Configuration Management in the X Window System

Jim Fulton

X Consortium
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

ABSTRACT

The X Window System† has become an industry standard for network window technology in part because of the portability of the sample implementation from MIT. Although many systems are designed to reuse source code across different platforms, X is unusual in its portability across software build environments. This paper describes several mechanisms used in the MIT release of the X Window System to obtain such flexibility, and summarizes some of the lessons learned in trying to support X on a number of different platforms.

1. Introduction

The X Window System† is a portable, network transparent window system originally developed at MIT. It is intended for use on raster display devices ranging from simple monochrome frame buffers to deep, true color graphics processors. Because of its client/server architecture, the non-proprietary nature of its background, and the portability of the sample implementation from MIT, the X Window System has rapidly grown to become an industry standard. This portability is the result of several factors: a system architecture that isolates operating system and device-specifics at several levels; a slow, but machine-independent, graphics package that may be used for an initial port and to handle cases that the underlying graphics hardware does not support; and the use of a few, higher-level tools for managing the build process itself.

1.1. Summary of X Window System Architecture

The X Window System is the result of a combined effort between MIT Project Athena and the MIT Laboratory for Computer Science. Since its inception in 1984, X has been redesigned three times, culminating in Version 11 which has since become an industry standard (see [Scheifler 88] for a more detailed history). X uses the client/server model of limiting interactions with the physical display hardware to a single program (the *server*) and providing a way for applications (the *clients*) to send messages (known as *requests*) to the server to ask it to perform graphics operations on the client's behalf. These messages are sent along a reliable, sequenced, duplex byte stream using whatever underlying transport mechanisms the operating system provides. If connections using network virtual circuits (such as TCP/IP or DECnet) are supported, clients may be run on any remote machine (including ones of differing architectures) while still displaying on the

---

† X Window System is a trademark of MIT; DECnet is a trademark of Digital Equipment Corporation; UNIX is a registered trademark of AT&T.

local server.

The details of how the client establishes and maintains connections with the server are typically hidden in a subroutine package (known as a *language binding*) which provides a function call interface to the X protocol. Higher level toolkits and user interface management systems are then built on top of the binding library, as shown in Figure 1 for the C programming language. Since only the underlying operating system networking interface of the binding (shown in *italics*) need be changed when porting to a new platform, well-written applications can simply be recompiled.

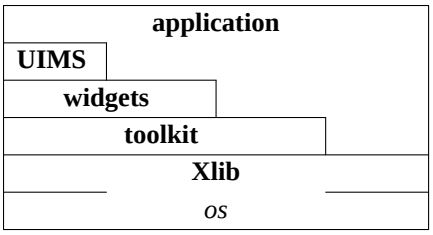

Figure 1: architecture of a typical C language client program

The server takes care of clipping of graphics output and routing keyboard and pointer input to the appropriate applications. Unlike many previous window systems, moving and resizing of windows are handled outside the server by special X applications called *window managers*. Different user interface policies can be selected simply by running a different window manager.

The MIT sample server can be divided into three sections: a device-independent layer called *diX* for managing the various shared resources (windows, pixmaps, colormaps, fonts, cursors, etc.), an operating system layer called *os* for performing machine-specific operations (managing connections to clients, dealing with timers, reading color and font name databases, and memory allocation), and a device-specific layer called *ddX* for drawing on the display and getting input from the keyboard and pointer. Only the *os* and *ddX* portions of the server need to be changed when porting X to a new device.

Although this is still a substantial amount of work, a collection of pixel-oriented drawing packages that only require device-specific routines (refered to as *spans*) to read and write rows of pixels are provided to allow initial ports of X to be done in a very short time. A server developer can then concentrate on replacing those operations that can be implemented more efficiently by the hardware. Figure 2 shows the relative layering of the various packages within the sample server from MIT. The *mi* library provides highly portable, machine-independent routines that may be used on a wide variety of displays. The *mfb* and *cfb* libraries contain versions of the graphics routines for monochrome and color frame buffers, respectively. Finally, the *snf* library can be used to read fonts stored in Server Natural Format. Typically, only the sections printed in *italics* need be changed when moving to a new platform.

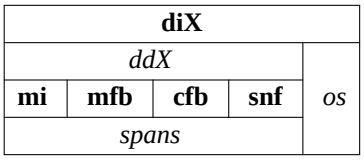

Figure 2: architecture of the MIT sample server

By splitting out the device-specific code (by separating clients from servers and *diX* from *ddX*) and then providing portable utility libraries (*mi*, *mfb*, *cfb*, and *cfb*) that may be used to implement the non-portable portions of the system, much of the code can be reused across many platforms, ranging from personal

computers to supercomputers.

2. Configuring the Software Build Process

In practice, porting X to a new platform typically requires adding support in the operating system-specific networking routines and mixing together pieces of machine-independent and device-specific code to access the input and output hardware. Although this approach is very portable, it increases the complexity of the build process as different implementations require different subsets. One solution is to litter the source code with machine-specific compiler directives controlling which modules areas get built on a given platform. However, this rapidly leads to sources that are hard to understand and even harder to maintain.

A more serious problem with this approach is that it requires configuration information to be replicated in almost every module. In addition to being highly prone to error, modifying or adding a new configuration becomes extremely difficult. In contrast, collecting the various options and parameters in a single location makes it possible for someone to reconfigure the system without having to understand how all of the modules fit together.

Although sophisticated software management systems are very useful, they tend to be found only on specific platforms. Since the configuration system must be working before a build can begin, the MIT releases try to adhere to the following principles:

- Use existing tools to do the build (e.g. *make*) where possible; writing complicated new tools simply adds to the amount of software that has to be bootstrapped.
- Keep it simple. Every platform has a different set of extensions and bugs. Plan for the least common denominator by only using the core features of known tools; don't rely on vendor-specific features.
- Providing sample implementations of simple tools that are not available on all platforms (e.g. a BSD-compatible *install* script for System V) is very useful.
- Machine-dependencies should be centralized to make reconfiguration easy.
- Site-wide options (e.g. default parameters such as directory names, file permissions, and enabling particular features) should be stored in only one location.
- Rebuilding within the source tree without losing any of the configuration information must be simple.
- It should be possible to configure external software without requiring access to the source tree.

One approach is to add certain programming constructs (particularly conditionals and iterators) to the utility used to actually build the software (usually *make*; see [Lord 88]). Although this an attractive solution, limits on time and personnel made implementing and maintaining such a system impractical for X.

The MIT releases of X employ a less ambitious approach that uses existing tools (particularly *make* and *cpp*). *Makefiles* are generated automatically by a small, very simple program named *imake* (written by Todd Brunhoff of Tektronix) that combines a template listing variables and rules that are common to all *Makefiles*, a machine- and a site-specific configuration file, a set of rule functions written as *cpp* macros, and simple specifications of targets and sources called *Imakefiles*. Since the descriptions of the inputs and outputs of the build are separated from the commands that implement them, machine dependencies such as the following can be controlled from a single location:

- Some versions of *make* require that the variable SHELL to be set to the name of the shell that should be used to execute *make* commands.
- The names of various special *make* variables (e.g. MFLAGS vs. MAKEFLAGS) differ between versions.
- Special directives to control interaction with source code maintenance systems are required by some versions of *make*.
- Rules for building targets (e.g. *ranlib*, lint options, executable shell scripts, selecting alternate compilers) differ among platforms.

- Some systems require special compiler options (e.g. increased internal table sizes, floating point options) for even simple programs.
- Some systems require extra libraries when linking programs.
- Not all systems need to compile all sources.
- Configuration parameters may need to be passed to some (such as -DDNETCONN to compile in DECnet support) or all (such as -DSYSV to select System V code) programs as preprocessor symbols.
- Almost all systems organize header files differently, making static dependencies in *Makefiles* impossible to generate.

By using the C preprocessor, *imake* provides a familiar set of interfaces to conditionals, macros, and symbolic constants. Common operations, such as compiling programs, creating libraries, creating shell scripts, and managing subdirectories, can be described in a concise, simple way. Figure 3 shows the *Imakefile* used to build a manual page browser named *xman* (written by Chris Peterson program of the MIT X Consortium, based on an implementation for X10 by Barry Shein):

```
DEFINES =  -DHELPFILE=\"$(LIBDIR)$(PATHSEP)xman.help\"
LOCAL_LIBRARIES =  $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB)
SRCS =  ScrollByL.c handler.c man.c pages.c buttons.c help.c menu.c search.c \
        globals.c main.c misc.c tkfuncs.c
OBJS =  ScrollByL.o handler.o man.o pages.o buttons.o help.o menu.o search.o \
        globals.o main.o misc.o tkfuncs.o
INCLUDES =  -I$(TOOLKITSRC) -I$(TOP)

ComplexProgramTarget (xman)
InstallNonExec (xman.help, $(LIBDIR))
```

Figure 3: *Imakefile* used by a typical client program

This application requires the name of the directory in which its help file is installed (which is a configuration parameter), several libraries, and various X header files. The macro *ComplexProgramTarget* generates the appropriate rules to build the program, install it, compute dependencies, and remove old versions of the program and its object files. The *InstallNonExec* macro generates rules to install *xman*'s help file with appropriate permissions.

3. Generating Makefiles

Although *imake* is a fairly powerful tool, it is a very simple program. All of the real work is performed by the template, rule, and configuration files. The version currently used at MIT (which differs somewhat from the version supplied in the last release of X) uses symbolic constants for all configuration parameters so that they may be overridden or used by other parameters. General build issues (such as the command to execute to run the compiler) are isolated from X issues (such as where should application default files be installed) by splitting the template as shown in Figure 4.

This template instructs *imake* to perform the following steps when creating a *Makefile*:

1. Using conditionals, *Imake.tmpl* determines the machine for which the build is being configured and includes a machine-specific configuration file (usually named *machine*.cf). Using the C preprocessor to define various symbols, this configuration file sets the major and minor version numbers of the operating system, the names of any servers to build, and any special programs (such as alternate compilers) or options (usually to increase internal table sizes) that need to be used during the build. Defaults are provided for all parameters, so .cf files need only describe how this particular platform differs from ‘‘generic’’ UNIX System V or BSD UNIX. Unlike previous versions of the *imake* configuration files, when new parameters are added, only the

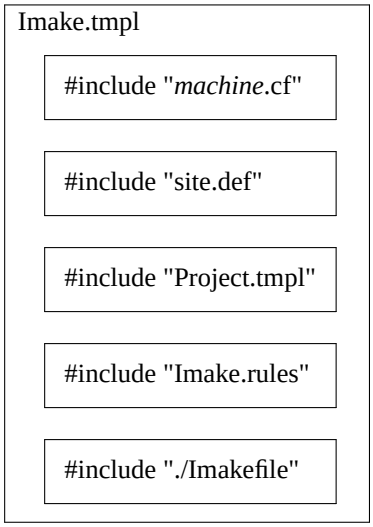

Figure 4: structure of *imake* template used by X

systems which are effected by them need to be updated.

2. Next, a site-specific file (named *site.def*) is included so that parameters from the .cf files may be overridden or defaults for other options provided. This is typically used by a site administrator to set the names of the various directories into which the software should be installed. Again, all of the standard *cpp* constructs may be used.
3. A project-specific file (named *Project.tmpl*) is included to set various parameters used by the particular software package being configured. By separating the project parameters (such as directories, options, etc.) from build parameters (such as compilers, utilities, etc.), the master template and the .cf files can be shared among various development efforts.
4. A file containing the set of *cpp* rules (named *Imake.rules*) is included. This is where the various macro functions used in the master template and the per-directory description files (named *Imakefile*) are defined. These rules typically make very heavy use of the *make* variables defined in *Imake.tmpl* so that a build's configuration may be changed without having to edit this file.
5. The *Imakefile* describing the input files and output targets for the current directory is included. This file is supplied by the programmer instead of a *Makefile*. The functions that it invokes are translated by *cpp* into series of *make* rules and targets.
6. Finally, *make* rules for recreating the *Makefile* and managing subdirectories are appended, and the result is written out as the new *Makefile*.

Imake, along with a separate tool (named *makedepend*, also written by Brunhoff) that generates *Makefile* dependencies between object files and the source files used to build them, allows properly configured *Makefiles* to be regenerated quickly and correctly. By isolating the machine- and site-specifics from the programmer, *imake* is much like a well-developed text formatter: both allow the writer to concentrate on the content, rather than the production, of a document.

4. How X uses *imake*

Development of X at MIT is currently done on more than half a dozen different platforms, each of which is running a different operating system. A common source pool is shared across those machines that support symbolic links and NFS by creating trees of links pointing back to the master sources (similar to the object trees of [Harrison 88]). Editing and source code control is done in the master sources and builds are done

in the link trees.

A full build is done by creating a fresh link tree and invoking a simple, stub top-level *Makefile* which:

1. compiles *imake*.
2. builds the real top-level *Makefile*.
3. builds the rest of the *Makefiles* using the new top-level *Makefile*.
4. removes any object files left over from the previous build.
5. builds the header file tree, and computes and appends the list of dependencies between object files and sources to the appropriate *Makefiles*.
6. and finally, compiles all of the sources.

If the build completes successfully, programs, libraries, data files, and manual pages may then be installed. By keeping object files out of the master source tree, backups and releases can be done easily and efficiently. By substituting local copies of particular files for the appropriate links, developers can work without disturbing others.

5. Limitations

Although the system described here is very useful, it isn't perfect. Differences between utilities on various systems places a restriction on how well existing tools can be used. One of the reasons why *imake* is a program instead of a trivial invocation of the C preprocessor is that some *cpp*'s collapse tabs into spaces while others do not. Since *make* uses tabs to separate commands from targets, *imake* must sometimes reformat the output from *cpp* so that a valid *Makefile* is generated.

Since *cpp* only provides global scoping of symbolic constants, parameters are visible to the whole configuration system. For larger projects, this approach will probably prove unwieldy both to the people trying to maintain them and to the preprocessors that keep the entire symbol table in memory.

The macro facility provided by *cpp* is convenient because it is available on every platform and it is familar to most people. However, a better language with real programming constructs might provide a better interface. The notions of describing one platform in terms of another and providing private configuration parameters map intriguingly well into the models used in object management systems.

6. Summary and Observations

The sample implementation of the X Window System from MIT takes advantage of a system architecture that goes to great lengths to isolate device-dependencies. By selectively using portable versions of the device-specific functions, a developer moving X to a new platform can quickly get an initial port up and running very quickly.

To manage the various combinations of modules and to cope with the differing requirements of every platform and site, X uses a utility named *imake* to separate the description of sources and targets from the details of how the software is actually built. Using as few external tools as possible, this mechanism allows support for new platforms to be added with relatively little effort.

Although the approaches taken by MIT will not work for everyone, several of its experiences may be useful in other projects:

- Even if portability isn't a goal now, it probably will become one sooner than expected.
- Just as in other areas, it frequently pays to periodically stand back from a problem and see whether or not a simple tool will help. With luck and the right amount of abstracting it may even solve several problems at once.
- Be wary of anything that requires manual intervention.
- And finally, there is no such thing as portable software, only software that has been ported.

7. References

[Harrison 88]

''Rtools: Tools for Software Management in a Distributed Computing Environment,'' Helen E. Harrison, Stephen P. Schaefer, Terry S. Yoo, *Proceedings of the Usenix Association Summer Conference*, June 1988, 85-94.

[Lord 88]

''Tools and Policies for the Hierarchical Management of Source Code Development,'' Thomas Lord, *Proceedings of the Usenix Association Summer Conference*, June 1988, 95-106.

[Scheifler 88]

X Window System: C Library and Protocol Reference, Robert Scheifler, James Gettys, and Ron Newman, Digital Press, Bedford, MA, 1988.