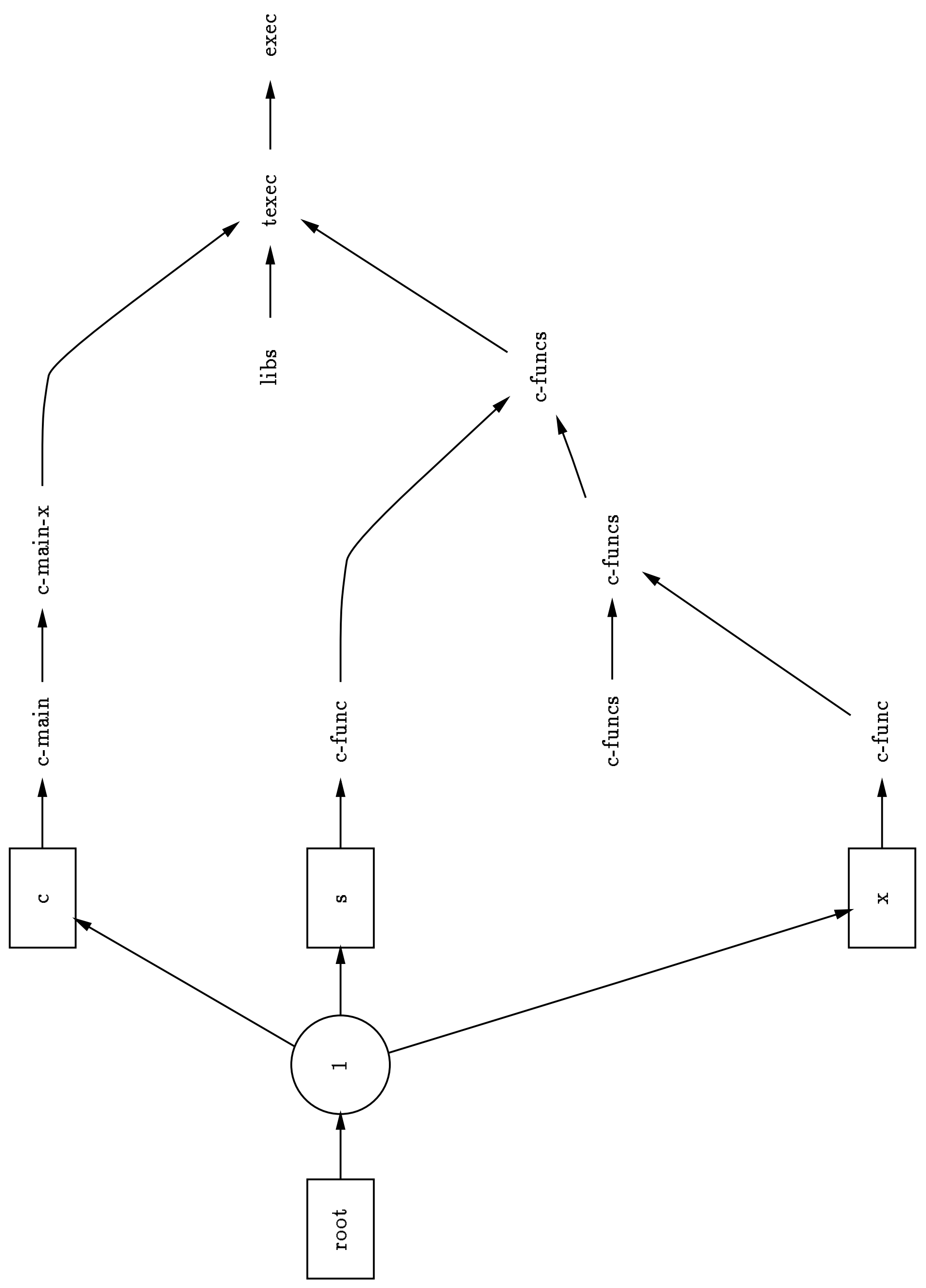

Figure 5.22: Production graph for the sequential ray tracing solution.

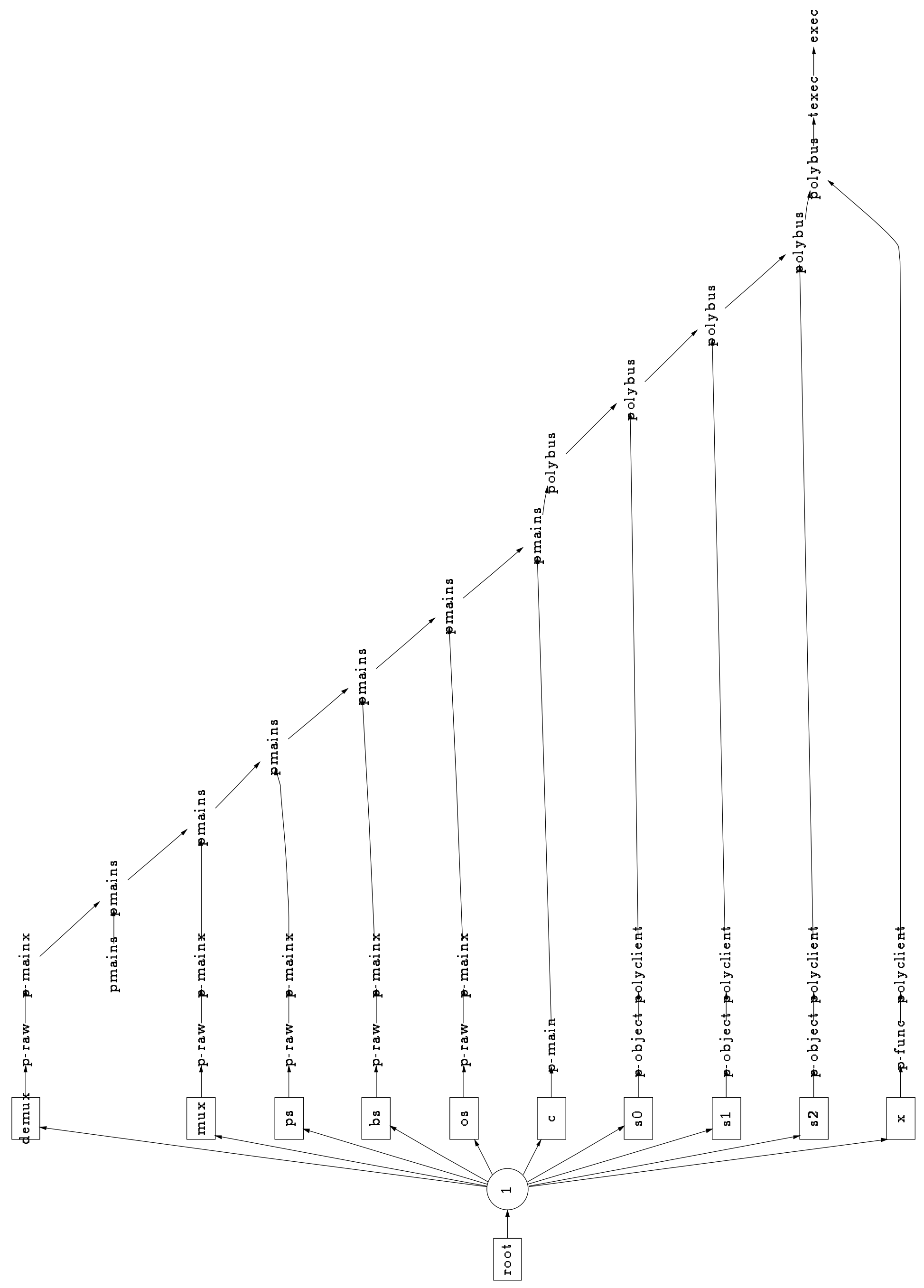

Figure 5.23: Production graph for the distributed ray tracing solution.

```
LOC=`hostname`
all: rtrace

rtrace: rtrace.bus
        echo "#!/bin/csh -f" > rtrace
        echo "bus rtrace.bus" >> rtrace
        chmod u+x rtrace

rtrace.bus: rcollect.co rserver18.co rserver16.co \
        rserver14.co rclient4.co splitorb6.co \
        splitbkgnd8.co splitparams10.co raymux12.co demux20.co rtrace.co
        csl -m rcollect.co rserver18.co rserver16.co \
        rserver14.co rclient4.co splitorb6.co splitbkgnd8.co \
        splitparams10.co raymux12.co demux20.co rtrace.co -o rtrace.bus

rtrace.co: rtrace.cl
        csc rtrace.cl

rtrace.cl: Map
        ../polyorchgen/polyorchgen rtrace 22 18 16 14 4 6 8 10 12 20 > rtrace.cl

rcollect.co: rcollect.cl rcollect
        csc rcollect.cl

rcollect.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu rcollect 22 18 16 14 4 6 8 10 12 20 > rcollect.cl

rcollect: rcollect.o rcollect-stub.o
        cc -o rcollect rcollect.o rcollect-stub.o -lith -lm

rcollect-stub.c: Map
        ../newpolystubgen/polystubgen -s 22 > rcollect-stub.c

rserver18.co: rserver18.cl rserver18
        csc rserver18.cl
rserver18.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu rserver18 18 22 16 14 4 6 8 10 12 20 > rserver18.cl

rserver18: rserver18.o rserver18-stub.o
        cc -o rserver18 rserver18.o rserver18-stub.o -lith -lm

rserver18.o: rserver.c
        cc -o rserver18.o -c rserver.c

rserver18-stub.o: rserver18-stub.c
        cc -o rserver18-stub.o -c rserver18-stub.c

rserver18-stub.c: Map
        ../newpolystubgen/polystubgen -s 18 > rserver18-stub.c

rserver16.co: rserver16.cl rserver16
        csc rserver16.cl

rserver16.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m harvey.cs.umd.edu rserver16 16 22 18 14 4 6 8 10 12 20 > rserver16.cl

rserver16: rserver16.o rserver16-stub.o
        cc -o rserver16 rserver16.o rserver16-stub.o -lith -lm

rserver16: rserver16.o rserver16-stub.o
        cc -o rserver16 rserver16.o rserver16-stub.o -lith -lm

rserver16.o: rserver.c
        cc -o rserver16.o -c rserver.c

rserver16-stub.o: rserver16-stub.c
        cc -o rserver16-stub.o -c rserver16-stub.c

rserver16-stub.c: Map
        ../newpolystubgen/polystubgen -s 16 > rserver16-stub.c
rserver14.co: rserver14.cl rserver14
        csc rserver14.cl

rserver14.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m thumper.cs.umd.edu rserver14 14 22 18 16 4 6 8 10 12 20 > rserver14.cl

rserver14: rserver14.o rserver14-stub.o
        cc -o rserver14 rserver14.o rserver14-stub.o -lith -lm

rserver14.o: rserver.c
        cc -o rserver14.o -c rserver.c

rserver14-stub.o: rserver14-stub.c
        cc -o rserver14-stub.o -c rserver14-stub.c

rserver14-stub.c: Map
        ../newpolystubgen/polystubgen -s 14 > rserver14-stub.c
```

Figure 5.24: Makefile specification for the distributed ray tracer (part 1 of 2).

```
rclient4.co: rclient4.cl rclient4
        csc rclient4.cl

rclient4.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu rclient4 4 22 18 16 14 6 8 10 12 20 > rclient4.cl

rclient4: rclient4.o rclient4-stub.o
        cc -o rclient4 rclient4.o rclient4-stub.o -lith -lm

rclient4.o: rclient.c
        cc -c rclient.c -o rclient4.o

rclient4-stub.c: Map
        ../newpolystubgen/polystubgen 4 > rclient4-stub.c

splitorb6.co: splitorb6.cl splitorb6
        csc splitorb6.cl

splitorb6.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu splitorb6 6 22 18 16 14 4 8 10 12 20 > splitorb6.cl

splitorb6: splitorb6.o
        cc -o splitorb6 splitorb6.o -lith -lm

splitorb6.o: splitorb.c
        cc -DFANOUT=3 -c splitorb.c -o splitorb6.o

splitbkgnd8.co: splitbkgnd8.cl splitbkgnd8
        csc splitbkgnd8.cl

splitbkgnd8.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu splitbkgnd8 8 22 18 16 14 4 6 10 12 20 > splitbkgnd8.cl

splitbkgnd8: splitbkgnd8.o
        cc -o splitbkgnd8 splitbkgnd8.o -lith -lm

splitbkgnd8.o: splitbkgnd.c
        cc -DFANOUT=3 -c splitbkgnd.c -o splitbkgnd8.o

splitparams10.co: splitparams10.cl splitparams10
        csc splitparams10.cl

splitparams10.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu splitparams10 10 22 18 16 14 4 6 8 12 20 > splitparams10.cl

splitparams10: splitparams10.o
        cc -o splitparams10 splitparams10.o -lith -lm

splitparams10.o: splitparams.c
        cc -DFANOUT=3 -c splitparams.c -o splitparams10.o

raymux12.co: raymux12.cl raymux12
        csc raymux12.cl

raymux12.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu raymux12 12 22 18 16 14 4 6 8 10 20 > raymux12.cl

raymux12: raymux12.o
        cc -o raymux12 raymux12.o -lith -lm

raymux12.o: raymux.c
        cc -DFANOUT=3 -c raymux.c -o raymux12.o

demux20.co: demux20.cl demux20
        csc demux20.cl

demux20.cl: Map
        ../polyservgen/polyservgen -h $(LOC) -m xring.cs.umd.edu demux20 20 22 18 16 14 4 6 8 10 12 > demux20.cl

demux20: demux20.o
        cc -o demux20 demux20.o -lith -lm

demux20.o: demux.c
        cc -DFANOUT=3 -c demux.c -o demux20.o
```

Figure 5.25: Makefile specification for the distributed ray tracer (part 2 of 2).

For all configurations, the server consists of the following components[1]

```
the device-dependent (ddx) init program
the device-dependent (ddx) hardware library
the device-independent (dix) xlib library
the operating system independent library
the authentication library
the utility library
the font library
the device-dependent (ddx) framebuffer library
the device-dependent (ddx) machine independent library
the extensions library
```

For instance, the server executable for a DEC 5000 workstation consists of the files

```
ddx/dec/ws/init.c
ddx/dec/ws/libdec.a
dix/libdix.a
os/libos.a
libXau.a
libXdmcp.a
/lib/font/libfont.a
ddx/mfb/libmfb.a
ddx/mi/libmi.a
/server/libext.a
```

The X configuration files are written in IMAKE and are highly dependent on the directory structure of the file system to structure implementation choices. Aside from the initialization program, each subsystem may be implemented by a single library or a set of libraries. These are stored in several subdirectories including the `ddx` (device-dependent) and `dix` (device-independent) directories. Below the device-dependent directory there are subdirectories of files for each vendor platform (e.g., sparc, dec). The pathname of a file used in a specific configuration of the software is found using the UNIX filepath convention. Thus, the initialization program for the server if configured on a DEC machine is `ddx/dec/ws/init.c`. In this case, there is a further differentiation because the `ws` stands for a particular type of DEC machine — a workstation.

This approach is also used by the NMAKE program to select appropriate implementations for components in an application based on environmental constraints, but it cannot handle more complex constraints easily. Software packaging allows developers to associated attributes with nodes (i.e., directories) at any level and establish constraints based on these attributes. The selection of candidate implementations occurs before packaging or during packaging by eliminating implementation that violate constraints.

We present an example of the usefulness of constraints within PACKAGE specifications by "repackaging" the X server. Figure 5.26 and Figure 5.27 show a top-level PACKAGE specification for the X11 server. The PACKAGE specification is organized along the same lines as the directory structure in the standard X Windows software. The major difference is that each component has a set of associated attributes such as ARCH, DIR, FILE, and MON. These attributes specify the properties associated with different implementations such as the machine architecture and type of video monitor.

[1] We do not configure the PEX extension libraries for the sake of brevity of this example.

The production rules specify the system and device constraints that control the selection of components for each execution environment. The two sets of production rules used in this example are shown in Figure 5.28 and Figure 5.29. In Figure 5.28, our environment is a SMCC Sparc workstation with a monochrome video monitor. The constraints related to these attributes are set on lines 7 & 8 of Figure 5.28 and Figure 5.29. The `==` operator states that in all subsequent searches, the ARCH attribute must equal `sparc` and the MON attribute must equal `mono` if specified. If an attribute is not specified by an implementation, it is independent of the constraint. In Figure 5.29, the rules are almost the same except that the environment is a DEC 5000 with a color monitor. The initialization program is different for each monitor. The packager controls the selection of the framebuffer library (either color or monochrome). This demonstrates the use of constraints where a selected implementation controls the selection of another implementation.

If we package the specification in Figure 5.26 and Figure 5.27 using the rules in Figure 5.28 (the Sparc), then the resulting production graph is shown in Figure 5.30. Only those components that are implemented for the Sparc and a monochrome monitor are selected and included in the manufacture graph. Components that are independent of these constraints are also included. Figure 5.31 shows the production graph for the same PACKAGE specification produced using the rules in Figure 5.29. Compared with Figure 5.30, the selected implementations in Figure 5.31 are quite different as a result of the ARCH and MON constraints.

Ports and bindings are not used in this PACKAGE specification. This is to demonstrate that existing applications can be quickly packaged as they exist. This implies that binding problems cannot be identified by the packager, but by later tools such as the link editor if there are missing components or name conflicts. One problem is that the packager must organize the object files and libraries in their correct linking order (lines 24-26 of Figure 5.28). This is accomplished using USE and DEF constraints. All attributes are string values but the `@=` operator (line 52 of Figure 5.28) converts both strings into sets with items separated by blanks. During the packaging phase, the USE = DEF constraint must hold for all searches below the point where the constraint is established. This ensures that the object files and libraries are ordered properly since the constructed manufacture graph consists of a sequence of nested nodes corresponding to libraries and object files that obey this constraint. A subsequent traversal of the constructed manufacture graph will yield the proper sequence for link editing.

The two MAKEFILES that are produced from the two production graphs are shown in Figure 5.32 and Figure 5.33. While these specifications are short, they are complex because of the selection and order of the library implementations. The PACKAGE specification, however, remains the same in this case. Even though the programmer should not have to deal with the production rules, it is interesting to note that the changes to the rules are less complex that those needed to change the Sparc MAKEFILES into the DEC MAKEFILE.

```
include stdpkg.pkg

implement Root as {
      APPNAME = xserver;
      Init init;
      Ddx;     # we don't need an instance name
      Dix;     # if an instance is only one of its
      Os;      # type in a composite implementation
      Auth;
      Util;
      Font;
      Framebuffer fb;
      Mi;
      Ext;
}
implement Init with c_main {
      ARCH     = dec
      DIR      = ddx/$(ARCH)
      FILE     = init.c
}
implement Init with c_main {
      ARCH     = sparc
      MON      = color
      FILE     = sparcInit.c
}
implement Init as {
      ARCH     = sparc
      MON      = mono
      Init;
      InitExt;
}
implement SubInit with c_main {
      ARCH     = sparc
      DIR      = ddx/$(ARCH)
      FILE     = sunInitMono.c
}
implement InitExt with c_func {
      ARCH     = sparc
      DIR      = ddx/$(ARCH)
      FILE     = sparcInitExtMono.c
}
implement Init with c_main {
      ARCH     = sparc
      MON      = color
      DIR      = ddx/$(ARCH)
      FILE     = SparcInit.c
}
```

Figure 5.26: PACKAGE specifications for the X11 window server.

```
implement Ddx with c_lib {
      USE   =
      DEF   =ddx
      ARCH  = dec
      DIR   = ddx/$(ARCH)
      FILE  = libddx.a
}
implement Ddx with c_lib {
      USE   =
      DEF   =ddx
      ARCH  = sparc
      DIR   = ddx/$(ARCH)
      FILE  = libddx.a
}
implement Ddx with c_lib {
      USE   =
      DEF   =ddx util
      ARCH  = x386
      DIR   = ddx/$(ARCH)
      FILE  = libddx.a
}
implement Dix with c_lib {
      USE   =ddx util
      DEF   =dix
      DIR   = dix
      FILE  = libdix.a
}
implement Os with c_lib {
      USE   =dix ddx
      DEF   =os
      DIR   = os
      FILE  = libos.a
}
implement Auth with c_lib {
      USE   =
      DEF   =auth
      DIR   = Xauth
      FILE  = libau.a
}
implement Util with c_lib {
      USE   =
      DEF   =util
      DIR   = ddx/mi
      FILE  = libXdmcp.a
}
implement Font with c_lib {
      USE   =util
      DEF   =font
      DIR   = fonts
      FILE  = libfont.a
}
implement Framebuffer with c_lib {
      USE   =font
      DEF   =fb
      MTYPE = color
      DIR   = ddx/cfb
      FILE  = libcfb.a
}
implement Framebuffer with c_lib {
      USE   =font
      DEF   =fb
      MTYPE = monochrome
      DIR   = ddx/mfb
      FILE  = libmfb.a
}
implement Mi with c_lib {
      USE   =fb
      DEF   =mi
      DIR   = mi
      FILE  = libmi.a
}
implement Ext with c_lib {
      USE   =mi
      DEF   =ext
      DIR   = extensions
      FILE  = libext.a
}
```

Figure 5.27: PACKAGE specifications for the X11 window server (con't).

```
exec               :
                   {
                   %
                   % set constraints on values of ARCH
                   % and MON attributes
                   %
                   ARCH==sparc
                   MON==mono
                   }
                             c_main
                   {
                   USE=
                   DEF=
                   }
                             c_libs
                   {
                   DEF=$2(DEFS)
                   }
                             c_funcs
                   :{
                   CC=cc
                   #all:  $1.APPNAME
                   #
                   #$1.APPNAME: $1.DIR/$1.FILE:r.o $3(OBJS)
                   #         $(CC) -o $1.APPNAME $1.DIR/$1.FILE:r.o $3(OBJS) $2(LIBS)
                   #
                   #$1.DIR/$1.FILE:r.o:  $1.DIR/$1.FILE
                   #         $(CC) -c $1.DIR/$1.FILE
                   #
                   $2
                   $3
                   }
                   ;

c_funcs            :         c_func c_funcs
                   [ (OBJS) $2(OBJS) " " $1.DIR "/" $1.FILE:r ".o" ]
                   :{
                   #$1.DIR/$1.FILE:r.o:  $1.DIR/$1.FILE
                   #         $(CC) -c $1.DIR/$1.FILE
                   #
                   $2
                   }
                   |
                   ;

c_libs             :
                   {
                   %
                   % establish constraint that USEs is a subset of DEFs
                   % to ensure that libraries are ordered correctly
                   %
                   USE @= $(DEF)
                   }
                             c_lib
                   {
                   DEF = $(DEF) $1.DEF
                   }
                             c_libs
                   [ (LIBS) $2(LIBS) " " $1.DIR "/" $1.FILE ]
                   |
                   ;
```

Figure 5.28: Rule specifications for the X11 window server example

```
exec                   :
                       {
                       %
                       % set constraints on values of ARCH
                       % and MON attributes
                       %
                       ARCH==dec
                       MON==color
                       }
                                 c_main
                       {
                       USE=
                       DEF=
                       }
                                 c_libs
                       {
                       DEF=$2(DEFS)
                       }
                                 c_funcs
                       :{
                       CC=cc
                       #all:  $1.APPNAME
                       #
                       #$1.APPNAME: $1.DIR/$1.FILE:r.o $3(OBJS)
                       #         $(CC) -o $1.APPNAME $1.DIR/$1.FILE:r.o $3(OBJS) $2(LIBS)
                       #
                       #$1.DIR/$1.FILE:r.o:  $1.DIR/$1.FILE
                       #         $(CC) -c $1.DIR/$1.FILE
                       #
                       $2
                       $3
                       }
                       ;

c_funcs                :         c_func c_funcs
                       [ (OBJS) $2(OBJS) " " $1.DIR "/" $1.FILE:r ".o" ]
                       :{
                       #$1.DIR/$1.FILE:r.o:  $1.DIR/$1.FILE
                       #         $(CC) -c $1.DIR/$1.FILE
                       #
                       $2
                       }
                       |
                       ;

c_libs                 :
                       {
                       %
                       % establish constraint that USEs is a subset of DEFs
                       % to ensure that libraries are ordered correctly
                       %
                       USE @= $(DEF)
                       }
                                 c_lib
                       {
                       DEF = $(DEF) $1.DEF
                       }
                                 c_libs
                       [ (LIBS) $2(LIBS) " " $1.DIR "/" $1.FILE ]
                       |
                       ;
```

Figure 5.29: Rule specifications for the X11 window server example

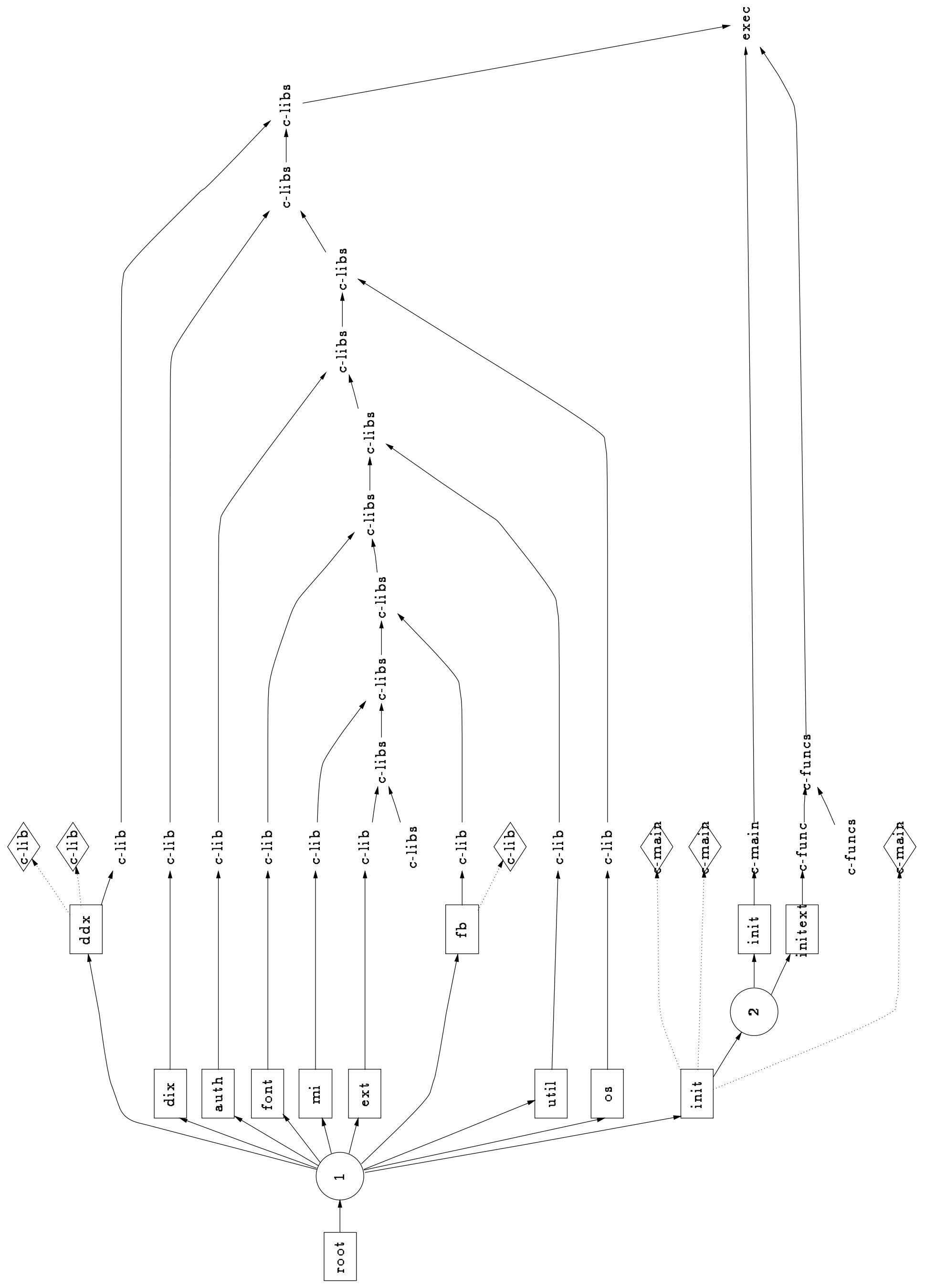

Figure 5.30: Production graph for the X Windows server (for Sun workstation).

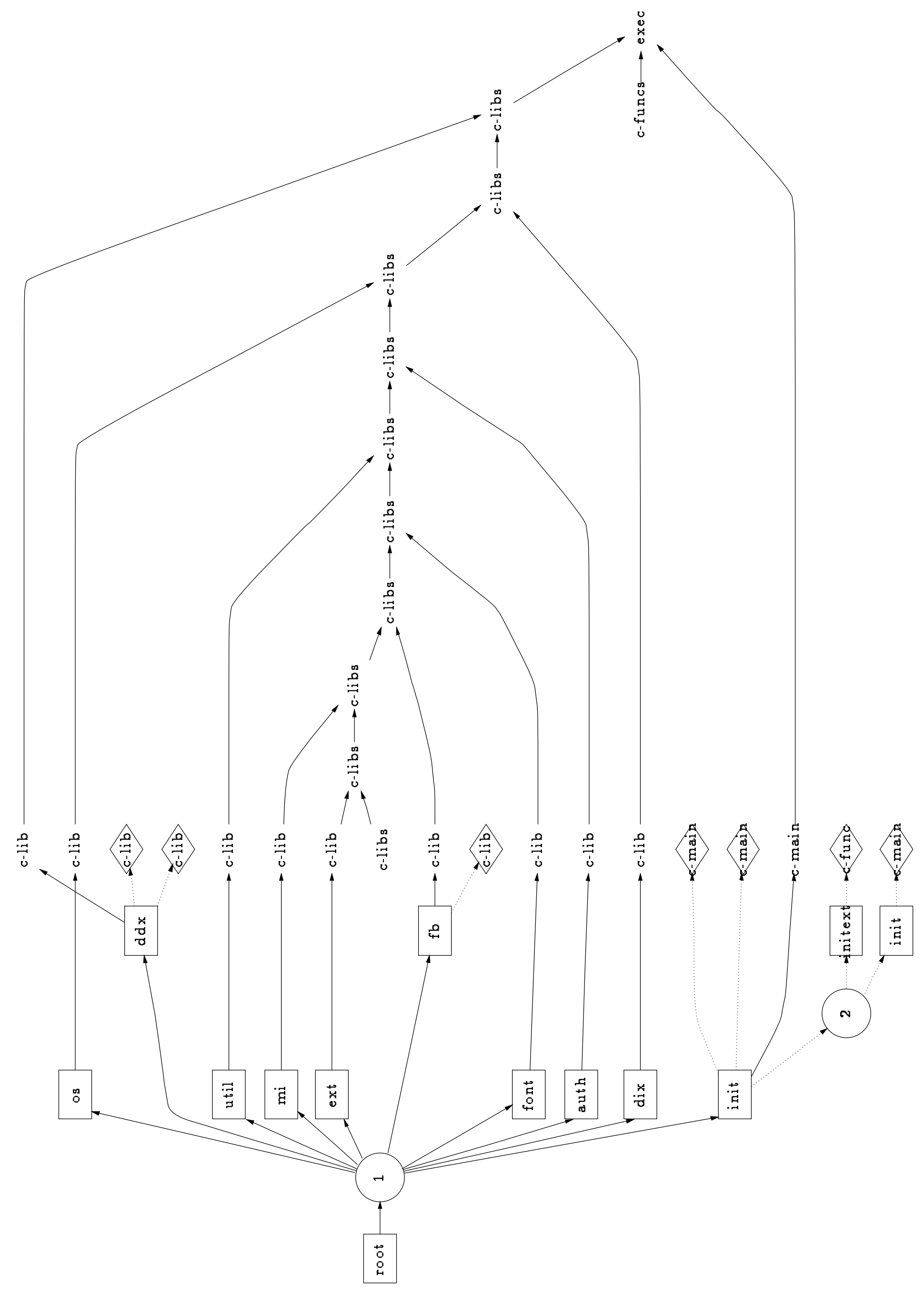

Figure 5.31: Production graph for the X Windows server (for DEC workstation).

```
all:  xserver

xserver:  ddx/sparc/sparcInitMono.o ddx/sparc/sparcInitExtMono.o
      cc -o xserver ddx/sparc/sparcInitMono.o ddx/sparc/sparcInitExtMono.o \
      extensions/libext.a mi/libmi.a ddx/mfb/libmfb.a fonts/libfont.a \
      ddx/mi/libXdmcp.a Xauth/libau.a os/libos.a dix/libdix.a \
      ddx/sparc/libddx.a

ddx/sparc/sparcInitMono.o:  ddx/sparc/sparcInitMono.c
      cc -c ddx/sparc/sparcInitMono.c

ddx/sparc/sparcInitExtMono.o:  ddx/sparc/sparcInitExtMono.c
      cc -c ddx/sparc/sparcInitExtMono.c
```

Figure 5.32: Makefile specification for the X Windows server (Sparc version).

```
all:  xserver

xserver:  ddx/dec/init.o
      cc -o xserver ddx/dec/init.o extensions/libext.a \
      mi/libmi.a ddx/mfb/libmfb.a fonts/libfont.a \
      ddx/mi/libXdmcp.a Xauth/libau.a os/libos.a \
      dix/libdix.a ddx/dec/libddx.a

ddx/dec/init.o:  ddx/dec/init.c
      cc -c ddx/dec/init.c
```

Figure 5.33: Makefile specification for the X Windows server (DEC version).

Chapter 6

Conclusions

Software packaging offers seamless integration to developers of software systems whose components are programmed in different languages, are distributed, or whose application must be ported to many execution environments. It is independent of any specific integration technology and extensible to include new technologies. Our use of this approach in our research has many current applications and possible future directions.

6.1 Applications

Our case studies and examples allude to some uses of software packaging to solve problems of heterogeneity and distribution when porting or configuring an application. The following is a list of possible application areas for software packaging. Our approach is extensible to many areas of integration problems in computing systems. We summarize these uses and some additional application areas for this technology.

6.1.1 Portability

By including alternate implementations of components and structuring an application so that only a few components must be reimplemented, developers can minimize the cost of porting applications between environments. Software packaging allows developers to organize applications in this fashion independent from the way an application is integrated. Current methods do not separate these tasks and embed the structure of a program within the integration steps.

6.1.2 Heterogeneity

The packager rules provided by the environment determine what types of “building blocks” are compatible and how to bridge their differences in a runtime environment. Connections between resource uses and their definitions can be manipulated abstractly without concern for their implementations. The packager determines the appropriate mechanism based on the total context of the application’s runtime environment. In this approach, connections are viewed as first-class entities in order to promote their abstraction from implementation.

6.1.3 Distribution

Like the heterogeneous components, distributed components that execute on different physical processors at runtime in an application need bridges to connect with one another.

6.1.4 Version control

If we associate a `VERSION` attribute with a primitive implementation, we can view different versions of code as alternate implementations. Constraints within a package specification and production rules may restrict the choices of implementations to thise with the same version.

6.1.5 Genericity

So far, the target object type has been an executable object (`exec`), but we are not limited to this situation. Libraries, documents, and databases can be packaged as well. For example, we may package an existing executable with input and output files to construct a single test in a test suite. Different "implementations" of the input and output components represent different test cases. Constraints on the input and output would ensure that the appropriate cases are matched together.

6.2 Future Directions

Our experience with the software packager has shown that it is a useful tool that elides the integration problems in heterogeneous, distributed environments. It is extensible so that future integration tools can be leverage when developed. We have plans to explore many avenues of integration gives this tool. We briefly summarize these areas.

6.2.1 Integration tools

Existing integration tools including OMG [OMG90] and Tk/Tcl can be incoporated into packaging production rules to demonstrate extensibility of the approach.

6.2.2 Incremental integration

When a component is reimplemented, the entire package must be regenerated and the integration must be build from the primitive implementations. An incremental approach is needed to reduce the costs of repackaging an application.

6.2.3 Graphical specification

A few developers who have used PACKAGE specifications have remarked that it is useful to visualize the structure graph when designing their applications, but the textual language does not provide the appropriate views. Several graphical tools have been developed to construct structure graphs through direct manipulation but none of these has been satisfactory to date. The reasons for this is primarily that the structure graph formalism itself was under evolution. A new graphical tool is needed to provide the visualization required by programmers.

6.2.4 Portability metrics

Since well-structured applications are modularized in such a way that reimplementation of a component has a low impact on the system as a whole, we suggest that portability can be quantified through measurements on software structure graphs. Such metrics would support claims of portability with values and give programmers guidelines during development regarding their design.

The software packager was built primarily to help integrate heterogenous, distributed programs in a fashion that hides the details of integration from the programmer. This was done to allow the programmer greater freedom in exploring alternate implementations quickly for prototyping. Our approach meets this requirement and opens up new areas of design methods.

6.3 Summary

Programming-in-the-large has long been a vision of many programmers who have wished to reuse existing software components by combining them together in a modular fashion, but are stopped by the barriers of heterogeneity. Configuration programming languages and IPC mechanisms allow developers to combine existing software into new applications at a modular level, but they rely on the programmer's knowledge of the component types and the capabilities of integration tools. The result is that the integration itself is a complex programming task that is just as difficult as programming-in-the-small.

Software packaging promotes the view of a software application as a modular collection of subsystems and alternate implementations. This view is practically applicable and in agreement with current experiences of software developers. Most software products that are portable and configurable in heterogeneous, distributed environments have mutliple implementations and are structured in a modular fashion to isolate dependencies with the design.

It is difficult to show that a programming language is abstract insofar that convenience to the programmer is increased. Convenience is defined loosely in terms of how "terse" it is for the programmer to specify a solution. The best we can hope to do is show that a new approach is more convenient than existing methods. Software packaging reduces the amount of work programmers must do to integrate applications in heterogeneous, distributed environments. By dealing with connections abstractly and using software packaging to infer how to implement the connections and select compatible implementations, the software developer reduces the amount of work necessary and reuses the bridges built by other developers. Software packaging also allows disparate applications to be composed and speeds the sythesis of new applications. Previous methods provided much of the integration technology, but left the programmer to specify the use of these tools explicitly. Software packaging relates such tools to the integration processes in an environment.

Software packaging embraces diversity and allows developers focus on composing different programs while ignoring incompatibilities that exist between them. This is important when changing the configuration of an application: porting it to new platforms, adding new implementations of components or features, and distributing it across a network of computers. This transparency, previously available only in homogeneous software development environments, is of most value when prototyping new applications from existing software. Often, prototype applications consist of existing programs that have been "patched" together. If the prototype is viable, components may be reimplemented so that they are more tightly bound together in a runtime configuration.

Finally, software packaging represents an extensible framework for software integration. It does not favor any particular environment, protocol, or programming language. Such mechanisms must be available, however, in order to integrate any application. This permits specialized tools to coexist and bridges to be built when more general standards do not exist. As new standards and tools are developed, the packager rules can be updated and existing applications can be repackaged.

Chapter 7

Glossary

The following is a list of terms and their definitions that introduced and used in this text.

abstract manufacture graph Integration processes in an execution environment characterized by production rules.

binding A binding is an instance of a `bind` connector between two ports (between module instances) in a composite implementation.

bridge Variant of *wrapper*. Term used loosely to describe implementation of a connection between two module instance ports. A bridge typically involves distributed components.

class A description of a structure in a PACKAGE specification.

component Any software artifact — an executable program, source file, services, etc.

composite implementation A PACKAGE specification unit that implements a module with a subsystem of module instances and connections between ports of those module instances.

connector A connector is any link that associated two or more module instance ports in a composite implementation.

converter A bridge that provides data conversion.

glue A stub or wrapper that implements a set of connectors.

implementation See *primitive* and *composite* implementations.

integrate To transform and combine.

item[integration process] The sequence (i.e., partial order) of tool invocations necessary to build a new software artifact from existing components.

instance See *module instance.*

manufacture graph See *software manufacture graph.*

module A "black box" that is described by a name, parameters, attributes, and ports.

module instance A copy of a module that is a member of a subsystem (within a composite implementation).

object An object is a declaration of an implementation type. File extensions are primitive object types in many current programming environments.

package A collection of software components, their interconnections, stubs, wrappers, *and* instructions needed to build the target system in a particular environment.

port A resource required or defined by a module.

port type A type of resource required or defined by a module. Some types of ports include `use`, `def`, `snk` (for asynchronous data "sink"), and `src` (for asychronous data "source").

primitive implementation A primitive implementation corresponds to a piece of source code, an existing service, or any base type of object that cannot be described as a composite of more primitive components in a programming environment.

production graph See *software production graph.*

seam Additional software needed to integrate software components.

item[software manufacture graph] Characterizes a specific integration process (i.e., partial order).

software production graph Union of a software structure graph and a software manufacture graph joined at the primitive implementation level.

software structure graph Description of the logical structure of a software artifact in terms of its (possibly multiple) implementations.

structure graph See *software structure graph.*

stub Additional code (usually generated from specifications) needed by remote procedure call (RPC) to perform bookkeeping and communications in distributed programming.

wrapper Additional code used to transform existign software component to a particular execution context.

Bibliography

[Apol83] Apollo Computer Inc., The Network Interface Definition Language, NIDL-1, Cupertino, CA, September 1983.

[BaBG88] Batory, D., J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise, GENESIS: An Extensible Database Management System, *IEEE Transactions on Software Engineering*, November 1988, Volume 14, Number 11, pp. 1711-1730.

[Begu90] Beguelin, A., The Parallel Virtual Machine (PVM) Environment, Oak Ridge National Laboratory Technical Report ORNL/TM-11826, July 1991.

[Bers84] Bersoff, E., Elements of Software Configuration Management, *IEEE Transactions on Software Engineering*, January 1984, Volume 10, Number 1, pp. 79-87.

[BiFo88] Bisiani, R. and A. Forin, Multilanguage Parallel Programming of Heterogeneous Machines, *IEEE Transactions on Computers*, August 1988, Volume 37, Number 8, pp. 930-945.

[BLLN85] Black, A., E. Lazowska, H. Levy, D. Notkin, J. Sanislo, J. Zahorjan, An Approach to Accommodating Heterogeneity, University of Washington Department of Computer Science Technical Report TR-85-10-04, October 1985.

[BHJL87] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter, Distribution and Abstract Types in Emerald, *IEEE Transactions on Software Engineering*, January 1987, Volume 13, Number 1, pp. 65-76.

[Bloo91] Bloomer, J., **RPC**, O'Reilly and Associates, Sebastopol, CA, 1991.

[Bori89] Borison, E., Program Changes and the Cost of Selective Recompilation, Carnegie Mellon University, Ph.D. Thesis, July 1989.

[CaPu90] Callahan, J. and J. Purtilo, A Packaging System for Heterogeneous Execution Environments, University of Maryland Computer Science Department Technical Report CS-TR-2542, October 1990.

[CaPu91] Callahan, J. and J. Purtilo, A Packaging Systems for Heterogeneous Execution Environments, *IEEE Transactions on Software Engineering*, June 1991, Volume 17, Number 6, pp. 626-635.

[Coop79] Cooprider, L., The Representation of Families of Software Systems, Carnegie Mellon University Computer Science Department, Ph.D. Thesis, January 1979.

[CrWW80] Cristofor, E., T. Wendt, and B. Wonsiewicz, Source control + tools = stable systems, *in Proceedings of the 4th Computer Science and Applications Conference*, Octiber 1980, pp. 527-532.

[DeKr76] DeRemer, F. and H. Kron, Programming-in-the-large versus Programming-in-the-small, *IEEE Transactions on Software Engineering*, June 1976, Volume 2, Number 6, pp. 80-86.

[ELNS92] Engels, G., C. Leweentz, M. Nagl, W. Schafer, and A. Schurr, Building Integrated Software Development Environments Part I: Tool Specification, *ACM Transactions on Software Engineering and Methodology*, April 1992, Volume 1, Number 2, pp. 135-167.

[ErPe84] Erickson, V. and J. Pellegrin, Build: A Software Construction Tool, *AT&T Bell Laboratories Technical Journal*, July-August 1984, Volume 63, Number 6, pp. 12-22.

[FeDD88] Feiler, P., S. Dart, and G. Downey, Evaluation of the Rational Environment, Carnegie Mellon University School of Computer Science Technical Report CMU/SEI-88-TR-15, July 1988.

[Feld79] Feldman, S., Make: A Program for Maintaining Computer Programs, *Software Practice and Experience*, April 1979, Volume 9, Number 4, pp. 255-265.

[Fowl85] Fowler, G., The Fourth Generation Make, *in Proceedings of the USENIX Conference — Summer '85*, June 1985, pp. 159-174.

[GaKN92] Garlan, D., G. Kaiser, D. Notkin, Using Tool Abstraction to Compose Systems, *IEEE Computer*, June 1992, Volume 25, Number 6, pp. 30-38.

[Gele85] Gelernter, D., Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, January 1985, Volume 7, Number 1, pp. 80-112.

[Gibb87] Gibbons, P., A Stub Generator for Multilanguage RPC in Heterogeneous Environments, *IEEE Transactions on Software Engineering*, January 1987, Volume 13, Number 1, pp. 77-87.

[Gorl90] Gorlen, K., S. Orlow, and P. Plexico, **Data Abstraction and Object-Oriented Programming in C++**, John Wiley & Sons, New York, NY, 1990.

[HaWe91] Harms, D. and B. Weide, Copying and Swapping: Influences on the Design of Reusable Software Components, *IEEE Transactions on Software Engineering*, May 1991, Volume 17, Number 5, pp. 424-435.

[HeLi82] Herlihy, M. and B. Liskov, A Value Transmission Method for Abstract Data Types, *ACM Transactions on Programming Languages and Systems*, October 1982, Volume 4, Number 4, pp. 527-551.

[Herl80] Herlihy, M., Transmitting Abstract Values in Messages, Massachusetts Institute of Technology, Ph.D. Thesis, 1980.

[Hoar85] Hoare, C.A.R., **Communicating Sequential Processes**, Prentice Hall, Englewood Cliffs, NJ, 1985.

[Hume87] Hume, A., Mk: A Successor to Make, *in Proceedings of the USENIX Conference Summer 1987*, June 1987, pp. 445-457.

[JoRT85] Jones, J., R. Rashid, M. Thompson, Matchmaker: An interface specification language for distributed processing, *in Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1985, pp. 225-235.

[KaHa83] Kaiser, G. and A. Habermann, An Environment for System Version Control, *in Proceedings of the COMPCON Spring '83*, February 1983, pp. 415-420.

[Lern91] Lerner, R., Specifying Objects of Concurrent Systems, Carnegie Mellon University School of Computer Science, Ph.D. Thesis, May 1991.

[Lewi83] Lewis, B., Experience with a System for Controlling Software Versions in a Distributed Environment, *in Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development*, November 1983, pp. 210-219.

[MaKS89] Magee, J., J. Kramer, and M. Sloman, Constructing Distributed Systems in Conic, *IEEE Transactions on Software Engineering*, June 1989, Volume 15, Number 6, pp. 663-675.

[MaLa89] Mahler, A. and A. Lampen, An Integrated Toolset for Engineering Software Configurations, *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 1988, pp. 191-200.

[McNu91] McNutt, D., Imake: Friend or Foe?, *SunExpert*, November 1991, Volume 2, Number 11, pp. 46-50.

[NaSc87] Narayanaswamy, K. and W. Scacchi, Maintaining Configurations of Evolving Software Systems, *IEEE Transactions on Software Engineering*, March 1987, Volume 13, Number 3, pp. 324-334.

[Nels81] Nelson, B., Remote Procedure Call, Carnegie Mellon University Department of Computer Science, Ph.D. Thesis, May 1981.

[Nels91] Nelson, G., **Systems Programming with Modula-3**, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Notk90] Notkin, D., Proxies: A Software Structure for Accommodating Heterogeneity, *Software Practice and Experience*, April 1990, Volume 20, Number 4, pp. 357-364.

[NoBL88] Notkin, D., A. Black, E. Lazowska, H. Levy, J. Sanislo, J. Zahorjan, Interconnecting Heterogeneous Computer Systems, *Communications of the ACM*, March 1988, Volume 31, Number 3, pp. 258-273.

[OMG90] Object Management Group Inc., Object Management Architecture Guide 1.0, OMG TC Document 90.9.1, 492 Old Connecticut Path, Framingham, MA 01701.

[Parn72] Parnas, D., On the Criteria To Be Used in Decomposing a System into Modules, *Communications of the ACM*, December 1972, Volume 15, Number 12, pp. 1053-1058.

[Perr89] Perry, D., The Inscape Environment, *in Proceedings of the IEEE 11th International Conference on Software Engineering*, May 1989, pp. 2-12.

[PuCa89] Purtilo, J. and J. Callahan, Parse Tree Annotations, *Communications of the ACM*, December 1989, Volume 32, Number 12, pp. 1467-1477.

[Purt85] Purtilo, J., Polylith: An Environment to Support Management of Tool Interfaces, *in Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, (Seattle, WA, June 25-28 1985), pp. 12-18.

[Reis90] Reiss, S., Connecting Tools Using Message Passing in the Field Environment, *IEEE Software*, July 1990, Volume 7, Number 4, pp. 57-66.

[Smit91] Smith, D., **Make**, O'Reilly and Associates, Sebastopol, CA, 1991.

[Snod89] Snodgrass, R., **The Interface Description Language**, Computer Science Press, Rockville, MD, 1989.

[StGi90] Stamos, J. and D. Gifford, Implementing Remote Evaluation, *IEEE Transactions on Software Engineering*, July 1990, Volume 16, Number 7, pp. 710-722.

[StKH86] Staudt, B., C. Krueger, A. Habermann, V. Ambriola, The GANDALF System Reference Manuals, Carnegie Mellon University School of Computer Science Technical Report CMU-CS-86-130, May 1986.

[Stra92] Strassmann, P., The Future Strategy of DoD's Computer Czar, *Washington Technology*, July 30, 1992, pp. 21-28.

[Stro86] Stroustrup, B., **The C++ Programming Language**, Addison-Wesley, Reading, MA, 1986.

[Sun85a] Sun Microsystems Computer Corp., External Data Representation Reference Manual, , Mountain View, CA, January 1985.

[Sun85b] Sun Microsystems Computer Corp., Remote Procedure Call Protocol Specification, , Mountain View, CA, January 1985.

[Sun89] Sun Microsystems Computer Corp., The Network Software Environment, NSE, Mountain View, CA, 1989.

[Sutt91] Sutton, S., D. Heimbigner, L. Osterweil, Managing Change in Software Development Through Process Programming, University of Colorado Technical Report CU-CS-531-91, June 1991.

[SwBa82] Swartout, W. and R. Balzer, On the Inevitable Intertwining of Specification and Implementation, *Communications of the ACM*, July 1982, Volume 25, Number 7, pp. 438-440.

[Terr87] Terry, D., Caching Hints in Distributed Systems, *IEEE Transactions on Software Engineering*, January 1987, Volume 13, Number 1, pp. 48-54.

[Vera89] The Sema Group, The Software Bus — its Objective: the Mutual Integration of Distributed Software Engineering Tools, PCTE-114, Trafalgar House, Richfield Ave., Reading, Berkshire RG18QA, UK, May 1989.

[Tich80] Tichy, W., Software Development Control Based on System Structure Description, Carnegie Mellon University Computer Science Department, Ph.D. Thesis, January 1980.

[WaHK88] Waite, W., V. Heuring, and U. Kastens, Configuration Control in Compiler Construction, *in Proceedings of the International Workshop on Software Version and Configuration Control*, January 1988, pp. 159-168.

[Wall87] Wall, L., Configuring the X11 Window System, *in Proceedings of the USENIX Summer 1987*, July 1987, pp. 161-190.

[Wegn90] Wegner, P., Object Oriented Programming, *OOPS Messenger*, November 1990, Volume 1, Number 4, pp. 3-54.

[WeOZ91] Weide, B., W. Ogden, S. Zweben, **Advances in Computers: Reuable Software Components**, Academic Press, New York, NY, 1991.

[Weis90] Weiser, M., PCR: The Portable Common Runtime Environment, *Software Practice and Experience*, May 1990, Volume 18, Number 5, pp. 112-130.

[Xero81] Xerox Corp., Courier: The Remote Procedure Call Protocol, XSIS 038112, Stamford, CT 06904, December 1981.