Object Make: A Tool for Constructing Software Systems from Existing Software Components

Yasuhiro Sugiyama

Department of Computer Science
Nihon University
Koriyama, Fukushima 963, Japan
e-mail: sugiyama@ce.nihon-u.ac.jp

1. Introduction

It is a common practice to build large-scale software systems as a combination of smaller components. However, it is quite costly if software developers need to design and develop the components from scratch for each software system. It is quite desirable that they build highly generalized components that can be used in many systems. However, it is not enough to prepare many reusable components. A mechanism to combine these components into a large system in a systematic fashion is quite essential.

This paper presents a tool, called Object Make, that (i) helps software developers to build reusable software components, and (ii) offers a mechanism to combine a large number of components into a system in a systematic way. The components of a system built with Object Make can be easily reused in other systems. The development of Object Make is motivated by the following background.

Make [2] 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 I found that it is not easy to reuse the components of a system that was built with Make. I developed Object Make to overcome the problems of Make without loosing the compatibility with Make.

This paper is organized as follows. The second section summarizes the problems of Make that I experienced. The third section describes the mechanism of Object Make to resolve the problems of Make, as well as the current implementation of Object Make. The fourth section explains the theoretical background of Object Make. The fifth section gives a quantitative evaluation of Object Make showing that Object Make is more effective than Make to produce large systems. The last section concludes the paper with my experience report on using Object Make.

2. Problems of Make

Make is a tool that builds or rebuilds software systems based upon *build rules* specified by system designers. Build rules are stored in files, called *description files*. Make reads a description file and builds a system based upon the build rules in the description file. Description files are often named **makefile** or **Makefile**, but they may have any other names. Typically, each system has its own description file. Build rules of a system state the *dependency* that lists the software components to be used to build the system, as well as the *build commands* to combine and/or transform the components into the system. A typical description file includes the build rules of the components to be used to build the system, in addition to the build rule of the system itself. This is because Make must build the components first before it combines the components into

This research was in part supported by the Grant-in-Aid for Scientific Research by the Education Ministry of Japan.

```
tree:tree.o parse.o lex.o makefile
  cc tree.o parse.o lex.o -o tree
tree.o:tree.c makefile
  cc tree.c -c -o tree.o
parse.o:parse.c makefile
  cc parse.c -c -o parse.o
parse.c:parse.y makefile
  yacc parse.y -o parse.c
lex.o:lex.c makefile
  cc lex.c -c -o lex.o
lex.c:lex.l makefile
  lex lex.l -o lex.c
```

Figure 1: A Sample Description File

the system.

Figure 1 shows a sample description file, named **makefile**, to build a simple compiler. The compiler, called **tree**, is produced from three object files: **tree.o** that contains a main program, a parser **parse.o**, and a lexical analyzer **lex.o**. The first line of the description file states that **tree** depends on the three object files as stated above, as well as the description file **makefile**. If **tree** does not exist or it is older than one of these files, Make executes the second line, which is a build command to produce **tree** by linking the three object files. This description file also contains the build rules to build the three object files from their source files. The fifth and sixth lines state that **parse.o** is compiled from **parse.c**, and furthermore, the seventh and eighth lines state that **parse.c** is produced from **parse.y** by Yacc, a parser generator. The build rules for other object files are specified in a similar way.

2.1. Large Description Files

One of the problems of Make is that it relies on the directory structure of the underlying file system to find out the description file to be used to build a system. When Make is invoked, Make reads a description file named **Makefile** in the current directory as its default. However, if the current directory includes two or more description files, Make is no longer able to determine the description file to be used. Users of Make are required to specify the description file explicitly when they invoke Make.

To avoid complex invocation operation, most users put all the related files of a system to be built into a single directory. They also create a single **Makefile** in the directory. They put the build rule of the system as well as the build rules for all the components, which are used to build the system, into the description file. As a result, description files tend to be large. Large description files result in the following difficulties.

(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. For instance, to reuse a source program, I need to know the compiler to be used to compile the source program. I need to know the invocation option to the compiler. However, Make does not allow me to reuse build rules straightforwardly. If 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.

For instance, in Figure 1, assume I am going to reuse only the parser of the compiler. Before I extract the necessary build rules for the parser, I need to analyze the description file and understand that (i) the object file **parse.o** is generated from the source file **parse.c** by **cc**, which is a C language compiler, and (ii) the source file **parse.c** is generated from **parse.y** by Yacc.

In this example, the description file is quite small. However, the description files of huge systems are quite large and complex. Extraction from huge description files is a quite complex task. As a result, large description files result in the difficulty of reusing the components of systems that are built with Make.

(2) Changing Build Rules

During the development of software systems, it often happens that changes are made to some of the components, but not to others, in order to fix a particular bug or to include a new design. Changes are not limited to the changes of the components themselves. Changes in the description files of the components need to be considered. When some of the components or their build rules are modified, Make needs to rebuild the components that relate to the modification before it rebuilds the system. It is not necessary to rebuild all the components. However, it is not a trivial task to find out the components that need to be rebuilt.

Although Make has an ability to find out the components

makefile

```
tree:tree.o parse.o lex.o makefile
    make -f makefile2  # build tree.o
    make -f makefile3  # build parse.o
    make -f makefile4  # build lex.o
    cc tree.o parse.o lex.o -o tree
```

makefile2

```
tree.o:tree.c makefile2
  cc tree.c -c -o tree.o
```

makefile3

```
lex.c.o:lex.l makefile3
  cc lex.c -c -o lex.o
lex.c:  lex.l makefile3
    lex lex.l -o lex.c
```

makefile4

```
parse.o:parse.c makefile4
    cc parse.c -c -o parse.o
parse.c: parse.y makefile4
    yacc parse.y -o parse.c
```

Figure 2: Sample Description Files with Recursive Invocation of Make

to be rebuilt, this mechanism works reasonably only when the contents of the components are changed. If changes are made in the build rules of components, Make fails to find out the components to be rebuilt.

Make rebuilds a component only when the build rule of the component states that the component depends on the modified components and/or the modified description files. However, if 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. In Figure 1, for example, you see that all the components depend on the same **makefile**. Even if only the build rule for **parse.c** is modified, Make will remake all the components. In other words, large description files result in loosing the precise control over the re-building processes of software systems.

2.2. Chaining of Build Rules

If I divide a huge description file into two or more separate description files, I can make each description file smaller. In Figure 2, for instance, I divided the description file in Figure 1 into four separate description files. However, unfortunately, decomposition of a single description file into multiple description files results in a new problem.

Generally speaking, when Make builds a component **P** from another component **Q**, Make must build **Q** first before it builds **P**. In other words, in order to build **P**, it is not sufficient to evaluate and execute the build rule of **P**. Make must evaluate and execute the build rule of **Q** first, before it evaluates and executes the build rule of **P**. This is called the *chaining* of the build rules. Unfortunately, Make establishes the chaining of the build rules only when the build rules of **P** and **Q** are described in the same description file. If the build rules of **P** and **Q** are included in different description files, Make is no longer able to establish the chaining. As a result, the building process of **P** will fail if **Q** does not exist. If **Q** exists, the building process will complete. However, the resulting **P** may not be the right one, because **Q** might need to be rebuilt, before it is used, to include some changes.

In order to avoid this problem, experienced users of Make often include commands to establish chaining in description files explicitly. In Figure 2, you see that **makefile** contains the commands to invoke Make recursively before the link operation. The recursive invocation of Make will assure that **makefile2**, **makefile3**, and **makefile4** are evaluated, and **tree.o**, **parse.o**, and **lex.o** are built before they are linked together to produce the executable file **tree**. If **makefile** does not include these three lines to invoke Make recursively, other description files are not evaluated at all. If one of the object files is missing, the linking process will fail. If all of them exist, the linking process will complete. However, the resulting executable file may not be the right one, because the object files might need to be rebuilt to include the latest changes made in their source files.

This method, to describe build rules in multiple description files and invoke Make recursively in the description files 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. Moreover, Make establishes the chaining of build rules when the build rules are stored in a single description file. It is desirable that the chaining of build rules is established even if the build rules are included in two or more separate description files.

3. Object Make

Object Make is a tool that fixes the problems of Make that I stated in the previous section without loosing the compatibility with Make. This section shows the way of Object Make to resolve the problems, I believe, and the current implementation of Object Make.

3.1. Multiple Description Files

First of all, 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**.

This simple naming rule assures that each component has its own description file, and Object Make can uniquely identify the description file of a component based upon the name of the component. Furthermore, since a single description file contains the build rule of a single component, users of Object Make can reuse the build rule of the component without extracting it from a huge description file. They just need to reuse the whole description file. As a result, components can be easily reused with their own description files.

Moreover, each description file can include a build rule stating that the component depends on the description file itself, but not other description files. As a result, when the description file is modified, Object Make updates only the components that relate to the modified description file, but it does not update any other components.

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.

In Figure 3, I divided the description file in Figure 1 into four separate description files to be used with Object Make. One of the major differences between Figure 2 and Figure 3 is that Figure 3 does not include lines to invoke Object Make recursively at all, while Figure 2 includes lines to invoke Make recursively. You also see that the description files in Figure 3 are named differently from those in Figure 2. The description file of **tree** is named **tree.make**, while the description file of **parse.o** is named **parse.o.make**,

tree.make

```
tree:tree.o parse.o lex.o tree.make
    cc tree.o parse.o lex.o -o tree
```

tree.o.make

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

lex.o.make

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

parse.o.make

```
parse.o:parse.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 3: Description Files for Object Make

and so forth. If I need to reuse the parser, I just need to reuse the source file **parse.y** and the description file **parse.o.make**. No extraction is necessary at all. Furthermore, **parse.o** depends on **parse.o.make** but not other description files. As a result, if **parse.o.make** is modified, Object Make rebuilds only **parse.c**, **parse.o** and **tree**, but it does not rebuild other components.

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 3 by the following command:

```
omake tree
```

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.

3.2. Chaining Build Rules

Object Make can establish the chaining of build rules even if those rules are included in different description files. Assume Object Make is about to build a component **P** from another component **Q**, and the build rules of **P** and **Q** are stored in **P.make** and **Q.make** respectively. Object

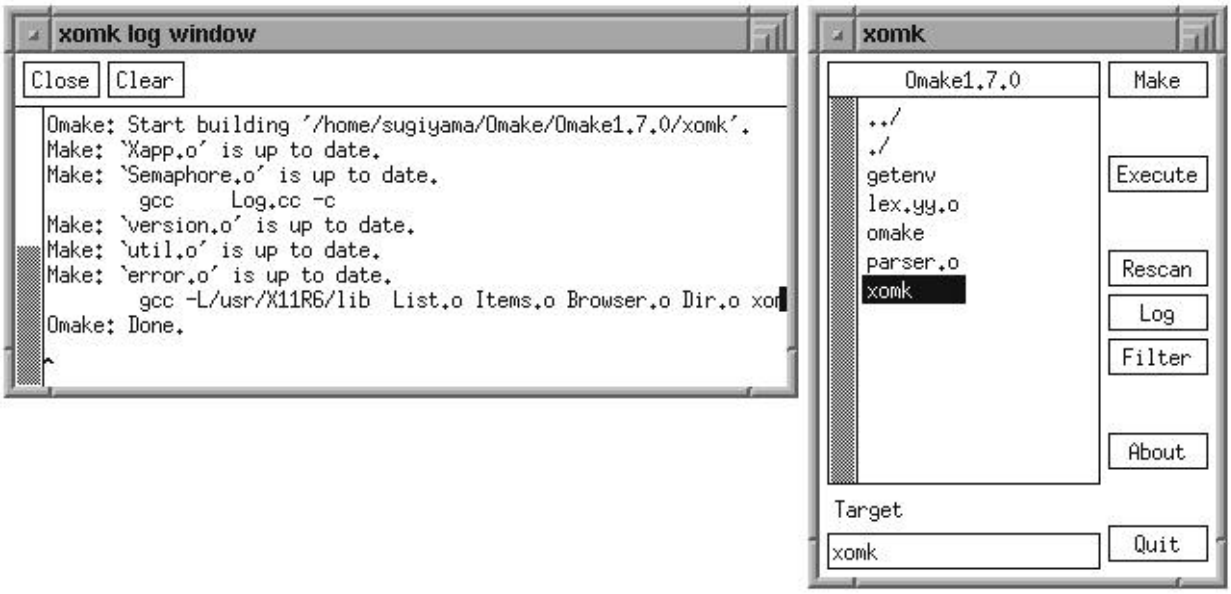

Figure 4: A Sample Session of xomk

Make first analyzes **P.make** and finds out that it needs **Q** to build **P**. Then it looks for the build rule of **Q** in **P.make**. If the build rule of **Q** is found in **P.make**, Object Make will use the rule in **P.make**. In this case, however, I assumed that the build rule of **Q** is included in **Q.make**, but not in **P.make**. Object Make will look for **Q.make**, and will use the build rule in **Q.make** to build **Q**. As a result, the chaining of the build rules is established. It is not necessary to specify the recursive invocation of Object Make at all.

In the example of Figure 3, the production of **tree** requires the three object files: **tree.o**, **lex.o**, and **parse.o**. Object Make will first search for their build rules in **tree.make**. In this case, **tree.make** does not include the build rules for these object files. Object Make, then, looks for **tree.o.make**, **parse.o.make**, and **lex.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 **lex.o.make** is established.

Object Make allows users to specify path names in description files instead of simple component names. Object Make assumes that a component and its description file are located in the same directory. If a path name is specified, Object Make will look for the corresponding description file in the directory which the path name specifies, and will perform the build operation in the directory. For instance, in the example of Figure 3, assume the related files of the parser are included in a sub directory called **parser**. If I specify **parser/parse.o** instead of **parse.o** in the description file **tree.make**, Object Make will look for **parse.o.make** in the **parser** sub directory, and will perform the build operation of **parse.o** in the sub directory. This allows users to organize software components in a hierarchical fashion.

3.3. Implementation and User Interface

Object Make is currently running on SunOS 4.1.3 and Solaris 2.3, versions of the UNIX operating system that run on Sun workstations, as well as A/UX, a version of UNIX that runs on Apple Macintosh computers. Object Make is implemented as a preprocessor to Make. However, users are not expected to invoke Make explicitly at all. From users' point of view, Object Make is a stand-alone tool that is upward compatible with Make. Object Make is able to read existing description files written for Make.

Object Make is a tool that is usually invoked from a command shell using a keyboard, like Make. However, in environments with a graphical user interface, like the X-window system [7], it is desirable that users can operate Object Make with a mouse instead of the keyboard. 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**.

Figure 4 is a picture showing a sample session of **xomk** when it is used to build itself. The right window is the main window to control **xomk**. When **xomk** is invoked, the main window will appear. Users can perform necessary operations on Object Make within this window. The left window is a log window to show messages generated by **xomk** itself and Object Make. It is possible to make the log window visible all the time. Usually, however, the log window appears only when there are messages to be displayed.

The main window shows the current directory name, "Omake1.7.0" in the figure, and a list of systems and components that can be built by Object Make in the directory. The list often includes the files that do not exist but can be produced by Object Make. The list also includes directories so that users can move around in the underlying file system. Users may specify some condition so that only the file names that satisfy the condition will be listed. Users can specify the condition by clicking on the "Filter" button.

In this list, users select a file that they need to build by clicking on it. Then, a click on the "Make" button invokes Object Make to build or rebuild the selected file. If the created file is executable, users may execute the file by clicking on the "Execute" button in the main window.

3.4. Related Works

Drawbacks of Make are not limited to those that I pointed out in the second section. A number of other serious drawbacks 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 [8] or RCS [12], its capability to handle multiple versions of software components is quite limited. DSEE [5], Cedar System Modeler [4], and ALS [11] 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 [3] 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 that I pointed out in the second section. On the other hand, I need to admit that Object Make is not designed to fix the problems addressed by these systems. However, I am currently developing a process programming environment OPM [10] that is aimed to fix all the problems in a reasonable fashion. Although it is out of the scope of this paper to cover OPM in details, the theory behind OPM, which is the key concept to design Object Make, will be briefly discussed in the next section.

4. Theory of Object Make

OPM is a process programming environment in which process programs are designed, executed, and tracked. OPM is capable of automating and assisting the execution of software build processes [9], because software build processes are special cases of software processes. Furthermore, OPM does not suffer from the problems of Make that I stated in the second section. Object Make is an implementation of the software build process support mechanism of OPM as a UNIX tool in a restricted form.

The theory behind OPM and Object Make is object-oriented approach. OPM has an object base that stores software components. In the object base, software components are treated as objects that are instances of their classes. The nature of an object is determined by the classes of the object.

The object-oriented approach is commonly characterized by (i) encapsulation and (ii) information sharing by inheritance or delegation [6]. OPM encapsulates 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 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 5 illustrates the building process of a system consisting of two components: **A** and **B** that is

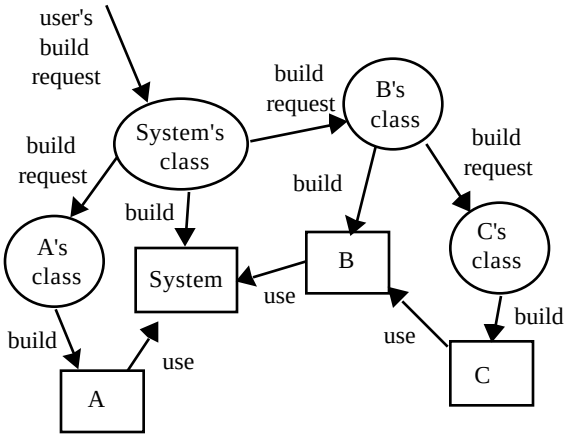

Figure 5: A Build Process in OPM

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.

5. A Quantitative Evaluation

Before concluding the paper, I would like to make some quantitative evaluation of Object Make. I will show how Object Make can save the effort to design and produce description files of a single software system compared with Make, when I reuse some of the components from existing systems.

I will use the COCOMO model [1] to compute the effort. Build rules are software written in a shell scripting language. As a result, the effort to build description files can be computed as the effort to build software systems is computed. COCOMO calculates the effort (*MM*) to build a software system based upon *DSI* (the number of Delivered Source Instructions). Here, I will use the following COCOMO Semidetached Mode Effort Equation:

$$MM = 3.0\left(\frac{DSI}{1000}\right)^{1.12}$$

However, when some of the components are adapted from existing software, I cannot directly use *DSI* . To handle the effects of adapted software, COCOMO calculates an *EDSI* (Equivalent number of delivered source instructions) which is defined as follows:

$$EDSI = DSI_{adapted} \times \frac{AAF}{100} + DSI_{new}$$

where $DSI_{adapted}$ is the *DSI* of the components that are adapted from existing software, DSI_{new} is the *DSI* of the components to be created from scratch, and *AAF* (adaptation adjustment factor) is defined as follows:

$$AAF = 0.40\,DM + 0.30\,CM + 0.30\,IM$$

where *DM* (percentage Design Modified) is the percentage of the adapted software's design which is modified in order to adapt it to the new system, *CM* (percentage Code Modified) is the percentage of the adapted software's code which is modified in order to adapt it to the new system, and *IM* (percentage of Integration required for Modified software) is the percentage of the effort required to integrate the adapted software into the new system and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

In order to show how Object Make reduces the effort compared with Make, I will compute:

$$ratio = \frac{MM_{make}}{MM_{omake}}$$

where MM_{make} is the effort when I use Make, while MM_{omake} is the effort when I use Object Make. The subscripts denote the tools in my concern. Notice that:

$$\frac{MM_{make}}{MM_{omake}} = \frac{3.0\left(\frac{EDSI_{make}}{1000}\right)^{1.12}}{3.0\left(\frac{EDSI_{omake}}{1000}\right)^{1.12}} = \left(\frac{EDSI_{make}}{EDSI_{omake}}\right)^{1.12}$$

Therefore, I will compute $\frac{EDSI_{make}}{EDSI_{omake}}$ first.

Consider a system consisting of *n* components, among which *p* components are adapted from existing systems that contain *k* components in total, while the remaining *n-p* components are newly built from scratch. Also assume that the average size of the components is *b*.

First I will compute $EDSI_{make}$ that is the *EDSI* when I use Make. As far as IM_{make} is concerned, if I adapt the components as I assumed, I have to examine the description files of the existing systems that contain the build rules of *k* components to write the description file of the new system. On the other hand, if I build these components from scratch, I simply need to figure out how to combine the build rules of *p* components to form the description file of the new system. As a result, I obtain:

$$IM_{make} = \frac{k}{p} \times 100 = \frac{10^4}{u}$$

where

$$u = \frac{p}{k} \times 100$$

denotes the percentage of the adapted components in the existing systems. To make the story simple, I assume that only the extraction from the containing description files, but no other extra work, is necessary to reuse the build rules. This implies:

$$DM_{make} = 0 \text{ and } CM_{make} = 0$$

I obtain:

$$AAF_{make} = 0.4 \times 0 + 0.3 \times 0 + 0.3 \times \frac{10^4}{u} = \frac{3 \times 10^3}{u}$$

As a result:

$$EDSI_{make} = pb \times \frac{30}{u} + (n - p)b = \frac{30\,pb}{u} + \frac{n}{100}(100 - t)b$$

where

$$t = \frac{p}{n} \times 100$$

denotes the percentage of the adapted components in the new system.

On the other hand, when I use Object Make, it is reasonable to assume that:

$$DM_{omake} = 0 \text{ and } CM_{omake} = 0 \text{ and } IM_{omake} = 0$$

because no extra work is necessary for adaptation. As a result, I obtain:

$$EDSI_{omake} = (n - p)b = \frac{n}{100}(100 - t)b$$

Therefore:

$$\frac{EDSI_{make}}{EDSI_{omake}} = \frac{\frac{30\,pb}{u} + \frac{nb}{100}(100 - t)}{\frac{nb}{100}(100 - t)} = \frac{30t}{u(100 - t)} + 1$$

Finally I obtain:

$$ratio = \frac{MM_{make}}{MM_{omake}} = \left(\frac{30t}{u(100 - t)} + 1\right)^{1.12}$$

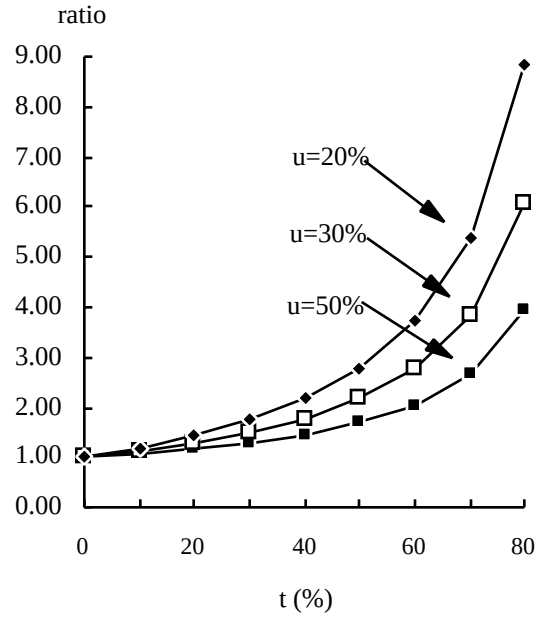

Figure 5: Effort comparison of Make and Object Make

Figure 5 shows *ratio* as a function of *t*. From this graph, I can observe several facts. First, the value of *ratio* is always bigger than 1. This implies that Make requires more effort than Object Make does. When *u* is fixed, *ratio* increases as *t* increases. This implies that Make requires more effort when I adapt a large number of components, than it does when I adapt a small number of components in the new system. When *t* is fixed, *ratio* increases as *u* decreases. This implies that Make requires more effort when I adapt components from large systems, than it does when I adapt components from small systems.

6. Conclusion

Currently, I am using Object Make and **xomk** in my research lab and my courses for programming in the C and C++ languages. Object Make is now quite popular among the students, and it is one of their indispensable tools now.

The students found that Object Make is quite handy to promote the reuse of classes when they use object-oriented programming languages, such as C++. One of the advantages of classes is that the students can reuse classes without knowing how the classes are implemented by virtue of encapsulation. Object Make allows them to reuse classes even if they don't know how to compile the source code of the classes. This was also quite effective for me to teach the concept of encapsulation to the students.

They also found that Object Make is handy to debug and test software components. They typically link a component with a main program that includes a test driver for a unit test, while they link the same component with other components in the system for an integration test. Please note that this is a special form of reuse of the components. Students reuse a single component in different subsystems. If they use Make, they need to write description files that are slightly different whenever they test the component in different combinations. However, with Object Make, they can use the same description file of the component for various tests.

I believe that, from the observation above, I can conclude that Object Make is a quite effective tool to reuse software components. Although Object Make is a fairly complete tool, I am still improving **xomk** to offer better user interface. The mechanisms that I am currently developing include an edit mechanism that allows users to create and modify their description files without leaving **xomk**, and a mechanism to show the relationship among components that allows users to see the whole structure of systems in a graphical notation.

References

[1] Boehm, B. W. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[2] Feldman, S. I. MAKE - A Program for Maintaining Computer Programs. *Software-Practice and Experience*, vol.9, pages 255-265, 1979.

[3] Kaiser, G. E. and P. H. Feiler. An Architecture for Intelligent Assistance in Software Development. in Proceedings of the *9th International Conference on Software Engineering*, pages 180-188, Monterey, California, ACM-IEEE, March, 1987.

[4] 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.

[5] 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.

[6] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. in Proceedings of the *ACM Symposium on Object-Oriented Programming Systems, Languages and Applications*, pages 214-223, Portland, Oregon, ACM, September, 1986.

[7] Quercia, V. and T. O'Reilly. *X Window System User's Guide*. O'Reilly & Associates, 1993.

[8] Rochkind, M. J. The Source Code Control System. *IEEE Transactions on Software Engineering*, vol.SE-1, no.4, pages 364-370, December, 1975.

[9] Sugiyama, Y. Producing and Managing Software Objects in the Process Programming Environment OPM. in Proceedings of the *First Asian Pacific Software Engineering Conference*, pages 268-277, Tokyo, Japan, IEEE and IPSJ, December, 1994.

[10] 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.

[11] Thall, R. M. Large-Scale Software Development with the Ada Language System. in Proceedings of the *ACM Computer Science Conference*, pages 55-67, Orlando, Florida, ACM, February, 1983.

[12] Tichy, W. F. RCS - A system for Version Control. *Software - Practice and Experience*, vol.15, no.7, pages 637-654, July, 1985.