FERMILAB-Conf-95/325

Software Standards, Methods and Quality Control for the Sloan Digital Sky Survey

R. Pordes, E. Berman, D. Petravick, D.Holmgren, C-H. Huang,
T. Nicinski, R. Rechenmacher, G. Sergey, C. Stoughton

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

R. Lupton

Astrophysics Department, Princeton University Observatory, P.O.
Peyton Hall, Ivy Lane, Princeton, New Jersey 08544

E. Fradelos

University of Illinois at Urbana Champaign
Champaign-Urbana, Illinois

October 1995

Proceedings of the *Computing in High Energy Physics 1995 (CHEP 95),*
Rio de Janeiro, Brazil, September 18-22, 1995

Software Standards, Methods and Quality Control for the Sloan Digital Sky Survey

Ruth Pordes, Eileen Berman, Don Petravick, Donald Holmgren, Chih-Hao Huang,

Tom Nicinski, Ron Rechenmacher, Gary Sergey, Chris Stoughton*

Computing Division, Fermi National Accelerator Laboratory, P.O. Box 500,Batavia, Illinois 60510, United States of America

Robert Lupton

Astrophysics Department, Princeton University Observatory, Peyton Hall, Ivy Lane,Princeton, New Jersey 08544, United States of America

Elias Fradelos

University of Illinois at Urbana Champaign

The Sloan Digital Sky Survey (SDSS) Collaboration involves upwards of 50 scientists, many of whom are involved in the development and use of the software needed to acquire, process and archive the image and spectroscopic data sets. Fermilab has major responsibilities in the development and maintenance of the project's software and for its software engineering practices. We report on the standards and methodologies we have developed in support of these activities.

1 Introduction

The Sloan Digital Sky Survey (SDSS) is a collaboration between the Fermi National Accelerator Laboratory, the Institute for Advanced Study, the Japan Promotion Group, Johns Hopkins University, Princeton University, The United States Naval Observatory, the University of Chicago, and the University of Washington. The SDSS will produce a five-color imaging survey of 1/4 of the sky about the north galactic cap and image 10^8 Stars, 10^8 galaxies, and 10^5 Quasars. Spectra will be obtained for 10^6 galaxies and 10^5 Quasars as well. The Survey will utilize a dedicated 2.5 meter telescope at the Apache Point Observatory in New Mexico. Its imaging camera will hold 54 Charge-Coupled Devices (CCDs). The SDSS will take five years to complete, acquiring well over 12 TB of data.

At the beginning of the project we launched a concerted effort to establish a base line for standards and methodologies in support of the software development activities - the details of which were reported at CHEP in 1994. These practices were ratified by the project at large and have been used and extended significantly. Through two years of development and support we have successfully organized the development of greater than two hundred thousand lines of source code, and the integration of more than one million-

*This work is sponsored by DOE Contract no DE-AC02-76CH03000

lines before the start of data taking. We successfully orchestrate centralized integration at Fermilab of software developed over six physically separate institutions.

The Survey continues to follow the spirit of the POSIX standards, and to incorporate them to the greatest extent feasible. ANSI-C, Extended F77 and C++ are the allowed programming languages. We include much public domain software into our central code repository and support infrastructure. As reported previously, other software standards and standard products in use are: X11, WWW and HTML, perl, TeX, CVS and RCVS, Fermilab UPS, Tcl/Tk, and an Object Oriented Database for the Survey's DBMS.

In this paper we report on our experiences and the perceived benefits in using and "enforcing" these standards. We report on further developments and directions we are taking for organizing and managing the software of the Survey. In particular, we are now placing emphasis on developing tools for quality control and analysis and we present details of our tools and methods in these areas. The successful implementation and adoption of our methodologies within the collaboration gives us the confidence that new tools we develop can and will be applied to good effect across the breadth of the collaboration's software.

2 SDSS Software Standards

When constructing a large software system, it is important to place enough controls on software to ensure that it possesses basic desirable qualities. The SDSS has a number of standards for its software. However, because many tens of scientists and computing professionals contribute code to the Survey, it is difficult to ensure that the standards are actually applied or applied consistently. While every attempt was made to simplify the standards, it was also decided that tools which test for standards compliance would be useful for all involved.

Several tools were integrated into scripts and makefiles common to all software development, and are utilized each time code is compiled. For instance, the makefile driver, *sdssmake,* sets up a uniform compilation environment featuring strict compliance to ANSI C, the presence of function prototypes, and other strict checks. Another script, *filter_warn,* suppresses trivial warnings, and allows us to make it a standard that all code compile without visible warnings. An example of a trivial warning message that we supress is that ANSI C does unsigned arithmetic differently than K&R C.

Others scripts are used only at periodic intervals, such as code walk- throughs, to perform specific checks. The *quasar* script scans source code, scripts, and compiled code hunting for less likely violations of standards. For example, this script checks for non-ANSI #include files, calls to FORTRAN not made via a distinct F77 binding module, and shell scripts coded in languages other then the Survey standard /bin/sh and /usr/local/bin/perl.

3 Experiences and Results

Although the advantages of software standards are numerous and don't need to be reiterated here, we believe our decision to set and comply to rigid standards has benefited us on many occasions. Most recently, for example, we've been able to port our software to previously unsupported platforms with relative ease.

Our system of tools and scripts was developed incrementally since the start of the survey several years ago, and we continue to build upon it and learn from our experiences. We feel that tools should not just flag discrepancies and chastise the user for non-compliance. We have learned to create tools which are useful to the developer for the creation of software systems and also help create a general awareness of the standards in the collaboration.

Overall, we've found that developers have been willing to comply with the Survey standards and take advantage of the tools we have provided. Since the standards are many, even the most diligent developer cannot possibly remember all the rules all of the time. We believe the success of our software walk-throughs (which have been without serious incident) is a testiment to the willingness of the developers to comply.

As the Survey software base increases in size and complexity, so too must the quality assurance infrastructure. We continue to improve checking for standards compliance during the software build process, but we also strive to perform checks during application execution, since Survey standards apply there as well. The next section describes in detail our current effort, the **Audit** Program, which inventories the layered software components used in our system.

4 Why We Need An Audit

Compliance to many aspects of Survey software standards is checked by our existing system. However, the Survey lacked a tool that could inventory which libraries and programs were actually used by a software package. For example, the Survey uses *Tcl/Tk* rather than *Motif*, *perl* rather than *gawk*, *make* rather than *gmake* and *lex* and *yacc* rather than *bison* and *flex*. How is possible to inform developers of these standards and to assist them in choosing the standard package?

One approach to this problem is to inventory the components in a software package, and to flag non-standard items used by that package. For these purposes, the inventory is simply a list of two kinds of files:

1) The files read during the software build.

2) The files (i.e. programs) executed during the software build, along with each program's full command line and environment (in the sense of UNIX environmental variables).

Unfortunately, it is painstaking and error-prone to establish software's identity by using visual inspection; yet knowing the packages used in the software process is critical as well. One can not claim to have a portable or well-defined software system unless one has a complete inventory of components. To build such an inventory one must monitor the software build process paying attention to the following characteristics:

1) The inventory of a build must include not only its root process, which might be a process executing the *make* utility, but all of its child processes as well. For example, the *make* utility creates child processes to do the actual work of compilating and linking, etc.

2) The inventory must include all files which were read. This includes the resolution of relative path names to full path names (implying knowledge of the current working directory) and the ability to resolve symbolic links to a true path.

3) The inventory must recover the full, absolute path to each program executed, the arguments to the program and the environment for that program.

5 The Audit Program

UNIX systems provide the /proc file system to monitor most aspects of a process. The /proc file system has a file for each process. These files do not refer to a disk, but to processes' address space and other process context items. Each process on a system corresponds to a file in /proc whose name is its Process Identification Number. It is possible to open a process's file and to read and write that process's address space. It is also possible to make ioctl() calls to the process's file and monitor calls to open(), fork(), exec() and cwd() to build the configuration inventory described above.

Audit is our implementation of this kind of tool. It produces a list of all files read and exec()'ed. The syntax of audit is very simple:

audit [-f file] program [arguments-to-program]

Audit runs the program, traces the required system calls, and writes an output file listing files read and programs exec'ed. by the program and all of its children.

6 Configuration Management

For the SDSS, every file which is read must have one of the following properties if a configuration error is to be avoided:

1) The file from another Survey controlled package, built and installed according to SDSS standards.

2) The file is a directory (from a Survey controlled package)

3) The file is under the /tmp or /usr/tmp directory.

4) The file is a POSIX or ANSI-C header file.

5) The file is a shared object, in the sense of a UNIX share object file.

6) The file is a terminal device.

7) The file corresponds to certain uncontrolled infrastructure, like TeX or perl.

8) The file is on a list of operating-system and compiler-specific list of exceptions.

Every file (program) which is executed must have one of these properties if a configuration management error is to be avoided:

1) The file is under a directory controlled by the Survey's configuration management system.

2) The file name corresponds to a POSIX utility.

3) The file name corresponds to other assumed infrastructure, like TeX or perl.

3) The file name is on a list of system-specific and compiler-specific list of allowed executables.

Audit provides the complete command line arguments for each program executed. However, up till now we have not used that information. Notice that it is not necessary to parse the command line to extract file names -- all files which are used are read, and will eventually be detected.

7 Results

A first version of **audit e**xists and will be used during code walk-throughs in the November, 1995 integration of our data system. If indicated, **audit** will be extended to analyze the command line options passed to compilers to ensure that all code is compiled to Survey standards. For example, the Survey has a rule that the C-compiler will be invoked in such a way as to insist that functions have prototypes. **Audit** can be modified to make this check, by looking for the required switch on a command line.

8 References

Documentation for the SDSS can be obtained through url http://www-sdss.fnal.gov:8000/ and links to collaborating institutions from this www server.