Comments on the X11R6.3 Configuration Files and Programs

Paul DuBois
dubois@primate.wisc.edu
1 January, 1997

I've been looking over the X11R6.3 configuration files and programs and have noticed some improvements over previous releases, as well as some things I find puzzling or that seem like bugs. I'm posting my observations here in hopes of eliciting discussion that will clarify things I don't understand. Also, I admit that these comments are a little propagandistic, because I suggest some changes that I believe would be useful.

Configuration File Changes

Architecture Comments

The configuration file architecture has changed. It's now something like this:

```
Imake.tmpl:
    Imake.cf
    site.def (with BeforeVendorCF defined)
    MacroIncludeFile (i.e., vendor.cf)
    site.def (with AfterVendorCF defined)
    Imake.rules
    most system parameters
    ProjectRulesFile (e.g., X11.rules)
    LocalRulesFile
    ProjectTmplFile (e.g., X11.tmpl)
    LocalTmplFile
    Imakefile
```

Imake.tmpl no longer includes `Project.tmpl`; instead it includes the two files named by the `ProjectRulesFile` and `ProjectTmplFile` macros. This pretty well completes the separation of system and project information, where system information is found in *Imake.tmpl* and *Imake.rules* on the one hand, and project information is found in project-specific files (*X11.rules*, *X11.tmpl*) on the other. This process began in X11R4 (when the two types of information were still very much intertwined), and now seems to be virtually complete. This clearer separation of system and project-specific information should make it easier to generate Makefiles for Motif or for CDE:

- To generate Makefiles for Motif, define `ProjectRulesFile` and `ProjectTmplFile` as `<Motif.rules>` and `<Motif.tmpl>.` The X11 rules and template files will still be used, because *Motif.rules* includes *X11.rules* and *Motif.tmpl* includes *X11.tmpl*.
- Similarly, to generate Makefiles for CDE, define `ProjectRulesFile` and `ProjectTmplFile` as `<cde.rules>` and `<cde.tmpl>.` The Motif and X11 rules and template files will still be used, because *cde.rules* includes *Motif.rules* (and thus *X11.rules*), and *cde.tmpl* includes *Motif.tmpl* (and thus *X11.tmpl*).

If you could simply redefine `ProjectRulesFile` and `ProjectTmplFile` in *site.def*, it'd be quite easy to use different project files. Unfortunately, it seems to be more complicated than is necessary. *Imake.tmpl* contains a section that looks like this:

```
#ifndef TopLevelProject
# define TopLevelProject    X11
#endif
#ifndef ProjectRulesFile
# define ProjectRulesFile       Concat3(<,TopLevelProject,.rules>)
#endif
#include ProjectRulesFile

#ifndef ProjectTmplFile
#define ProjectTmplFile         Concat3(<,TopLevelProject,.tmpl>)
#endif
#include ProjectTmplFile
```

The idea here, I think, is that to affect how `ProjectRulesFile` and `ProjectTmplFile` are defined, you only have to redefine `TopLevelProject`, e.g., in *site.def*. Unfortunately, when we look at *site.def*, we see this:

```
/* On systems where cpp doesn't expand correctly macros in include directives
 * the two following macros need to be defined directly (where "X11" is
 * really whatever the TopLevelProject macro is defined to be).
 */
# if defined(SunArchitecture) || defined(AIXArchitecture) \
     || defined(USLArchitecture) || defined(UXPArchitecture) \
     || defined(SCOArchitecture)
#  ifndef ProjectRulesFile
#   define ProjectRulesFile <X11.rules>
#  endif
#  ifndef ProjectTmplFile
#   define ProjectTmplFile  <X11.tmpl>
#  endif
# endif
```

So it seems that there are a lot of systems for which redefining the `TopLevelProject` macro won't work. (And who knows whether the list of faulty systems is complete, since R6.3 hasn't been tested on all systems for which there is a *vendor.cf* file.) The upshot? To use different project files, you must know whether you can set `TopLevelProject`, or whether you have to set `ProjectRulesFile` and `ProjectTmplFile` directly instead.

It seems to me that this could simpler, both in terms of what goes in the configuration files, and for people who are trying to figure out how to use the files. How? Junk `TopLevelProject` entirely and work only with `ProjectRulesFile` and `ProjectTmplFile`. That would make the section in *Imake.tmpl* look like this:

```
#ifndef ProjectRulesFile
# define ProjectRulesFile       <X11.rules>   /* or maybe <noop.rules> */
#endif
#include ProjectRulesFile

#ifndef ProjectTmplFile
# define ProjectTmplFile        <X11.tmpl>    /* or maybe <noop.rules> */
#endif
#include ProjectTmplFile
```

And the section in *site.def* would look like this:

```
/*
 * Redefine these values to use different project-specific files
 */
#ifndef ProjectRulesFile
# define ProjectRulesFile   <X11.rules>
#endif
#ifndef ProjectTmplFile
# define ProjectTmplFile    <X11.tmpl>
#endif
```

Now there's no decision to make about whether to redefine `TopLevelProject` or not, and it's clearer what to do to use the Motif or CDE files: redefine `ProjectRulesFile` and `ProjectTmplFile` in *site.def*. In addition, since this suggested change eliminates the `XXXArchitecture` tests, it has none of the system-dependent stuff that is in the original machinery.

A side effect of introducing `ProjectRulesFile` and `ProjectTmplFile` into the architecture appears to be that `LocalRulesFile` and `LocalTmplFile` are effectively vestigial now. They were introduced in R6 as a way of allowing the architecture to accommodate the Motif configuration files: to cause the Motif files to be processed, you'd put the following in *site.def*:

```
    #define LocalRulesFile <Motif.rules>
    #define LocalTmplFile <Motif.tmpl>
```

The new Motif files themselves include the relevant X11 files, so you can now achieve the same effect by redefining `ProjectRulesFile` and `ProjectTmplFile` instead. As a result, `LocalRulesFile` and `LocalTmplFile` appear to be unnecessary. Is there still some use for the latter two macros?

Other Configuration File Comments

- *X11.rules* doesn't contain any rules! Why is the stuff in that file there and not in *X11.tmpl*?
- *Imake.rules* up through R6.1 has defined `ComplexProgramTarget_1()` through `ComplexProgramTarget_3()` rules, and `ComplexProgramTarget_1()` has defined the `OBJS` and `SRCS` variables in terms of `OBJS1` through `OBJS3` and `SRCS1` through `SRCS3`. There are now additional rules `ComplexProgramTarget_4()` through `ComplexProgramTarget_10()`, and `ComplexProgramTarget_1()` defines `OBJS` and `SRCS` in terms of `OBJS1` through `OBJS10` and `SRCS1` through `SRCS10`. (The C++ rules have been extended in a similar way.)
- *X11.tmpl* defines the variables `DEPLIBS1` through `DEPLIBS10`, rather than just `DEPLIBS1` through `DEPLIBS3` that the old *Project.tmpl* used to define. (This change parallels the *Imake.rules* changes discussed in the preceding item above.) However, *Motif.tmpl* still defines only `DEPLIBS1` through `DEPLIBS3`. (On the other hand, since *X11.tmpl* already defines all the `DEPLIBSn` variables, it's not clear to me why *Motif.tmpl* defines any `DEPLIBSn` variables at all.)
- *Motif.tmpl* no longer messes around with `IMAKE_CMD`; that's good.
- Recent Motif distributions that have included R5-based configuration files used generator macros to construct variable names for referring to libraries. However, those macros were incompatible with the generator macros used in the X11 configuration files from R6 on. The *Motif.tmpl* that ships with R6.3 uses the same generator macros that are used elsewhere in the R6.3 configuration files. That's good, too.
- In the configuration files that I've seen ship with Motif thus far, `UseInstalledMotif` is a macro that has been either defined or left undefined. In the *Motif.tmpl* that ships with the R6.3 configuration files, `UseInstalledMotif` has become a Boolean (`YES`/`NO`) macro.
- `UseInstalledX11` is a Boolean macro (i.e., has a `YES` or `NO` value), in contrast to `UseInstalled`, which is either defined or left undefined. This is a bit inconsistent perhaps. Maybe that's what leads to what appears to be a problem in *ibmLib.rules* and *ibmLib.tmpl*: they test `UseInstalledX11` on the basis of whether or not it's defined, rather than testing its value. Some other similar inconsistencies (which may be bugs) are:
  - `UseImports` and `ImportX11` are also Boolean (`YES`/`NO`) macros. However, *ibmLib.rules* and *ibmLib.tmpl* test each of them on the basis of whether or not they're defined, rather than testing their value.
  - *Motif.tmpl* tests `UseImports` using `#ifdef UseImports` at one point and `#if UseImports` at another.

- Since an `all::` target entry is now generated right at the beginning of *Imake.tmpl*, it's no longer necessary for the `emptytarget::` to be generated near the end of *Imake.tmpl*. The latter entry can be removed.
- The `_XUsed()` and `_XUseCat()` rules in *Imake.rules* seem to have problematic definitions. They make decisions based on the value of `UseInstalledX11`. But `UseInstalledX11` is defined in *X11.rules*, which is processed after *Imake.rules*.
- **BUG?** The leading comments in *site.def* appear to advise that the `OS{Major,Minor,Teeny}Version` macros can be changed in *site.def*, and also that they should not be changed in *site.def*.

Program Changes

imake

imake now provides a means of yanking the operating system major, minor, and teeny version numbers using the `uname()` system call. It then passes those values to *cpp* so that the `OSMajorVersion`, `OSMinorVersion`, and `OSTeenyVersion` macros can be set automatically. This is good, because you don't have to set the values in your vendor files manually. However, I think the method by which this is done could be simpler. I'll describe how it's actually done, and then how I think it could be simplified.

To use `uname()` to get the OS version numbers, you add a section to *imakemdep.h* that describes how to process the `uname()` return value on your system. For instance, the FreeBSD section looks like this:

```
/* uname -r returns "x.y[.z]-mumble", e.g. "2.1.5-RELEASE" or "2.2-0801SNAP" */
# define DEFAULT_OS_MAJOR_REV   "r %[0-9]"
# define DEFAULT_OS_MINOR_REV   "r %*d.%[0-9]"
# define DEFAULT_OS_TEENY_REV   "r %*d.%*d.%[0-9]"
```

If the `DEFAULT_OS_XXX_REV` macros are defined, *imake* calls `uname()` and pulls apart the return value using the `scanf()` patterns in the macro values. Then it writes lines like the following to *Imakefile.c*:

```
#define DefaultOSMajorVersion 2
#define DefaultOSMinorVersion 1
#define DefaultOSTeenyVersion 5
```

By themselves, these definitions are not usable. So it's necessary to modify your vendor file as well. In R6.1, the OS version numbers were set in vendor files using constructs like these:

```
#ifndef OSMajorVersion
#define OSMajorVersion      2
#endif
#ifndef OSMinorVersion
#define OSMinorVersion      1
#endif
#ifndef OSTeenyVersion
#define OSTeenyVersion      0
#endif
```

For R6.3, these must be changed as follows:

```
#ifndef OSMajorVersion
#define OSMajorVersion      DefaultOSMajorVersion
#endif
#ifndef OSMinorVersion
#define OSMinorVersion      DefaultOSMinorVersion
#endif
#ifndef OSTeenyVersion
#define OSTeenyVersion      DefaultOSTeenyVersion
#endif
```

The result is that when *imake* runs, it passes definitions of the `DefaultOSXXXVersion` macros via *Imakefile.c*, and those values are used to set the `OSXXXVersion` macros.

I think the `uname()` support is a good thing, but I suggest that the implementation could be simpler. Were *imake* to write definitions into *Imakefile.c* for the `OSXXXVersion` macros, rather than for the `Default-OSXXXVersion` macros, it would be necessary only to put the `uname()` stuff in *imakemdep.h*. It wouldn't be necessary to change the vendor files at all. That's because in R6.1 the vendor files were already set up to allow the `OSXXXVersion` macros to be overridden by prior definitions. If *imake* defined the `OSXXXVersion` macros itself, its definitions would take precedence.

This would allow the *imake* to provide override values for the version numbers, yet allow the vendor files to be left just as they were in R6.1. As it is now, there is a tight dependency between modifying *imakemdep.h* and changing the vendor file and to match.

I'm curious as to why the additional complexity introduced by the `DefaultOSXXXVersion` macros was used. Is there some benefit to requiring a change to the vendor file when *imakemdep.h* is changed?

Suggestion: Those vendor files that have been changed to use the `DefaultOSXXXVersion` macros should be "unchanged", and *imake* should define the `OSXXXVersion` macros instead. This would make extending the `uname()` support mechanism to other systems an easier and less error-prone process because fewer changes would be needed.

Another aspect of the current `uname()` mechanism is that its use of `DefaultOSXXXVersion` makes the R6.3 configuration files incompatible with all previous versions of *imake*, even recent R6.x versions. The suggested change allows the R6.3 configuration files to continue to work with any R6.x version of *imake*.

Question: How does one parse this bit of stuff from *imake.c*?

```
*   5. Start up cpp and provide it with this file.
*  Note that the define for INCLUDE_IMAKEFILE is intended for
*  use in the template file.  This implies that the imake is
*  useless unless the template file contains at least the line
*      #include INCLUDE_IMAKEFILE
```

What should the phrase "the imake is useless" really be? "The resulting Makefile is useless"?

makedepend

makedepend now properly evaluates macros that are defined as hex constants. Formerly these always evaluated as zero. This meant that in a code fragment such as the following, `CONST1` would evaluate as zero and *ar.h* would not be considered a dependency by *makedepend*:

```
#define CONST1 0x1

#if CONST1
#include <ar.h>
#endif
```