Using Imake for Multiple Projects
(document version 1.02)

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 24 July 1990

1. Introduction

imake is a tool for generating makefiles and is used to allow software projects to be configured easily on a wide variety of target machines. It is distributed with projects such as the X Window System and the Kerberos authentication system. The tool *makedepend* is typically used in conjunction with *imake* to generate dependencies in makefiles. *xmkmf* is also used, for bootstrapping makefiles from imakefiles.

This document describes the particular way *imake* is used to configure projects at WRPRC. The method is patterned after the way *imake* is employed in the X Window System (version 11, release 4). The X11R4 method is significantly more developed than in X11R3. Presumably, *imake* as used in Kerberos (currently at protocol revision 4) might have been used as a base, but I judged it unsuitable for two reasons: (i) I learned *imake* from studying X, so I'm probably imprinted to it; (ii) development of the configuration files in Kerberos appears to be at a more primitive level even than in X11R3.

The X11R4 approach allows configuration files to be stored either in a well-known system-wide location (e.g., */usr/lib/config*), or in a *config* directory under the project top-level directory.

A more-than-passing acquaintance with *imake* is assumed here. Helpful companion documentation includes ‘‘Configuration Management in the X Window System,’’ the configuration and *README* files in the X11R4 distribution (in the *mit/config* directory), and the WRPRC configuration files (see below for information on distribution and availability).

2. Statement of Problem

Because of the flexibility *imake* provides as a configuration tool (once you figure out how to use it!), I had begun to use it on all my projects. Since these were all developed at the same site, the projects naturally had quite a bit of configuration information in common. On the other hand, different projects just as naturally had their own individual requirements.

What I had been doing was creating a *config* directory in each project directory, with a copy of the configuration files in each, modified as necessary for the project (otherwise known as ‘‘using brute force’’). I did not use configuration files in a system-wide location because of differing project requirements. At the same time, there was a great deal of redundancy between sets of configuration files since projects were built and installed in similar ways. It seemed rather wasteful and silly not to take advantage of this overlap, besides the fact that the multiple copies of files contributed to a maintenance inefficiency when changes were made that affected all projects.

This (unsatisfactory) situation prompts the question: how can the number of files to be maintained be minimized and how can configuration files be shared among projects without sacrificing the ability to specify project-specific information as necessary?

3. Solution to Problem

Since *imake* possessed the ability to look through multiple directories when looking for configuration files, the simplest solution to the problem seemed to be:

(i) Create a set of standard configuration files and store them in a well-known location (referred to here as StdConfigDir).

(ii) Allow projects to define an alternate configuration directory (referred to here as ProjConfigDir, typically the *config* directory under the project directory).

(iii) Tell *imake* to look through ProjConfigDir, then StdConfigDir for configuration files. This means changing *xmkmf*, which executes *imake*, and the configuration file definitions that construct *imake* commands.

In other words, a standard set of configuration files is created and stored in a well-known location for use by multiple projects. However, should it be required, individual projects place configuration files in their own *config* directory, which takes precedence over StdConfigDir when searching for files. In the degenerate case, projects need supply nothing if the standard stuff is sufficient; only files in StdConfigDir will be used.

4. Implementation of Solution

4.1. Imake.tmpl Architecture

The architecture of the *imake* template, *Imake.tmpl,* as used in X11R4 is, roughly:

header block section	
#include <*platform*.cf>	(platform-specific definitions)
#include <site.def>	(site-specific definitions)
system description, build and installation definitions	
#include <Project.tmpl>	(project-specific definitions)
#include <Imake.rules>	(*imake* rules)
#include "./Imakefile"	(*Imakefile* for current directory)

The header block section determines the machine type and the platform-specific file *platform*.cf to be included.

To accommodate other project-specific files, extra #include’s are introduced into *Imake.tmpl*. The resulting modified architecture becomes:

header block section

#include <*platform*.pcf>
#include <*platform*.cf>

#include <site.pdef>
#include <site.def>

system description, build and installation definitions

#include <Project.ptmpl>
#include <Project.tmpl>

#include <Imake.prules>
#include <Imake.rules>

#include "./Imakefile"

The header block section now additionally determines the name of the project-specific platform file *platform*.pcf.

The modification is based on the notion of override files. Before any of the platform, site, project or rules files are included, another related file is included which may contain project-specific information. Standard versions of these project-specific files (**.pcf*, *site.pdef*, *Project.ptmpl* and *Imake.prules*) are present in StdConfigDir (to prevent file-inclusion errors) but they are all empty. If a given project has special configuration requirements, it may supply its own private versions of any of these files in its *config* directory. Since *imake* will search that directory before StdConfigDir, the standard (empty) versions will be overridden. The project does not have to actually replace them.

Example 1: At the one extreme, when a project has no special project-specific configuration requirements (no *config* directory, or an empty one), only the files in StdConfigDir are used to create makefiles. Since the default project-specific files are empty, the new architecture of *Imake.tmpl* is functionally equivalent to the old. At the other extreme, a project can replace *all* the configuration files; nothing from StdConfigDir is used then.

Example 2: The standard *site.def* file might be used by the system administrator to set the value of BinDir to the directory in which programs are installed. A project might want them installed somewhere else; this could be done by defining BinDir in the project-specific site file *site.pdef* to override the value in *site.def*.

Example 3: Override files may also be used to extend the standard configuration by defining symbols not appearing at all in the standard files. For instance, if some programs were to be installed in the standard BinDir, but others were to be installed in a private directory, a symbol such as ProjBinDir could be defined for the latter purpose in *Project.ptmpl*.

One bothersome thing about this architecture is the naming of *Project.tmpl* and *Project.ptmpl*. If *Project.tmpl* contains ‘‘project-specific’’ information, then what does *Project.ptmpl* contain? ‘‘Project-project-specific’’ information? Ugh. I prefer to consider *Project.tmpl* to contain ‘‘standard’’ (i.e., best-guess) project configuration parameters, and *Project.ptmpl* to contain the parameters which actually are specific to a given project.

The disadvantages of this architecture are obvious:

- *Imake.tmpl* is somewhat more complex.
- StdConfigDir must be populated with the additional **.pcf*, *site.pdef*, *Project.ptmpl* and *Imake.prules* files (although this is easy since they are all empty).

The advantages are:

- Configuration information is centralized, but flexibility is maintained. Projects may override definitions in the the standard configuration files without replacing those files directly, and multiple projects may use files in StdConfigDir cooperatively without interfering with each other.
- Override files minimize the need for individual projects to modify the standard non-project specific files (*Imake.tmpl*, **.cf*, *site.def*, *Project.tmpl* and *Imake.rules*) directly. The philosophy underlying the architecture is that the standard files describe a common ‘‘baseline’’ configuration, and individual projects need specify configuration information (using private **.pcf*, *site.pdef*, *Project.ptmpl* and *Imake.prules* files) only to the extent that they *differ* from the standard.
- When improvements are made to the standard files, the changes are immediately available to any project using them via a normal configuration step, i.e., ‘‘make Makefile;make Makefiles’’. (Of course, it also becomes possible to break a lot of projects if an erroneous change is made.)
- Overall, fewer files need to be maintained, and less space is used.

4.2. Other Changes

xmkmf is used to bootstrap makefiles from imakefiles. To accommodate the above scheme, it was modified from the version distributed with X11R4 to search the project’s *config* directory, then the standard configuration directory (i.e., ProjConfigDir then StdConfigDir, not just StdConfigDir).

The definition of the makefile variable IMAKE_CMD in the configuration files was also changed:

```
Old (from X11R4):
#ifdef UseInstalled
        IRULESRC = $(CONFIGDIR)
        IMAKE_CMD = $(IMAKE) -DUseInstalled \
                -I$(IRULESRC) \
                $(IMAKE_DEFINES)
#else
        IRULESRC = $(CONFIGSRC)
        IMAKE_CMD = $(NEWTOP)$(IMAKE) \
                -I$(NEWTOP)$(IRULESRC) \
                $(IMAKE_DEFINES)
#endif

New:
#ifdef UseInstalled
        IMAKE_CMD = $(IMAKE) -DUseInstalled \
                -I$(NEWTOP)$(PROJCONFIGDIR) \
                -I$(STDCONFIGDIR) \
                $(IMAKE_DEFINES)
#else
        IMAKE_CMD = $(NEWTOP)$(IMAKE) \
                -I$(NEWTOP)$(PROJCONFIGDIR) \
                -I$(STDCONFIGDIR) \
                $(IMAKE_DEFINES)
#endif
```

The old rule determines whether to use versions of *imake* (and *makedepend*) already installed in public directories or build them in the source tree based on whether UseInstalled is defined or not. That symbol also determines where to look for configuration files and defines the value of IRULESRC accordingly. If UseInstalled is undefined, searches look in CONFIGSRC, the project's *config* directory, otherwise in CONFIGDIR, the standard configuration directory.

This behavior is unsuitable for the purposes described here, since both of those directories should be searched, if they exist. The new rule makes UseInstalled relevant only for determining where to find (and possibly build) *imake* and *makedepend*, not where to look for configuration files.

5. Test of Implementation

After making the changes described above, a flight test was arranged. Nine projects developed under six different logins were converted to use the standard files. Projects supplied override files where necessary. These projects were all initially developed using configuration files that bore a fairly strong relationship to the standard set, so it was expected that conversion would not be unbearably odious. The degree of relationship broke down as follows:

- Two projects used configuration files that were identical (or nearly so) to the standards. Each of these projects, unlike the other seven, comprised only a single source directory. As expected, conversion was completely trivial, once the top-level *Makefile* was rebuilt with *xmkmf*.

- Four projects required additional project-specific specification beyond the standard configuration (special include file directories, link libraries, installation directories, etc.). These usually required only *site.pdef* and *Project.tmpl* files in their local *config* directories. One project required a *platform.pcf* file. Project owner and group names (used for install operations) were moved out of *site.def* files into *site.pdef*.

- Three projects diverged from the standard files fairly significantly, because although their configuration files were initially set up from files similar to the standards, less effort had been exerted to keep them in sync as project development progressed. Besides the changes made to the projects described in the previous paragraph, there were additional special commands for compilation, special *Imakefile* rules, etc. Nevertheless, conversion proceeded smoothly and without incident.

One thing I had done in the original *Project.tmpl* files was to define certain makefile variables directly (e.g., ‘‘INCLUDESRC=$(TOP)/h’’), rather than by using a previously #define’d *cpp* symbol (e.g., ‘‘INCLUDESRC=IncludeSrc’’, where IncludeSrc is #define’d within a #ifndef/#endif block). This was a short-cut, that works in a non-shared environment but needed to be fixed to work in a shared environment. A default value of IncludeSrc needs to be placed in *Project.tmpl*, and individual projects redefine that symbol as necessary in *Project.ptmpl*.

It was necessary to place a number of *cpp* symbol definitions in *Project.tmpl* within #ifndef/#endif blocks, since they became liable to being overridden in *Project.ptmpl*.

In the top-level imakefiles for the multiple source directory projects, the ‘‘make World’’ rules had to be adjusted to account for whether UseInstalled is defined or not. If so, the build process does not need to try to build *imake* and *makedepend* in the project source tree. (That would be a mistake since the source for those programs would not necessarily be present.)

Overall, conversion was simple and straightforward. I judged the conversion odiousness coefficient acceptably low.

6. Another Problem—Project Groups

The mechanism described above allows a group of similarly-configured projects to share configuration files and selectively extend or override them as necessary. This provides a solution to the original problem (how to avoid maintaining multiple full sets of files while maintaining flexibility), but leads to another. Suppose that instead of a group of projects, there exist several groups of projects, such that projects within a group are similar, but differences between groups are quite large. This can happen, for instance, if projects designed and developed at one site under certain conventions are installed at another site where different conventions are used.

The difficulty here is that there are two sets of ‘‘standard’’ configuration files, and they cannot both be installed in the same well-known location. This problem can be overcome quite simply. Instead of using StdConfigDir as the single well-known location in which the standards are installed, define StdConfigPath to be that directory, and use it as the root of a set of subdirectories, each of which corresponds to a project group. Each group’s standard configuration files are installed in the appropriate directory, and StdConfigDir for a given project group will be defined as that directory (e.g., StdConfigPath/*abc* for project *abc*).

To allow *xmkmf* to be used for bootstrapping, it was modified slightly again to select one of the subdirectories as the hardwired-in default, but allow another to be selected through a **–C** option. For convenience, the argument following **–C** may be absolute or relative. If absolute, it is used unchanged. If relative, StdConfigDir is prepended.

This modification makes *xmkmf* incompatible with the version distributed with X11R4, which is a problem for sites using that version. Assuming X configuration files are installed in */usr/lib/X11/config* and the value of StdConfigPath is */usr/lib/config*, this incompatibility can be obviated as follows:

```
mkdir /usr/lib/config
cd /usr/lib/config
ln -s /usr/lib/X11/config X11
ln -s X11 default
```

For sites without symbolic links, the files in */usr/lib/X11/config* can be copied into */usr/lib/config/X11*, and *xmkmf* can be hacked to use */usr/lib/config/X11* instead of */usr/lib/config/default* as the default. Ugly, but workable. (Also known as ‘‘crude, but effective’’.)

With these conventions, projects within groups can share redundant information, as before, and project groups can coexist peacefully.

The nine projects previously converted were tested against these additional modifications. They required only ‘‘xmkmf;make Makefile;make Makefiles’’ to properly reset the configuration information in the makefiles.

7. Distribution of Projects

The mechanism described in this document appears to solve the problem of reusing configuration files while allowing for extension and override within individual projects. However, when a project is packaged for distribution, it’s very little use to include only those configuration files which are unique to the project, if the receiving site doesn’t have the standard files. Several alternatives come to mind.

- Provide the standards as a separate distribution, along with the sources to *imake* and *makedepend*. This is perhaps inconvenient for the receiver, as it is another distribution that needs to be retrieved.
- Copy all the standard files into the project’s *config* directory and distribute along with the project. One has to be careful when doing the copying not to destroy the project’s override files, the resulting distribution size is larger, and distributing the standards is unnecessary if receiving site has them already. Also, it is not clear which files are actually unique to the project when this is done.
- Keep the project’s unique configuration files in *config* (where they are normally anyway), and copies of the standard files in *config-std*. If the receiver has the latter already, they can be jettisoned easily, otherwise installed in the project’s StdConfigDir.

It is unclear whether any of these has any compelling advantage over the others.

8. Beginning New Projects

If a new project has special configuration requirements, a file *README.{project-name}* should be created in *config* explaining what they are.

8.1. Define UseInstalled, or Not?

If a project wants to use the installed versions of *imake* and *makedepend*, the initial makefile should be created from the the imakefile by executing *xmkmf* with no arguments (or just **–C** *configdir*) in the project top-level directory. This will cause -DUseInstalled to be passed to *imake* which will in turn cause the version of IMAKE_CMD that includes -DUseInstalled to be selected from the configuration files. Thus -DUseInstalled will propagate nicely through all future ‘‘make Makefile’’ operations.

If *imake* and *makedepend* are to be built in the project source tree, UseInstalled should not be defined, and you must have the source under the project. *imake* source is in *config* and *makedepend* source is in *util/makedepend* or *util/scripts*, depending on the definition of UseCCMakeDepend. Execute ‘‘xmkmf **.**’’ (note the **.**!) in the project top-level directory. This will run *imake* without -DUseInstalled, and the proper IMAKE_CMD in the config files will be selected.

One problematic aspect of *xmkmf* is that it really should always pass -DUseInstalled, since it assumes there is an installed version of *imake* anyway (it doesn't specify a path to it). But then the problem becomes how to turn UseInstalled off later? You can't run ‘‘make Makefile’’ because the version of IMAKE_CMD that uses -DUseInstalled has already been selected.

9. Distribution and Availability

The standard WRPRC configuration files are (or will be soon) available for anonymous *ftp* access in the *~ftp/pub/imake-stuff* directory, in file *config-WRPRC.shar.Z* on host *indri.primate.wisc.edu* (Internet address 128.104.230.11). These files illustrate the concepts discussed in this document.

This distribution also includes parts of the X11R4 *mit/util* source tree (where *makedepend* and *xmkmf* are found) as well as *msub*, a locally-developed utility for yanking the values of variables directly from makefiles to produce target files from templates.

The X11R4 configuration files are in the same directory, in *config-X11R4.shar.Z.*