Conf-940456--5

WSRC-RP-93-1464

UNIX PROGRAMMER'S ENVIRONMENT AND CONFIGURATION CONTROL

by
P. W. Wyatt

Savannah River Site
Aiken, South Carolina 29808

A document prepared for:
Society for Computer Simulation-1994 Multiconference, SCS Multiconference Proceedings
at P. O. Box 17900, San Diego, CA 92177-7900
from 4/11/94 thru 4/15/94

DISCLAIMER

DOE Contract No. **DE-AC09-89SR18035**

This paper was prepared in connection with work done under the above contract number with the U. S. Department of Energy. By acceptance of this paper, the publisher and/or recipient acknowledges the U. S. Government's right to retain a nonexclusive, royalty-free license in and to any copyright covering this paper, along with the right to reproduce and to authorize others to reproduce all or part of the copyrighted paper.

DISCLAIMER

UNIX PROGRAMMER'S ENVIRONMENT AND CONFIGURATION CONTROL

T. R. Arnold
Mead Data Central
Dayton, Ohio

P. W. Wyatt
Westinghouse Savannah River Company
Savannah River Site
Aiken, South Carolina

ABSTRACT

A package of UNIX utilities has been developed which unites the advantages of the public domain utility "imake" and a configuration control system. The "imake" utility is portable. It allows a user to make Makefiles on a wide variety of platforms without worrying about the machine-dependent idiosyncracies of the UNIX utility "make". Makefiles are a labor-saving device for compiling and linking complicated programs, and "imake" is a labor-saving device for making Makefiles, as well as other useful software (like a program's internal dependencies on included files). This "Environment", which has been developed around "imake", allows a programmer to manage a complicated project consisting of multiple executables which may each link with multiple user-created libraries. The configuration control aspect consists of a directory hierarchy (a baseline) which is mirrored in a developer's workspace. The workspace includes a minimum of files copied from the baseline; it employs soft links into the baseline wherever possible. The utilities are a multi-tiered suite of Bourne shells to copy or check out sources, check them back in, import new sources (sources which are not in the baseline) and link them appropriately, create new low-level directories and link them, compare with the baseline, update Makefiles with minimal effort, and handle dependencies. The directory hierarchy utilizes a single source repository, which is mirrored in the baseline and in a workspace for a several platform architectures. The system was originally written by T. R. Arnold to support C code on Sun-4's and RS6000's. It has now been extended to support FORTRAN as well as C on SGI and Cray YMP platforms as well as Sun-4's and RS6000's. This means that large FORTRAN programs can be managed in much the same way that UNIX developers managed the development of X -- with order and planning in the way the code is controlled, updated, compiled, linked, and installed. It also means that Makefiles can be created and used by people who are not expert UNIX programmers.

INTRODUCTION

Development of engineering applications has often begun without a means of managing the software in an ordered, modular fashion. This is particularly true of applications that were originally coded under operating systems which do not lend themselves to a development environment. Sometimes, you may still find applications consisting of a few (perhaps one) large FORTRAN sources in a simple directory structure (or lack of structure). On a system where a user's "own" space is limited, they may not even fit into an edit buffer. This situation was later addressed with Makefiles (or their rough equivalents). The Makefile is input for the UNIX utility "make". This utility uses other UNIX executables to find sources, compile and link them, and deliver an executable somewhere. It may also embody a list of dependencies on included files. This means that when you add a new (perhaps hidden) dependency to a source, the "make" utility will know which sources need to be recompiled and will do the operation for you. You may compile, link, and deliver a complicated executable by typing "make".

So why not just use "make", which is a powerful tool, to manage code development? There are several reasons. In the first place, "make" does not recognize conditionals. So to port from one platform to another, you must edit each Makefile to suit the platform. If you have a lot of Makefiles, you must do a lot of editing. But "imake", which uses a series of data files that allow conditionals, requires little editing. Also, "make" has no flow control, and any Makefile dependencies on headers are inherently non-portable. We can limit editing to a few key points with "imake" and take most of the burden off the user. The user will have to tailor some of the "Environment" directory structure to be

mnemonically named. But that's not a big job.

UTILITY PACKAGE DESIGN

The "Environment" combines configuration control with "imake", whose data files like "Imake.tmpl", "Project.tmpl", etc., are edited extensively. There is a baseline directory featuring places for "imake" and its data files, two suites of Bourne shells layered over "imake", source locations, architecture-dependent residences for object code and executables, and delivery points for "finished" stuff. The user creates a "workspace", which is a mirror image of the baseline but with certain simplifications. The "imake" delivery and the "Environment" ".ws" and "ws" shells are not copied into the workspace. Nor is any unnecessary object code copied from the baseline to the workspace. Instead, such resources are "automatically" pointed to with soft links whenever a new list of dependencies (a ".depends" file) is created. This reduces the likelihood of error and conserves disk space. Also, Imakefiles ("imake" input) for standard directories like "lib" are copied into the workspace. The user may create other directories (for new libraries or executables) with one of the "ws" shells. Initial Makefiles are also created at this time. The user may then copy sources out of the baseline or may invent new ones. Each time new ones are invented, the Imakefile must be edited and the local Makefile must be rebuilt with "make Makefile" -- in which the make target is input for "make" itself. Then the dependencies on headers (the ".depends" file, which starts out empty) must be updated with "make depend". Typing "make" creates a new executable, and "make install" delivers it to a point where it may be run from outside the workspace. The dependencies are discovered with the ".ws_srcdepend" shell. It replaces the "imake" release's "makedepend" utility and searches for included files with the C preprocessor. The "Environment" has been modified for use with FORTRAN sources so that these may be processed with the C pre-processor just like C sources.

There are many useful utilities in the two groups of Bourne shells. Some are accessed from outside the "Environment" workspace. Others may only be used inside it. Sources may be copied or checked out from the baseline. They may be checked into the baseline by "authorized" users. The baseline itself is typically protected by limiting both ownership and write permission. There are utilities for quick comparison between the baseline and the workspace, for importing and linking blocks of new code, for identifying who has something checked out (and thereby denied to other users until it is checked back in), and for quickly showing what local files are new, or different from, or the same as the baseline. This automates a great deal of what is sometimes done administratively. What might well be more important is that it could encourage the use of mechanically generated Makefiles (and a code structure that mirrors a baseline template) among programmers who are not experts in constructing Makefiles on multiple platforms.

APPLICATIONS

Figure 1 shows a small part of a Makefile in the big window on the left. The Makefile is being managed via "imake", whose input file, or "Imakefile", is shown in the lower of the two windows on the right. The rules for creating all those "make" targets and processes in the Makefile are embodied in the "imake" templates (more data files). Some are standards which ought not to be changed much. You can dream up whatever rules you like and put them into others. The rules are identified in the Imakefile with a shorthand. So the programmer who is using "imake" may refer repeatedly to "imake" rules in many Makefiles but only edit the rule into the "imake" templates once. An engineer can manage a complicated program structure with very little effort. Most of the stuff that might be fat-fingered into oblivion is put where it is seldom in harm's way. In essence, you get the names of your sources typed in right and "imake" takes care of the rest for you. As an added bonus, the Imakefile has none of the Makefile's annoying dependence on TABs.

The "Environment" is illustrated schematically in Figure 2. The "workspace" is a directory hierarchy that mirrors some part (usually a single architecture) of the baseline. The "share" directories hold sources. Typically, "bin" holds delivered executables, "lib" holds delivered libraries, and "include" holds delivered headers that are shared among different libraries and/or executables. The lowest levels of the directory hierarchy include whatever names the user wants to assign to libraries and executables. In this example, "lib" is for libraries and the "prog[i]" are for various programs that link to the libraries. Many applications may be supported by the "Environment". Figure 3 shows some graphical results from a movie generator that is managed by the "Environment". An

application (not managed in this way since it exists as an executable purchased from a vendor) writes data to a client-server application (which IS managed by the "Environment"), which in turn feeds the X-based graphics. In fact, the directory hierarchy may be tinkered with to save anything you like -- including data and trash that really ought to be thrown away. This has the advantage of physically tying the sources to the executables so that it's harder to lose them. It also allows you to collect in an ordered way all the junk you may think you'll need some day. That's an advantage/disadvantage of having your own configuration control software, including sources.

This paper was prepared in connection with work done under Contract No. DE-AC09-89SR18035 with the U.S. Department of Energy.

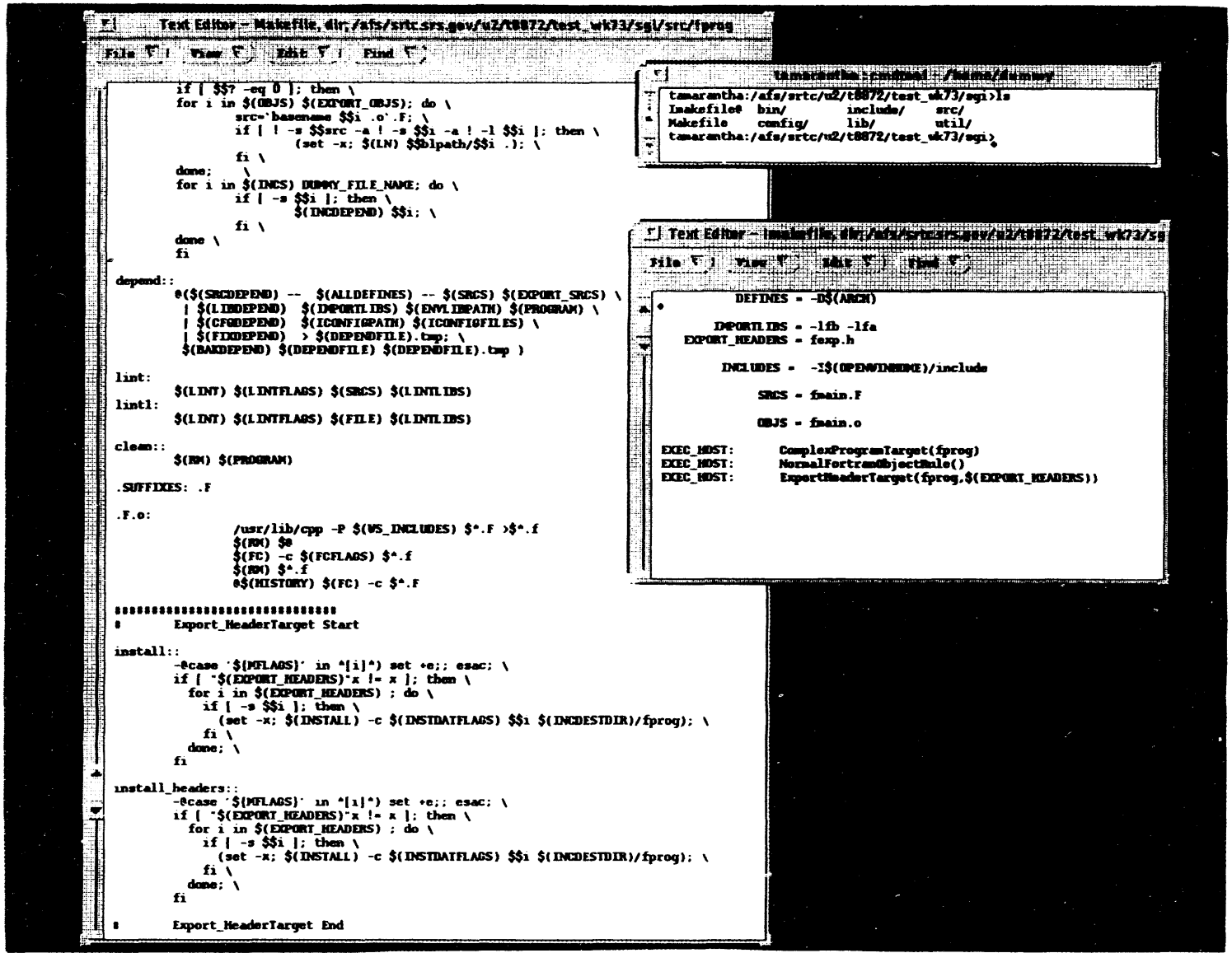

Figure 1: Makefile and Imakefile

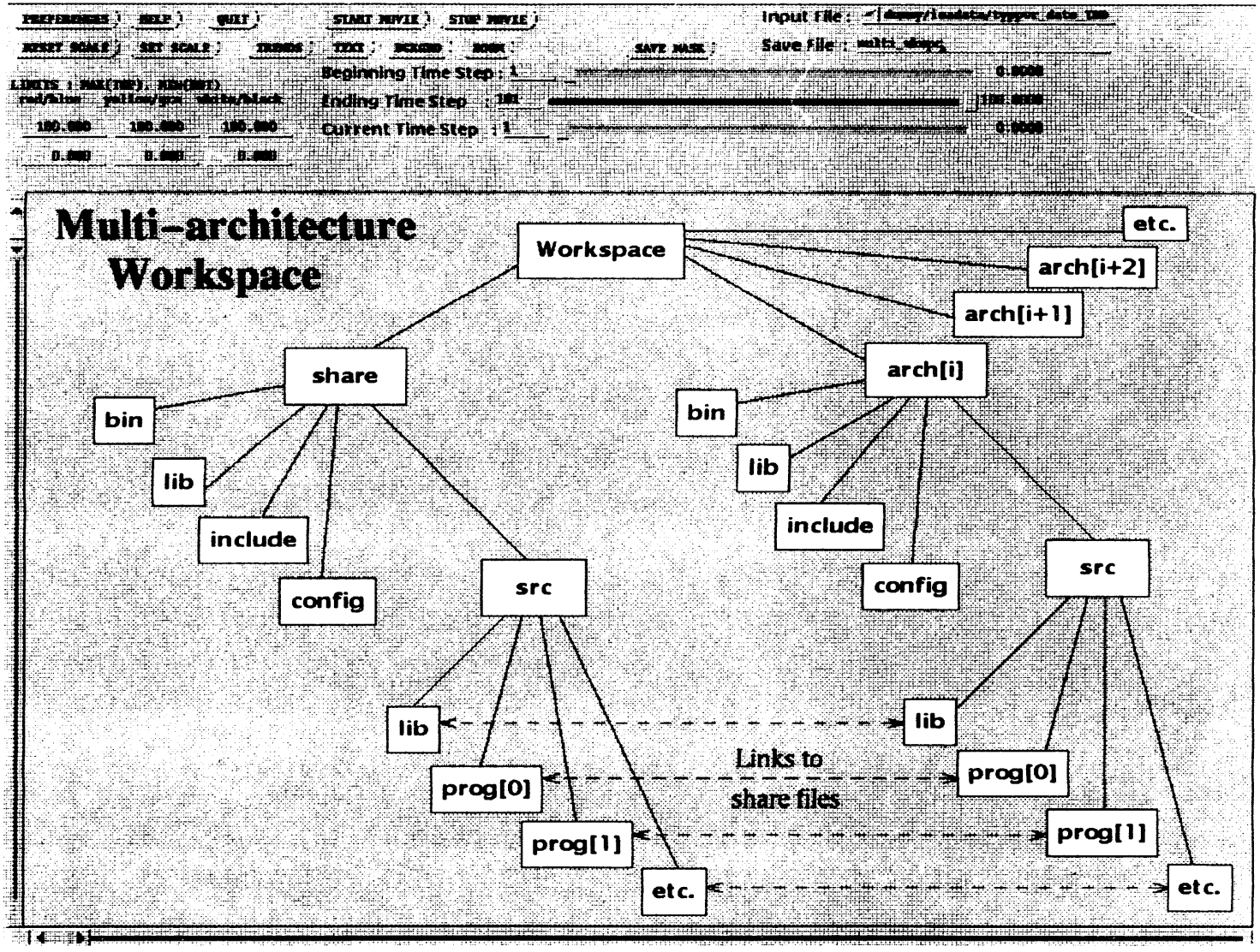

Figure 2: "Environment" Directory Hierarchy

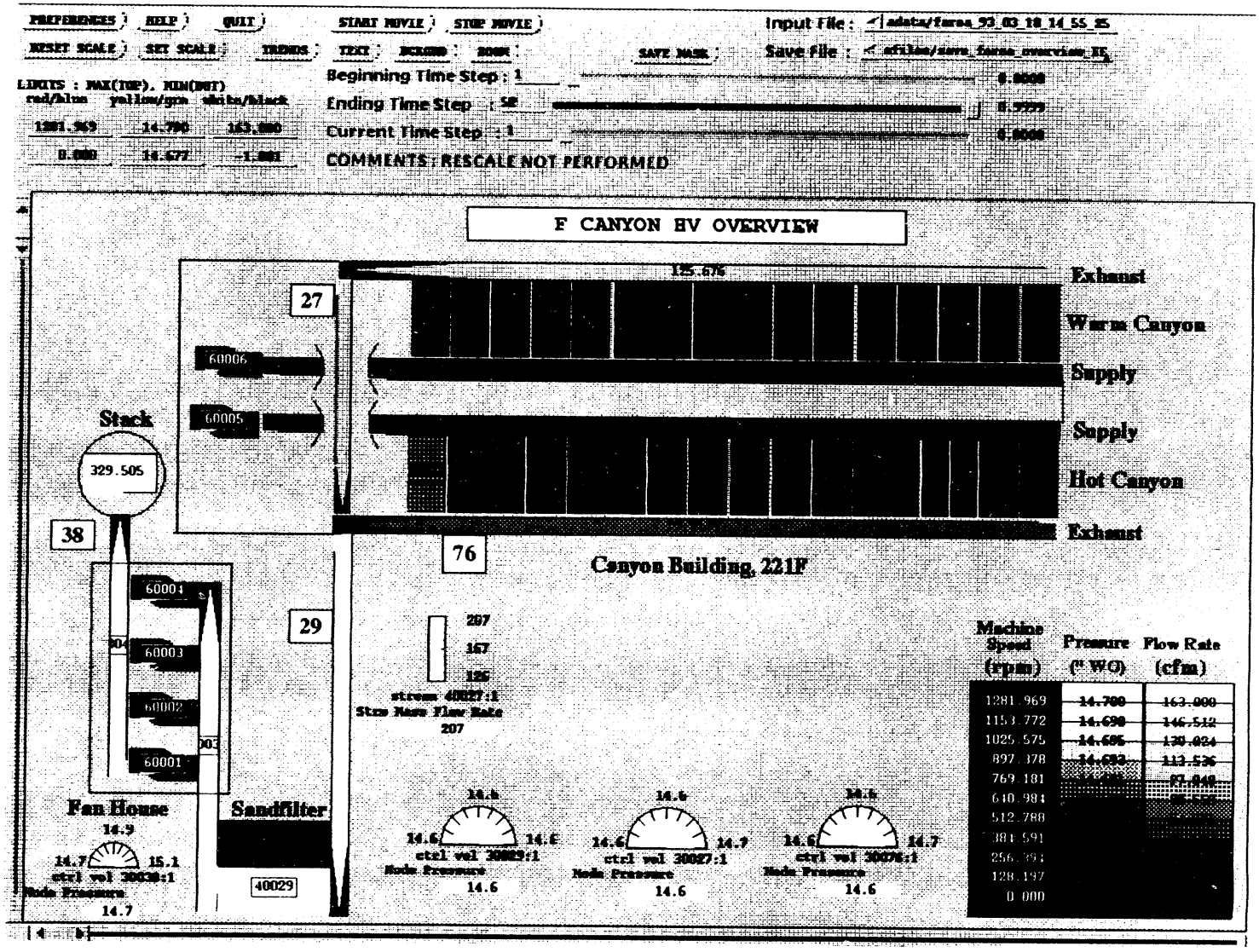

Figure 3: Graphics for a Ventilating System Model

END

DATE
FILMED
3/9/94