The Source Code Control System

MARC J. ROCHKIND

Abstract-The Source Code Control System (SCCS) is a software tool designed to help programming projects control changes to source code. It provides facilities for storing, updating, and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change, when and where it was made, and why. This paper discusses the SCCS approach to source code control, shows how it is used and explains how it is implemented.

Index Terms-Configuration management, program maintenance, software control, software project management.

I. INTRODUCTION

COMPUTER programs are always changing. There are bugs to fix, enhancements to add, and optimizations to make. There is not only the current version to change, but also last year's version (which is still supported) and next year's version (which almost runs). Besides the problems whose solutions required the changes in the first place, the fact of the changes themselves creates additional problems. The most serious are the following.

1) The amount of space to store the source code (whether on disk, tape or cards) may be several times that needed for any particular version. For example, there might be "customer," "system test," and "development" source libraries, with most modules represented by a different version in each.[1]

2) Fixes made to one version of a module sometimes fail to get made to other versions.

3) When changes occur it is difficult to tell exactly what changed and when.

4) When a customer has a problem it is hard to figure out what version he has.

The Source Code Control System (SCCS) attempts to solve these problems by an approach which treats each module as a set of related sequences of source code, each member of which represents one version of the module. There are as many different versions in the set as there were changes to the module since its original coding. The key features of SCCS fall into these categories.

Storage: All versions of a module are stored together in the same file. Source code common to more than one version is not duplicated. All versions are accessible.

Protection: A programmer may be restricted to updating only designated modules, and only designated versions of those. The only access to a module is through SCCS.

Manuscript received August 5, 1975.

The author is with Bell Laboratories, Murray Hill, N. J. 07974.

[1] The term *module* refers to a convenient unit of source code, usually a subroutine or macro.

Identification: The system automatically stamps load modules with information such as version number, date, time, etc. The source code that was used to make the load module may later be retrieved from this information alone.

Documentation: The system automatically records who made each change, what it was, where it was made, when it was made and why.

There are two implementations of SCCS: one for the IBM 370 under the OS and one for the PDP 11 under UNIX [1]. Most of this paper applies to both implementations, but where a difference is relevant it is noted. More detailed comments about the implementations are in Section IX.

II. MODEL AND NOMENCLATURE

At the heart of SCCS is its technique for storing the changes made to a module. This technique is based on the following model.

Each time the module is changed (each change usually corresponds to one editor session) the change is stored as a discrete *delta.* Conceptually, the deltas resulting from a series of changes are strung together in a chain. Fig. 1 shows a module which has been changed three times. For simplicity, the original module is also shown as a delta. The source code of the module is accessible at each of the four points at which deltas were added. To produce the latest version, SCCS follows the chain from the beginning, *applying* deltas as it goes. Each delta is applied to the source code as it existed just prior to that delta. Similarly, the source code as it was just before the last change is accessed by applying only the first three deltas, and so on.

It is important to note at this point that Fig. 1 is only a model; how the module is physically stored and how the deltas are actually applied will be described in Section VIII.

When a new module is coded, it is said to be at *release* 1. Each delta represents a new *level.* Deltas are named by their release and level numbers. In Fig. 1, the first delta represents release 1, level 1, the second represents release 1, level 2, etc. Usually (but not necessarily), the first few deltas correspond to the initial changes that are made, to new modules: correcting syntax errors and bugs found by "unit testing."

Let us continue this example by assuming that the module of Fig. 1 is now turned over to a system-test group. This group will be testing release 1. The programmer then begins working on enhancements which will be incorporated into release 2. Traditionally, the programmer

Fig. 1.

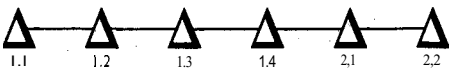

Fig. 2.

makes a copy of the module at this stage and begins modifying that copy. However, with sees the programmer just adds more deltas to the end of the chain, specifying that they belong to release 2. In Fig. 2 two deltas have been addedtb release 2. Although the programmer has added these new deltas to the same file that the system-test group is using, the release 2 deltas cannot change the Source code under system test. The reason is simply that when the system-test group wants the source code they request it at release 1, causing sees to stopapplying deltas with delta 1.4.

Further development on the module follows the same pattern. Eventually release 1 will be distributed to customerS, release 2 will undergo system test, and the programmer will begin work on enhancements for release 3. Note that the deltas have stayed put, but the names (e.g., "development," "system test," and "custom6r") for the releases have moved. Contrast this with the usual situation in which the names for the source code libraries stay the same but the modules are moved about.

The particular delta chain used in our example applies to only this one module. In general, each module of a software system will have a different number of deltas per release.

Of course, things rarely work out as smoothlv as we have described them. Suppose that after the programmer began addIng release 2 deltas a bug is discovered during system test. This bug cannot be fixed by adding a delta to the end of the chain, because that would make it a release 2 delta, and the system-test' group is accessing the module at release 1. Clearly, the delta (or deltas) needed to fix'the bug must go in the middle of the chain, between deltas 1.4 and 2.1. To do this the progr'ammer just specifies release 1 when he makes the delta (the process of making a delta will be described in Section VII). Fig. 3 shows the delta chain with delta 1.5 'added.

When the module is next accessed at release 1, delta 1.5 will be applied. However, when the module is next accessed at release 2, delta 1.5 'will not be applied. If sees were to apply it and then apply deltas 2.1 and 2.2, the result could be disastrous.[2] A warning message is issued when a delta like 1.5 is skipped, to ensure that it is riot forgotten'in the case where the bug it fixed is also present in newer (higher numbered) releases.

Deltas may only be added at' the *end* of a release; the system would not permit a delta to be inserted'between deltas 1.2 aud1.3, for example. In practice, the point at which new releases are begun is determined both by the implications of this constraint and by the requirements for protection, which will be discussed in Section IV.

[2] Actually, in earlier versions of sees deltas like 1.5 *were* applied to release 2. ThIS was .orIgima.lly thought to be desirable, but experience showed' otheI'Wlse.

The allowable number of releases and number of levels per release are, for all practical purposes, unlimited (255 and 32 767).

Additional flexibility in controlling the effect of deltas is provided by two kinds of special deltas. The first is *optional deltas.* Optional deltas are like normal deltas in. all respects, except that when added they are associated with an arbitrary *option letter*. Optional deltas are only applied if their associated option letter is specified by the user. The original intention was that option letters would be assigned to specific customers, and that optional deltas would be used to install "temporary fixes" appropriate only for one customer, with the idea that such fixes would be incorporated into the standard product in the next release. Optional deltas can be, and have been, used for other, similar purposes.

The second kind of special delta is one which, when applied, explicitly forces other deltas to be applied or not, by either *including* or *excluding* them. A list of deltas to be included' or excluded is specified when such a delta is created. The exclusion facility is most often used simply to correct mistakes. For example if, after delta 3.14 is added, it is found to be undesirable, the programmer might add delta 3.15 which excludes it. If the module is accessed at level 3.14, delta 3.14 itself would be applied. If the module is accessed at level 3.15, though, delta 3.14 would not be applied. From the viewpoint of control, this form of error correction is safer than allowing the programmer to actually delete a delta, since no potentially necessary information is lost.

The inclusion facility is most often used to either make optional deltas effectively nonoptional, or else to force a delta which has been added in the middle of the delta chain (such as delta 1.5 in Fig. 3) to be applied in higher numbered releases.

A delta which includes and/or excludes other deltas may be optional. Additionally, a (possibly optional) delta which includes and/or excludes other deltas may in turn be included or excluded by some other delta. If one delta includes a delta, and another delta excludes, that same delta, the chronologically newer of the two including/excluding deltas has precedence.

III. IDENTIFICATION

The purpose of the sees identification facilities is to permit the correct version of the source code to be determined from information associated with a load module. With sees this information is especially useful because the system can regenerate the correct version of source code from it. In general, it is best to place the. identification within the source code in a way that will cause the information to .ap'pear in the load module also. For example, a, PL/I program might be identified with a

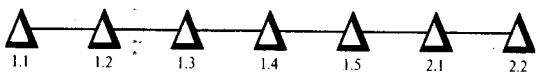

Fig. 3.

variable initialization, like this:

```
• DCL REL_LEV CHAR(6) INIT('5.31');
```

This is sufficient to ensure that the string "5.31" appears someplace in the object code; the only problem is finding it when it is needed. To help, utility programs have been written for each computer system with which SCCS is used to find the identification information within a load module automatically.

Rather than impose a particular content and form for identification information, SCCS instead provides a general facility that allows projects to adopt their own schemes. This approach also avoids the problem of providing separate support for each of the numerous languages used at Bell Laboratories. Essentially, identification is provided by a set of predefined, parameterless macros, called *identification keywords.* Each keyword stands for one piece of identifying infonnation. For example, R stands for the release, L stands for the level, D stands for the date, etc. In all there are about a dozen keywords. When SCCS generates a version of a module, it replaces every appearance of an identification keyword within control characters (% by default) by the identification datum appropriate to that generation. For example, if the stored source code for a module contains

```
DCL ID CHAR(13) INIT('%R%.%L% %D%');
```

and the user requests level 2.16 on Mar. 17, 1975, the same line in the generated source code would appear like this:

```
DCL ID CHAR(13) INIT('2.16 750317');
```

The particular set of identification keywords used depends on the needs of the project using SCCS. The most common set is keywords representing release, level, option letter, date and time. Actually, no matter how complex the delta chain, the release, level, and option letter alone are sufficient to identify a version.

SCCS also has a facility which allows identification keywords to be used in conjunction with IBM *identification records* [2]. An identification record is an area of a load module specifically designed for identification information.

IV. PROTECTION

The goal of protection with SCCS is not so much to prevent sabotage as to ensure that there is no confusion about what the management of a project does and does not want programmers to change. Very clever programmers can compromise the protection mechanism; however, conscientious programmers need only be reminded when they are trying to change something they should not.

The SCCS protection facilities are oriented towards limiting what modules a programmer can add deltas to and in which releases. More basic protection, against, for example, "zapping" an SCCS module to change the meaning of a delta, is provided by the operating system under which SCCS runs.

To be completely flexible, each module would have to be associated with a bit matrix with programmer's names along one dimension and releases along the other.[3] Neither SCCS implementation goes this far; instead they treat each dimension separately. Both implementations keep a list of *locked* releases for each module. No one can add a delta to a locked release. For the programmer dimension, the PDP 11 implementation keeps a list of programmer's names for each module. The IBM 370 implementation does not limit updating by programmer at all. It does, however, maintain *release passwords* in addition to the release locks. A programmer must supply the proper password in order to add a delta to a password-protected release.

On another level, both implementations require an *administrator's password* for certain functions, such as setting the locks and passwords, changing the list of programmers permitted to add deltas, creating or deleting modules, and so on.

The degree of protection employed depends on the project using SCCS. Often when individuals use it for their personal programs no protection at all is set up: the individual is his own administrator, and he can add deltas to any release of any module.

V. DOCUMENTATION

As soon as something goes wrong with a program, the first question an experienced programmer asks is "What changed?" This is usually a difficult question to answer. Often the answer is not found until the bug which caused the problem is itself found. SCCS provides an easy answer to this question, and having the answer at the outset usually helps in finding the bug.

For each delta, SCCS automatically records who added it and when it was added (date and time to the nearest second). Where the change was made, that is, to what source lines, and what the change actually was is recorded by the nature of the deltas themselves. (The technique actually used to store this information will be explained in Section VIII.) The reason for the delta is not recorded automatically; it must be supplied by the programmer adding the delta, but it is required. The quality of the reason (like the quality of the change itself) depends on the conscientiousness of the programmer. Reasons like "Trouble Report 5576; change SUM header" are what one likes to see. Sometimes, unfortunately, one sees instead things like "Another bug" or "Tried again."

SCCS incorporates the delta documentation into several reports. The most common is issued whenever a module is accessed. It is a chart giving, for each delta, the release, level, option letter (if any), date and time of

[3] Of course, the information would not actually have to be stored in matrix form; it could be compressed in a variety of ways.

creation, who added it, why it was added, whether it was applied in this particular access, and why it was or was not applied (for example, if it was optional, included or excluded).

VI. ACCESSING MODES

This and the next section are intended to illustrate how SCCS is actually used. There are, of course, hundreds of details which will not be described here. Hopefully, though, we can give some idea of what the system is like.

A user communicates with SCCS through a set of commands. The set used in the PDP 11 implementation will be shown here. The simplest and most common way in which modules are accessed is by name alone:

```
get modx
```

causes SCCS to generate the source code corresponding to the very end of the delta chain. If Fig. 3 represents the chain for module "modx," then this command would produce the source code corresponding to level 2.2. The generated source code is written into an auxiliary file named "modx.a."

If the source code is needed at some other release, the desired release is specified with the r parameter:

```
get modx - r1
```

would be used to access module "modx" at the highest level of release 1 (1.5 in this case). A level number may also be given:

```
get modx - r1.3
```

accesses "modx" at level 1.3.

An option letter may be specified with the *o* parameter:

```
get modx - r3.11 - oX
```

requests "modx" at level 3.11 with optional deltas associated with letter "X" applied.

Finally, modules may be accessed with a *cutoff date* given by the c parameter:

```
get modx - r3.4 - c7502
```

accesses "modx" at level 3.4, but without the application of any deltas newer than the last day of February, 1975.

VII. MAKING A DELTA

With the PDP 11 implementation, making a delta is a three-step procedure. The first step is to obtain a copy of the module to be changed. This is done with the *get* command:

```
get modx - r2 - e
```

The e parameter specifies that this access is for the purpose of making a delta. Among other things, it suppresses the substitution of identification keywords by particular values. Essentially, in this step the programmer is saying, "I want to change module 'modx'. Give me a copy to mark up."

In the second step, the programmer "marks up" the file named "modx.a" generated by the *get* command with the UNIX editor, an interactive context editor styled after QED [3]. If necessary, this step may span several days, or even weeks, and may involve several editor sessions and compilations.

Whenever the programmer is ready, he takes the third step, which is the adding of the delta to the chain. The command

```
delta modx
```

performs a heuristic comparison between the file originally generated in the first step (which is regenerated for the comparison) and the file as modified by the programmer. The changes found are incorporated into a delta, in a way to be described in the next section.

Between steps one *(get)* and three *(delta)* any other attempts to add a delta to the same module are locked out. However, access for read-only purposes is allowed.

VIII. DELTA STORAGE AND ACCESSING ALGORITHM

The attempt by SCCS to record every version of every module that ever existed is rather ambitious. The system would be impractical unless it used a storage technique and accessing algorithm that allowed many deltas to be kept at a reasonable cost in terms of disk space and processing time. SCCS meets both of these criteria. The space required to store a delta is only slightly greater than the amount of text inserted by that delta. The accessing algorithm allows any level to be reached in essentially equal time, by applying deltas in parallel during a single pass over the file containing the module.

All changes are stored in terms of two primitives: *insertion* of an entire line and *deletion* of an entire line. This means, for example, that the replacement of a single character on a line is stored as a deletion of the entire old line and an insertion of an entire new line. Also, a movement of a block of lines from one point in the module to another is stored as a deletion of the block at the old location and an insertion of an identical block at the new location. By using only these two primitives information is lost. However, experience has demonstrated that this information is not really essential; it is sufficient that SCCS be able to reproduce the required versions accurately.

Each module is stored in a separate sequential file. The part of the file where the actual deltas are kept is called the *body.* The body consists of *text records,* containing source code inserted by deltas, and *control records,* which specify the effects of each delta. The relationship between text and control records is best explained by showing how a new delta is added to an existing module.

The typical source code change to be incorporated into a delta consists of several insertions and deletions at different points in the module. Where an insertion is to be made, the new source code is physically inserted at the appropriate place, as a sequence of text records, one per source line. The new text is bracketed with two control

records. The first, placed just before the start of the text, is called an *insertion control record.* It consists of a code indicating insertion and the release and level numbers of the new delta. The text is followed by an *end control record,* which consists of a code indicating end and, again, the release and level numbers (these are needed because the various bracketing control records do not necessarily nest). The control-record brackets are used during the accessing process to delimit text to be either kept or skipped, depending on whether the delta is to be applied or not.

Text to be deleted by the new delta is also bracketed with control records. The opening bracket is a *deletion control record,* consisting of a code indicating deletion and the release and level numbers. The closing bracket is an end control record. Hence, there are. three types of control records in all. Fig. 4 shows the body part of a typical file. The letters I, D, and E represent codes for insertion, deletion, and end control records, respectively.

The source code corresponding to a particular level is generated in the following way. The body is scanned sequentially record by record. When a text record is encountered, it is either kept or skipped depending on the setting of a toggle called the *keep switch.* A yes setting indicates that the record is to be kept, and a no setting indicates that it is to be skipped. Wheri an insertion or deletion control record is encountered, an entry is made into a linked list called the *control queue.* The entry COntains the release and level numbers of the delta, the date and time of the delta, and a flag field (to be explained shortly). The control queue is maintained in chronological order, with the entry corresponding to the newest delta at the head. When an end control record is encountered the matching control queue entry is removed from the list.

Note that there will never be two entries in the control queue which correspond to the same delta.

The flag field of a control queue entry has one of three values: yes, no or null. Whenever the control queue is changed, the keep switch is reset according to the flag field of the entry at the head of the control queue, if the flag value is yes or no. If the value is null, the next entry is examined to see if its flag is yes or no. The search continues until a value other than null is found. The search will always be successful.

A flag field value depends on whether the corresponding control record is an insertion or deletion, and whether the corresponding delta is to be applied or not. If an insertion is to be applied, the value is yes. If an insertion is not to be applied, or if a deletion is to be applied, the value is no. If a deletion is not to be applied, the value is null.

The body part of a file is preceded by three other parts called the *header,* the *release table,* and the *delta table.* The header contains assorted information associated with the module, such as the release locks, the list of programmers authorized to add deltas, the language the moduie is written in, an English description of the module, and

I 1.1
1 1.4
text of 1.4
EIA
text of 1./
D 1.2
more text of J, I
E 1.2
I 1.2
text of 1.2
D 1.3
more text of 1.2
E 1.2
more text of /.1
E 1.3
more text of 1. J
E 1.1

Fig. 4.

other similar" information. SOme of this information is not used by SCCS directly, but is recorded only to help document the structure of the software project.

The release table consists of just a count of the number of deltas in each release. Its only purpose is to enable SCCS to configure certain internal storage structures in preparation for processing the body.

Finally, the delta table contains all the information associated with each delta which is required to decide whether, in a particular access, that delta is to be applied or not. This required information is the release number, the level number, the option letter (if any) and the date and time the delta was added. Each delta table e,ntryalso contains a list of other deltas included or excluded by that delta, if any, and information'about who made the delta and why (as described in Section V). Prior to the execution of the accessing algorithm itself, the decision as to which deltas are to be applied is made based on the information about each delta in the delta table and the release, level, optioll letter, and cutoff date specified on the *get* command. These decisions are stored in an internal array indexed by release and level number. Later, while processing the body, each control record encountered is looked up in this array to find out if the corresponding delta is to be applied.

The time required to access a given level of a module depends most strongly on the total number of records (control and text) in the body; It depends less strongly on the amount of activity on the control queue, which is determined by the total number of control records. It depends hardly at all on the number of deltas. Therefore, if, say, 17 insertions have to be made, it makes no difference. to the accessing time whether they are made in one delta or 17.

IX. IMPLEMENTATION AND HISTORY

Because SCCS represented such a radical departure from conventional methods for controlling source code, it became clear when we began development of it (in late

1972) that a paper specification would not be sufficient to "sell" the system to the software projects for which it was intended; we would have to have a working prototype. It was essential that this prototype be available quickly for trial by the projects; it was also important that it be reliable from the start, so that sees would not have to begin life with a bad reputation to overcome. We decided to implement sees on the IBM.370, underOS/MVT, and, to code the system in SNOBOL 4 [4J, using the SPITBOL compiler [5]. We chose SNOBOL 4 because of its power, which would allow us to spend a minimum amount of time on coding details, particularly those relating to dynamic storage management, and because of its debugging facilities, especially symbolic tracing and dumping. Experience has justified our choice. The first release of sees was coded, debugged, and tested by one person in less than three months.

In most cases, a penalty for using SNOBOL 4, even when compiled with SPITBOL, is that the resulting program is bigger[4] and slower than it would be if coded (with considerably more effort, of course) in a lower level language, such as, say, BLISS [6J or PL 360 [7]. These factors were irrelevant during the six-month trial period following completion of the prototype, in which several large applications experimented with the system. When one project decided to adopt sees, the efficiency of the system became very relevant, but it was efficient enough so that they simply continued to use the prototype. Since this project began using SCCS many enhancements have been made to it, but it is still basically the same SNOBOL 4 program.

In the fall of 1973, with several additional projects eager to use SCCS, we decided that, rather than continue to enhance the IBM 370 implementation, or to recode it entireiy, we would take an entirely different, much more ambitious approach. This approach was prompted by two problems. First, a very large project wanted to use sees on a Univac 1110, and it looked like we would have to implement sees on that machine. The second problem was that, although our IBM sees users were satisfied, we felt that sees should provide interactive text editing, which would require a major redesign of the IBM implementation. We found a novel solution for both these problems: we would design a new SCCS which would run on neither the Univac nor the IBM, but on a PDP 11/45 under UNIX. Source code would be stored, via sees, on the PDP 11, rather than on the Univac or IBM machines. To compile the source programs, jobs, containing the source code along with appropriate job control language; would be sent to either the Univac or the IBM. In fact, we went beyond the original,limited, goals of SCCS and set out to design a complete project design, development and maintenance facility on the PDP 11/45 which we called the "Programmer's Workbench." sees is only a part of this facility; other components include remote job-entry to both Univac and IBM, project documentation tools,[5] a trouble reporting system, and a load and regression testing facility for IBM IMS [8J "and Univac BIeSlloo [9J projects.

With respect to the Programmer's Workbench version of sees, the term *"source code* control system" is actually a misnomer; sees is used to control changes to documents, such as user manuals or program logic manuals, as well as to source code. The system might better be called a *"text* control system."

The Programmer's Workbench has proven to be very popular with both management and programmers, and is now used by almost all software projects at the author's installation. The current users of the original IBM sees implementation plan to switch to the Programmer's Workbench in 1976, at which time the "prototype" will finally be abandoned.

X. STATISTICS

In order to give an idea of the size of the projects using sees, we present some statistics gathered from the largest user of the IBM 370 implementation. This project has about 100 programmers. They have been using sees for about two years, although conversion of their subsystems to sees occurred gradually during their first year of use. When our statistics-gathering program was run, it reported the following figures: 2964 modules, 14 455 deltas, and 1 016 766 total records (including hot.h text and control records in the body and the header, release table and delta table). This works out to an average of about 5 deltas per module. However, 1209, or 40 percent, of the modules had only one delta; i.e., they had been placed under sees control but had not yet been edited with sees. For the modules which had been edited with sees, the average is 7.5 deltas per module. Also, 126 modules had more than 25 deltas.

When all of the modules were ,accessed at their latest level, they totaled 740719 lines. This number may be taken as the minimum number of lines which must be kept, assuming that only the latest version is needed. At an additional space cost of 37 percent, sees not only keeps the necessary versions (one each for customer, system test, and development), but also can regenerate any module at any point since it was placed under sees control, as well as maintain a complete history of the changes to the project's software.

ACKNOWLEDGMENT

The author wishes to thank D. A. Nowitz who worked with him on the original design of sees. The idea of having deltas came from IBM's CLEAR [10J, although, for the most part, similarities between the two systems stop there. Even as far as deltas are concerned, in CLEAR there tend to be relatively few deltas which, once added, may be changed. In sees deltas may not be changed once

[4] At our installation the size penalty is not too severe because we now use OS/VS2.

[5] Which were used to prepare this paper.

added, although the effects of them may be altered by adding more deltas. At various times, S. F. Coppage, S. T. Feczko, and A. L. Glasser worked with the author on the design and coding of enhancements to the IBM 370 implementation. The idea of the Programmer's Workbench came mostly from E. L. Ivie, and a dozen or so people have worked on various parts of it; the author was responsible only for the SCCS component.

REFERENCES

[1] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. Ass. Comput.Mach.,* vol. 17, pp. 365–375, July 1974.
[2] *OS/VS Linkage Editor and Loader,* IBM, Form GC26-3813.
[3] L. P. Deutsch and B. W. Lampson, "An online editor," *Commun. Ass. Comput. Mach.,* vol. 10, pp. 793-799, 803, Dec. 1967.
[4] R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language,* 2nd ed. Englewood Cliffs, N. J.: Prentice-Hall, 1971.
[5] R. B. K. Dewar, *SPITBOL Manual, Version 2.0,* Illinois Inst. of Technol., Chicago, Ill., Feb. 1971.
[6] W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A language for systems programming," *Commun. Ass. Comput. Mach.,* vol. 14, pp. 780-790, Dec. 1971.
[7] N. Wirth, "A programming language for the 360 computers," *J. Ass. Comput. Mach.,* vol. 15, pp. 37-74, Jan. 1968.
[8] *Information Management System Virtual Storage (IMS/VS) General Information Manual,* IBM, Form GH2Q-1260.
[9] *BICS1100 System Information Manual,* Univac, Form NA8300.
[10] H. M. Brown, "Presentation on Clear," *Software Engineering Techniques,* (Proc. of conf. sponsored by NATO Sci. Committee) Apr. 1970 (available from Scientific Affairs Division, NATO, Brussels, Belgium).

Marc J. Rochkind was born in Baltimore, Md., on June 12, 1948. He received the B.S.M.E. degree from the University of Maryland, College Park, and the M.S.M.E. degree from Rutgers University, New Brunswick, N.J., in 1970 and 1972, respectively.

Since 1970 he has been a member of the Technical Staff at Bell Laboratories, Murray Hill, N.J. He is currently engaged in the development of software systems for use by Bell Telephone companies.

Program Design by a Multidisciplinary Team

SUSAN VOIGT

Abstract-The use of software engineering aids in the design of a structural finite-element analysis computer program for the CDC STAR-IOO computer is described. Since members of the design team came from diverse backgrounds, both the unique features of the CDC STAR computer and structural analysis concepts and computing requirements had to be understood before design began.

Nested functional diagrams to aid in communication among team members were used, and a standardized specification format to describe modules designed by various members was adopted. This is a report of work in progress where use of the functional diagrams provided continuity and helped resolve some of the problems arising in this long-running part-time project.

Index Terms-Modularity, program design, program specifications, software engineering, structural finite-element analysis, top-down design.

INTRODUCTION

GENERALLY, a computer program begins as an idea in the head of a potential user. The user enlists the aid of a software analyst to help design a program, and a program development project is born. Over the years, experienced programmers and analysts have developed techniques which are useful in designing programs. For example, a flow chart is one way of diagramming the sequence of operations to be performed in a program. Flow charts are a convenient way for assembly language programmers to gain insight into program logic. Since the advent of higher level languages, flow charts are no longer essential to sequence the logical steps of a program. For example, experienced programmers think in Fortran and, consequently, can write a program directly without first diagramming the logic with a flow chart. Flow charts, therefore, often have become an "after the fact" means of documentation and have ceased to be an aid to program design. In many cases, functional flow charts describing the general logic flow in a program are developed as part of a program's documentation after implementation of the program.

For most large program developments, more than one program designer is involved, more than one user must be satisfied, and usually other programmers implement the design. In large program developments program

Manuscript received August 5, 1975.
The author is with NASA Langley Research Center, Hampton, Va. 23665.