主動式的軟體維護方法
AN ACTIVE APPROACH TO SOFTWARE MAINTENANCE

朱正忠 張志宏 楊宏戟
William C. Chu Chih-Hung Chang Hongji Yang*

逢甲大學資訊工程研究所
Department of Information Engineering
Feng Chia University
Taichung, Taiwan, R.O.C.
chu@fcu.edu.tw

*Computer Science Department
De Monfort University
England
*hjy@dmu.ac.uk

摘要

在本篇論文中，我們將提出一套主動式的軟體維護方法，以一連串優良的改善步驟，以改善軟體的可維護性。透過正規化的推論方法來改寫現存軟體，使得程式中正確的部分能被保存下來，而隱含在程式中不易被發現的錯誤能被找尋出來。同時，程式中語意和法法上的錯誤能提早被發現，減少軟體的大小，撰寫的方式能夠統一，間接降低軟體的花費。

關鍵字: 軟體維護, 轉換, 推論, 可維護性

Abstract

We propose an active approach which is a continued quality improvement process to improve software maintainability. By using a formal inference methodology to re-write existing software, the correctness of program can be preserved and many imbedded errors can be revealed. Meanwhile, the semantically unreachable statements and possible incorrect semantics can be revealed, the size of software be reduced, writing style be unified, and the total software cost be indirectly reduced.

Keywords: Software Maintenance, Transformation, Inference, Maintainability

1. INTRODUCTION

Although software engineering technologies have been used to improve the design of software systems, in real world, most of software systems are still underdeveloped. Also, on-going program maintenance, e.g. patches and newly added code, can result in even worse software systems. Most existing software systems are very difficult to understand and maintain due to their improper representations or designs. Studies show that nearly 80% of all software resources are allocated to improving existing systems [1].

Software maintenance activities include error corrections, enhancements of capabilities, porting, and software transformation. Traditionally, software maintenance activities were requested when problems happened. Most of maintenance approaches are usually designed for solving one maintenance problem at a time. Maintainers are discouraged to improve software as a whole due to the high cost and complexity of software systems. As a result, software size is getting bigger and bigger and quality is getting worse and worse. Traditional maintenance activities have the following characteristics, we call it *passive* maintenance.

- The system is diffuse. Parts and pieces are scattered about, and little in the way of or ganization is visible. Whatever organization once existed is in the process of breaking down.
- An inventory of the system is either lacking or is too lengthy. In either case, it is al most impossible for anyone to overview the system and, thus, to police it.
- No one is responsible for policing the system anyway.
- The volume of the system always increases, never decreases.
- Work on the system is driven purely by customer demand. As in war, periods of boring nonactivity are interspersed with periods of frantic activity.
- During the frantic times, the system plays the role of enemy. It resists change to the utmost. Nothing can be done without ten attendant complications.
- Inconsistencies between original systems premises and current requirements develop which cannot be resolved because the original system cannot be modified safely.

As a result, software size is getting bigger and bigger and quality is getting worse and worse. Without proper management, very likely, fixing one maintenance problem may introduce more maintenance problems later. In this paper, we propose an *active* maintenance approach, which has the following characteristics:

- It is carefully and thoroughly organized.
- Excellent means of overview have been provided.
- Organization is scrupulously maintained, implying that the system is supervised.
- Obsolete components are ruthlessly tracked down and eliminated.

- A policy of self-initiated internal enhancement is followed under which any component of the system, including programs, can be modified.

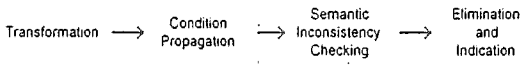
Active maintenance can be viewed as a sort of continuous, low-key recovery project. Conversely, a recovery project can be thought of as a concentrated dose of very active maintenance. At either end of the scale of activity, the aim is the same.

A detailed description of our active maintenance approach is given in section 2. In section 3, some experimental results will demonstrate the feasibility of our approach. In section 4, will compare our approach to other research in this field. In the final section, we will summarize our research.

2.ACTIVE MAINTENANCE

Behind the "maintenance iceberg" lies the reality of incomplete system development, that is, a huge, mostly unsuspected potential for continuing the development of the average system. Active Maintenance (AM) is a treatment that aims at completing this development. We use a technique known as *Inferential Systems Analysis* (ISA), which stipulates that the system in question already be known to work, at least to some degree. This stipulation is unavoidable because ISA attempts to improve a system through a process of revision that preserve its functionality, much as the process of revising a rough draft preserves its meaning. Since the revision only repeats what the original copy says, if what a system says is untrue-- that is, if the system does not work correctly -- then neither will its recovered image.

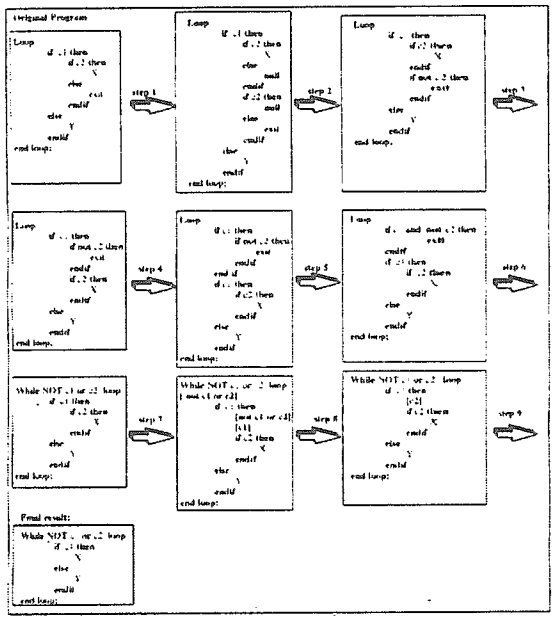
Of course, AM would be nearly worthless if it could only be performed on systems that already well designed and ran perfectly, or if it is required that every bug in a system be perpetuated like a fly in amber. Fortunately, neither is the case. Rather we must be a



little loose and define a "working system" as one that "mostly" works. Nevertheless, with or without bugs, a working system is the only kind that can be analyzed.

2.1 Inferential Software Analysis (ISA)

Inferential Software Analysis (ISA) is a methodology by which both the design and the implementation of a system can be simultaneously analyzed and revised. As shown in Figure 1, ISA is based on the iterative application of certain semantic equivalent transformation rules, conditions lion techniques, and semantic inconsistency checking, and invalid semantics elimination to source codes. ISA, in general, has the momentous property that they can be applied without either expert or global knowledge of the system at hand; instead, the analysis progressively teaches the analyst about the system. For this reason, an ISA analyst is



more like an editor of technical papers than he is like a technical writer.

Much of ISA is based on the principle that even when little is known about a thing's purpose of function, it can often be recognized to have a simpler form. Software fragments can sometimes be demonstrated to be equivalent without fully comprehending the function of either. Software fragments sometimes can also be demonstrated to be equivalent irrespective of their surrounding contexts (which may differ).

Figure 2 shows an example of ISA process applied to a *loop* statement with nested *if* statement constructs. In the original program, the *exit* condition of loop statement is not explicitly represented in the syntax. The purpose of applying ISA process is to rewrite to this program fragment, so the *exit* condition can be explicitly represented. In this example, *cl* and *c2* represent boolean expressions; X and Y are either a single statement or a sequence of statements. The transformation in step 1 expands *if* statement with *exit* statement into two *if* statements with *null* statements, where we assume X statement does not affect *c2* condition. In step 2, *if* statement with *null* statement is simplified. Step 3 switches two *if* statements, where we assume X does not affect *c2*. In step 4, we factor the condition *cl* to statements of *then* part of *if* statement, where we assume statements in the *then* part do not affect the value of *cl*. Step 5 simplifies condition part of nested instatement. Step 6 transforms *loop* statement to *while* statement. Step 7 propagates condition to each statement. Step 8 checks semantic consistency. In the final step, the syntax of redundant *if* condition checking is eliminated. From this example, we have noted that transformation also requires pre-condition checking.

Some pre-conditions have to be assured before a transformation rule can be applied.
Although we can argue that explicitly revealing the *exit* condition in the syntax is nothing better than the original form, it does propose another clear way of program representation and, if applied to entire software system, unify the writing style.

Some ISA Rules

Before we list some of transformation rules, let's define notations used in the following discussion. Let X and Y be a statement, a sequence of statement, or null statement, "X;Y" denotes that the execution of X is followed by, where ";" is the delimiter of the sequence of statements. A symbol, say $\Delta:$, represents a label and serves as an entry point of control flow of a program. A loop statement is represented by **loop X end loop.** C, C 1, and C2 are boolean expressions and [ ] is denoted as a pre-condition.

Some Transformation Rules

Rule 1: loop X; exit when C; Y; end loop $\Leftrightarrow$ loop X if C then goto $\Delta:$; Y; end loop; $\Delta:$
Rule 2: loop X; if C then goto $\Delta:$; Y; end loop V: $\Leftrightarrow$ $\nabla:$ X; if C then goto $\Delta:$; Y; goto $\nabla:$; V:
Rule3: $\Delta:$X; $\nabla:$Y;goto$\Delta:$ $\Leftrightarrow$ X; $\nabla:$Y;X; goto$\nabla:$
Rule 4: goto $\nabla:$; $\nabla:$ $\Leftrightarrow$ $\nabla:$
Rule 5: if C then X else Y $\Leftrightarrow$ if **NOT** C then Y else X
Rule 6: $\Delta:$If C then X; goto $\Delta:$ $\Leftrightarrow$while C loop X; end loop
Rule 7: if C or **NOT** C then X endif $\Leftrightarrow$ X
Rule 8: if C then X else Y endif $\Leftrightarrow$ [X dose not affect C] if C then X else null endif; if C then null else Y endif
Rule 9: X;Y $\Leftrightarrow$ [X and Y are semantically independent] Y;X
Rule 10: if C then X;Y endif $\Leftrightarrow$ [X does not affect C] if C then X endif; if C then Y endif
Rule 1 1: if C 1 then if C2 then X endif endif $\Leftrightarrow$ [C 1 does not affect C2] if C 1 and C2 then X endif;
Rule 12: if C then X else null endif $\Leftrightarrow$ if C then X endif
Rule 13: if C then null else X $\Leftrightarrow$ if NOT C then X endif

where we informally define *semantic independence* of two statements X and Y as follows: if the execution order X and Y does not affect execution results, X and Y are semantic independent.

Some Condition Propagation Rules

Rule 14: [C 1] if C2 then X else Y endif $\Leftrightarrow$ if C2 then [C 1 and C2] X else [C 1 and NOT C2] Y endif
Rule 15: [C] X;Y $\Leftrightarrow$ [X does not affect C] X; [C] Y
Rule 16: [C] X;Y $\Leftrightarrow$ [X affects C] X; [update-condition(C,X)] Y

Where function *update-condition* takes condition C and statement X as input and returns an updated condition based on semantics of X statement. For example, the pre-condition "a > 0 " in "[a > 0] a=l; b=l" will become "a = 1" after the execution of statement "a=1", where the condition a=1 is returned by update-condition ([a>0], a=1).

Some Elimination Rules

Rule 17: [C] if NOT C then X $\Leftrightarrow \Phi$
Rule 18: [C] if C then X endif $\Leftrightarrow$ X

In appendix A, we show two more ISA examples, one is a Fortran program fragment and the other is a command procedure. Two examples being reclaimed all show much more clear and smaller results.

3. EXPERIENCES

Active Maintenance pays for itself over the long run by decreasing maintenance costs; hence, a short-lived systems, with only a short maintenance tail, is an unlikely target for this treatment. The cost of actively maintaining a system is sensitive to the amount of verification demand for it. One never wants to let a customer exaggerate his real needs in this respect, because during the project every component will have to be modified and put into service, perhaps repeatedly.
As a classroom exercise, to formally demonstrate the purely inferential character of this approach, the author has recovered a Fortran program of about 2200 lines, starting with no inkling of its purpose and no documentation other than its listing and a sample of its output. The program was reduced to fewer than 500 lines.
Experience gained from similar work on nearly a hundred programs, derived from many applications, strongly suggests that the average application program, when it is thoroughly recovered, loses over half its volume. Most of reduced volume were the redundant, inconsistent, or dead-code statements. From our experience, with the continued improvement process the cost of software maintenance had been much less. Therefore, we claim that maintaining a robust software system is much cost effective than curing a "sick" system.

3.1. What's Wrong with Programs?

What is wrong with the average program that leads to over half of it being discarded? While this subject could fill a book, it can summed up by saying that most programs suffer from various sorts of indirectness. They spend too much time beating around the bush and not coming to the point. The business that the program is supposed to carry on is overwhelmed by its busyness with itself –a trait that might be described as "software narcissism".
One common instance of useless processing is unnecessary copying of data back and forth inside a program. One wonders how many CPU hours are wasted each year on pointless shuffling of data. Another related practice is unnecessary transformation of data: converting data into mysterious formats or codes before they are used, when they can be used just as efficiently in their original form. Sometimes, the only effect of encrypting data in this way is the need to decrypt them again.
Overly conditional writing is the rule in present day programming practice. Almost no programming

problem is as "iffy" as it first seems. Incorrect identification of the unit of work often leads to a lot of trouble. One form of this is the Big-Array style of program. A Big-Array program brings all the data in the world into memory at once, so that it will be easily accessible in case it ever discovered what to do with it. This style of programming seems to be founded on the idea that a programming problem can be solved by overwhelming it with sheer computational power.

At the opposite extreme is the Half-a-Loaf style of programming. Here, the data necessary to perform a unit of work are fetched in pieces, and the program spends most of its time trying to figure out what fractional stage of the work cycle it is in. People in this camp seem to prefer writing "ifs" instead of "while's".

Say to say, specimens of meticulously "goto-less" coding are prize trophies in the present exhibit. Program flags and switches are too often used as convert means of implementing explicit branches.

This is very unfortunate because reliance upon internal logical variables practically guarantees that real solution to the problem at hand will be missed. eliminating flags is difficult, but it increases comprehensibility like nothing else.

One's goal in recovering a program should be to eliminate everything in it that is not directly related to the business of the program, in other words, what the customer understands and cares about. The ideal program is one whose every statement translates straightforwardly into an assertion about the customer's business that the customer could immediately agree was both true and relevant to the job he has in mind. "Business-centric" style is correct: egocentric style is not.

We believe many programmers suffer from Representation-Fixation: the unnecessary and undesirable distinction between objects and their representation within the program. Variable names that end with suffixes such as code, proc, flag, or ptr, show an undesirable preoccupation with representation. This trait in programmers suggests an overly literal, poorly abstractive, mind set. It is reminiscent of the psychology conditioned by programming in assembly language, where the definite tendency is to think of a symbolic identifier as naming the memory address of a datum, rather than as naming the datum itself. This is precisely contrary to the essential idea of data abstraction

Clarity of format is another important area where programs at present often fall down. Touching only briefly on this subject, one should answer question like these while recovery is in progress:

- Are all names as well chosen as they could be? Have all names been checked or just the names of variables?
- Is there enough white space in the program. A program can be nearly unreadable, without it.
- Do I understand all the comments in the program? If not, will anyone else?

3.2. Debugging Programs

We believe that most program bugs are the result of unclear expression, that is, inadequate iteration of the program draft. Although, strictly speaking, it is impossible to infer the existence of any bug, since this would require knowledge of the program's intent, active maintenance is nevertheless a practical means of locating and removing bugs. Indeed, debugging is an almost inevitable outcome of thorough recovery.

The author once discovered several dozen, suspicious-looking files in a system he was maintaining. After much effort it was determined that these files were nowhere used and, in fact, had never been used during the two years they had existed. Forty cylinders were released.

A program under recovery is like a visibly shrinking carpet. The bugs hiding under it are either automatically uncovered by the process, or they resist the shrinking, which also reveals their presence. For instance, statements may knot up into a logical tangle that defies every effort to pick apart. Once a few strings in the knot are carefully cut, the reduction of the program can be resumed. This involves extra-inferential decision, of course, but the recovery process, plus common sense, do at least offer clues as to what they should be. Heretofore unrecognized bugs can be discovered in this manner.

One might be concerned that compressing a program beyond a certain point would start to complicate, rather than simplify, it. However, experience shows that there is little real danger of a program becoming too densely written, in other words, of brevity defeating clarity.

In fact, the importance of studying trade-offs between goals has generally been exaggerated. All the commonly accepted desirable attributes of program text, understandability, reliability, maintainability, seem to be characterized more by mutual reinforcement than by mutual exclusion. They are aspects of a single super-attribute, called quality. An unfortunate corollary to this is this is that whatever degrades a system in one of these aspects, will also degrades it in other aspects as well.

4. RELATED WORKS

Much research has been worked on transformation for program development [2-9] Transformation from initial specification to final program has the correctness merit. Only few research has been applied transformation for software maintenance. Balazer suggested a method to modify original specification and then re-implement it [7]. A Maintainer's Assistant has applied transformation to existing source codes. It's ultimate objective is to facilitate the transformation of existing source program to high level specification. It restructured source code to an easily understood representation by improving control flow, removing dead code, and introducing a good procedural structure. The application of transformation rules were metrics guided. However, it did not apply semantics propagation to remove logically unreachable or redundant statements. Pleszkoch etc

applied semantic propagation to control variables, which are boolean variables whose purpose is to control program flow in decision and iteration structures and eliminated logically *non-traversable path,* a control path through the program logic that cannot possibly be traveled during execution, from structured programs [9]. Chu and Patel incorporated software engineering principles, such as localization, information hiding, and abstraction to program transformation and restructuring [10,11]. Kozaczynski etc classified program transformation to text-level, syntactic-level, semantic-level, and concept-level transformation. Their tools support automated concept recognition by defining concept pattern in the transformation rules. This approach greatly elevates the levels of transformation specification [12].

In all these cases, maintenance activities are held to solve one problem at a time. Active Maintenance approach reduced the maintenance efforts of evolutional software systems by transforming under-developed into quality software systems with reduced size and unified style.

5. SUMMARY

The Active Maintenance approach is proposed to reduce software maintenance cost by refining source codes. The applicability of this approach is due to the under-development of source codes of software systems and improper patches during maintenance phase. Most software systems present different styles, unnecessary complexity, redundant codes, logically unreachable statements, etc. which all directly reduce software maintainability. Traditional passive maintenance approach usually solve one problem at a time, but may also introduce more problems later. Although maintenance activities usually modify small portion of source code, they require to read and understand relatively large portion of source codes before a proper modification action can be taken. Therefore, the representation and quality of source code will affect maintenance efficiency. From our experience, with the continued improvement process the cost of software maintenance had been much less. Therefore, we claim that maintaining a robust software system is much cost effective than curing a "sick" system.

By using a formal inference methodology to re-write existing software, the correctness of program can be preserved and many imbedded errors can be revealed. Meanwhile, the semantically unreachable statements and possible incorrect semantics can be revealed, the size of software be reduced, writing style be unified, and the total software cost be indirectly reduced.

Current approach has been manually applied in more than a dozen projects and has shown its feasibility of cost reduction on software maintenance. Our next step is to implement this approach, so the transformation can be automatically applied.

6. REFERENCE

[1] Lientz, B. and Swanson, "Software Maintenance Management," Addison Wesley, 1980.

[2] Bauer, F. L., Moller, B., Partsch, H. and Pepper, P., "Formal Construction by Transformation-Computer Aided Intuition Guided Programming," *IEEE Trans. on Software Engineering,* Vol. SE-15, No. 2, Feb., 1989.

[3] Bull, T., "An Introduction to the WSL Program Transformer," *IEEE Conference on Software Maintenance 1990,* San Diego, California, Nov. 26-29, 1990.

[4] Feather, M. S., "A Survey and Classification of Some Program Transformation Techniques," in *Program Specification and Transformation,* 1987.

[5] Partcsh, H. and Steinbrugen, R., "Program Transformation Systems," *Computing Surveys,* Vol. 15, No. 3., Sept., 1983.

[6] Yang, H., "The Supporting Environment for a Reverse Engineering System-The Maintainer's Assistant," *IEEE Conference on Software Maintenance 1991,* Sorrento, Italy, 1991.

[7] Balzer, R., "A 15 Year Prospective on Automatic Programming," *IEEE Trans. on Software Engineering,* Vol. SE- 11, No. 11, pp. 1257-1267, Nov. 1985.

APPENDIX A

Example 1. Fragment from Filght Test System

```
BEFORE
C COMPUTE STTM, MLM (MISSILE LAUNCH MODE),
C LTE, TFL, ACM, LAM.
 STTM=0
      IF (ACTLZ .EQ. 1) GOTO 180
      IF (LM .EQ. 1) GOTO 160
      IF (LM.EQ. 3) GOTO 170
 250  IF(IFLT(7) .LT. 15 GOTO 255
      XRSKA = 8.4
      IF (RSKA .EQ. 13.2) XRSKA = 13.2
      IF (ACMS .EQ. 1 .AND.
      * .(LM .EQ. 0 AND (HPRF .EQ. 1 OR .LM.EQ. 2))
      * .AND.RL.LT. XRSKA) GOTO 180
 255  IF (HPRF .NE. 1 GOTO 180
 257  MLM =1
      IF (LM .EQ. 0) GOTO 190
      TFL=9.08
      GOTO 200
 190  TFL=8.12
 200  IF(ACTLAU .NE. 1) GOTO 210
      MLM=2
      IF (LM .NE. 0) GOTO 220
      TFL=.79
      GOTO 210
 220  TFL=1.81
      IF (LM .GT. 1) GO TO 230
      STTM= 1
 230  LTE=3
      GOTO 240
 160  IF (MODE .EQ. 1 .OR. MODE .EQ.3) .AND. HPRF .EQ. 1)
      * GOTO 250
      LTE = 3
 180  IF ((COS(EL( 1 )/R)*COS(AZ( 1 )/R) .GE. .966) LTE=1
      IF (IFLT(7) .LT. 16) GOTO 260
      LTE=3
      IF(ABS((N/RL1 *COS(HDGI/R) + E/RL1 *SIN(HDGI/R))
```

```
     *COS(PITCH/R) I -D/RL I *SIN(PITCH/R)) .GE9659) LTE= I
        GOTO 260
 170    LTE=I
 260    MLM=3
        TFL=-.05
 240    IF (MLM .EQ. 3) ACM=I
        LAM=0
        IF(MLM .EQ. 3.0R. IRID.EQ. 0) GOTO 245
        IF(MLM .EQ. 3 .AND. HI .LE. 2600 .AND. RMTAS .LT. 1)
      GOTO 244
        IF((TYPE .NE. I .OR. HT .GT. 2600) .AND. TYPE .NE. 2)
      GOTO 245
        IF(ECM .LT. 2 .AND. RMTAS .LT. 1) GOTO 244
        IF(ALPHAL .LT. 90.) GOTO 245
 244    LAM= I
 245    .....
```

```
AFTER
C COMPUTE STTM, MLM (MISSILE LAUNCH MODE), C LTE,
TFL, ACM. LAM.
STTM=0
MLM=3
LTE=3
TFL=-.05
IF (ACTLZ) THEN
LAMBDA=LAM_P( I )*COSHI+LAM_P(2)*SINHI
LAMBDA = LAMBDA*COSPI -LAM_P(3)*SINPI
IF (ABS(LAMDA) > .9659) LTE=I
ELSE IF (LM .EQ. 3) THEN
LTE= I
ELSE IF(ACTLAU) THEN
MLM=2
TFL= 1.81
IF (LM .EQ. 0) TFL=0.79
IF(LM.LE. 1) STTM= I
ELSE
MLM= I
TFL=9.08
IF(LM .EQ.0) TFL=8.12
IF(LM .LE.) STTM=I
ENDIF
ACM=(MLM .EQ.3)
LAM= .NOT. ACM .AND. (IRID .NE. 0) .AND (TYPE .EQ.
* I .AND HT .LE. 2.6) .AND (ECM .LT. 2 AND RMTAS .LT. I
* .OR. ALFL .GE. 90.)
```

Example 2. Command Procedure Fragment

BEFORE

```
$ on warning then exit
$ if p I .eqs. " 1 " then goto kbatch
$ if p I .eqs. "2" then goto kprint
$ if pl.eqs. "3" then goto klqpunt
$ exit
$ KBATCH:
$ on warning then goto dintry
$ on error then goto dintry
$ START
$ if p2 .eqs. "" then goto ask
$ goto haltit
$ ASK:
$ inquire p2 "Job_number"
$ goto start
$ HALTIT:
$ stop/entry='p2'sys$batch
$ goto ok
$ DLNTRY:
$ on warning then goto dintrypq
$ on error then goto dintrypq
$ del/entry='p2' sys$batch
$ write sys$output "Pending job deleted"
$ exit
$ OK:
$ write sys$output "Job deleted."
$ exit
$ KPRINT:
$ STARTP:
$ on warning then goto dintrypq
$ on error then goto dintrypq
$ if p2 .eqs. "" then goto askp
$ goto haltitp
$ ASKP:
$ inquire p2 "Job number"
$ goto startp
$ HALTITP:
$ stop/queue/entry= 'p2' sysprinter
$ goto okp
$ DLNTRYP:
$ on warning then goto dintrypq
$ on error then goto dintrypq
$ del/entry='p2' sys$print
$ write sys$output "Pending job deleted."
$ exit
$ 0KP: write sys$output "Bad job number."
$ exit
$ KLQPRINT
$ STARTLQP:
$ on warning then got dintrylqp
$ on error then goto dintrylqp
$ if p2 .eqs. "" then goto asklqp
$ goto haltitiqp
$ ASKLQP:
$ inquire p2 "JOb number"
$ goto startiqp
$ HALTITLQP:
$ stop/queue/entry='p2' Iqp
$ goto oklqp
$ DLNTRYLQP:
$ on warning then got dintrylqp
$ on error then goto dintrylqp
$ del/entry='p2' Iqp
$ write sys$output "Pending Job deleted."
$ exit
$ OKLQP:
$ write sys$output "Job deleted."
$ exit
$ DLNTRYLQPQ:
$ write sys$output "Bad job number."
$ exit
```

AFTER

```
$ if pl .eq. I then q := sys$batch
$ if p I .eq. 2 then q := sys$print
$ if p I. eq. 3 then q;= Iqp
$
$WHILE: if p2 .ne. O then goto END WHILE
$ inquire p2 "Job Number?"
$ goto while $END_WHILE:
$
$ msg = "Job has been stopped."
$ on error then continue
$ stop 'q'/entry='p2'
$
$ if $severity then goto end_if
            msg = "Pending job deleted."
            on error then msg = "Bad Job nutuber."
            delete 'q' /entry='p2'
$END_IF:
$ write sys$output msg
$ exit
```