

# Distributed Development of Complex Software Systems with Object Make

Yasuhiro Sugiyama  
Department of Computer Science  
Nihon University  
Koriyama, Japan  
sugiyama@ce.nihon-u.ac.jp

## Abstract

*Production of a large-scale software system involves quite a few software components. It is very common to develop such software components in a distributed environment consisting of multiple computer systems connected by computer networks. In order to support effective software development in a distributed environment, a mechanism to combine a large number of components, which are distributed over the network, into a single system in a systematic way is quite essential. Object Make is a tool that automates the building process of software systems consisting of software components that are stored in the file systems of distributed computer systems, as if they are stored in a single file system. Software developers are not required to transfer these remote components to their own local computer systems explicitly, even if the computer systems do not share a common file system. This paper will give an overview of the distributed software build process for large-scale software systems using Object Make.*

## 1. Introduction

Production of a large-scale software system involves quite a few software components. It is very common to develop such software systems in an environment consisting of multiple computer systems connected by computer networks. These computers may be located in a single office, or may be geographically distributed. Software engineers own their own computer systems, and develop these software components on their own computer systems independently. These separately developed software components are combined into a single large software system.

In such cases, the way of sharing of separately built software components is one of the critical issues to be considered for an effective software development process. If these computers share a single common file system by

NFS, for example, direct file sharing is one of the possible solutions. Files being developed by each software engineer are supposed to be accessible for other software engineers for system integration. In this case, however, precise access control of shared files is quite desirable, because people may accidentally and/or intentionally read, modify, and/or destroy other people's files.

In case when the computers do not share their file systems, software engineers are required to transfer necessary files manually and explicitly using **ftp** and/or some other methods. It is quite costly, however, to exchange files during the development processes.

In order to support effective software development in a distributed environment, a mechanism to combine a large number of components, which are distributed over the network, into a single system is quite essential. It is quite desirable that the mechanism works as if all the software components are locally available, even if some of them are stored in remote computers' file system.

Make [3] is a tool that is widely used to automate the building and re-building processes of software systems. I have a long experience in using Make. However, I have noticed several difficulties of Make, when I use it for the development of large-scale software systems. Particularly, it is not easy to reuse the components of a system that was built with Make. Moreover, Make is not designed to be used in distributed environments.

I have developed a tool, called Object Make to overcome the problems of Make without losing the compatibility with Make. The first version of Object Make [15] was built in order to (i) help software developers to build reusable software components, and (ii) to offer a mechanism to combine a large number of components into a system in a systematic way. Object Make is now in its second generation. Object Make now allows software developers to combine software components that are stored in the file systems of remote computer systems, as if they are stored in the developer's local file system. Software developers are not required to transfer these

```

tree.make
tree:  tree.o parse.o gen.o tree.make
      cc tree.o parse.o gen.o -o tree

tree.o.make
tree.o: tree.c tree.o.make
       cc tree.c -c -o tree.o

gen.o.make
gen.o:  gen.c gen.o.make
       cc gen.c -c -o gen.o

parse.o.make
parse.o: parse.c lex.c parse.o.make
        cc parse.c -c -o parse.o
parse.c: parse.y parse.o.make
        yacc parse.y -o parse.c

lex.c.make
lex.c:  lex.l lex.c.make
       lex lex.l -o lex.c

```

**Figure 1. Description files for Object Make**

remote components to their own local computer systems explicitly.

This paper gives an overview of Object Make focusing on its use in distributed environments. I will show how one can use Object Make to develop software systems by integrating software components distributed over networks.

This paper is organized as follows. The second section gives an overview of Object Make for non-distributed use. The third section will describe the distributed software build mechanism of Object Make, including the current implementation of Object Make. The fourth section will conclude the paper with my experience report on using Object Make, and my future plans for extending the capability of Object Make based upon the experience.

## 2. An overview of Object Make in non-distributed environments

This section gives an overview of Object Make focusing on its use in non-distributed environment.

A large-scale software system consists of quite a few software components. These software components, in their turns, consist of other smaller components. These components are integrated in a hierarchical fashion to form a single software system or a larger component. Object Make is a tool that automates the building processes of software systems as well as their constituent software

components. Object Make is also capable of rebuilding software systems with a minimal cost when changes are made to some of the constituent components.

Please note that Object Make does not distinguish “components” from “systems” as far as their build processes are concerned. This is because a system, which consists of multiple components, may be a constituent component of a larger system.

### 2.1. Build rules

Object Make builds or rebuilds software components based upon *build rules* specified by the designers of the components. Build rules are stored in files, called *description files*. Object Make reads a description file and builds a component based upon the build rules in the description file. Object Make assumes that each software component has its own description file. The standard name of the description file of a component is given by adding the extension “.make” to the name of the component. For instance, the description file of **tree** is named **tree.make**.

Figure 1 shows sample description files to build a simple compiler. The compiler, called **tree**, is produced from three object files: **tree.o** that contains a command analyzer, a parser **parse.o**, and a code generator **gen.o**. These three object files are components to be used to build the compiler. The description file of **tree** is named **tree.make**.

The three object files are in their turns produced from other software components. The description file of **parse.o** is named **parse.o.make**, while the description file of **gen.o** is named **gen.o.make**, and so forth. The parser relies on a lexical analyzer **lex.c** to read source files to compile. The source file **lex.c** is included from **parse.c** when **parse.c** is compiled to produce **parse.o**. The lexical analyzer **lex.c** is also a software component produced from its source file **lex.l** by a lexical analyzer generator **lex**. The description file of **lex.c** is named **lex.c.make**.

Build rules of a system or a component state the *dependency* that lists the software components to be used to build the system or the component, as well as the *build commands* to combine and/or transform the components into the target system or component. The first line of the description file **tree.make** states that **tree** depends on the three object files, as well as the description file **tree.make**. The second line is the build command to link the three object files to produce the executable file **tree**.

Dependencies may include components that are not explicitly specified in their associated build command. For instance, **parse.o.make** states that **parse.o** depends on **lex.c** which is included from **parse.c** at the compile time. But, in the build command to produce **parse.o**, **lex.c** is not explicitly specified at all.

The command to build a system or a component with Object Make takes the name of the system or the component as an invocation parameter to Object Make. The executable file of Object Make is named **omake**. For instance, Object Make will start building the system **tree** in Figure 1 by the following command:

**omake tree**

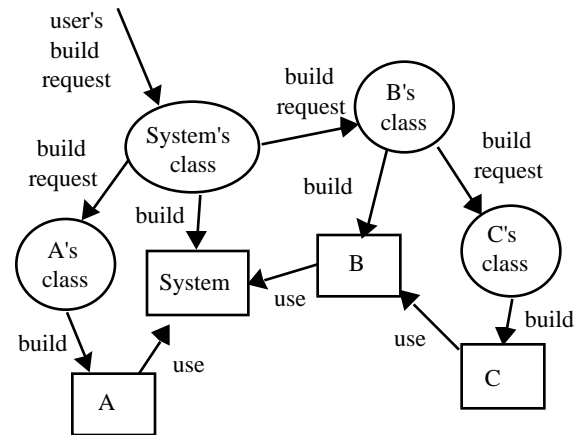
When the command above is invoked, Object Make will look for the description file **tree.make** first. If the description file is found, Object Make will start building **tree** using the build rules in the description file. If **tree** does not exist yet, Object Make will build it by executing the build command in the second line of the description file **tree.make**. If **tree** exists, Object Make takes a look at the time stamp of **tree** and the four components on which **tree** depends. If **tree** is older than any of the three object files or the description file, Object Make will rebuild it.

Please note that it is a common style of defining a description file to state that the component to be built by the description file depends on the description file itself. As a result, when the description file is modified, Object Make is able to update the components that relate to the modified description file.

## 2.2. Chaining build rules

Object Make is able to establish the chaining of build rules even if those rules are included in different description files, in contrast Make does not establish chaining of build rules in different description files. In the example of Figure 1, the production of **tree** requires the three object files: **tree.o**, **parse.o**, and **gen.o**. In order to execute the build command to produce **tree** successfully, **tree.o**, **parse.o**, and **gen.o** must exist. In case some of them do not exist, Object Make will try to build them automatically. Object Make will first search for their build rules in **tree.make**. However, **tree.make** does not include the build rules for these object files. Object Make, then, looks for their description files **tree.o.make**, **parse.o.make**, and **gen.o.make**. Since these description files exist, Object Make will use the build rules in these files. Consequently, the chaining from **tree.make** to **tree.o.make**, **parse.o.make**, and **gen.o.make** is established.

Moreover, chaining of two build rules is often extended to third or fourth build rules. The first line of the description file **parse.o.make** states that **parse.o** depends on its source file **parse.c**, as well as the lexical analyzer source file **lex.c** and the description file **parse.o.make**. As far as **parse.c** is concerned, this description file also contains the build rules to build **parse.c** from its



**Figure 2. The object-oriented build process model for Object Make**

corresponding source file **parse.y**. The third and fourth lines state that **parse.c** is generated from **parse.y** by **Yacc**, a parser generator. Object Make will use this build rule to produce **parse.c**. On the other hand, the description file does not include the build rule for **lex.c**. Object Make looks for **lex.c.make** and will use this description file to produce **lex.c**. As a result, chaining of the three description files is established in a hierarchical fashion.

## 2.3. The Build Process Model of Object Make

Object Make is built on top of the build process model developed in [13][14]. The theory behind Object Make is object-oriented approach. Object Make treats software components as if they are objects that are instances of their classes.

The object-oriented approach is commonly characterized by (i) encapsulation and (ii) information sharing by inheritance or delegation [8]. The object-oriented build process model tries to encapsulate build rules of objects in their classes, and shares the encapsulated build rules by delegation. The build rule of each component is hidden from the outside of the class. Users of a component do not need to know how to build the component. They just need to send a request to the class to build the component.

Delegation shares build rules by forwarding build requests to the classes that own the build rules. Each class includes the build rules of the components of the class. The class does not include the build rules of other components that are necessary to build the component of the class. When a class needs some other components to produce its instance, the class will suspend its building process temporarily, and will send requests to the classes

	Object Make	Make
Reuse of description file	Easy	Not easy
Precise build control	Component-wise	System-wise
Chaining build rules	May involve multiple description files	Within a single description file

**Table 1. A comparison of Object Make and Make**

of the components to produce the required components. The class resumes its build process only after all the necessary components are ready for use.

For instance, Figure 2 illustrates the building process of a system consisting of two components: **A** and **B** that is produced from a third component **C**. A user of the system will send a request to the class of the system to build the system. The class of the system will send requests to the classes of **A** and **B** to build their instances before it starts producing the system. When the class of **A** receives the request, it starts building **A**. However, the class of **B** does not start building **B**, because **C** is necessary to build **B**. The class of **B** sends a request to the class of **C** to build **C**. When **C** is built, the class of **B** builds **B** from **C**. The class of the system starts building the system only after **A** and **B** are made.

In Object Make, a description file corresponds to a class, and the software components built by the description file correspond to the instances of the class. Encapsulation is implemented in terms of the naming rule, which I stated in the previous section. The naming rule allows that the description file of a component can be uniquely identified, as the class of an object can be uniquely identified. Although the contents of description files are not hidden from the outside, users are allowed to build components without knowing the details of the build rules.

Delegation is implemented as the chaining of build rules that allows reuse of software components in multiple software systems without duplicating the build rules of the shared components in the multiple systems.

## 2.4. Advantages over Make

This section will give a brief summary of advantages of Object Make over traditional Make focusing on the use in non-distributed environment. A full detail of advantages of Object Make in non-distributed use is already reported in [15]. Table 1 shows a summary of comparisons of Object Make and Make in three aspects.

Typical use of Make is to create a single description file named **Makefile** for the entire system. As a result, description files tend to be large. Large description files result in the following difficulties.

### 2.4.1. Reusing build rules

When I reuse a component of a system in other systems, I often wish to reuse its build rule in the system's description file. However, Make does not allow me to reuse build rules straightforwardly. When the description file includes the build rules of two or more components, first I must understand how the component is currently used in the system, and then I need to extract the necessary build rule of the component from the description file.

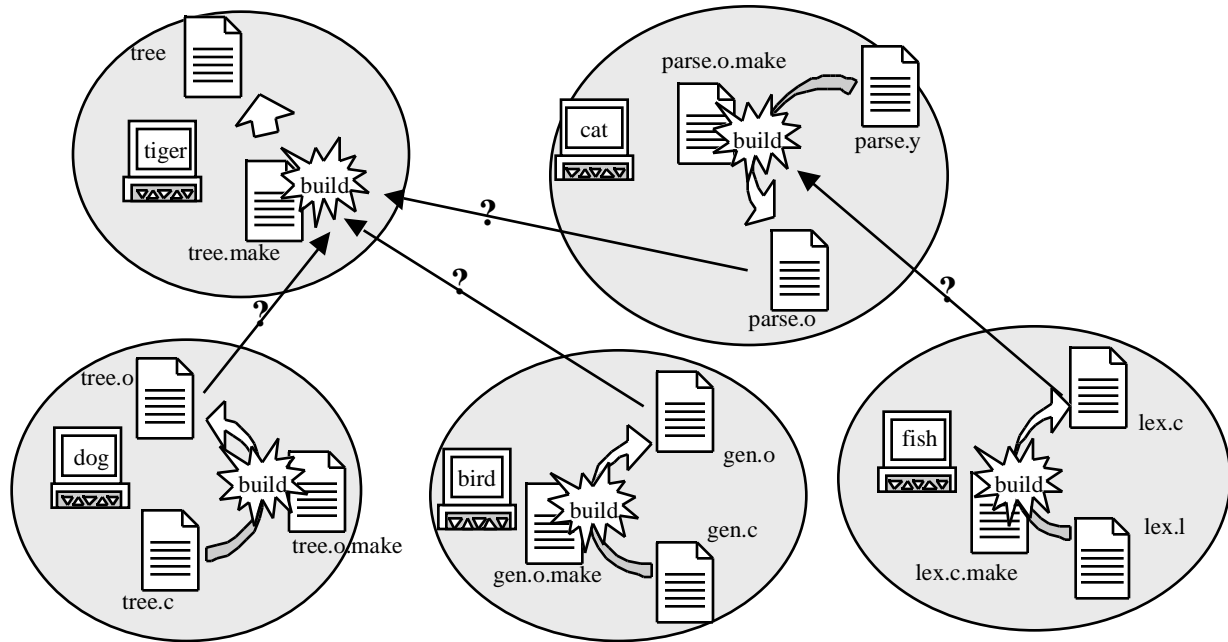
On the other hand, Object Make assumes that each constituent software component has its own description file. Consequently, reuse of build rules is quite straightforward. When I reuse the code generator in Figure 1, for example, I just need to reuse the source file **gen.c** and the description file **gen.o.make**. No extraction is necessary at all.

Please note although Object Make allows users to include the build rules of all the related components in a single description file like Make, this is only to keep the compatibility with Make, and is not the way in which Object Make should be used. The users will suffer from the same problems as they do when they use Make, if they use Object Make with a single huge description file.

### 2.4.2. Changing build rules

Changes, which are made during software development, are not limited to the changes of the components. Changes in the description files of components also need to be considered. When the description file of a component is modified, the component needs to be rebuilt with the new build rule. Make rebuilds the component only when the build rule of the component states that the component depends on the modified description file. It is a typical style of writing a description file to state that all the components to be built by the description file depend on the description file itself. However, since the build rules of all the components of a single system reside in a single description file, all the components depend on the same description file. As a result, a change in the build rule of a single component will result in the reproduction of all the components.

On the other hand, Object Make accepts separate description files for individual software components. As we saw in Figure 1, each component can be stated that it depends on its own description file, but not other's. Consequently, modification of the description file of a



**Figure 3. Distributing the compiler project**

component will not result in the reproduction of non-related components.

### 2.4.3. Chaining of build rules

If I divide a huge description file of Make into two or more separate description files, I can resolve the problems above. However, unfortunately, decomposition of a single description file into multiple description files results in a new problem. Unfortunately, Make establishes chaining of build rules only when the build rules are described in the same description file. If the build rules are included in different description files, Make is no longer able to establish the chaining of these build rules.

In order to avoid this problem, experienced users of Make often include commands to invoke Make recursively in description files. This method, to describe build rules in multiple description files and invoke Make recursively to establish chaining, is commonly used in the description files of many public domain software systems. However, recursive invocation of Make is only necessary to remedy the inappropriate behavior of Make to establish the chaining of build rules, and is not related to the "real" production process of the software systems. Object Make fixed this problem as I stated in the earlier section.

## 2.5. Related works

The history of the research on this area is quite long. A number of serious drawbacks of Make have been pointed

out, and systems aimed to overcome the problems have been developed. Although Make can be used with version control tools, like SCCS [11] or RCS [16], its capability to handle multiple versions of software components is quite limited. DSEE [7] and Cedar System Modeler [6] handle software components with versions more reliably by integrating their version control mechanism and the build process support mechanism into a single environment. Make uses the time stamp of software components to determine if the components need to be rebuilt. As a result, it often rebuilds components even if the reproduction is not necessary. Marvel [5] uses predicates to describe the scheduling rule of build processes more precisely. Although these systems focus on the problems in their own concern, they do not address the issues on build processes in distributed environment at all.

## 3. Distributed software development with Object Make

Object Make is capable of building software systems combining remote components stored in the file system of some other computer systems, in addition to the local components, as if all the components are located in the local host.

Please note that Object Make does not assume any distributed software architecture, like CORBA, DCOM [2], and RMI. Object Make treats software components as black boxes. Object Make is only interested in the build

processes, like compilation and linking processes. Non distributed software as well as distributed software can be developed in a distributed environment with Object Make.

### 3.1. Distributing Build Processes

Consider the case when the compiler project in Figure 1 is carried out in a distributed environment. Assume, as Figure 3 shows, four different programmers on hosts: **dog**, **cat**, **bird**, and **fish** develop the command analyzer **tree.o**, the parser **parse.o**, the code generator **gen.o**, and the lexical analyzer **lex.c** respectively. In addition, assume that the system integration is performed on a host **tiger**. I assume that these five hosts do not share a common file system.

Since Object Make allows that each software component has its own description file, it is quite easy to distribute the development tasks of these software components onto the five hosts, as shown in Figure 3. The software engineer on the host **dog** writes the source code **tree.c**, and uses Object Make on **tree.o.make** to compile the source code. The engineer will probably create another description file to combine a test driver with **tree.o** to test the functionality of the component. This is a typical use of Object Make on a single host. On the hosts **bird** and **fish**, software engineers pursue software development activities in a similar fashion. On these hosts, there is no interaction necessary with other hosts.

### 3.2. Issues on distributed software build

The two similar questions arise here are: (i) how the system integrator on the host **tiger** combines the three components on three different hosts into a final executable file, and (ii) how the programmer on the host **cat** uses **lex.c** being built on the host **fish**.

#### 3.2.1. Downloading

One possible solution is that the system integrator gets copies of the three components from the programmers' hosts. The system integrator links these copies of the three components into a single executable file.

Since the system integrator's host, by assumption, does not share a common file system with the three programmer's hosts, the system integrator must get a copy of the components explicitly and manually. Downloading the components with **ftp** command is one of the possible ways to get a copy of the components. Making a copy of files by **rep** command is another solution. Manual and interactive operations, like **ftp**, are not only time consuming but also not easy to automate, while remote access commands, like **rep** and **rlogin**, are famously known that they can be a security hole due to their ways of

access control. Consequently, a more sophisticated and transparent way of accessing remote components is quite desirable.

#### 3.2.2. Inconsistency

Making copies of remote components results in the following inconsistency problem. The programmers may make changes to their own files on their hosts for bug fixes and enhancements. Then the copies that the system integrator has are not identical to the original components. It is quite desirable to keep the original files and their copies consistent all the times.

Inconsistency problem can be more serious and complex when a system is organized in a hierarchical and encapsulated fashion. In the example of Figure 3, the production of the component **parse.o** requires **lex.c** being developed on the remote host **fish**. The build process of **tree** does not directly involve **lex.c**. As a result, when **lex.l** is modified, the system integrator on **tiger** has no way to know that the system **tree** needs to be rebuilt, until the component **lex.c** is rebuilt from the new **lex.l**.

#### 3.2.3. Traffic

One possible solution to avoid inconsistency is to get copies of all the remote components whenever the system integrator carries out the build process. However, this results in creating unnecessary traffic on the networks between hosts.

Re-transfer of a component is necessary only when the original file is updated after the copy is made. Not all the components require retransmission. Only the modified components after the copies are made need to be retransmitted. It is desirable to have a mechanism to minimize the network traffic by watching the activities performed on the components.

#### 3.2.4. Security

Security is another critical issue to be considered when we develop software systems in a networked environment. Software developers need to log onto remote computer systems to get program files and document files. Password of each software developer can be easily snooped and stolen, if they send their password in a plain text form. Sometimes, program and document files need to be protected from unauthorized access. Transferring these files between computer systems in non-encrypted form is quite dangerous.

### 3.3. Specifying remote components

Object Make allows the system integrator to explicitly specify the name of hosts, which hold the remote components, in his description file using a URL notation. Figure 4 shows the description files to build the compiler

tree.make on the host tiger

```
tree:    ${//dog/tree.o} ${//cat/parse.o} ${//bird/gen.o} tree.make
cc -o tree    ${//dog/tree.o} ${//cat/parse.o} ${//bird/gen.o}
```

parse.o.make on the host cat

```
parse.o: parse.c ${//fish/lex.c} parse.o.make
cc parse.c -c -o parse.o
parse.c: parse.y parse.o.make
yacc parse.y -o parse.c
```

**Figure 4. A description file for combining remote components**

system in the distributed environment in Figure 3. Although only two description files are shown here, the other description which are not shown here are identical to those in Figure 1. In Figure 4, for example, `${//dog/tree.o}` designates a component **tree.o** on host **dog**. A host name may be a fully qualified domain name, like `${//dog.animal.com/tree.o}`. A file name may include a path name. Please note that this URL notation is an abbreviated form. The non-abbreviated form is `$(omtp://dog/tree.o)`, where **omtp** (Object Make Transfer Protocol) is the name of the protocol which Object Make implements for accessing remote software components. The details of **omtp** protocol will be described in the following subsections.

When Object Make is invoked on this description file, Object Make works as if it directly reads the contents of the requested components from the remote hosts. The system integrator is freed from downloading the necessary components into the integrator's personal directory. Moreover, Object Make does not create local copies of these remote components in the system integrator's directory. Consequently, the system integrator is not required to maintain the copies of the components in the personal directory.

Object Make allows directory names in `VPATH` variable include host names. `VPATH` variable is an extension to the original Make introduced by GNU Make[12]. The value of `VPATH` variable is a list of directory names. When a file name in a description file is specified without an absolute pathname, the name is usually interpreted as being relative to the current directory where GNU Make is invoked. However, in case `VPATH` variable is defined in a description file, the file names with non-absolute path name are first interpreted as being relative to the current directory. If the files are not found, GNU Make tries to interpret the file names as being relative to the directories listed in `VPATH` variable. This feature can be used to search a file in two or more directories. In Object Make, `VPATH` variable accepts directory names with host names in the URL notation.

Consequently, Object Make is capable of searching files in multiple directories in multiple hosts.

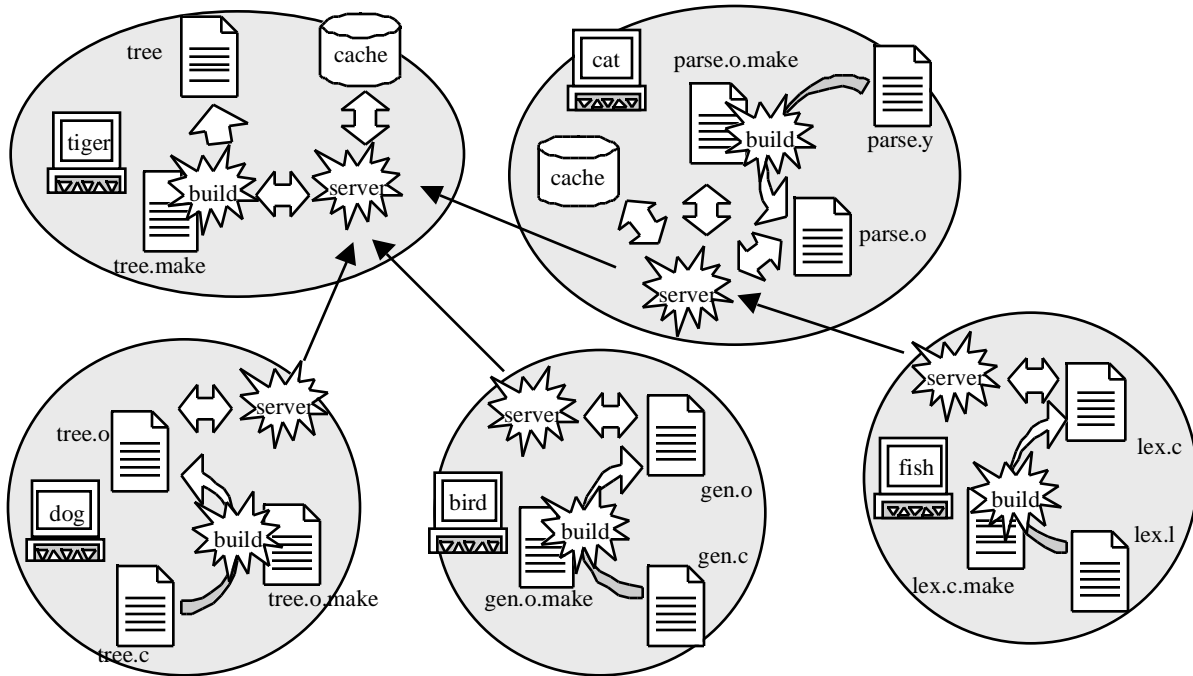
### 3.4. System configuration of Object Make

Figure 5 shows a whole picture of the build processes of the compiler system using Object Make. Object Make consists of two components: a client program **omake** and a server program **omkd**. The client is a program that is directly invoked by software engineers to produce a build process. The build processes created by the client are denoted as “build” in Figure 5. If all the components for a build process are locally available, the client program can execute the building process without any servers. When the client program executes a building process, which involves remote software components, the client program communicates with servers.

Hosts, which wish to publish any components to other hosts, need to run an Object Make server. The Object Make server is a program that runs in background as a daemon process. The daemon processes produced by the Object Make server are denoted as “server” in Figure 5. When a client is invoked on a description file, which includes a remote component, the client reads the contents of the remote component from the server. The client is able to read the contents of the component only when a server is running on the remote host, and the server allows the client to read the contents of the component. The server implements a security protocol to protect itself from unauthorized accesses. The security issues will be discussed more precisely in a later section. The server also implements a caching mechanism, which I plan to discuss in a later section, to reduce unnecessary traffic on the networks.

### 3.5. Remote chaining

When a server receives a request from a client to read the contents of a component, which the server maintains, the server checks to see if the component is already



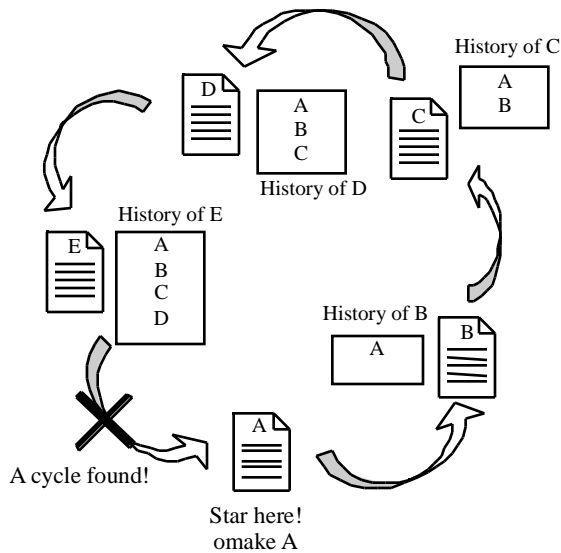
**Figure 5. Distributed build processes of Object Make**

available for the client. If the component is not built yet or is not up-to-date, the server builds or rebuilds the requested component before it processes the request from the client. The build or rebuild process is performed along with the component's description file stored in the server's host. Consequently, chaining of build rules between the client and the server is established.

When a server builds or rebuilds a component, which is

requested by a client, the server often communicates with other servers. The build rule of the requested component may require software components on some other remote servers. Then, the server tries to read the contents of the required component from the remote servers. Servers do not distinguish clients' requests from servers' requests. Consequently, chaining of build processes among servers is also established. In the example of Figure 5, for example, the production of **tree** on the host **tiger** requires **parse.o** on the host **cat**. The production of **parse.o** involves the production of **lex.c** on the host **fish**. Consequently, the chaining from **tiger** to **cat** is extended to the host **fish**.

One of the issues to be resolved to implement the remote chaining of build rules is detection of cycles in build rules. Dependency among components in two or more description files might result in a cyclic dependency. If a cycle exists in dependency, the chain of build rules will never terminate. Detection of a cycle in multiple description files is not a trivial task, because these description files are stored in two or more hosts separately and it is not possible to examine all the dependency files at one time. Object Make is capable of detecting cycles in dependency at build time. During a build process generated by Object Make, each component carries an attribute called history that is a list of components which depend on the component being built, as Figure 6 shows. If the component being built again depends on any components in its history, a cycle exists. Such a cyclic build process will be immediately stopped.



**Figure 6. Detecting a cycle in dependencies**

### 3.6. Caching

During software development, build processes are often carried out repeatedly to carry out changes and error corrections. However, it is not common that changes made between successive build processes involve all the components. Only some of them are modified. Consequently, successive build processes often require transferring the contents of remote components even if they are not modified.

To avoid extra traffic on computer networks, Object Make implements its own caching mechanism. The Object Make server performs the caching activities. Each client designates a near-by Object Make server as its cache server. The client and the cache server may be running on the same host, but it is not necessary. A client's cache server need to run on a host that shares a file system with the client's host. In other words, hosts with a common file system can share a common cache server.

Let's go back to Figure 5 to see how the caching mechanism of Object Make works. When a client requests a component on a remote server, the client does not contact with the remote server directly. The client contacts with its cache server first. The cache server forwards the client's request to the remote server. The remote server sends the requested component to the cache server. The cache server saves the components into its cache preserving the time stamp of the original component. The cache is a directory in the file system of the cache server's host. The cache server allows the client to use the copy of the components in the cache directly.

When the client requests the same component later, the request again goes to the cache server first. The cache server checks to see if a copy of the component is available in the cache. If it does not exist in the cache, the cache server forwards the client's request to the remote server in order to get a copy of the component. If a local copy in the cache exists, the cache server sends a request to get the time stamp of the component in the remote server. If the cached copy is up-to-date, that is the cached copy has the same time stamp as the original file, the cache server does not forward the client's request to the remote server. Instead, the cache server allows the client to use the copy in the cache. If the cached copy is out-of-date, the cache server forwards the client's request to the remote server to get a copy of the latest component.

Please note that when a cache server sends a request to get the time stamp of a component to a remote server, the remote server might perform the building activities of the component, depending on the status of the remote host. If the component exists in the remote server and is up-to-date, only its time stamp will be returned to the cache server. Otherwise, the remote server builds or rebuilds the

component, and sends the time stamp of the newly built/rebuilt component to the cache server. This may involve chaining to some other servers.

### 3.7. Security

Object Make provides three integrated mechanisms to improve the security of distributed software build processes.

#### 3.7.1. Client verification

Object Make provides a client authentication mechanism so that only authorized clients can access software components in servers. Each Object Make server is capable of keeping a database of valid users for the server along with their password. Only the users listed on that user database are allowed to log onto that server. Each Object Make server may consult with the underlying operating system for user verification. In this case, only users who have a valid account on the operating system are allowed to log onto the Object Make server.

When an Object Make client needs to access some component on a remote server, the client shows a prompt to get a user name and a password for that server. The user on the client needs to supply the valid user name and a password to proceed.

Client verification using client's certificate, instead of a password, is also possible. Object Make uses SSL (Secure Sockets Layer) [10] for the client verification. SSL is a widely used protocol for encryption. The most common use of SSL can be found on the World Wide Web. Netscape Communicator includes a build-in support for SSL.

Object Make clients carry their own certificate signed by a certificate authority. When a client connects to a server, the client sends its certificate to the connecting server. The server makes sure that the certificate is valid using the public key of the certificate authority who signed the client's certificate. If the certificate is intact, the server checks to see if the client's name in the certificate is included in the valid user list. If a match found, the server grants the access from the client. Otherwise the server does not accept the connection request from the client.

#### 3.7.2. Server verification with SSL

Server verification is another issue to be discussed. When a client or a server connects to another server, it must be verified that the server to be connected is the "real" one. Client and servers do not want to connect to any server masquerading as a particular server and disclose confidential identity information.

Object Make uses SSL for the server verification. Object Make servers carry their own certificate signed by a certificate authority. When a client connects to a server,

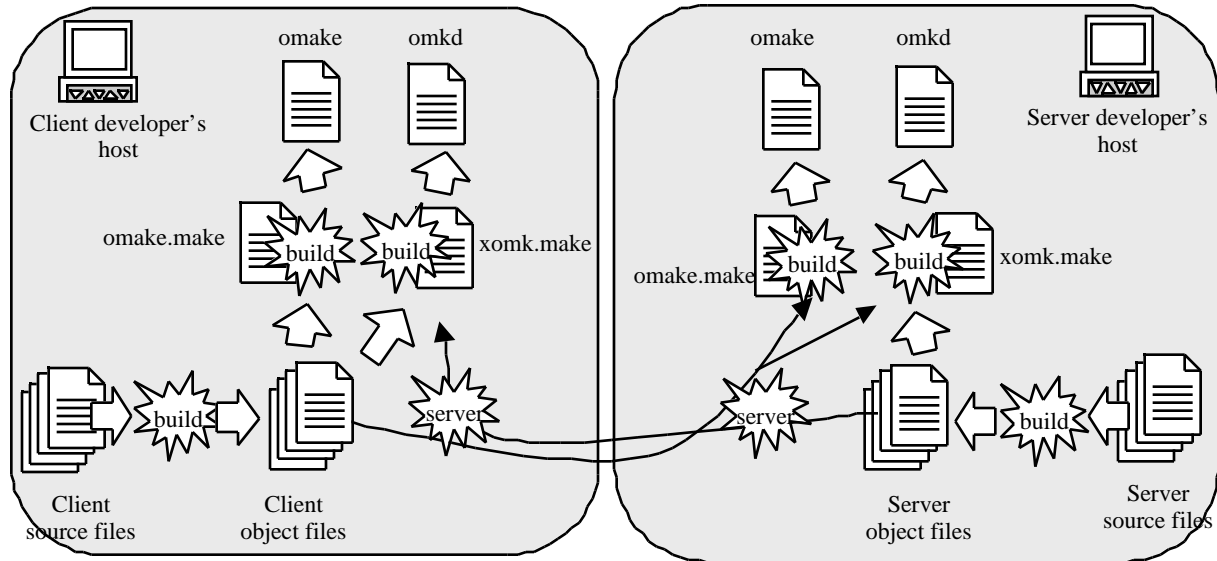


Figure 7. Using Object Make for building Object Make

the client requests for the certificate of the server. The client will verify the server's certificate with the certificate authority's public key that the client has. If this verification process fails, the client disconnects from the server immediately.

When the server's certificate is valid, the client compares the name of the server in the description files against the name in the server certificate. Only when the two names match, the client establishes a connection with the server.

### 3.7.3. Encryption

Reliability of the client verification procedure using a user name and a password will be lost if the password is stolen by other people. Object Make is capable of encrypting data being transferred between clients and servers to avoid sniffing. Object Make includes a mechanism to encrypt users' password as well as files being sent between Object Make servers and clients.

Object Make uses SSL for data encryption. When a client connects to a server, the client and the server negotiate with each other to establish a set of keys for data exchange. This process is called an SSL handshake process. After the handshake process successfully completes, the client and server exchange data, such as password, in encrypted format using the keys established in the handshake process.

### 3.8. Implementation

Object Make is currently running on SunOS 4.1.3 and Solaris 2.6, versions of the UNIX operating system that run on Sun workstations, as well as LinuxPPC R5, a

version of the Linux operating system that runs on Apple Macintosh computers. Object Make uses SSLeay [4] to implement the SSL protocol.

The Object Make client is implemented as a preprocessor to Make. However, users are not expected to invoke Make explicitly at all. From users' point of view, the Object Make client is a stand-alone tool that is upward compatible with Make. The Object Make client is able to read existing description files written for Make.

The Object Make server is a program that runs in background as a daemon. Although the Object Make client is a stand-alone program, the client is, at the same time, one of the constituent components of the Object Make server. When an Object Make server establishes chaining of build rules, the server invokes the Object Make client.

The Object Make client is a tool that is usually invoked from a command shell using a keyboard, like Make. However, Object Make is accompanied by a tool, called **xomk**, that offers a graphical user interface to invoke Object Make. **xomk** allows users to invoke Object Make simply by clicking a mouse. **xomk** is implemented on X-window version 11 release 5. It can be used with most window managers like **twm**, **mwm**, and **olwm**. I will not cover **xomk** here because the details of **xomk** have been already discussed in [15].

## 4. Evaluation and Future Considerations

The first project, which I used Object Make for development, was the development of Object Make itself. As I described in this paper, Object Make consists of a client **omake** and a server **omkd**. Although the client and

tree.make

```
tree:    ${omtp://dog/tree.o} ${mftp://cat/parse.o} ${omtp://bird/gen.o} tree.make
cc -o tree    ${omtp://dog/tree.o} ${mftp://cat/parse.o} ${omtp://bird/gen.o}
```

**Figure 8. A description file using mftp**

the server were developed on two different hosts by two different programmers, the client and the server share a library that was built by the client's developer. In other words, the source files required to build the server are distributed on two different hosts. In addition, the server cannot be tested without a client, and the client cannot be tested without a server. This implies that the two hosts must have both the client and the server. However, we did not want to make copies of source files on the two hosts.

Within these requirements, we developed Object Make in the configuration as shown in Figure 7. We put the two description files **omake.make** and **omkd.make** on these two hosts. This simple arrangement allowed us to build the server and the client on both machines, while we were able to keep the source files of the server and the client on two different hosts. This way of using Object Make worked out quite reasonably.

However, I also found several issues to be considered in the future releases of Object Make. Currently my students and I are working on the following features to improve the capability of Object Make.

#### 4.1. Uploading and version control

It is not rare that a software engineer works on two or more locations. The engineer will probably have computer systems at each work place. In order to perform a single job on multiple locations, the engineer will have to keep the copies of the files at all the locations, unless the engineer carries all the files in a removable media with him/her as he/she moves. Inconsistency among these copies can easily arise. Whenever the engineer modifies a file at a work place, the engineer need to propagate the changes to all the copies of the file at the other work places.

The current implementation of Object Make offers a limited support for such a software engineer. The engineer will select one of the hosts as a primary location of his/her work. The engineer then runs Object Make server on this primary host. When the engineer needs to work at some other location other than the primary host, he/she is not required to take all the files to the location. The engineer takes or downloads files to be modified with their description files. The engineer modifies the files and will use Object Make to compile the files on the remote

location. Object Make will take care of obtaining any other read-only components, which are necessary for the build, from the primary host. When the changing task completes, the engineer will upload the modified files to the primary host for synchronization.

The issue here is that such manual downloading and uploading of files will easily result in conflicts. Changes made by two or more engineers may result in conflicts when they upload the modified components to the server.

One of the possible solutions to this problem is to use a version control tool like SCCS and RCS along with **ftp**. CVS [1] may be more convenient because it does not require **ftp** for downloading and uploading files. These version control tools guarantee that only one engineer can check out a single file at a time for modification purpose.

Rather than relying on these external tools, I plan to implement download and upload commands within Object Make. The downloading command will allow software engineers to get a copy of a software component stored in a server host, without using some other tools for downloading. The download command will be executed in a mutual exclusive way, so that two or more engineers are not allowed to download the same component. The download command will lock the component so that other people cannot download the same component until the component is uploaded. Upload command will send the modified component back to the server, and will unlock the component.

Version control is another issue to be considered. When a modified component is uploaded, a new version may be created instead of overwriting the original component.

#### 4.2. The mftp protocol

The **omtp** protocol, which I stated in the section 3, assumes that each component specified in a URL notation is built on the host that holds the description file. However, this protocol does not work well if hosts involved in a build process do not have an identical architecture. It is quite common, for instance, that an object file produced on some architecture cannot be used on other architectures.

I plan to implement a new protocol **mftp** (Make File Transfer Protocol) to remedy this problem. When a build process of a component requires another component on a

remote host, the **omtp** protocol establishes the chaining of build processes on these hosts. On the other hand, in the **mftp** protocol, the description file of the remote component is sent to the host that requires the remote component, and the description file will be processed there. Consequently, the related build processes are carried out on the single host. In this case, all the file names in the transferred description file, which cannot be resolved locally in the host, are treated as the components on the original remote host.

For instance, consider the case when the host **cat** in Figure 5 has a different architecture from other hosts, and therefore the object file **parse.o** build on **cat** cannot be used on **tiger** for linking. In such a case, we use the **mftp** protocol instead of the regular **omtp** protocol. I need to modify the description file in Figure 4 as shown in Figure 8. In this description file, **parse.o** is specified by the URL `$(mftp://cat/parse.o)`.

When **parse.o** is built using **mftp**, first its description file **parse.o.make** is transferred from the host **cat** to the host **tiger** that executes Object Make on **tree.make**. Then **tiger** starts the build process on the description file **parse.o.make**. The component **parse.o** depends on **parse.c** and **lex.c**. Since **parse.c** is not locally available on **tiger**, Object Make checks to see if the build rule for **parse.c** is specified in the description file. In this case, the description file includes a build rule for **parse.c**. Object Make evaluates the build rule to produce **parse.c**. However, **parse.c** depends on **parse.y** which is not available on **tiger**. Again Object Make checks to see if **parse.y** has its build rule in the description file, and will find out that no build rule for **parse.y** is available. Object Make assumes that **parse.y** exists in the server **cat** where the description file **parse.o.make** is downloaded. Finally, Object Make executes the build rule by obtaining the **parse.y** from the host **cat**.

On the other hand, **lex.c** is specified with a host name in the **omtp** protocol, Object Make dispatches the build process of **lex.c** on the host **fish** and get a copy of it. Please note that moving description files around multiple hosts with **mftp** is not a simple task, because the build commands in the description files may strongly rely on the underlying architecture of the host where the description file is originally defined. In order to make **mftp** protocol successful, a precise care regarding the difference between architectures or hosts is quite essential.

There is a popular tool called **autoconf** [9] to be used to build open-source systems. **autoconf** generates a script named **configure** that is able to generate description files which are customized for the builders' environments. I am considering using **autoconf** with Object Make to automate the customization of description files to various architectures.

## 5. References

- [1] Cederqvist, P. et al. Version Management with CVS, Signum Support AB, Sweden, 1993.
- [2] Eddon, G. and Eddon H. Inside Distributed COM. Microsoft Press, 1998.
- [3] Feldman, S. I. MAKE - A Program for Maintaining Computer Programs. Software-Practice and Experience, vol.9, pages 255-265, 1979.
- [4] Hudson, T. J. and E. A. Young. SLeay Programmer Reference.
- [5] Kaiser, G. E. and P. H. Feiler. An Architecture for Intelligent Assistance in Software Development. in Proceedings of ICSE9, pages 180-188, Monterey, California, ACM-IEEE, March, 1987.
- [6] Lampson, B. W. and E. E. Schmidt. Organizing Software in a Distributed Environment. in Proceedings of the ACM SIGPLAN 83 Symposium on Programming Language Issues in Software Systems, pages 1-13, ACM, 1983.
- [7] Leblang, D. B., R. P. Chase Jr. and G. D. McLean Jr. The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. in Proceedings of the IEEE Conference on Workstations, pages 266-280, San Jose, California, IEEE, November, 1985.
- [8] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. in Proceedings of OOPSLA'86, pages 214-223, Portland, Oregon, ACM, September, 1986.
- [9] MacKenzie, D. and B. Elliston. Autoconf: Creating Automatic Configuration Scripts, Free Software Foundation, 1998.
- [10] Netscape Communications. SSL 3.0 Specification, <http://www.netscape.com/libr/ssl/ssl3/index.html>
- [11] Rochkind, M. J. The Source Code Control System. IEEE Transactions on Software Engineering, vol.SE-1, no.4, pages 364-370, December, 1975.
- [12] Stallman, R. M. and R. McGrath. GNU Make: A Program for Directing Recompilation, Free Software Foundation.
- [13] Sugiyama, Y. Producing and Managing Software Objects in the Process Programming Environment OPM. in Proceedings of APSEC'94, pages 268-277, Tokyo, Japan, IEEE and IPSJ, December, 1994.
- [14] Sugiyama, Y. and E. Horowitz. OPM: An Object Process Modeling Environment. in Proceedings of the 5th International Software Process Workshop, pages 134-136, Kennebunkport, Maine, ACM-IEEE, October, 1989.
- [15] Sugiyama, Y. Object Make: A Tool for Constructing Software Systems from Existing Software Components. in Proceedings of the ACM Symposium on Software Reusability, pages 128-136, Seattle, Washington, ACM, April, 1995.
- [16] Tichy, W. F. RCS - A system for Version Control. Software - Practice and Experience, vol.15, no.7, pages 637-654, July, 1985.