1. Distributed Object Oriented Make Development System

1.1 Introduction

The basic concept of data processing is that we have some kind of data then we apply a program to it to generate other data for further processing or use. That is quite simple, but if we are working with large amount of data it becomes hard to know what data sets and programs we use to produce a particular result. Moreover some of the data sets sooner or later will be replaced, in which case all of the data we produced with the use of this data set becomes *obsolete* (or *out-of-date)*.

So the problem is not to write a program which can generate one type of data from other type of data, but a program that can decide how to generate a specified data from the available data.

There are two ways to generate more complex applications: write a brand new program or combine simpler programs to produce a complex processing system. The former requires more effort to develop the application, but the result will be more effective. The latter will be developed much faster, and the maintenance will be much easier (*Figure 1-1*).

To develop simple and small programs and build more complicated applications from them has many advantages such as easier development, re-usability and less hardware requirements. However the interfaces between these applications must be designed very well.

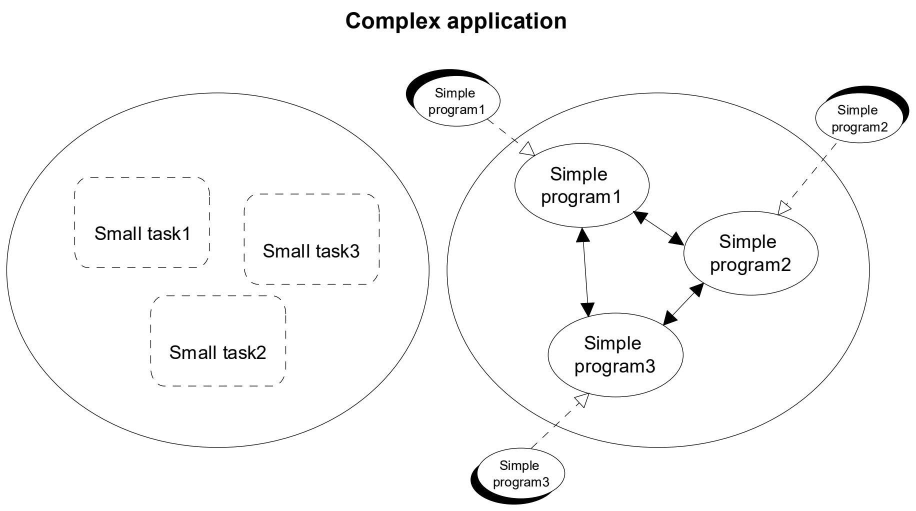


Figure 1-1 Two ways to develop application

Let us take a closer look at the software development process. First we apply compilers to the source files, and many cases we apply further programs to the generated object code (i.e. linkage editors etc.). In different applications we use different compilers and we apply them in different orders.

The problem becomes more complicated if after the compilation some of the data sets have been modified. In that case not all the source files must be compiled, but the modified ones. And of course object files generated from the modified source files will be modified and the results or other object files generated from them must be recompiled (re-linked).

The most straightforward algorithm to solve this problem is to recompile (or re-link) all files right after they were modified. However it is a very cost ineffective method as the following example shows. Assume that an application was built. After that a set of the source files was modified. By the algorithm after the first source file were modified the file will be compiled and the generated object file will be linked, and this will happen to all the source and generated object files. Of course in that case not all the linking were necessary but the last one since the result of the previous ones were overwritten. Everyone can see that this algorithm produces a computational overhead.

Another approach turns the previous algorithm upside-down. This means that we not look at what files generated from a certain file, but wee look at what files needed to generate a certain file. In that case after modifying a source file nothing happens. The action is taken when a result is needed. First we should examine if it is available and if it is generated from the appropriate latest source files. If not the algorithm is applied recursively and the result is built.

A considerable effort has been made to solve this problem, particularly for software development tasks. A utility called *MAKE* was developed to keep track of dependencies between files. There are other solutions, but these programs are rather platform dependant and not a standalone utility, but a part of an integrated development environment (IDE).

In the following paragraphs we introduce some example problems we will use to demonstrate the process of making data up-to-date. Our first example is the standard MAKE software development process.

Example 1:

The example is maintenance of an image displayer/converter application. It contains a resource definition file *DRAW.RC*, which includes a header file *DRAW.RCH*. The project contains three C++ source files *DRAWING.CPP*, *MAIN.CPP* and *CONVERT.CPP*. The source file *CONVERT.CPP* includes a header file *CONVERT.HPP* and the source file *DRAWING.CPP* includes a header file *DRAWWING.HPP*. The source file *MAIN.CPP* includes both *CONVERT.HPP* and *DRAWING.HPP* and also includes a header file *MAIN.HPP*. A resource file *DRAW.RES* is produced by the resource compiler from the file *DRAW.RC*. The C++ source files are compiled by a C++ compiler, producing three object files *DRAWING.OBJ*, *CONVERT.OBJ* and *MAIN.OBJ*. The object files and the resource file then linked together by a linkage utility producing the executable file *DRAW.EXE*. The structure of the project is represented on the *Figure 1-2*.

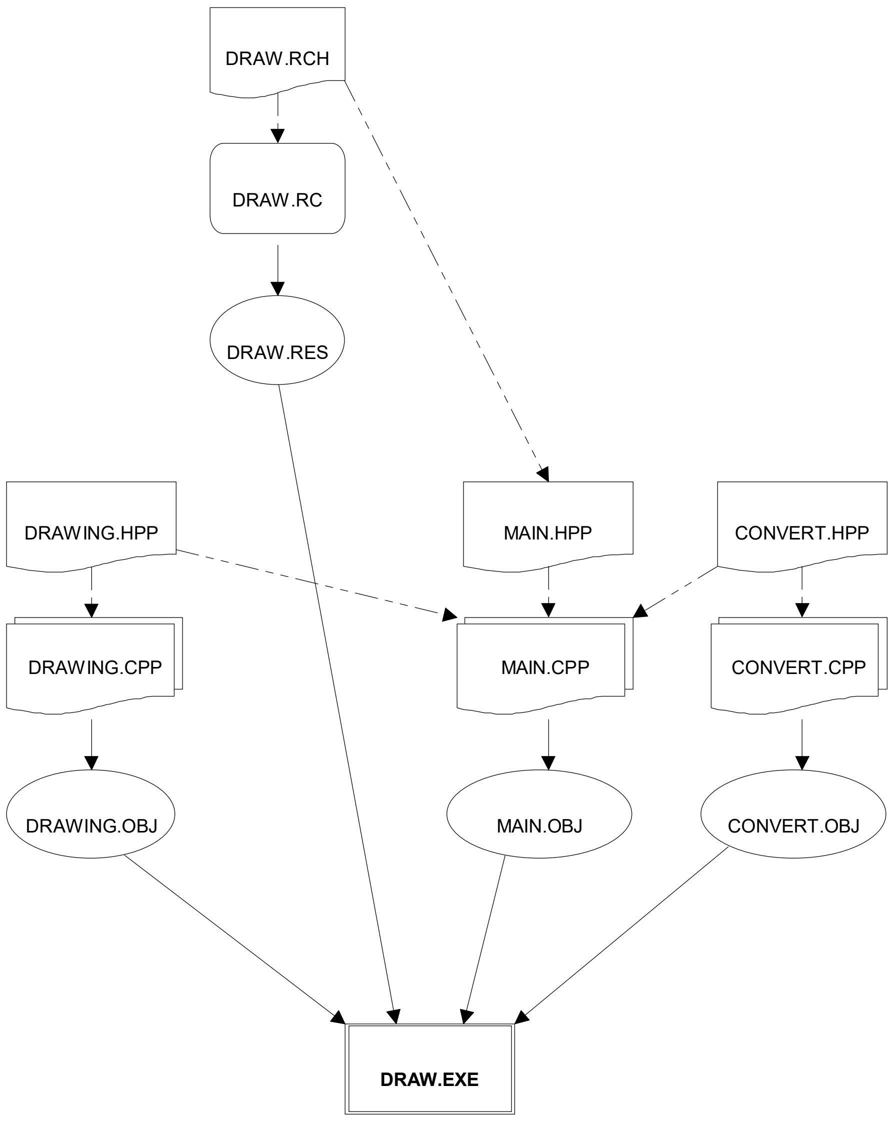


Figure 1-2 Software development example

2. The Traditional Make Utility

One of the major achievements in the software development area is the MAKE utility. Here we will discuss the abilities and weaknesses of this tool. To understand how the MAKE works, here we discuss the concepts and terminology. The files represent computer data. No other data types supported (i.e. the file is a smallest unit the make can deal with). The files have one property, the last time they were modified. Based on this information, and of course information provided by the user in the makefile the MAKE utility can decide which files must be recompiled (and how) to make the required file up-to-date.

2.1 *Concepts*

The primary goal of the developers of the MAKE utility was to aid software development. To understand why they used that particular solution we describe here the most widely used way to develop software products. The program is built from different object files which are produced mostly by C or other compilers from source code or code generated by sophisticated source-code generators. When a source code changed all object or other source files should be recompiled which is derived from this source file. However, in most of the cases, when a new feature or bug fix is introduced to the program, more than one file should be changed. For this reason it is a waste of computer resources to recompile the project when a file is changed. The logic behind the MAKE utility is not “This file is changed. What, should I remake?”, but rather “I want this, what should I remake to make it up-to-date?”. This concept is like the top-down design method.

Each generated file is called a **goal**. A goal depends on its **dependant**s. The goal is called **up-to-date** if its dependants exist and are up-to-date, and the goal exists and newer than any of its dependant. The source files (which are not generated, but given) are always up-to-date.

In the *Example 1* the main goal is the executable file *DRAW.EXE*. It depends on the files *DRAWING.OBJ*, *MAIN.OBJ*, *CONVERT.OBJ* and *DRAW.RES*. The executable file is generated by the shell command:

```
LINK MAIN.OBJ DRAWING.OBJ CONVERT.OBJ DRAW.RES
```

2.2 *Files*

A makefile is used to control the make process. The makefile contains information about the project goals, the commands used to rebuild them and the dependency relations between them.

The makefile consists of explicit and implicit rules, variable and option definitions. For example if we want to compile the program HELLO which is compiled by the command.

```
CC -O HELLO HELLO.C
```

we use a makefile which defines a **goal** *HELLO* and defines its **dependant**s *HELLO.C* and *HELLO.H* if *HELLO.H* is included from *HELLO.C*. Here we call the command producing the program *HELLO* the **make action**. It is possible to use other utility than compiler, for example the *LEX* or *YACC* which generate a C source file from some specific files.

The term **relation** means a binary relation between two objects. In the traditional make only the *newer-than* relation is available. The makefile contains the project goal, its dependants and the make actions necessary to produce the goals. The UNIX makefiles contain *option*s, *macro definition*s, *comment*s, *explicit* and *implicit rule*s. The first three provided for convenience and customisation, for us the implicit and explicit rules are important. The structure of the explicit rule is the following (*Figure 2-1*):

```
goal: dependant1 ... dependantn
  action1
  ...
  actionn
```

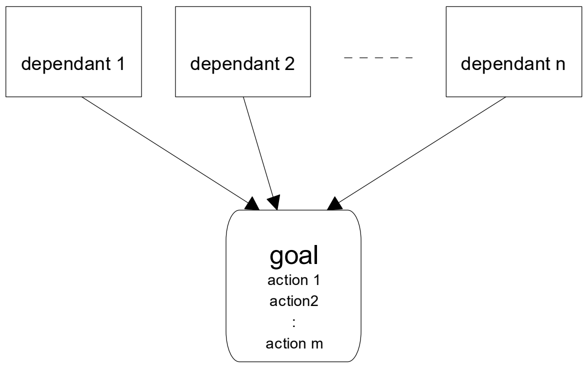


Figure 2-1 Goal and its dependants

This construct describes that the goal 'goal' depends on 'dependant$_1$' ... 'dependant$_n$' and to build the goal it is necessary to execute the shell commands 'action$_1$' ... 'action$_n$'. The dependant list and the make actions are optional but naturally it is senseless to omit both. There are utilities that can produce the dependant list automatically based on the make actions. For example the *MAKEDEPEND* utility is widely used to produce dependency information for makefiles scanning the C source code for include files. Also C compilers have options which can be used the produce such dependency lists. The typical example of implicit rules is this:

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

This describes that files end with '.o' can be produced from files with the same name except '.c' instead of '.o' by using the shell command

```
cc -c filename
```

when *filename.o* produced compiling *filename.c*.

```
CC = cl
LINK = link
RCOMP = rc
RM = del
BJS = drawing.obj main.obj convert.obj
CFLAGS = -c

# implicit rule
.cpp.obj:
  $(CC) $(CFLAGS) $<

.res.rc:
  $(RCOMP) $<

# explicit rule (draw.exe)
draw.exe: $(OBJS) draw.res
  $(LINK) $(OBJS)
  $(RCOMP) draw.res

# non file goal
clean:
  $(RM) *.obj *.res

drawing.obj: drawing.cpp drawing.hpp
convert.obj: convert.cpp convert.hpp
main.obj: main.cpp main.hpp drawing.hpp convert.hpp
draw.res: draw.rc draw.rch
```

In the example makefile we can see almost all kind of makefile definitions. This makefile include variable definitions, implicit and explicit rules. The variable definitions are used to define command names of compilers, linkers and command line options and the names of the object files. The implicit rules define the compilation method of C++ source files and resource definition files. Implicit rules can be applied to filename suffixes listed in the SUFFIXES make option. The makefile contains explicit rules with and without specified make actions. Here we can see, how explicit rules can be used with implicit rules to define actions by implicit rules and dependencies by explicit rules. The generation of this kind of explicit rules can be highly automatic. There is a goal 'clean' which not really refers to a file. Since the defined make action does not generate the file named 'clean' (and we can assume there is no such file in the current directory) the goal 'clean' is never up-to-date and the command 'make clean' will always take that action.

Notice that the object files do not depend directly on the C++ header files. For example *MAIN.CPP* depends on *MAIN.HPP*, *CONVERT.HPP* and *DRAW.HPP* can be expressed in a quite difficult way in the makefile. It is easier to write things like above instead of things like this:

```
main.cpp: main.hpp drawing.hpp convert.hpp
  touch main.cpp
main.obj: main.cpp
```

Since the *MAKE* utility was developed for compilation all the goals and dependants are supposed to be files. However, it is possible to define goals which are not files, for example the goal *clean* is usually defined to be a non-file goal. Of course, because there is no file named *clean* created during the making the goal *clean*, the goal *clean* will never be up-to-date unless there is a file named *clean* in the actual directory. The clean goal is usually defined in the following way:

```
clean:
  rm -rf *.o core
```

That means all object files, and a core file should be removed to make the project clean. In this case the goal *clean* is a non-file goal, and it has no dependants.

2.3 Make process

The make works in the following way:

1. the make is called with the name of the goal and other options
2. the make reads the makefile in the current directory unless other makefile is specified with command-line options (in UNIX systems the file named *makefile* of *Makefile* used in that order, if both exist a warning is printed).
3. if no goal is specified at the command line the first goal is used
4. if the specified goal does not exist in the makefile the make utility prints an error saying that it does not now how to make the goal.
5. the make utility checks if the goal is up-to-date by checking whether the file with same name as the goal and the dependants are existing, and the goal is newer than its dependants. If all the files exist, and the goal is the newest then no action is taken, and the goal is said to be up-to-date.
6. the make checks if the dependants are up-to-date and if not, it regenerates them
7. the goal is made by the specified shell command(s)

The structure of the make file allows the users to define a wide range of activities, not only to build programs, but build other things (for example documentation), and take different actions (for example installing the software).

Because the makefile is a simple text file it is easy to write makefile generators, and other make-related tools. Some of them:

- *MAKEDEPEND*: extract dependency information from source files, based on the information contained in the makefile, and checking the source files it creates a complete dependency information of the project goals.
- *IMAKE*: create a makefile from a machine independent imakefile, and machine dependent make templates. The templates are installed at the machines, and the imakefiles are provided with the software.

3. Distributed Make Utility

This chapter describes the concepts of the Distributed Make, and the requirements the Distributed Make should meet.

To illustrate the Distributed Make we will slightly modify our previous example. In that example we introduced a program that converts and displays images. We assume there are two types of image (*BMP* and *PNM*). The program should display these images. The two image type processing functions will be implemented in two computers. For example a *BMP* specific part can be implemented in an IBM PC and a *PNM* specific part in a workstation. The rest of the program can be implemented in any of the computers. Then the Distributed Make is used to build the project. Let us assume that all the not image specific parts are implemented in the PC. In that case we can see the structure of the project in the *Figure 3-1*. If we build a project in the PC than we see that the files *PNM.HPP* and *PNM.OBJ* are remote dependants, and a remote action may be necessary to compile the object file *PNM.OBJ*.

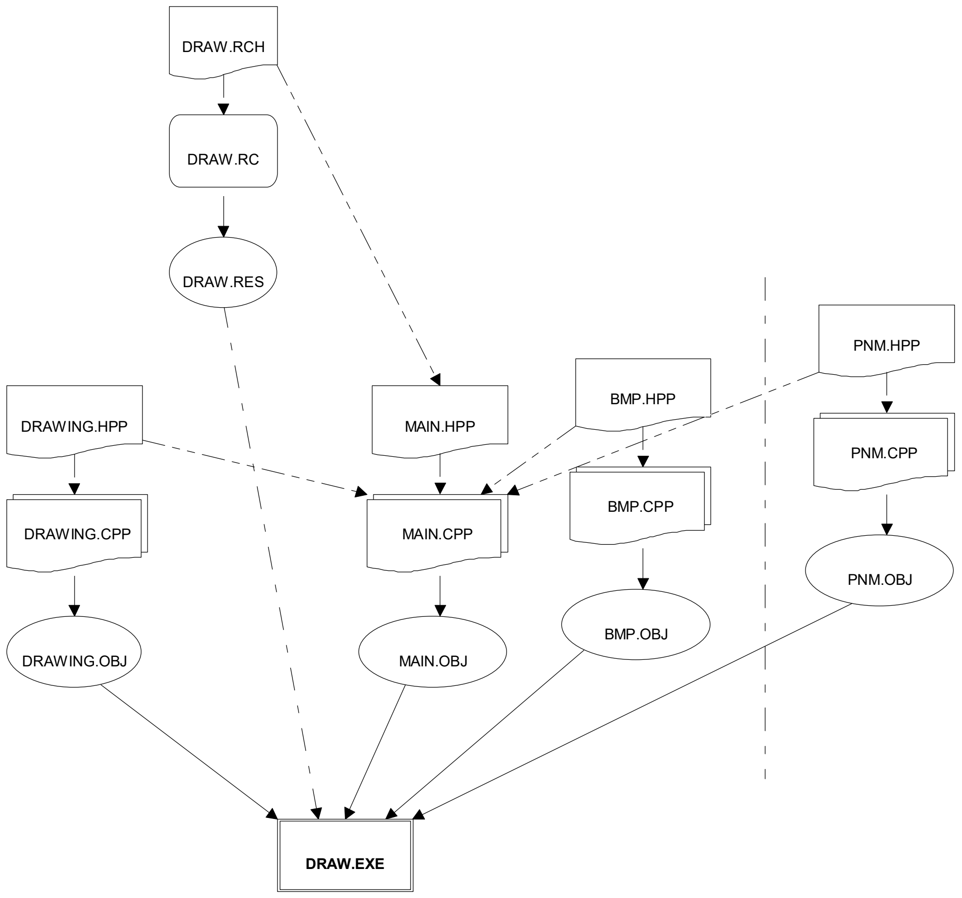


Figure 3-1 Software development example on multiplatform environment

3.1 Network, distribution concept

Computers can be used more efficiently if they are connected to each other by a computer network. The set of physically connected computers is not yet a network environment, the computers must communicate with each other with some kind of network protocols. The computers can communicate with different protocols. In fact different size of computers mostly use different protocols to communicate.

The Distributed Make utility should know all the necessary protocols to be able to communicate with each host, and carry out all the make action and remote copies. Software development on distributed environment has a number of advantages.

The major advantage is *sharing code*. This means, that a code developed on one host can be used on another host, and the newer versions become available immediately. Some softwares are running on several hosts. The software products based on the client-server model contain several components running in different type of hosts communicating with each other by a network. Obviously this type of software must be developed on a networked environment.

A networked environment also useful when a *multiplatform software* is developed. If we want to develop a software which is capable of running under different operating systems, a network can connect the development sites to each other to transfer the core code (which is machine independent). In the case of large software project it is possible that some parts of the software were written on one machine and other parts of the software are written on different machines. So the source files or the object file should be transferred between the machines to build the complete software. With distributed make it is possible to use the same makefile on all the hosts and build it on any of the host. However, a makefile can define the host where the actual compilations take place.

3.2 Problems with network distribution

One of the problems in the networked environments is the *wide range of available protocols*. This is not a problem itself but implies a problem that different computer communicating with each other with different protocols will never understand each other. The best solution would be the use only one protocol (but which?) that all the host can understand.

Other problem is the *data representation*. Different computers represent data in different forms, and the conversion between these forms is not always easy.

Another problem can be the *temporary unavailability* of some hosts. This means that some of the hosts on the network is unreachable due to some hardware problem in the lines (this is mostly a problem for wide range networks). In that case it is possible to suspend the execution of a (probably remote) command, take other actions, and if the connection is established again the command can be resumed.

In wide area networks (WANs) the different time zones may introduce additional problems. Even in the case of a local area network (LAN) unsynchronised system clocks may cause problems.

3.3 Data conversion on multiplatform network

In heterogeneous computer network a different file representations should be converted to each other. For example the DOS and WINDOWS based systems represent the end-of-line with a two character sequence (CR and LF) while in UNIX systems only the LF character terminates the lines, in other computers (for example the Apple Macintosh) the CR character is a line terminator. However, not all the files should be converted (for example images must remain the same).

There are other data than files, for example the last modification time of a file, represented in different forms on different platforms.

One of the major problems is the naming of files. Some operation systems use very limited file names for traditional reasons.

One solution of these problems is to define object (time, filename, etc.) types and in these objects to define default conversion functions to each two of supported platforms. An other solution is to support a standard format, and provide conversion functions to this format. In this case the conversion requires more computational effort, but the number of conversion functions is much less, and the set of conversion functions is easier to extend.

3.4 Make process

We introduce the following technical terms for Distributed Make:

- ***local host*** is the host where the Make utility is executed
- ***remote hosts*** are all the other hosts
- ***remote makefile*** is the makefile that resides on remote host
- ***remote goal*** is a goal residing on a remote host
- ***remote dependant*** is one of the dependants of a goal resides on a remote host
- ***remote action*** is a make action that should be performed on the remote host. For example a remote copy when one of the files are copied between different machines is a remote action.

We intend to use the same makefile on all hosts. This means that each object can be local or remote. In that case all objects described as remote objects (the description of the object contains the host it resides on).

When the MAKE command is executed on a host, first the Make looks for the makefile. If not specified in the command with the '-f' flag the make utility looks for the file named 'makefile' and 'Makefile' (in this order, a warning message is issued when both files exist) in the current directory. In the case of the Distributed Make utility a remote makefile can be used.

After the utility found the makefile parses it. After that the Make looks for the goal (which is the first goal in the makefile unless specified in the command line). If the Make encounters a remote dependant it makes a local copy of it. In the local command this local copy is used. If a goal is remote it is either made locally and later copied into its host or made by a remote command in which case the Make must make sure that all dependants are copied into the remote host where the remote command is executed.

While the host is temporary unavailable the normal behaviour of the make is to issue a warning message, and continue the make process with other independent actions, or issue an error message and stop. A command line switch determines what the Make will do in this case.

Example 2: In this example we look at the work of a network manager. He or she maintains host, information services and databases in the network. In this example we can illustrate remote and shared objects and actions. One activity of the administrator is to make a backup from a specific host to a specific device. In this case the host where the backup made from and the backup device are the most important objects. Both of these objects can be remote or local. It is possible that both of these objects can be remote in the same time. The backup process can be a remote action. Also some special relation can be introduced, for example a relation can be if a cartridge mounted in the tape device used as backup device. Mounting a tape device itself can be a make action. The backup may depend on that action. The backup may also depend on the relation between the 'free space' attribute of the mounted backup device and the 'allocated space' attribute of the disks the backup made from.

The maintenance of a database or network information system can be described with activities. Such as the database update may be a make action. Let us assume that new data files arrived. The new files are represented by objects the database depends on. When the make is executed the make actions take place that incorporate the new data to the database. The database and the data files can be remote or shared objects.

4. Object Oriented Make

The key idea of the whole development is the object-oriented design. In the traditional MAKE utility every goal and dependent is file, so the Object Oriented Make everything is represented as object. The *make activities* are described as objects. A set of predefined object types (**class**es) can be used to define activities. This object definitions may take parameters (mostly other objects, but other parameter types are also possible) to describe the unique attributes of the activity. The activities has dependants (objects which the activity depends on), and the program checks if an object is completed (up-to-date) before trying to make it. The general improvement in this concept is to use objects as project goals, rather than files (as in the standard Make).

We can say that in the traditional MAKE all objets are files, but it is not true in some viewpoint. Lets say that all the files are objects. The objects has properties such as their last modification time and command how they can be created. We can divide the objects in different classes, such as C source files, object files, etc. Let us allow different object types than files for example variables, options or actions. Then we can express make actions as objects.

Here we can ask how these objects relate to each other, and we can define relations between objects. With relations we can express dependencies. So we can express make actions as object properties (a make action can be itself an object) and dependencies as relation between object. Than we start to extend the object types (classes) and relations, and we achieve a utility that is a generalisation of the standard MAKE utility. The result will be rather different than the MAKE, but it works with the same concepts, and can handle not only files but any (implemented) type of objects.

The applications of the Object Oriented Make will depend on the implemented Make classes. So the extensibility of the Object Oriented Make is very important.

4.1 General introduction to the object oriented concepts

In this chapter we introduce the basic concepts of object oriented programming, such as ***encapsulation*** and ***inheritance***.

While in the traditional function oriented programming concept data and functions that process the data are handled separately, in the object oriented concept they are encapsulated into one structure called object. The techniques used in object oriented programming are also different because of the specific properties of the objects. An application may contain different kind of objects. The same type of objects are called object classes, a specific object is called an ***instance*** of its object class.

The most used technique to create new object classes is inheritance. Inheritance means that we use previously defined object class or classes to define a new class. The concept is that the most general object classes are created first, then a more specific object classes are created by inheritance from the general ones.

The new classes are inherited from their ***ancestor***s, by adding new data and function fields or redefining existing function fields. The classes have specific function fields called ***constructor***s and ***destructor***. An object class may have more than one constructor, but only one destructor. A constructor is used to construct a new object instance and initialise its data fields. The destructor is used to destroy an existing object instance by deallocating the dynamically allocated data, or perform other actions before the deallocation of the object's data.

The access to the data fields of the classes can be restricted. Three kind of access level available in C++ the ***private***, the ***public*** and the ***protected***. The private data and function fields are available from the functions of the class. The public fields are available from any place in the program, while the protected fields are available from the class and its successors. One may specify access levels in inheritance to restrict the successor's access to it's ancestor's fields. The access rights may be overwritten by defining ***friend***s.

In some cases an inherited functions should call one of the actual object's function. To do this a use of ***virtual function*** is necessary. If in the ancestor a function is defined as virtual, and the same function is defined in the successor, that means the last fuction overwrite all of the previous definitions.

It is possible you do not need to implement (only declare and use) virtual functions in an object class. Such a class is called an ***abstract class***, since it is not functional (the virtual functions are unimplemented). These kind of classes can be used as ancestors of other objects, which can be a normal (non-abstract) class by defining all the missing virtual functions. To express that a class is abstract we assign a zero (0) value to the unimplemented virtual function.

```
class CAbstractClass
{
public:
   ...
   virtual int Function( void ) = 0;
   ...
};
```

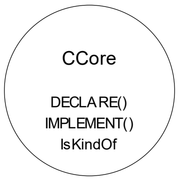
4.2 Object Oriented Make classes

The Object Oriented Make has some basic make classes. A part of these classes are abstract classes. That means they are never used in makefiles but serve as an ancestor of definable make classes. The abstract classes make the creation of new make and property classes easier.

The main abstract class is the *CCore* class. The *CCore* class has an important virtual function, the *IsKindOf* function. This function returns a pointer to the object instance if its parameter is the string "CCore", otherwise it returns NULL. If this function is implemented correctly in the derived objects, then each object can return its components (parts type of its ancestors). This function recursively calls the same functions of the ancestors which can return a pointer to the part of the object given by the parameter.

```
class CCore
{
public:
   ...
   virtual void * IsKindOf(
                       const char * pClassName );
   ...
};
```

```
void * CCore::IsKindOf(
                  const char * pClassName )
{
  if ( !strcmp( pClassName, "CCore" ) )
      return (CCore *) this;
  return NULL;
}
```



The implementation of the *IsKindOf* function is quite straightforward and some macros also provided to make it easier. However these macros are difficult to use in some cases due to the possible large number of parameters. The make and property class implementors must know how to create the *IsKindOf* function of the new class since it is important for the makefile parser. First the object should save its class name. The class name can be saved into a static variable since all object instances of the same class have the same class name. This class name can be set in the construction or can be hardwired into the class code. Then the *IsKindOf* function must be implemented. This function must check if its parameter is the same string as the object's class name and return a pointer to the object (the *this* pointer) if it is. Otherwise it should call the *IsKindOf* functions of the ancestors of the object class.

```
class CExample : public CCore
{
public:
  static const char *  classCExample;
  virtual void * IsKindOf( const char * pClassName );
  ...
};

void * CExample::IsKindOf( const char * pClassName )
{
  if ( !strcmp( pClassName, classCExample ) )
      return (CExample *) this;
  return CCore::IsKindOf( pClassName );
}
```

When the class has only one ancestor this can be done by the macros DECLARE and IMPLEMENT. However in most of the cases the object has more than one ancestor, since properties are defined by inheritance, other macros can be used (IMPLEMENT2, IMPLEMENT3). To learn more about the usage of these macros see the example.

Helper macros:

```
  DECLARE( <class name> );
  IMPLEMENT( <class name>, <base class name> );
  IMPLEMENT2( <class name>, <base class name1>,
                                  <base class name2>
  CLASS( <class name> )
```

Derived class declaration:

```
class CExample : public CCore
{
  DECLARE(CExample);
public:
};
IMPLEMENT(CExample, CCore);
```

Using derived class and *IsKindOf* virtual function:

```
CExample Example;
CCore *  pCoreClass = &Example;
void *   ret = pCoreClass->IsKindOf( CLASS(CExample) );
  Result: ret = pointer to Example
void *   ret = pCoreClass->IsKindOf( CLASS(CCore) );
  Result: ret = pointer to CCore part of Example
```

4.3 Object Oriented Makefile

The Object Oriented Make uses *Object Oriented Makefile*, i.e. the makefile contains object declarations and dependency information. The make actions are determined by the object types. In the makefile we can use instances of predefined classes. The class definitions are 'wired' into the Make program, and can be extended by extending the Object Oriented Make itself. Of course, we can set attributes in the object declarations as well.

4.4 Makefile parser and Make Objects

In our concept the role of the Make utility is to parse the makefile and create the objects defined in it. Everything else is done by the objects. However, it seems to be necessary to provide information on the host (where the make is running) and the software environment (i.e. operating system, environment variables, available programs, utilities, etc.). This can be done with the use of a core make class, in which case this information is provided by the functions implemented in the core make class. All the Make Objects are the dependants of this class.

The Make Object definitions consist of the object type, the object name (the name of this object instance) and the object parameters which give the actual values of the member variables of the object type. The parameters may include the host (to define remote objects) or this parameter can be specified in the object name.

Every Make Objects are derived from *CCore*. Using the *IsKindOf* virtual function has the following advantage:
We should see what can be done in compilation time and what should be done in runtime. At the parse of the makefile a certain property of an object is needed. We can not use type-casting because the property class is available at runtime. Instead we use the *IsKindOf* virtual function.

The dependence relations are defined in a traditional makefile-like way. First the goal is specified, then a colon then the dependants. The dependants can be not only objects, but properties of the objects (for example myfile.modtime). If no property specified the default (the last-modification-time for file objects and other reasonable properties for other objects) property is used.

4.5 Special object types for Object Oriented Make

In the Object Oriented Make implementation we want to use two kind of object types, the Property classes and the Make Object classes. The Property classes define properties of the Make Object classes. If an object type has a certain property the given Property is inherited by the Make Object class.

4.5.1 Property Object Type

The Property Object type (class) represents a certain property. Relations are defined with the help of properties. All Property classes are derived from a core *CProperty* property class.

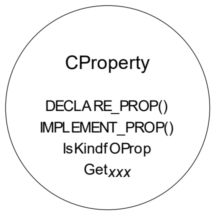
As you will see later the properties define relations and these relation functions are needed for the makefile and the objects and the relation (property) names are available only at runtime, it is necessary to retrieve a property ancestor of the given object. Fortunately that can be done by the *IsKindOf* function. The *IsKindOf* function must be implemented the same way as described at the introduction of the *CCore* class. Unfortunately the relation functions in the property objects can not be declared as normal or virtual functions due to the inheritance used in the definition of make classes, so relations must be declared (and implemented) as static.

This makes necessary to implement a virtual function named *IsKindOfProp* in the base property class (*CProperty*). The *CProperty* class is an abstract class, and it is used as the ancestor of all property classes. The *IsKindOfProp* virtual function is like the *IsKindOf* function except it has a second parameter, which used to return the relation function defined by the property. This is needed, because relations should be declared as static functions. The *IsKindOfProp* function must be declared and defined the same way as the *IsKindOf* function except is must return a relation function in its second parameter.

Macros are also provided for declaration and implementation of this function, which have the same limits as in the case of *IsKindOf* function, however they are more useful since property classes are mostly inherited from another property class. There are macros called DECLARE_PROP and IMPLEMENT_PROP, that makes easier the declaration and implementation of these functions. See the examples to learn more about the usage of these macros.

The *CProperty* class has another unimplemented virtual function, the *Get*xxx function, where the xxx is the name of property value. This function simply returns the value of the property.

```
//////////////////////////////////////////////////////////
// CProperty
class CProperty : public CCore
{
  DECLARE_PROP(CProperty);
public:
  // example: value of property
  virtual CValue   GetValue( ... ) = 0;
  ...
  virtual CProperty * IsKindOfProp(
                        const char * pPropertyName,
                        RELATION_FUNCT* * pRelation = NULL
);
  ...
};
IMPLEMENT_PROP(CProperty, Ccore);
```



4.5.2 Make Object Type

The Make Object types (classes) are derived from the core *CMakeObject* (and *CCore*) make object type. These classes can be used to define objects in the makefile. All make actions are performed by these objects. The core Make Object has three basic functions. The first is the completed function (*IsCompleted*) which determines that the object is up-to-date or not. The second is the *MakeMe* function which takes the necessary actions to make the object up-to-date. The return values of these functions are either yes or no. The yes means that 'yes - the object is up-to-date' or 'yes - it is successfully made up-to-date', the no means that the object is out-of-date or the remake of the object was unsuccessful due to some errors. There is a third reasonable answer for the first function which says that the answer is not available, i.e. during the computation of the answer an error occurred. The *Construct* function allows to dynamically construct Make Objects.

```
//////////////////////////////////////////////////////////
// CMakeObject
class CMakeObject : public CPropCompleted
{
  DECLARE(CMakeObject);
protected:

  CMakeObject( void );

  CMakeObject( const char * Name, int Type );

public:
    virtual ~CMakeObject();

    enum EState {
                    STATE_INITIAL,
                    STATE_NOTCOMPLETE,
                    STATE_BLOCKED,
                    STATE_READY,
                    STATE_MAKING,
                    STATE_COMPLETE,
                    STATE_ERROR
    };

  // Make Object operators

    virtual int      Communicate( HCOMM hComm, const char *
  Command );

    virtual int      ServerGetValue( HCOMM hComm, const char
  * Property, const char * Command ) = 0;

  // abstract Make Object operators

    virtual BOOL     IsCompleted( void );

    virtual const char * GetDefProp( void ) = 0;

    virtual BOOL     IsExist( void ) = 0;

    virtual BOOL     CleanUp( void );

  const char *       GetName( void ) const;

    virtual char *   GetFullName();

    static CMakeObject * NameToObject(
                    const char *  ObjectName );
    CProject *       GetProject( void );
    CProject *       GetTopLevelProject( void );

    CDependList *    m_Dependants;
    virtual BOOL     BeginMake(
                        BOOL        Error );

    virtual BOOL     EndMake(
                        BOOL        Error );

    virtual const CHostObject *      GetHost( void );

    virtual COutput *    GetOutput( void );
    void                SetName(
                            const char * Name );

    virtual Estate      Request( CMakeObject *obj );

    virtual int         Notify( int succes, CMakeObject *obj );

    virtual int         Transfer( HCOMM hComm = NULL );

    virtual int         NotifyClients( int Success = -1 );

    void                SetState( EState State );

    CMakeObject::EState  GetState( void );

  protected:

    int                 IsPropCompleted( DepListItem& Dependant
  );

    int                 IsPropCompleted( const char *
  pPropertyName, CProperty& OtherProp );

    virtual BOOL        Init( void );

    virtual BOOL        OnChangedDeps( CDependList& Dependants
  );

    int                 Execute(
                            const char * Title,
                            const char * CmdLine );

    static const CORE_ENTRY * serverGetValue(
                            HCOMM        hComm,
                            const char * Property,
                            const CORE_ENTRY * Entry );

    char *              m_Name;

    char *              m_ParamsString;

    CObList_            m_Clients;

    CObList_            m_Depends;

    CHostObject *       m_pHost;

    CProject *          m_pProject;
    CParamList *     m_pParams;

  private:

    EState           m_State;

  #ifdef UNIX
    pid_t            pid;
  #endif

  // overridable Make Object operators

    virtual BOOL     MakeMe( void );

  friend MakeDlg;

  friend CProject;

  friend CServerProject;

  friend CProperty;

  friend CHostObject;

  };
```

The member functions and variables will be explained below.

4.5.2.1 The Communicate function

This function is called upon communication requests. Override this function to define protocol extension to the make protocol.

4.5.2.2 The ServerGetValue function

Convenience function to return certain data to remote parts of the object.

4.5.2.3 The IsCompleted function

Returns TRUE if the object is up-to-date. It uses the m_Depandants list and the IsPropCompleted and IsExists functions to determine if the object is up-to-date.

4.5.2.4 The GetDefProp function

Returns the name of the default property (see properties). This may be different for each type of objects. For most of the classes this is the ModifyTime property (for traditional applications).

4.5.2.5 The IsExist function

Returns TRUE if the object's data exist.

4.5.2.6 The CleanUp function

Performs cleanup after make actions.

4.5.2.7 The GetName function

Returns the object name. Each object must have a unique name within the same project. See also the SetName, GetFullName functions and the m_pName field.

4.5.2.8 The GetFullName function

Returns the full object name of the current object, that is the full name of the parent project appended a '#' (hash mark) and the object name to it.

4.5.2.9 The NameToObject function

Returns an object with the given name if exists, otherwise returns a NULL pointer. This function searches the entire hierarchy. See also the GetName, GetFullName and GetTopLevelProject functions.

4.5.2.10 The GetProject function

Returns the project of the object. Objects are organised into a directed tree. The root of the tree is the top-level project, the branches of the tree are the projects and the leafs are the simple objects. This function returns the parent node of the object. See the m_pProject field, and the GetTopLevelProject function..

4.5.2.11 The GetTopLevelProject function

Returns the toplevel project. It is useful if a search should be performed in the entire object hierarhy.

4.5.2.12 The M_Dependants list

The dependants of the object. For remote objects the list is empty.

4.5.2.13 The BeginMake function

This function is called before the make action enabling the obejcts to perform initialization tasks.

4.5.2.14 The EndMake function

This function is called after the make actions.

4.5.2.15 The GetHost function

Returns the host assigned to the object. For local object a NULL pointer is returned. See also the m_pHost field.

4.5.2.16 The GetOutput function

Returns the object where the output from the make actions and other results should be redirected.

4.5.2.17 The SetName function

Sets the objectname during initialization. The object name should not change during the make process.

4.5.2.18 The Request function

This function is used to insert the object to another objects m_Clients list. The function may return an immediate answer (either ready or error) in which case the object is not inserted to the other objects m_Clients list, and won't be notified later about the change of the state of the other object. This function requests the state of the object given as the parameter, and if the object is not ready it requests the object to be made. The return value may indicate three results:

- the object is up-to-date (complete)
- an error occurred during the creation of the object (error)
- the object acknowledged the request and will send a notification when ready

See also the Notify, NotifyClienat functions and the m_Clients list.

4.5.2.19 The Notify function

This functin is used to notify clients about the change of the state of the object. If the object reaches a terminal state (COMPLETE or ERROR) it notifies all the other objects that are waiting for it. The notification specifies only that the object is complete or an error occured. See also the NotifyClients function and the m_Clients list.

4.5.2.20 The Transfer function

This function is used to transfer the actual data of an object. This function is rather object specific. In the CMakeObject it is not implemented. It must be overwritten in the inherited classes. An example implementation of this function is available in the CFileObject class.

4.5.2.21 The NotifyClients function

This function calls the Notify function for each object in the m_Clients list. For more information see the Notify function and the m_Clients field.

4.5.2.22 The SetState function

An object must always have a state. States play a very important role in Distributed make. The state of an object may change only in certain ways. For more information about states see the Objact states section. See also the m_State field.

4.5.2.23 The IsPropCompleted functions

These functions are used to determine if the object is up-to-date.

4.5.2.24 The Init function

This function is used to initialise the object. It is called from the constructor, and before the object is unloaded from the memory.

4.5.2.25 The OnChangedDeps function

This function will be called when the dependants are changing.

4.5.2.26 The Execute function

This function executes a shell command. This is the preferred way of executing a shell command rather than using the low level functions such as fork and exec. The command is executed in a separate process. The current process will be blocked until the shell command is completed, but incoming connection will be accepted and served. In a UNIX box the process ID of the child process will be stored in the pid field, and if possible the user ID and the group ID will be set to appropriate values. See the features for more information.

4.5.2.27 The ServerGetValue function

Returns a property value for a communication link. This function makes easier the access of object properties.

4.5.2.28 The m_Name field

The name of the object. Each object within a project must have a unique name. Names are assigned to objects at the time of the creation and never changed during the make process. Names are used to identify objects in the makefile and in the make communication. This field is set by the SetName function and can be queried by the GetName function. The naming of make objects is described in more detail later. See also the GetFullName, GetName and SetName functions.

4.5.2.29 The m_ParamString field

The parameters of the make action are stored in this field.

4.5.2.30 The m_Clients field

The clients of an object are the set of objects which requested the object. The members of this list should be completed to accomplish the goal of the make process, but can be performed only when this object is available. Upon completition of the object these objects will be notified. The CHostObject instances in this list have a special role. A ChostObject instance in the list indicates that the remote part of this object acknowledged a request. This list must be NULL for remote objects. See also the Notify and NotifyClients functions.

4.5.2.31 The m_Depends field

The dependants of the object are stored in this list. The objects in the list are checked to decide if the object is up-to-date. The relations between the objects are represented with these lists.

4.5.2.32 The m_pHost field

In the Distributed Make each object has a host assigned to it. This is considered to be the actual place of the object. For remote objects this pointer points to a CHostObject instance, for local objects this is a NULL pointer (it does NOT point to the local host). This field can be accessed with the GetHost function.

4.5.2.33 The m_pProject field

Each object is a member of a project (except the top-level proect). The projects and objects are organised into an ordered tree. The projects are the branches of the tree, while the objects are the leafs of the tree. The root of the tree is the top-level project. The parent of the make object (the object one level up in the hierarchy) is stored in this field. In the case of the top-level project (which has no parent) this field points to the top-level project itself (it is NOT NULL!). This field should be accessed through the GetProject function.

4.5.2.34 The m_pParams filed

This field contains the parameters for the make object. These parameters are used during the make object creation.

4.5.2.35 The m_State field

This field stores the state of the object. The state of the objects has an important role in distributed make. The scheduling of make actions is mostly controlled by object states. For more information on object states see the MakeMe function, the SetState function and the state section.

4.5.2.36 The pid field

This field exists only on UNIX boxes. If a make action takes place, the process ID of the child process is stored here. This enables the make object to abort a running make action. See the Execute function.

4.5.2.37 The MakeMe function

The MakeMe function implements the probably most important feature of each object. The MakeMe member function of the CMakeObject class has a special role in the make actions, since the logic that controls the make actions is implemented here. These actions are desctibed in the States section.

4.5.3 Error Handling

There can be errors in both of the makefiles and objects. For example a C source code may contain errors. The object should distinguish different type of errors and perform necessary actions. The result of the make action can be one of the following cases:

- First the make action can be successful. This does not requires special treatment, a success return code must be returned to the make utility.
- In the second case a fatal error occurred. This means that the required make action can't be performed or it reported an error. In this case a fatal error must reported and the make process must be aborted.
- The third case is a non fatal error. For example one part of the network is unreachable and a required remote file or action is not available. In this case the a non-fatal error code is returned and it depends on the make options if the make process should be aborted or not. In the third case the make may decide to perform other necessary make actions if possible and try this action later. In the third case a warning message is issued. This gives the user an opportunity to take actions in order to correct the error.

4.5.4 Non-file Make Object Type

We saw that in the traditional make all goals and dependants are files, even if these files never exists (for example in the case of the goal clean). In the Object Oriented Make it is possible to create non-file object. The values of the *non-file* Make Objects are defined in the time of object definition by explicit parameters or defaults. The values and properties of the non-file object can be saved to and restored from the Make status-file. In this way it is possible to determine if a non-file object has changed. The values and the properties of the file type object can also be saved and restored. The Make handles the file and non-file type objects in the same way. The important advantage of the Distributed Make is that since the non-file objects can save or restore their ‘up-to-date status’, while in the traditional Make non-existent files (non-file objects) are never up-to-date (even if they have no dependants).

4.6 Object Instances

4.6.1 Property Object

Property object instances are never used. Property objects are used to inherit the make classes. Property objects are not defined for standalone use but as parts of make objects, because of property class is an abstract class.

4.6.2 Make Object

Make Objects are derived from other Make Objects and Property Objects. The concept is that simple and less specific classes can be extend with properties to define more specific make classes.

```
CFileObject         Object1( “src1.c“ );
CMakeObject         Object2;
```

4.7 Relations

Relations are defined between property classes. Traditionally the only used relation is the newer-than relation. Here we extend the set of relations to define new type of dependencies and conditions. Each property can support one unary and one binary relation. If more relations are needed the property object must be inherited and the new relation can be defined between the new property classes. To determine if an object has a certain property, or to determine if two object can be compared with subject to a certain relation the inheritances of the object are examined, i.e. the object must have a certain ancestor.

The Relation function returns TRUE if the value of first property is ‘up-to-date’ compared to second property value, otherwise it returns FALSE.

```
  BOOL          Relation1(
                    CPropCompleted&  Left,
                    CPropCompleted&  Right );
  BOOL          Relation2(
                    CPropTime&       Left,
                    CPropTime&       Right );
```

4.8 Properties and Make Objects

The object attributes can be defined as member functions in the object, so why do we need these Property classes? Assume that many objects have the same property. If we define properties as member functions, we must define the Property in all of the object types.

Instead we define Make Objects by inheritance. If an object class has a certain property then the class is derived from that property (and possibly from another make class and other properties too). In the case of Make Object both the *IsKindOf* and *IsKindOfProp* functions must be declared and implemented. The declaration of these functions is the same as in the case of properties. However implementation of the *IsKindOfProp* is different. Since the actual class is not a property class (however it is derived from the *CProperty* too) the *IsKindOfProp* function implemented in a different way than in the case of property objects (which are derived from only another property objects). In this case the *IsKindOfProp* function must be implemented that it should call the *IsKindOfProp* functions of all the ancestors. The *IsKindOfProp* function of non-property object types must be called too since the object has all the properties that its ancestors have. The *IsKindOf* function must be implemented in the same way as described at the introduction of the *CCore* object.

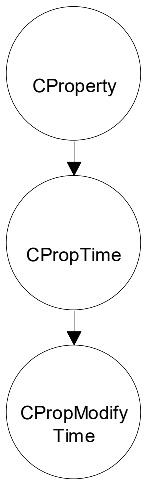
The macros DECLARE_OBJ and IMPLEMENT_OBJ are provided to make the declaration and implementation of these functions easier. See the examples for more information on usage of these macros.

```
/////////////////////////////////////////////////////////
// CMakeObject
class CMakeObject : public CPropCompleted
{
   DECLARE(CMakeObject);
public:
   virtual CProperty * IsKindOfProp(
                   const char * pClassName,
                   RELATION_FUNCT* * pRelation = NULL ) =0;
   ...
};
IMPLEMENT(CMakeObject, CPropCompleted);
```

4.8.1 Property inheritance rules

Here we describe some rules of Property inheritance. If a Property inherited twice the function contained by the property must be overwritten. Otherwise reference to these functions will be ambiguous, since it is not defined in the C++ specification which instances of these functions used.

To solve this problem the successors 'hide' some of the virtual functions of they ancestors by redefining them as private. The example shows how the successor hides the ancestors virtual function. The successor is the *CPropModifyTime* class, the ancestor is the *CPropTime* class and the 'hidden' virtual function is the *GetTime* function.



```
///////////////////////////////////////////////////////////
// CPropTime
class CPropTime : public CProperty
{
  DECLARE_PROP(CPropTime);
public:
  virtual CGMTime  GetTime( void ) = 0;
};
IMPLEMENT_PROP(CPropTime, CProperty);


///////////////////////////////////////////////////////////
// CPropModifyTime
class CPropModifyTime : public CPropTime
{
  DECLARE_PROP(CPropModifyTime);
public:
  virtual CGMTime  GetModifyTime( void ) = 0;
private:
  virtual CGMTime  GetTime( void ) = 0;
};
IMPLEMENT_PROP(CPropModifyTime, CPropTime);
```

4.8.2 Make Object inheritance

Make Object types are derived from the core Make Object Type. More specific make classes derived from less specific classes. For example a C-source-file object type (*CCSourceFile*) derived from a more general source-file object type (*CSourceFile*).

```
/////////////////////////////////////////////////////////
// CFileObject
class CFileObject : public CMakeObject, public
CPropModifyTime
{
  DECLARE_OBJ(CFileObject);
public:
  // overwrite all property functions of base properties
      (Getxxx)
  virtual CGMTime  GetTime( void );
  virtual CGMTime  GetModifyTime( void );

  // overwrite CMakeObject functions
  virtual void *   Construct(
                       const char * ParamLine );
  virtual BOOL     MakeMe( void );
  virtual const char * GetDefaultProp( void );
  ...

protected:
  CFilePath        m_FilePath;
  ...
};
IMPLEMENT_OBJ2(CFileObject, CMakeObject, CPropModifyTime);
```

4.8.3 Default Property Type

Each Make Object has a default property which is used if none of its properties defined in the makefile. This property are *default property*. For file-type objects it is the last-modification-time property.

The default property defined by *GetDefaultProp* member function of *CMakeObject*:

```
/////////////////////////////////////////////////////////
// CMakeObject
class CMakeObject : public CPropCompleted
{
  DECLARE_OBJ(CMakeObject);
public:
  ...
  virtual const char * GetDefaultProp( void );
  ...
};
IMPLEMENT_OBJ(CMakeObject, CPropCompleted);
```

4.9 Properties and Relations

As described above the connection between relations and properties are very strong. The set of relation can be extended by extending the set of properties. So relations are defined as property class member functions. A relation can be defined as unary relation or binary relation between identical property types.

Since the relations are implemented as static functions, so the *this* pointer is not available in these functions, the unary relations have one parameter while the binary relations have two parameters. This is because the *Relation* functions have the fixed type and so can be automatically declared (but not implemented) by the DECLARE_PROP macro.

```
////////////////////////////////////////////////////////
// CPropCompleted
class CPropCompleted : public CProperty
{
  DECLARE_PROP(CPropCompleted);
public:
  virtual BOOL     IsCompleted( void ) = 0;

  // autimatic declaration in DECLARE_PROP macro
  static BOOL      Relation(
                       CProperty *  Left,
                       CProperty *  Right );
};
IMPLEMENT_PROP(CPropCompleted, CProperty);


BOOL CPropCompleted::Relation(
                       CProperty *  Left,
                       CProperty *  Right )
{
  return ((CPropCompleted *) Left)->IsCompleted();
}


////////////////////////////////////////////////////////
// CPropTime
class CPropTime : public CProperty
{
  DECLARE_PROP(CPropTime);
public:
  virtual CGMTime  GetTime( void ) = 0;
  ...

};
IMPLEMENT_PROP(CPropTime, CProperty);


BOOL CPropTime::Relation(
                       CProperty *  Left,
                       CProperty *  Right )
{
  return ((CPropTime *) Left)->GetTime() >=
                       ((CPropTime *) Right)->GetTime();
}
```

5. The Object Oriented Makefile

The Object Oriented Makefile consists of object declaration and dependency relations. Object declarations contain the object type the object name and optional parameters. The object type must be defined in the Make utility. The object name is a unique identifier, which used later to reference the object specified by the parameters. When a non existent object is referenced the make utility tries to apply the automatic object declaration. This means that based on the information in the object name the make utility creates an object declaration. For example the type of an object called *MYPROG.C* defaults to a C-source-file (*CCSourceFile*) object type, the parameters defaults to *MYPROG.C* (the name of the file). Based on the default information it is possible to declare the object. Of course, exact declaration of the objects are preferred.

The dependency relations are represented in a traditional like form. Each dependency consists of a goal (or activity) and a set of properties of object instances.

5.1 Concepts

We treat all options, variables. goals, and dependants as objects. The performed make actions depend on the type of the goal object. The role of the Make utility is to parse the makefile and call the functions of the goal objects. The make also provides information to the objects using default objects such as current *hardware* or *environment*. The objects do not necessarily represent files. It is necessary to maintain another file to track the object states of non-file or file objects. For example we should save the compiler options used to produce a goal to decide next time if the options are changed. The object properties are represented also as objects. We bind relations to object properties so relations are object oriented too. Both object types and relations (as property object types) are extensible with the definition of new make classes. The taken action is not explicitly defined in the makefile, but basically coded into the make classes.

We mostly use the same concept that we defined in the sections Traditional Make Utility and Distributed Make with some exception. The main difference is that this utility is capable of handle other object types than files, so we replaced some file related terms with not file-related terms. For example if nothing has to be done to achieve a certain goal (it is called **up-to-date** in the traditional make) we call it **completed**. This expresses that a goal other than a (creation of a) file, for example a test is completed (it does not make too much sense to say a 'test is up-to-date').

Make actions are implicitly defined by object types and they have hidden implementation.

Example:

```
...
CExeFile          myprog( "myprog" );
CAction           test( "test_my_program" );
CInstall          install( "/usr/local/bin", myprog );
...
myprog: myobj1, myobj2
test: myprog
install: test
...
```

In this case the goal *myprog* depends on the objects *myobj1* and *myobj2* (possibly object files), which are not discussed here. The goal (activity) test depends on *myprog*, since the program should be made before tested, and the activity *install* depends on the test. If the test is not successful the program will not be installed (so the last working version will be in the directory */usr/local/bin*). You may observe that no make actions specified on the example. This is general since make actions are implicitly defined by object types, and custom make action can be specified as *CAction* make objects. The make utility will keep track of the activities completed so if a program is not modified and the last test was successfully completed after the building of the program the test will not be executed again, and this applies also to the activity install in particularly in this example and to any objects in generally.

Object Oriented Makefile for Example 1:

```
CHeaderFile       MainHdr( "MAIN.HPP" );
CHeaderFile       DrawingHdr( "DRAWING.HPP" );
CHeaderFile       ConvertHdr( "CONVERT.HPP" );
CHeaderFile       DrawRCHdr( "DRAW.RCH" );
CSourceFile       MainSrc( "MAIN.CPP" );
CSourceFile       DrawingSrc( "DRAWING.CPP" );
CSourceFile       ConvertSrc( "CONVERT.CPP" );
CRCSourceFile     DrawRCSrc( "DRAW.RC" );
CObjectFile       MainObj( "MAIN.OBJ" );
CObjectFile       DrawingObj( "DRAWING.OBJ" );
CObjectFile       DrawingObj( "CONVERT.OBJ" );
CResourceFile     DrawRes( "DRAW.RES" );
CExeFile          DrawExe( "DRAW.EXE" );

DrawRCSrc : DrawRCHdr
DrawingSrc : DrawingHdr
ConvertSrc : ConvertHdr
MainSrc : MainHdr, DrawingHdr, ConvertHdr

DrawRes : DrawRCSrc
DrawingObj : DrawingSrc
ConvertObj : ConvertSrc
MainObj : MainSrc

DrawExe : DrawRes, MainObj, DrawingObj, ConvertObj
```

5.2 Makefile syntax

Besides of the object declarations and dependencies the makefile can contain empty lines and comments for user convenience. The object declarations and the dependencies can be mixed but the preferred order is the declarations first and the dependencies later. An object can not be multiple defined (i.e. only one object may exist in one name). If an undefined object is referenced in a dependency it is automatically defined so it can not be defined later.

The makefile syntax is roughly defined by these grammar rules:

```
makelines →         declaration makelines |
                    dependency makelines |
                    EMPTY_LINE makelines |
                    comment makelines |
                    ; /* nothing */
declaration →       TYPE identifier '(' STRING ')' ';' |
                    TYPE identifier ';' ;
dependency →        identifier ':' list ';' ;
list →              identifier list |
                    ; /* nothing */
comment →           '#' comm_chars '\n' ;
```

The semantics are defined by adding attributes to the grammar.

5.3 Defaults

Defaults are assigned to object names and object types. Instead of giving certain information the user can rely on the defaults. The object type and object parameter default values depend on the object name. The default object property depends on the object type.

If an object is not declared in declaration section of the makefile and later referred in a rule, then the object will be implicitly declared (if possible).

The defaults that depends on the object type must be implemented in the object classes as member functions (for example *GetDefaultProp*, which should be implemented in the Make Object). The defaults that depends on the object name partially must be implemented in the makefile parser. For example the default object type can not be implemented in the objects classes since no object class available when the default needed (the default itself will be the object class). For example the object *HELLO.C* probably has the class of *CCSourceFile*.

5.4 Makefile parser

The Object Oriented Make parses the makefile, and creates the objects defined in it. The dependencies are also processed by the make, but evaluated by objects. The parsing of a makefile is not a trivial task. For example in the case of an object declaration the parser has to interpret the parameters to find out their types and find a proper object creator function. If the make utility is extended with new classes, the parser has to be modified also to interpret the new object types and their parameters.

The Object Oriented Make utility maintains an object list. This list is empty in the beginning, then the make utility inserts predefined Make Objects (currently no predefined objects exist). After the list was initialised with the predefined objects the make utility starts to parse the makefile. The makefile probably contains more object declarations. The declared objects will be created and inserted to the list. The dependency relation will be attached to the objects in the list. The dependency relations may contain implicit object declarations which will be inserted to the list also.

At object creation time the previously saved object parameters must be restored. The object parameters may change in which case the object will be obsolete.

After the list is built the Make utility looks for the goal object and calls its up-to-date function which returns the logical value if the object is up-to-date. If the object is up-to-date the make process terminates, otherwise the goal objects make-me method will be called.

The Object Oriented Make must provide some service functions to the objects. The main role of these files is to store object data, such as creation or modification time or other properties. These functions are:

```
FILE *     OpenStatusFile(
                   const char *     FileName );


void       CloseStatusFile(
                   FILE *           hFile );


BOOL       SearchObjectSection(
                   const char *     ObjectName,
                   FILE *           hFile );


int        ReadStatusFile(
                   const char *     ObjectName,
                   const char *     Property,
                   char *           Buffer,
                   int              BufferLen,
                   FILE *           hFile );


const char *ReadStatusFile(
                   const char *     ObjectName,
                   const char *     Property,
                   FILE *           hFile );


const char *ReadStatusFile(
                   const char *     ObjectName,
                   const char *     Property,
                   const char *     StatusFile );


BOOL       WriteStatusFile(
                   const char *     ObjectName,
                   const char *     Property,
                   const char *     Buffer,
                   FILE *           hFile );


BOOL       WriteStatusFile(
                   const char *     ObjectName,
                   const char *     Property,
                   const char *     Buffer,
                   const char *     StatusFile );
```

5.4.1 Example

We examine Example 2 in a network containing only UNIX workstations, since other operating systems such as DOS does not provide support for remote actions. The backup process can be defined as a traditional make goal.

For example this makefile can create a backup of the entire root filesystem.

```
backup:
  tar -cv1 /
```

However using remote tape devices makes the makefile different and more complicated. For example the use of a remote tape device yields:

```
backup:
  tar -cvf - / >rsh other.host cat >/dev/mt1
```

while saving a remote filesystem yields:

```
backup:
  rsh other.host tar -cvf - / > /dev/mt1
```

The disadvantages of these solutions is that no checks performed before the make actions take place (the goal does not depends on anything).

Object Oriented Make example:

```
CTapeDev           TapeDev("/dev/mt1");
CTape              Tape("vol13");
CFileSystem        Fsystem("/");
CSave              Backup(Tape,FSystem);

Backup: TapeDev.mounted Tape Fsystem
```

There are four objects.

1. *TapeDev* is the backup device object in which the tape cartridge will be mounted.
2. *Tape* is the tape cartridge (identified by its volume label).
3. *Fsystem* is the filesystem where the backup will be made from.
4. *Backup* is the activity backup.

The *Backup* activity has two parameters, the tape cartridge and the filesystem objects. The backup action (which is coded in the *CSave* make class) will take place if the tape cartridge is mounted in the tape device and the backup activity is older either the data on the tape or the filesystem (the age of the backup activity is the time since it was performed last time).

If the backup activity has never taken place before it will be older than the tape and the filesystem. If the backup is performed successfully the Backup object will be saved, thus next time it can be checked against the last modification time of the data in the tape or in the filesystem.

Here a special relation 'mounted' is used to establish if the tape is on the tape device. In the case of the *Tape* and *FSystem* objects the default property (last modification time) is used.

5.5 YACC

We intend to use the YACC utility as the makefile parser. The YACC utility is a parser generator, which works with LALR languages. So the syntax of the makefile will be clear, and the semantics will be expressed with an attribute grammar. The extension of the Object Oriented Make means the extension of makefile language. This is done by writing new make classes, and defining grammar rules that can create these objects.

5.6 Object Oriented Makefile parsing

The parsing of the Object Oriented Makefile is done by a program generated by the YACC utility from an attribute grammar, and a lexical analyser which reads makefile and makes tokens out of it for the attribute grammar parser. The makefile parsing will take place in one pass, that is the makefile will be read once and parsed immediately, so parsing of a special file (pipe) will be possible.

6. Distributed Object Oriented Make

The Object Oriented Make uses a special makefile to describe the activities (the activities are similar to goals in the traditional make). Further extension is the support for heterogeneous distributed (computing) environment. This means that a set of different computers are connected to a network, and the distributed object oriented make development system can carry out actions on different hosts to complete an activity. Generally the Distributed Object Oriented Make can be used not only in the development of software projects, and other activities that creates files from other files, but any kind of computer applications which can be described with this concept.

You may observe that the distributed extension of traditional MAKE utility is *only one application* of Distributed Object Oriented Make Development System.

Another example can be a company which has several departments all over the country. In this case wide area network can be used to connect the different departments. An activity can be the creation of a report of the activities of the department. In this case the report depends on the reports of the department, which can be generated by remote actions. Most of the department report are remote dependants. In this example special viewpoint can be examined, because of the use of a wide area network. For example the company may stretch over time-zone boundaries or the network can be disconnected for a certain period of time (we can assume the use of telephone lines in the network, since connecting the departments all the time can be quite expensive). In this example the parallel make actions can be represented, because it is practical to generate the reports of the departments parallel.

From the above example you can see that the user is encouraged to derive own version of the standard make classes to handle special situations or to implement perfectly new make classes with relations the designer of make did not even think of.

Here we describe the basic concepts of the Distributed Object Oriented Make. One of goals is to develop a Make utility, which is capable of maintaining a distributed program development project. The utility should be capable of making all files of the project on all machines up-to-date. It will be object-oriented and easy-to-use. The following part of the document contains ideas about of the distributed make. The consistency of these ideas is object of further examinations, so not necessarily all of them will be included in the final specification.

6.1 Necessary information to decide what is remote

Since the same makefile can be used on several hosts, a file can be remote or local depending on where the make was executed. In the case of those objects which are either goals or dependants it is necessary to establish whether they are remote or local. To do this information must be provided on which host the makefile is running. This information (and other information which can be determined by low-level, machine dependant functions) must be provided by the make utility. This can be done in the core make object (the core object must contain machine dependant code), so the make should provide predefined object instances before it starts to parse the makefile.

6.2 The Client-Server modell

To implement the Distributed Object Oriented Make we used the client-server model for the application. This communication mode is working in the following way:

- First the client initiates a connection to the server.
- The server accepts the connection.
- The client issues a request.
- The server sends an answer to the request and closes the connection.
- The client receives the answer from the server and closes the connection.

The dmake utility consists of servers and starters. Both servers and starters play the role of both a server and a client. The server functions has priority over the client functions to avoid the system hang while waiting for service. The servers must be running prior to the starting of the starter applications. Normally servers are running continously (like other daemons), but performing no actions (not eating up the CPU time). The role of a starter application to initiate a connection to a server and request some actions from it. The server will probably issue other requests to other servers and the system will start to work. The starter applications play another quite important role, as they collect the output of the make actions performed by the make servers. In this case the starter applications play the role of a server.

6.3 The network distribution concept

The Distributed Object Oriented Make works in the following way. The system consists of make servers which are running in the hosts of the network. One make process is running in each host. The entire project update is done by these make processes, which can communicate with each other. Updating an object can be requested by connecting to a make process. The protocol used for the communication is described later. The make servers are playing the role of both servers and clients, but the server functions have a priority over the client function (i.e. the make server will serve incoming connections before initiating outgoing connection). This reduces the response time of the servers and makes the system load (resource consumption) lower.

6.4 How does the make process work?

After booting the computers the Distributed Object Oriented Make servers should be started. Each server will parse its makefile, which describes the environment for the server (objects on the hosts, relations to other hosts, etc.). If the servers are running the user may initiate a connection to one of the servers to request the recreation of an object. After that the Distributed Object Oriented Make decides which objects must be created to build the specified object. Than the creation of these objects (including the specified object) takes place. In a time only one object will be made in each host, but different hosts will work in a parallel way. If an error occurs during the creation of the objects the make process will be shut down and the error will be reported. However,because different actions take place in a parallel way, some actions may take place after an error is spotted. This is because the termination of these actions is either not possible or may result in memory loss in the Windows operating system.

6.5 Accessing remote objects

Remote objects are accessed through their local counterparts. Each distributed object has multiple functionality. First, an object must perform the tasks of a normal (not distributed) make object, such as performing make actions. Besides these tasks the distributed make object must work as a server and as a client. If it is defined as remote object it must propagate all requests to its remote counterpart. If it is defined as local object it must serve the queries issued by its remote parts. This way we can provide a transparency of remote objects. (i.e. the remote objects can be treated almost in the same way as the local objects.)

6.6 Make Communication Protocol

We implemented a communication protocol for transferring object properties, data and information, and issue requests for actions. This protocol is based on TCP/IP connections. There are two types of communications.

- External messages
- Internal messages

External messages are used by starter applications to communicate with make servers. External messages have the form of :

!1.2.3.4,1234 -options#PROJECT#object action

,where 1.2.3.4 is the IP address of the starter application in the form of a hostname (with an optional domain name) or a dotted quad. The 1234 is the port number where the starter application accepts output messages. The -output represents one or more make options (which will take effect during the action). The '#PROJECT#object' part is an object reference. The action is the requested action (usually the MakeMe action). This message is sent to the make server where the object '#PROJECT#object' resides. The server creates the object if not exists (loads a project file if necessary), the object communicate function is invoked to perform the desired action. The object responds with a message that indicates that the request is acknowledged or an error occurred. With this type of messages a make session is created. Each session has the following attributes:

- The starter's IP address
- The starter's port
- The project options
- The session identifier

The first three are described above, the session identifier is assigned to a session by the make server and will be used in all further communication actions involved in the desired make action.

Internal messages are used to communicate between make servers. Internal messages have the following form:

@12345678#PROJECT#object action

The 12345678 number is the make identifier described above. Make servers access the project options and the starter applications communication parameters through this identifier.

The entire communication is designed in a way that servers only propagate connection to objects. As we described earlier communications occur between the local and remote part of the same object, so the extension of the protocol is easy to implement. Basically individual objects may implement an arbitrary protocol.

6.7 Make Object Types for Distributed Processing

Several classes and structures provided to utilities the communication. These are:

- The CHostObject class
- The MPC_Command class
- The _HCOMM class

The CHostObject class represents a host in the network. Host objects are used to establish a connection and validate incoming connection. This means that the communication peer is provided as a Host object in most of the cases, and a Host object is assigned to each incoming connection (the incoming connections are rejected if no corresponding host found). This way with host objects we establish a host-level access control (user level access control is not common in Windows environments). The Host class provides the following interface to objects:

```
class CHostObject : public CMakeObject
{
  DECLARE_OBJ(CHostObject);
public:
  CHostObject( void );
  ~CHostObject();


// override Make Object operators


  virtual BOOL     IsCompleted( void );


  virtual BOOL     IsExist( void );


virtual BOOL       MakeMe( void );


  static const char *  ConvertPath( char * FilePath );


  const char *     HostPath( void );


  KWSocket *       Connect();


static
int
  Reply(
          HCOMM   hComm,
          const char * Message,
          int     MessageLength = -1 );


  int     Send(
                  const char *     pSendData,
                  int              SendLen,
                  KWSocket *       pSocket = NULL,
                  BOOL             EnableIdle = TRUE );
  int      Send(
                   char *           pData,
                   int              Len,
                   const char *     pSendData,
                   int              SendLen = -1 );

  KWStreamSocketClient * Send(
                   char *           pSendData,
                   int              SendLen = -1,
                   BOOL             EnableIdle = TRUE );

protected:

  char     m_HostPath[32];

  char     m_IPAddress[32];

  char *   m_ServerName;

friend CProject;

};
```

The MPC_Command class provides an interface to simple internal make messages. The class sends a message and waits for the response. This is the highest level interface to the make messages. However this interface can handle the most basic messages. More complicated protocol extensions may be implemented in make objects.

```
struct MPC_response {
  int              ReturnCode;
  int              Length;
  char *           Response;
};

class MPC_command
{
public:

  MPC_command();

  MPC_command(CHostObject *hostobj, CMakeObject *obj, char
*commandstr, const char * pResult = NULL );

  ~MPC_command();

  void             SetHost(CHostObject *hostobj);

  void             SetObject(char *objrefstr);

  void             SetObject(CMakeObject *obj);
  void              SetCommand(char *comm);

  void              SetCommand(int size, char *comm);

  BOOL              Send( void );

  const char *      Result( void );

private:

  CHostObject *     host;
  CMakeObject *     m_pObject;
  int               length;
  char *            command;
  const char *      m_pResult;
  MPC_response      response;

};
```

The _HCOMM class is a lower level interface of make connections. Both incoming and outgoing connections can be represented with the _HCOMM class. The _HCOMM class is used in the request functions of the make classes. The _HCOMM class contains all the parameters of the connection. The _HCOMM class is used to pass connections to objects. The file transfer is also implemented with this class.

```
class _HCOMM
{
public:

  _HCOMM(CMakeObject *object, char *message);

  _HCOMM(KWSocket *socket, CMakeObject *object = NULL);

  ~_HCOMM();

  int Connect();

  int Send(const char *Buffer, int Bufflen = -1);

  int Receive(char *Buffer, int Bufflen);

  char *Gets();

  CMakeObject *     pObject;
  KWSocket *        pSocket;
  CHostObject *     pHost;
  const char *      Message;
  long              MakeID;
private:

  char *            buffer;
  int               bufsiz;
  int               datalen;

};
```

7. How to extend the Distributed Object Oriented Make?

1. Designing of new property and make classes: Designing the new property and make classes are very important. We have to decide, what kind of objects we want to introduce. After deciding what type of objects needed and what kind of properties may it has, we must specify the most important aspects of the new make (or property class). We must answer the following questions. What dependencies the new object may have? What make action the new object class must perform? How it relates to other objects and properties?
2. Implementation of the new object or property class: After we answered the questions we can implement the new class in C++. The method how the new make or property classes must be implemented is described in the next section.
3. Specifying the new class for the makefile parser: After the implementation of a new class we must inform the makefile parser what is the name of the new class and how to create it. This can be done by modifying the makefile grammar.
4. Rebuild the make program: Compile the source code of the new make or property class. Rebuild the make utility.
5. Test the new make utility: After the make is successfully rebuilt it must be tested, with some test projects. The test projects must cover as many of the possible cases as possible. For example there must be errors both in some of the test projects, and makefiles. An important test can be it the make utility is able to make itself again.
6. Install the make utility: Replace the old make executable with the new one. The last three steps can be done by the make itself.

7.1.1 Example

In the Example 2 we can use a quite similar makefile we used in the Object Oriented section, except objects are extended with locations. So the makefile looks like this:

```
CHost                RemoteHost1("remote1.host");
CHost                RemoteHost2("remote2.host");
CTapeDev             TapeDev("RemoteHost1:/dev/mt1");
CTape                Tape("vol13");
CFileSystem          FSystem("RemoteHost2:/");
CSave                Backup(Tape,FSystem);

Backup: TapeDev.mounted Tape FSystem
```

In that case the backup device is on the host named "remote1.host" (the *RemoteHost1* object) and the filesystem of the host "remote2.host" will be saved. The make action will be performed locally. We can see that making some objects remote is quite simple in the makefile. The objects can access remote files and actions through remote host objects.

8. Projects

Projects are used to specify object sets. A project is a set of objects. Each object in a project depends on objects in the same project. Projects are also represented as objects. A project may contain sub-projects (i.e. project type objects). As objects projects have a unique name. Names are unique in a project, but different projects may contain objects with the same name. To avoid confusion objects are referred as objects of a specified project.

8.1 Make processes

A make process is running on each machine. These make processes govern the communication between objects in different hosts. The make process is working by the definitions in the top-level makefile. In the top-level makefile we can define projects. The dmake program can only create objects that defined in the makefile. The top-level makefile works as a configuration file for the make processes. Even host level access control is implemented with top-level makefiles.

8.2 Object states

From the point of view of the make process the objects may be in several states. These states control the make process. The states are implemented in the CMakeObject class. For each object the following states exist:

- INITIAL - the object is not checked to be complete
- NOTCOMPLETE - the object is not complete, but no action has been performed to make it complete
- BLOCKED - the make action corresponding to the object should not be started yet
- READY - the make action should take place
- MAKING - the make action is in progress
- COMPLETE - the object is complete
- ERROR - an error occurred making the object

The object states have a different meaning for remote, local and project objects. The concept is to treat remote objects as local object with one dependant of its remote part. The make action corresponding to a remote object is to transfer its actual data. So the meaning of the states for remote objects are the following:

- INITIAL - same as in the case of local objects
- NOTCOMPLETE - the remote part is not compete
- BLOCKED - a request has been sent to the remote part to make itself, but the remote part is not yet complete
- READY - the remote part is complete, we are ready for the data transfer
- MAKING - the data transfer is in progress
- COMPLETE - the data has been successfully transferred to the local host
- ERROR - an error occurred during the process

Because the states have a special meaning remote objects must be treated in a slightly different way than local objects. It is because in a make process it is possible (and likely) that transferring the data of the object is not necessary, but only some checks should be performed on the object. To make it clear let's see an example. Assume that we have a local object file and a remote source file and there is no local copy of the source file available. If the object file exists and newer than the source file (in the remote machine), than no action should take place. But in that case the source file won't be in the state COMPLETE but in the state READY. So remote objects are not necessarily made complete. The object states are implemented in a way that only certain changes should be made to the states. These restrictions are applied to make sure that the make process will terminate. Fortunately the object class implementors have little or nothing to do with the object states.

9. Customising the Object Oriented Make Components

The Distributed Object Oriented Make is extensible. There is a way to introduce new make classes and properties (relations). The new objects must be implemented in C++. In the following sections we describe how to write new make and property classes for the Distributed Object Oriented Make. Besides writing the C++ code the class implementor must define the name of the new class, and the construction of an object instance of the new class. These definitions are necessary for a makefile parser to detect and create new objects.

9.1 Inheritance, virtual functions

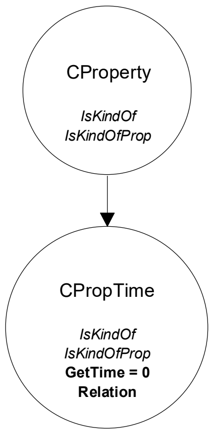
In this chapter we describe how to use inheritance and virtual functions to define new make and property object classes. We use the definitions and implementations of the property classes *CPropTime* and *CPropModifyTime* and the make object class *CFileObject*.

9.2 Adding a new property

In this section we use the property class *CPropTime* as an example. The *CPropTime* class is inherited from the *CProperty* class. The *CPropTime* class has no more ancestors, so the DECLARE_PROP macro is used to declare the *IsKindOf*, *IsKindOfProp* and *Relation* functions (which are automatic defined in macro). This macro has one parameter, the name of the object class (*CPropTime*). Then a public virtual function is declared (*GetTime*). This function is unimplemented in the *CPropTime* class (we assigned a zero to it). This virtual function enables us to inherit the *CPropTime* property by different make classes. In these classes the *GetTime* function must be implemented. This is followed by the object specific data and function fields. In the example one function (*SetTime*) and one data field *m_Time* is declared. After the class declaration, the implementation part follows. The *IsKindOf* and *IsKindOfProp* functions are implemented by using the IMPLEMENT_PROP macro. This macro has two parameters, the property class name and its ancestors class name (*CPropTime* and *CProperty*). After that the function *Relation* is implemented. This function is declared by the DECLARE_PROP macro, but can not be implemented by the IMPLEMENT_PROP macro, since its implementation may be different in each class. In the last part the function field (*SetTime*) is implemented.

```
/////////////////////////////////////////////////////////
// CPropTime
class CPropTime : public CProperty
{
  DECLARE_PROP(CPropTime);
public:
  virtual CGMTime  GetTime( void ) = 0;
};
IMPLEMENT_PROP(CPropTime, Cproperty);
```

```
BOOL CPropTime::Relation(
                      CProperty *  Left,
                      CProperty *  Right )
{
  return ((CPropTime *) Left)->GetTime() >=
                      ((CPropTime *) Right)->GetTime();
}
```



9.2.1 Adding a relation

A new relation can be added by defining a new property, and implementing its Relation function according to the needs. Each property can define only one relation. For defining more relations new property classes must be implemented.

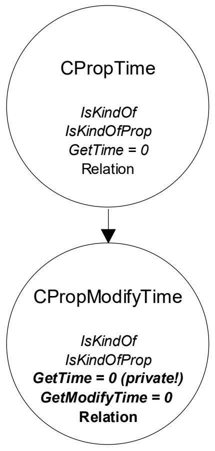
```
BOOL CPropTime::Relation(
                      CProperty *  Left,
                      CProperty *  Right )
{
  return ((CPropTime *) Left)->GetTime() >=
                      ((CPropTime *) Right)->GetTime();
}
```

9.2.2 Steps to define a new Property

1. new Property class is inherited from the *CProperty* class
2. *DECLARE_PROP* macro is used to declare standard part of the new Property class
3. declare additional *Get*xxx virtual function to enable accessing property value
4. *IMPLEMENT_PROP* macro is used to implement standard part of the new Property class
5. implement `BOOL Relation( CProperty *, CProperty * )` member function

9.3 Customising existing property

The property classes are either inherited from the *CProprty* class or from another property class. We illustrate the latter with the help of the class *CPropModifyTime*. The *CPropModifyTime* class is inherited from the *CPropTime* class. The *CPropModifyTime* class declared the same way as the *CPropTime* class were declared except that the inherited *GetTime* function is redeclared as private. The *IsKindOf* and *IskindOfProp* functions are declared and implemented by using macros as in the declaration of the *CPropTime* class. The *CPropModifyTime* class has an unimplemented virtual function too (the *GetModifyTime* function), and it makes private the *GetTime* inherited virtual function. The *Relation* member function is declared by the DECLARE_PROP macro. The *CPropModifyTime* class has no other data or function fields.



```
//////////////////////////////////////////////////////////
// CPropModifyTime
class CPropModifyTime : public CPropTime
{
  DECLARE_PROP(CPropModifyTime);
public:
  virtual CGMTime  GetModifyTime( void ) = 0;
private:
  virtual CGMTime  GetTime( void ) = 0;
};
IMPLEMENT_PROP(CPropModifyTime, CPropTime);

BOOL CPropTime::Relation(
                      CProperty *  Left,
                      CProperty *  Right )
{
  return CPropTime::Relation( Left, Right );
}
```

9.3.1 Steps to costumising an existing Property

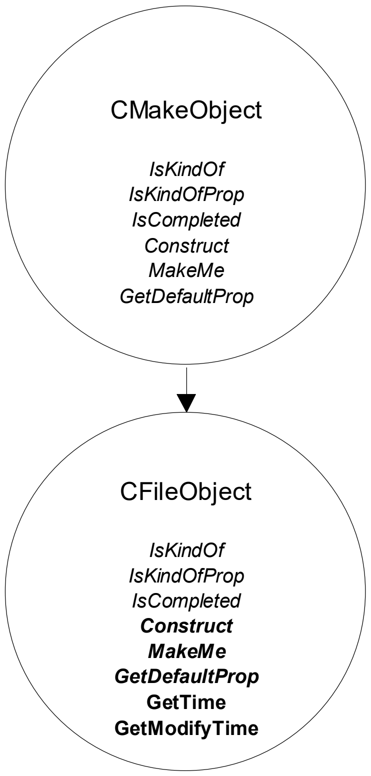
1. new Property class is inherited from the predefined class

2. *DECLARE_PROP* macro is used to declare standard part of the new Property class
3. hide all property value functions of all ancestor Property class redeclaring from public to private access
4. declare additional *Get*xxx virtual function to enable accessing property value
5. *IMPLEMENT_PROP* macro is used to implement standard part of the new Property class
6. implement `BOOL Relation( CProperty *, CProperty * )` member function

9.4 Adding a new make class

Make classes are inherited from a descendant of the *CMakeObject* class and property classes. The most simple make object classes are inherited from the *CMakeObject* class. Here we describe the creation of a new make object class by using the *CFileObject* class as an example. The *CFileObject* class is derived from the *CMakeObject* class and the property classes *CPropCreateTime* and *CPropModifyTime*. In the declaration of the class we can see the usage of the DECLARE_OBJ macro which declares the *IsKindOf* and *IsKindOfProp* functions. Then the virtual functions are declared which should be implemented to make this class a non-abstract class (the functions which are unimplemented in the ancestor classes). In the implementation part we can see that the functions *IsKindOf* and *IsKindOfProp* are not implemented by the macro IMPLEMENT_OBJ, but implemented individually. The *IsKindOf* functions checks if the object class name (*CFileObject*) equals to the parameter of the *IsKindOf* functions and if it is then returns a pointer to the actual object. It the match is failed then it calls the *IsKindOf* functions of the ancestors. The *IsKindOfProp* function does the same thing except the actual object class name is not checked and only the property ancestors are called. If one of the ancestors recognised the object class name then the function returns it immediately.

After these functions you can extend the object functionality using other member functions. The *CFileObject* class is not an abstract class since all virtual functions that are unimplemented in the ancestor classes are implemented.



```
/////////////////////////////////////////////////////////////
// CFileObject
class CFileObject : public CMakeObject,
                        public CPropModifyTime
{
  DECLARE_OBJ(CFileObject);
public:
  // overwrite all property functions of base properties
      (Getxxx)
  virtual CGMTime  GetTime( void );
  virtual CGMTime  GetModifyTime( void );

  // overwrite CMakeObject functions
  virtual BOOL     MakeMe( void );
  virtual const char * GetDefaultProp( void );
  ...

protected:
  CFilePath        m_FilePath;
  ...
};
IMPLEMENT_OBJ2(CFileObject, CMakeObject, CPropModifyTime);


CMakeObject * CFileObject::Construct(
                        const char * ParamLine )
{
  CFileObject *    pObject = new CFileObject( ParamLine );
  pObject->m_FilePath = ParamLine;
  return pObject;
}
```

```
const char * CFileObject::GetDefaultProp( void )
{
  return PROP(CModifyTime);
}

CGMTime CFileObject::GetTime( void )
{
  return GetModifyTime();
}

CGMTime CFileObject::GetModifyTime( void )
{
  return <file time stamp>;
}
```

9.4.1 Steps to define a new Make Object

1. new Property class is inherited from the *CMakeObject* and other Property class
2. *DECLARE_OBJ* macro is used to declare standard part of the new Make Object class
3. redeclare all *Get*xxx virtual value functions of ancestor Properties to public
4. redeclare *MakeMe* virtual function if necessary
5. declare member variables
6. *IMPLEMENT_OBJ* (or *IMPLEMENT_OBJ2*, *IMPLEMENT_OBJ3* etc. on multiple inheritance) macro is used to implement standard part of the new Make Object class
7. implement all *Get*xxx functions
8. implement *Construct* function for dynamic construction
9. implement *GetDefaultProp* function
10. implement *MakeMe* function if necessary (based on *CMakeObject MakeMe*)

9.5 Customising existing make classes

Make classes can be inherited from other make classes. This way existing make classes can be extended with new properties and relations. Inheritance from other make classes can be done the same way as in the previous example where a new make class were derived from the *CMakeObject* class.

10. Other features

10.1 Options

The object oriented make utility may take command line options which effect the way that the objects are working. Some options may effect the work of the makefile parser while other options may effect the work of the objects. These options can be implemented as a predefined make object. For example make options can describe defaults or may effect the work of certain object defined in the makefile.

10.2 Automatic Makefile skeleton building

The Object Oriented Makefile describes the structure of the project. For the automatic generation of the makefile, all the aspects of the project building must be considered, so it is impossible for complex projects. However the makefile skeleton building is possible based on some obvious information. For example dependencies may be generated automatically in some cases or automatic object definitions can be made explicit by adding the proper declarations to the makefile.

10.3 Makefile conversion utilities

Another important utility can be a makefile converter, which convert traditional makefiles or other information to object oriented makefiles. This kind of utility will allow the make users to switch to an object oriented make without learning the entire makefile language at once.

10.4 Problems and solutions

In this section we describe some problems which arose implementing a Distributed Object Oriented Make program. We think these problems are the basic problems each Distributed Object Oriented Implementor should solve. In some cases we show other solutions than the chosen in our implementation, in these cases we give pros and contras for each solution.

10.4.1 Multiplatform implementation

The problem is that the program should run under UNIX and MS Windows. To write a C++ program that runs under both UNIX and Windows is quite trivial. Problems arise when the program contains operating system specific parts. In this case these parts was OS specific:

- The file naming
- The communication handling
- The shell command execution
- Preemption of processes (preemptive multitasking is not supported by MS Windows for Workgroups)
- User interfaces

The DOS operating system handles only file names 8+3 characters long. The UNIX systems can handle almost arbitrary long file names (usually the limit of 256 characters never exceeded). Another problem is the representation of path names. In the UNIX systems the '/' character is used to separate the directory names, in the DOS the '\' character is used. Another problem may be the DOS drive concept, where letters are assigned to hard disk partitions and floppy disk drives (and other devices). Another problem is that the UNIX file names are case sensitive, while the DOS file names are not, so the DOS may treat two different UNIX files (for example 'Example' and 'eXample') as a same file. Besides these purely operating system specific problems another problem is the different file naming conventions (which partly arises from the operating system specific limitations). This last problem means that the same type of files (such as object files) are named in different way (for example an object file generated from a C source file example.c named example.o on a UNIX box and example.obj under the DOS). Many cross-platform software shows solutions for a part of these problems (such as DOS NFS clients), but we used a different approach. The files are represented in an operating system independent form (i.e. always '/' used as separator), and DOS drive specifications are discarded under UNIX.

The communication handling is quite similar under UNIX and Windows, however in the Windows implementation constants and variables are named differently. The solution for this problem is to write the communication specific code twice and use ifdefs to include the correct version to the operating system specific code. As a solution to this problem the communication functions may be encapsulated into objects. Our implementation is based on the KW object set, which contains such an encapsulation. Unfortunately some of the features of the communication protocol are not included to these object encapsulations.

The shell command executions is one of the major problems in this multiplatform implementation. The main problem is with the Windows operating system which provides no functions to execute DOS commands. In our solution we used a quite complicated trick to execute a DOS command, which involved creation of Windows program information files and made the resulting program much more complicated and harder to install and maintain. Even the accessing of the return code of a command remained unsolved. We found no way to retrieve the return code of the command. The termination of the shell command for a user request is also not possible under Windows.

Another problem with the Windows environment that it is not primitive (i.e. a program may eat up all the CPU time, leaving no processing time to other programs). This limitation made the entire implementation more complicated. For code level compatibility we did not used some features of the UNIX operating system. We must implement a scheduling scheme that releases CPU time for other applications and to handle parallel communication requests. We can not emphasise the problems arisen with the parallel execution (in the sense that parallel actions take place in different hosts) and the parallel message handling. This is one of the key problems since timings are extremely important to avoid deadlocks and execution errors (timeouts that are detected as errors by the application).

The problem of user interfaces seems to be not relevant since the servers have small and simple user interfaces. Even with these interfaces a bunch of problems may arise. Not to mention that the interfaces should be implemented twice (since not too much common was found in the UNIX and Windows version), the communication may result problems in the interfaces. For example if an error occurred in the initialisation of the communication (which is necessary under Windows) the application window was not displayed (since the initialisation of the communication protocol was the part of the initialisation of the application), so the user has no indication of the error.

10.4.2 Project termination

Another great problem is the termination of a make process in the case of an error. The make process creates a quite complicated data structure representing a process. The cleanup of these complicated structures are not an obvious task if an error occurs (or an error message arrives from another host). These cleanups must be performed in all hosts involved in the project. Even on some hosts make actions may be in progress. The termination of these processes may be quite difficult and may result in unpredictable errors. In our implementations these processes won't be terminated, but the entire cleanup process will be suspended until the termination of these processes.

10.4.3 Communication

There are many possible problems with the communication. These problems partly depend on the network environment. There are many possible problems from the most basic connection problems to such a complicated problem such as the misconfiguration of the network software of the absence of name resolution.

10.5 Bugs and weaknesses of the implementation

The program runs only an IP network. The network must be configured properly to use host name resolution, etc. The program will not start up without the networking software even if make actions should be performed in one host.

The host are identified by their Internet addresses. This is sufficient in most of the cases, but a host may have more Internet addresses. If a host have multiple internet addresses the explicitly given or the first returned by the gethostbyname function is used.

The system only uses TCP connections. The use of the User Datagram protocol (UDP) may improve the effectively of the server, but makes it much more complex and may introduce brand new problems with message handling.

Many of the make applications uses the modification times of files. The network may span across time zone boundaries. The time zones are handled correctly by the CGMTime class. However if the clocks of the hosts is not synchronised correctly even on local networks many strange thing may happen. This problem can be used by using a time service in the net. (The description of the network time service is not in the scope of this documentation.) The TZ (time zone) environment variable must be set correctly.

The provided object set is only capable to handle the most basic actions. A well designed and implemented object set is necessary to effectively use the application.

In the Windows version only one starter application can be started once. Fortunately the use of multiple starter applications is not typical.

Because of some object oriented design concept the dmake utility lacks the flexibility of the traditional make utility. However by implementing a more general object set the flexibility may be greatly improved.

10.6 Most important features

There are many important features in this implementation. Probably the most important feature is the easy extensibility, and the use of the networking concept, which yields an effective parallel execution.

10.6.1 Easy network distribution

The Distributed Object Oriented make provides an easy to use interface to build distributed projects and keep them up-to-date. First you need to design your project, and describe it in an object oriented makefile. The object oriented makefile consists of object declarations and dependency relations. Define the hosts and objects (object, source and executable files, options etc.) in the makefile. Then describe the relation between the objects, by specifying dependency relations. Distribute your object in the hosts by specifying host dependencies. Redistributing the project is easy this way. Finally you must copy the data of source objects to the specified hosts.

10.6.2 Parallel execution

The make actions taken in different hosts will be executed in a parallel way. Design your project to take the advantage of parallel processing, but also note that the dependency relations between objects on different hosts may increase the network traffic. Make actions on the same host are executed one after the other. Parallel execution of these actions makes no sense, since this would result in a slower execution even if no swapping is required. However on multiprocessor machines the parallel execution may result a better performance.

Contents