PROCEEDINGS

THIRD ANNUAL

PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

TOOLS FOR SOFTWARE QUALITY

September 27, 1985
Red Lion/Lloyd Center
Portland, Oregon

1985 Pacific Northwest Software Quality Conference

TABLE OF CONTENTS

CHAIRMAN'S MESSAGE

Ronald K. Swingen

Welcome to the Third Annual Pacific Northwest Software Quality Conference. We are pleased that you took advantage of this opportunity to share knowledge and ideas on "Tools for Software Quality."

The requirements for producing high-quality software have never been greater than they are today. Our "computer industry" is under extreme pressure to be profitable and productive. We cannot meet these requirements merely by working harder. We must avail ourselves of every opportunity to leverage our efforts--hence, the importance of software tools.

The Proceedings contains 15 papers from software engineers and managers who responded to our Call for Papers. One of these will be reproduced in a future issue of IEEE Computer. This year's Conference includes a new element--exhibits by selected vendors who offer products related to our theme. A listing of those vendors is included for your reference.

Watch for an announcement of the Fourth Annual Software Quality Conference during the summer of 1986. We welcome your comments on this year's Conference and your suggestions for the 1986 program.

ORGANIZING COMMITTEE

1985 Pacific Northwest Software Quality Conference

Chairman:
Ronald K. Swingen
Mentor Graphics
8500 S.W. Creekside Place
Beaverton OR 97005
503/626-7000

Program:
Monika Hunscher
Floating Point Systems, Inc.
P.O. Box 23489, MS S-150
Portland OR 97223
503/641-3151, x1516

Exhibits:
Richard A. Martin
Intel Corporation, EY2-01
5200 N.E. Elam Young Parkway
Hillsboro OR 97124
503/681-2246

Treasurer:
Kenneth P. Oar
Hewlett Packard Co.
Portable Computer Division
1000 N.E. Circle Blvd.
Corvallis OR 97330
503/757-2000, x 4248

Committee

Sue Bartlett
Metheus C-V, Inc.
P.O. Box 959
Hillsboro OR 97123
503/640-8000, x231

Chuck Martiny
Tektronix, Inc. MS 50-487
P.O. Box 500
Beaverton OR 97077
503/627-6834

Dale Mosby
Sequent Computer Systems, Inc.
14360 N.W. Science Park Drive
Portland OR 97229
503/626-5700

LeRoy Nollette
Tektronix, Inc. MS 78-528
P.O. Box 500
Beaverton OR 97077
503/627-5012

Steve Shellans
Tektronix, Inc. MS 50-487
P.O. Box 500
Beaverton OR 97077
503/627-4954

Janet Sheperdigian
Intel Corporation EY2-01
5200 N.E. Elam Young Parkway
Hillsboro OR 97124
503/681-2284

Sue Strater
Mentor Graphics
8500 S.W. Creekside Place
Beaverton OR 97005
503/626-7000

George Tice
Tektronix, Inc. MS 92-525
P.O. Box 500
Beaverton OR 97077
503/629-1310

AUTHORS

1985 Pacific Northwest Software Quality Conference

Ms. Susan Bartlett
Metheus-CV
P.O. Box 959
Hillsboro OR 97123

Dr. David B. Benson
BENTEC
NE 615 Campus St.
Pullman WA 99163

Ms. Deborah Caswell
Hewlett Packard
3500 Deer Creek Rd.
Palo Alto CA 94304

Dr. Curtis Cook
Oregon State University
Computer Science Department
Corvallis OR 97331

Mr. Bob Grady
Hewlett Packard
3500 Deer Creek Road
Palo Alto CA 94304

Mr. Warren Harrison
University of Portland
5000 N. Willamette Blvd.
Portland OR 97203

Mr. William H. Hodges
Boeing Computer Services
P.O. Box 24346
Seattle WA 98124-0346

Dr. Peter Hruschka
Promod, Inc.
22981 Alcalde Dr.
Laguna Hills CA 92653

Mr. Arun Jagota
Intel Corporation
5200 N.E. Elam Young Parkway
Hillsboro OR 97124

Mr. David J. Kerchner
Floating Point Systems
P.O. Box 23489, MS S-150
Portland OR 97223

Dr. Edward Miller
Software Research Associates
P.O. Box 2432
San Francisco CA 94126

Mr. Tom Milligan
Tektronix, Inc.
14460 N.W. Hunters Dr.
Beaverton OR 97006

Mr. Chris Olson
Tektronix Inc.
P.O. Box 4600 MS 92-525
Beaverton OR 97075

Mr. David A. Rodgers
Boeing Commercial Airplane Co.
12707 N.E. 120th, Unit B3
Kirkland WA 98034

Mr. Gordon Staley
Hewlett Packard Co.
1000 N.E. Circle Blvd., P.C. Div.
Corvallis OR 97330

Mr. George D. Tice, Jr.
Tektronix, Inc.
P.O. Box 4600 MS 92-525
Beaverton OR 97075

Mr. David Vomocil
Hewlett Packard Co.
1000 N.E. Circle Blvd.
Corvallis OR 97330

Mr. Walter Webb
Tektronix, Inc.
P.O. Box 4600 MS 92-525
Beaverton OR 97075

Mr. Rainer Wieland
Tektronix, Inc.
P.O. Box 4600 MS 92-525
Beaverton OR 97075

EXHIBITORS

1985 Pacific Northwest Software Quality Conference

Database Design, Inc.
Contact: Kim Frazier
2006 Hogback Rd.
Ann Arbor MI 48104
313/971-5363

Higher Order Software
Contact: Robert S. Dane
2067 Massachusetts Ave.
Cambridge MA 02140
214/257-3758

Interactive Systems Corporation
Contact: Kristie Korte
2401 Colorado Ave., 3rd Floor
Santa Monica CA 90404
213/453-8649

Productivity Products International
Contact: Chet Wisinski
27 Glen Road
Shady Hook CT 06482
203/426-1875

Promod, Inc.
Contact: Thomas L. Scott
22981 Alcalde Dr.
Laguna Hills CA 92653
714/855-8560

Software Research Associates
Contact: Dr. Ed Miller
580 Market St., Suite 350
San Francisco CA 94104
415/957-1441

Teledyne Brown Engineering
Contact: Rusty Bynum
300 Sparkman Dr.
Huntsville AL 35807
205/532-1661

Wiley Learning Technologies
Contact: Jacqueline Philpotts
605 Third Avenue
New York NY 10158
212/850-6000

KEYNOTE

Adversaries in Software Development

Dr. Richard Hamlet
Professor of Computer Science
Oregon Graduate Center for Study and Research

Although designers and programmers want to make their software work well, the pressure of circumstances can compromise a project. An independent quality assurance (QA) group can defend standards, but only if there is agreement about measures of software quality. Unfortunately, our understanding of how to measure quality is still very poor.

A careful look at a number of accepted quality measures shows that each can be subverted. That is, software may be given the appearance of quality (accidentally or on purpose) without having the substance. For the measures to have meaning, software developers must cooperate with QA in their application, observing the spirit rather than the letter of the law.

Biography

Dr. Richard Hamlet has had a distinguished career in higher education as teacher and researcher. In the last 20 years he has taught at the University of Washington, University of Maryland, and University of Melbourne, and since 1984 has been Professor of Computer Science at the Oregon Graduate Center.

Dr. Hamlet also has had practical experience as systems programming director of a university computer center and a commercial timesharing service bureau. He has rewritten and maintained a commercial operating system, and written several production-quality compilers. He is the author of a textbook on theory of computing and is working on two other texts, one on theory, and the other (with Harlan Mills and others), an introduction to programming from a mathematical point of view.

Session 1

METHODOLOGIES

Titles and Speakers:

"Extending Structured Analysis to Become a Design Tool"
Walter Webb, Rainer Wieland, and Chris Olson, Tektronix, Inc.

"Implementing the Software Review Process"
David Kerchner, Floating Point Systems, Inc.

"Software Configuration Management: A Tool for Software Quality"
George Tice, Jr., Tektronix, Inc.

Extending Structured Analysis to Become a Design Tool

Walter Webb

Rainer Wieland

Chris Olson

Software Development Products Division
Tektronix, Inc.

ABSTRACT

This paper presents an extension of the Structured Analysis method of writing specifications. Data flow diagrams previously defined using interactive graphics tools directly dictate the architecture of the program. The software design is the requirements definition, bypassing the traditional structured design step. Consequently, the resulting program closely reflects the Structured Analysis document. This approach was successfully employed in building a software product at Tektronix.

July 2, 1985

Extending Structured Analysis to Become a Design Tool

Walter Webb

Rainer Wieland

Chris Olson

Software Development Products Division
Tektronix, Inc.

Introduction

In July, 1984, Tektronix began selling software tools (SA Tools) which aid a software engineer in using Structured Analysis.[1] The tools themselves were specified with Structured Analysis. The resulting specification showed all processes and their data interfaces down to the mini-specification level. The next step was to develop a structured design by the application of transform and transaction analysis on the structured specification. This procedure, however, is ill-defined.

> When you carry out transform analysis, remember that it is a strategy. You cannot unthinkingly follow its steps as you could those of an algorithm. From time to time, to stay on the right track, you must bring to bear your knowledge of what the system is supposed to accomplish. And, when you derive your first structure chart, you must use all the design criteria you have learned to improve it.
>
> One day, transform analysis may become an algorithm. But if it does, the structure chart will disappear and the DFD will be implemented directly, for a machine can obey an algorithm much better than can a human being. . . . perhaps, we shall see DFDs being executed on a horde of dynamically reconfigurable microprocessors.[2]

At about this time, the project team was exposed to the Large-Grain Data Flow technique.[3] With this approach, the data flow diagrams (DFDs) directly dictate the architecture of the program. Contrary to the popular convention of developing structure charts from the DFDs, structure charts are never drawn. This technique was adapted by the project team in designing the software.

A New Model for Program Design

Processes in a DFD are defined either by a lower level DFD or by a mini-specification (mini-spec). The DFDs exist in a hierarchical tree structure where the mini-specs are the leaves on the tree. A mini-spec contains a structured English description of a primitive process.

The new program design model is based on the DFD: only mini-spec processes do any work (are executable) and they may execute in parallel. In addition, all data flows are thought of as being single-entry queues that are either full or empty. The essential question is "When should a mini-spec (module) be activated?". This question is answered as follows.

Control of Module Execution

A module is able to perform its task when all of its input queues are full *and* when all of its output queues are empty. The destination of some output data flows is external to the system. Their queues are fixed as always being empty. This approach is similar to techniques proposed by

others.[4,5] Some modules do not need all of their inputs to be present or all of their outputs to be consumed. Such modules require state memory to track which data flows are present or which have been used.

Each module is responsible for filling its output queues and for emptying its input queues. Failure to do so results in a static (or deadlocked) system. In this manner each module can be executed independently based on the state of its inputs and outputs. If no queues change state (are emptied or filled) after all modules have executed, the entire program is done and halts. A single main program controls the execution of all modules.

This model is implemented by associating a boolean flag for each queue. A queue is full if its corresponding flag it *set*, empty if its flag is *clear*. The setting and clearing of these flags is performed using compile-time macros.

The specific application of this model to develop a program is described in the subsequent sections.

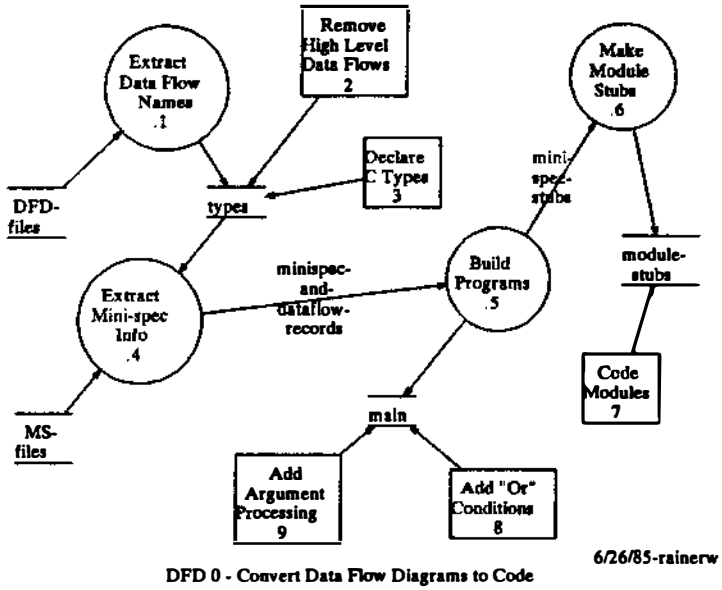


Figure 1

From Data Flow Diagrams to Code -- an Overview

It is assumed that one starts from a structured specification consisting of data flow diagrams, mini-specs, and a data dictionary. It is also assumed that these documents are consistent and correct. The following examples and descriptions are based on the use and output of Tektronix' SA Tools.

These tools are grouped into five categories: graphics editing tools, evaluation tools, correction tools, display tools, and auxiliary tools. The cumbersome task of drawing, correcting, and verifying the data flow diagrams is simplified by using these automated tools.

To produce high-level code (C in this case) from the data flow diagrams the steps shown in Figure 1 are followed. Figure 1 is itself shown as a DFD with circles, representing automated steps and rectangles representing manual steps. The steps are outlined in the Table 1.

Table 1	
Step	Description
Extract Data Flow Names	The data flow names are extracted from all DFDs, sorted and saved in the file *types*.
Remove High Level Data Flows	The data flow names not attached to a mini-spec are deleted. The output data flows that terminate on the boundary of DFD 0 (the top level DFD) are also deleted.
Declare C Types	The C language declarations for the remaining data flows are defined.
Extract Mini-spec Info	The mini-spec body is converted to C language comments. Records indicating the name and type of each data flow and the names and parameters of each module are created.
Build Programs	The main program containing the code to call each mini-spec, the data declarations for all data flows with their associated queue states, and the module stubs containing the parameter declarations and mini-spec body as comments are all created.
Make Module Stubs	The module stubs are split into separate files.
Code Modules	The mini-spec inserted as comments into each module is converted to code.
Add "Or" Conditions	This step is optional. It consists of adding logical "or" conditions to the "if" statements preceding a module's call from the main program.
Add Argument Processing	This step is optional. It consists of adding code to process any command line parameters that are needed by the program.

An Example

The above technique was used to produce the SA Tools. Each DFD and mini-spec must be in a separate file. A directory must be created containing only those DFDs and mini-specs that are directly involved in the code generation. For this example the files for the SA Tools' **lookdd** command are in the current directory:

dfd0	dfd2	dfd9	dfd9.1	ms2.1
ms9.1.1	ms9.1.2	ms9.1.3	ms9.1.4	ms9.1.5

To generate code for one of the other list commands, a different set of lower level DFDs and mini-specs would be used with the same top level DFD, The leveled DFDs for the **lookdd** command are shown in Figures 2 through 5.

Figure 2 is the top level DFD for the SA Tools' *list* commands. It contains no mini-specs and ends up with no executable modules.

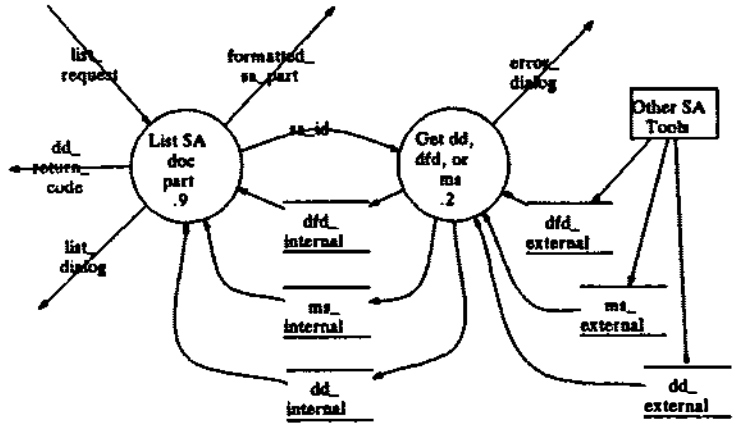


Figure 2

Figure 3 contains the input modules for the three types of files supported by the SA Tools. Once created, these modules serve as library modules for all of the tools. The DFDs are thereby used to define modules which are common to several individual programs. The program **lookdd** only needs the mini-spec 2.1 to read the data dictionary. The other processes in Figure 2 will not contribute to the code.

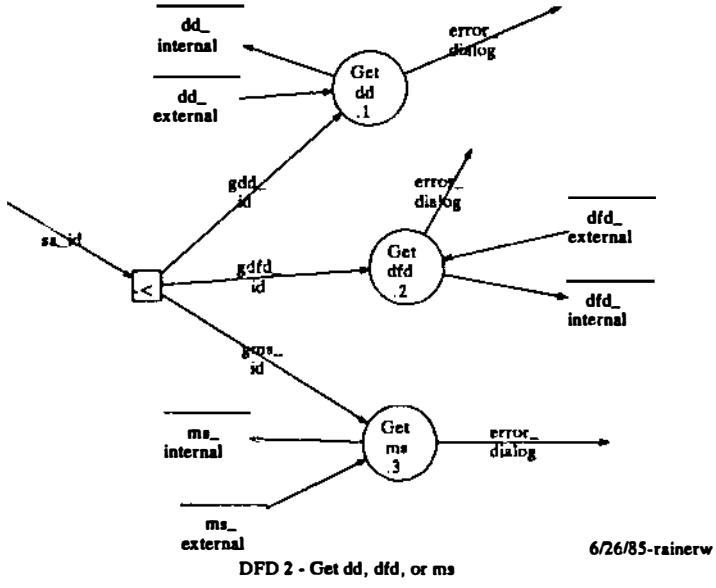


Figure 3

Figure 4 shows the *list* commands as separate entities. None of the processes in this figure are mini-specs. Only the mini-specs under process 9.1 (the **lookdd** command) will expand into code.

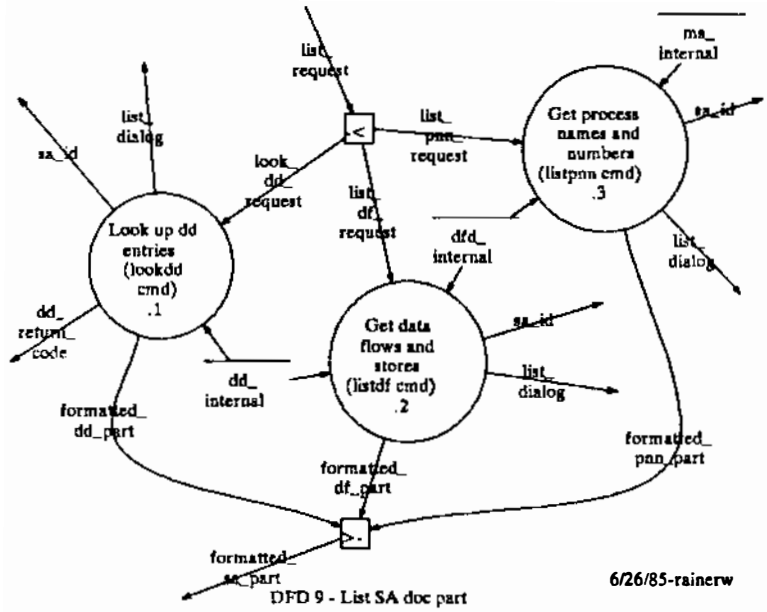


Figure 4

Figure 5 shows all of the processes that comprise the **lookdd** command. Each process is a mini-spec and will have a corresponding code module.

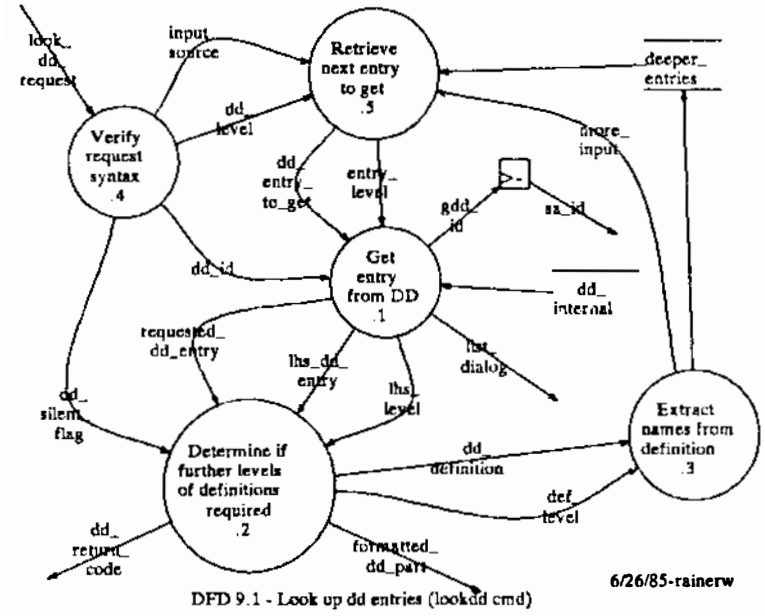


Figure 5

From Data Flow Diagrams to Code -- Step by Step

The automated steps are composed of standard UNIX* commands, SA Tools' commands, and special programs developed to support this code generation technique. The manual steps are performed using a text editor.

UNIX is a Trademark of Bell Laboratories.

Step 1 - Extract Data Flow Names

This step consists of the command sequence:

listdf dfd* | sort | uniq >types

Data flow names are extracted from all the data flow diagrams (**listdf dfd***), sorted by name (**sort**), duplicates dropped (**uniq**), and saved in a file (**>types**).

Performing this step on the DFDs in Figures 2-5 results in a list of all of the data flow names used in the diagrams. The contents of the file *types* are show below:

dd_definition	dd_entry_to_get	dd_external
dd_id	dd_internal	dd_level
dd_return_code	dd_silent_flag	deeper_entries
def_level	dfd_external	dfd_internal
entry_level	error_dialog	formatted_dd_part
formatted_df_part	formatted_pnn_part	formatted_sa_part
gdd_id	gdfd_id	gms_id
input_source	lhs_dd_entry	lhs_level
list_df_request	list_dialog	list_pnn_request
list_request	look_dd_request	more_input
ms_external	ms_internal	requested_dd_entry
sa_id		

Step 2 - Remove High Level Data Flows

This step consists of deleting all data flow names that do not flow into or out of a mini-spec. Since only mini-specs are executed the intermediate level data flow names can be discarded. The data flow names that are outputs (terminate on a boundary point) of the top level DFD must also be deleted. Such outputs are not cleared by the modules in the current program since they are consumed external to this system.

Step 3 - Declare C Types

In this step, the designer specifies the C data declarations for the remaining names in the file *types*. If the data dictionary was constructed correctly, this information should be readily available. The results of this step are shown below:

dd_definition	char *
dd_entry_to_get	char *
dd_id	char *
dd_internal	struct dd
dd_level	int
dd_return_code	int
dd_silent_flag	int
gdd_id	char *
look_dd_request	char *
requested_dd_entry	char *

Step 4 - Extract Mini-spec Info

This step consists of the following command sequence

dfdtolist dfd*

This step and the next two are usually performed together from a UNIX shell script for ease of use. We normally put the following into a shell script called *dtoc*:

dfdtolist dfd | awk -f awk.script ; mkproc proc.c*

The **dfdtolist** program uses the same DFDs as are used in the previous steps. The file *types* as well as all of the mini-spec files for the DFDs are also required by **dfdtolist**. These files need not be specified on the command line that invokes **dfdtolist**; the program accesses them directly. The output from this step consists of C comment blocks, dataflow records, and minispec records for each mini-spec. The output is directed to standard output so that it can be piped to the next step.

The comment blocks indicate information derived from mini-spec such as author, date, parent name, as well as the entire mini-spec body.

The dataflow record indicates the name and type of a data flow originating or terminating on the mini-spec's process bubble in the parent DFD. Each dataflow record has the format:

dataflow	name	flag	C-type

The *name* is the data flow name. The *flag* indicates whether the data flow is an input or an output. Both fields are derived from the parent DFD. The *C-type* field is taken from the file *types*.

The minispec record indicates the name of a process and all of its required data flow names. All fields are derived from the parent DFD. Each minispec record has the format:

minispec	file-name	function-name	parameters

file-name is the name of the file from which the mini-spec was read.

function-name is the name of the C function representing this mini-spec. Ideally, the name of the mini-spec process would be the name of the function. However, compilers and linkers have severe restrictions on lengths and of function names. Thus, the convention was adopted to name each function with a *p* followed by the process number (periods being replaced by underscores).

parameters are the names of all data flows used by the mini-spec. Again, the name of the data flow should become the name of the parameter. However, hyphens must be converted to underscores in order to disambiguate parameter names from arithmetic expressions.

A sample of the output from **dfdtolist** is shown below.

```
/*
 ***************************************************
 *
 * DFD        - 9.1 - Look up dd entries (lookdd cmd)
 * MINI SPEC   - 9.1.1
 * TITLE       - Get entry from DD
 * AUTHOR      - rainerw
 * DATE        - 6/26/85
 ***************************************************
 */

dataflow      dd_id  i     char   *
dataflow      dd_entry_to_get i     char   *
dataflow      dd_internal    i     struct dd
dataflow      requested_dd_entry    o     char   *
dataflow      gdd_id o     char   *
minispec ms9.1.1 p9_1_1 gdd_id  requested_dd_entry     dd_internal     dd_entry_to_get dd_id
     /*
      ***************************************************************************
```

```
*
* Get entry from DD
* rainerw
*  6/26/85
* 9.1.1
*
* Repeat {
*     if left hand side of dd entry != dd_entry_to_get {
*           skip this dd entry.
*     } else {
*           SET requested_dd_entry = right hand side of dd entry.
*           SET lhs_level = entry_level.
*           SET lhs_dd_entry = left hand side from a dd entry.
*           CLEAR  dd_entry_to_get.
*           CLEAR  entry_level.
*           Return.
*     }
* } until the entire dd_internal has been read one time.
*
* if (dd_silent_flag = FALSE) {
*     print "Name <dd_entry_to_get> not found in DD <gdd_id>".
* }
*
* CLEAR  dd_entry_to_get.
* CLEAR  entry_level.
* Return.
*
*************************************************************************
*/
```

Step 5 - Build Programs

This step consists of an *awk* program that produces the main program and the module stubs from the output of the previous step. The main program contains the data declarations for all data flows, the data declarations for all flags associated with each data flow, and the main loop that calls each mini-spec process in turn. The main program generated for the example is shown below. The include file references were automatically generated for this application.

```
/*
 * Main loop
 */
#include      "flag.h"
#include      "io.h"
#include      "error.h"
#include      "globals.h"

main (argc, argv)
      int    argc;
      char   *argv[];
{
      FLAG   loop_flag;

      do {

      loop_flag = 0;
      if (IS_SET(Fdd_internal)) goto skip001;
      if (IS_CLEAR(Fgdd_id)) goto skip001;
      p2_1 ();
      loop_flag = 1;
skip001:
      if (IS_SET(Fgdd_id)) goto skip002;
      if (IS_SET(Frequested_dd_entry)) goto skip002;
      if (IS_CLEAR(Fdd_internal)) goto skip002;
```

```
        if (IS_CLEAR(Fdd_entry_to_get)) goto skip002;
        if (IS_CLEAR(Fdd_id)) goto skip002;
        p9_1_1 ();
        loop_flag = 1;
skip002:
        if (IS_CLEAR(Frequested_dd_entry)) goto skip003;
        if (IS_SET(Fdd_definition)) goto skip003;
        if (IS_SET(Fdd_return_code)) goto skip003;
        if (IS_CLEAR(Fdd_silent_flag)) goto skip003;
        p9_1_2 ();
        loop_flag = 1;
skip003:
        if (IS_CLEAR(Fdd_definition)) goto skip004;
        p9_1_3 ();
        loop_flag = 1;
skip004:
        if (IS_CLEAR(Flook_dd_request)) goto skip005;
        if (IS_SET(Fdd_silent_flag)) goto skip005;
        if (IS_SET(Fdd_level)) goto skip005;
        if (IS_SET(Fdd_id)) goto skip005;
        p9_1_4 ();
        loop_flag = 1;
skip005:
        if (IS_SET(Fdd_entry_to_get)) goto skip006;
        if (IS_CLEAR(Fdd_level)) goto skip006;
        p9_1_5 ();
        loop_flag = 1;
skip006:
        ;
        } while (loop_flag);
}
```

The include file *flag.h* contains the global data declarations for each data flow and its associated queue state flag. All queue states are automatically initialized to empty (FALSE).

```
FLAG   Fdd_definition = FALSE;
FLAG   Fdd_entry_to_get = FALSE;
FLAG   Fdd_id = FALSE;
FLAG   Fdd_internal = FALSE;
FLAG   Fdd_level = FALSE;
FLAG   Fdd_return_code = FALSE;
FLAG   Fdd_silent_flag = FALSE;
FLAG   Fgdd_id = FALSE;
FLAG   Flook_dd_request = FALSE;
FLAG   Frequested_dd_entry = FALSE;
char   *dd_definition = {NULL};
char   *dd_entry_to_get = {NULL};
char   *dd_id = {NULL};
char   *gdd_id = {NULL};
char   *look_dd_request = {NULL};
char   *requested_dd_entry = {NULL};
int    dd_level = {NULL};
int    dd_return_code = {NULL};
int    dd_silent_flag = {NULL};
struct dd     dd_internal = {NULL};
```

Each module stub contains the correct external data declarations for the data flows used by a module, and the rudimentary C statements to make the file suitable for compiling. Even though all of the data flows are global variables in this implementation, each module can only access those data flows that are directly attached to its process bubble since other data flows are not explicitly declared. The following is an example of a mini-spec stub.

```
#include    "io.h"
#include    "error.h"
```

```
    BEGIN
    STATE(STATE0)
    END
}
```

Step 6 - Make Module Stubs

This step consists of the following command sequence:

mkproc filename

The program **mkproc** splits up *filename*, the module stubs produced by the previous step, into separate files. One file is created for each module. Having each module stub in a separate file permits better management of the system components and lets the user take advantage of UNIX utilities like *make*.

Step 7 - Code Modules

This step is the coding of the mini-spec from the algorithm described by the mini-spec body. The body of the mini-spec has been put into each module file as a comment to aid in this translation step.

In some instances the module for a mini-spec may have multiple internal states. The states are a means of introducing control inside the module. The need for multiple states arises when a module is used to control the sequence of execution of other modules. Macros are used to define states and state transitions. This allows the source code to remain readable.

Step 8 - Add "Or" Conditions

This step is optional. Some modules must execute even if not all of their inputs are set. Such modules must have their conditional invocation in the main program modified. These modifications consist of adding a logical "or" to the list of conditions preceding the module's call.

Step 9 - Add Argument Processing

This step is optional. It is required if the main program must obtain user-supplied parameters from the invoking command line. In this case, the designer must supply the code required to process the command line.

Advantages of This Technique

Generating code from the DFDs ensures that the specification is very close to the final code in the implemented product. If the specification is correct, the implementation will be correct.

The use of compile-time macros for the module entry, module exit, module state control, and queue state control makes it easy to add (and subtract) debug hooks into various parts of the system. The macros need simply be changed to include the desired debug print statements.

The conversion of DFDs to structure charts is skipped. This saves time. It also preserves the original information about the system. Usually DFDs are discarded after structure charts are drawn. This does not happen here.

Disadvantages of This Technique

Reading the code without the original DFDs is difficult. You must have the specification to understand the code.

```
#include      "globals.h"

/*
 ************************************************
 *
 * DFD        - 9.1 - Look up dd entries (lookdd cmd)
 * MINI SPEC  - 9.1.1
 * TITLE      - Get entry from DD
 * AUTHOR     - rainerw
 * DATE       - 6/26/85
 ************************************************
 */

/*
 * FLAGS
 */
extern FLAG   Fdd_id;
extern FLAG   Fdd_entry_to_get;
extern FLAG   Fdd_internal;
extern FLAG   Frequested_dd_entry;
extern FLAG   Fgdd_id;

/*
 * GLOBALS
 */
extern char   *dd_id; /* i */
extern char   *dd_entry_to_get; /* i */
extern struct dd     dd_internal; /* i */
extern char   *requested_dd_entry; /* o */
extern char   *gdd_id; /* o */

p9_1_1()
{
     /*
      ************************************************************************
      *
      * Get entry from DD
      * rainerw
      *  6/26/85
      * 9.1.1
      *
      * Repeat {
      *     if left hand side of dd entry != dd_entry_to_get {
      *           skip this dd entry.
      *     } else {
      *           SET requested_dd_entry = right hand side of dd entry.
      *           SET lhs_level = entry_level.
      *           SET lhs_dd_entry = left hand side from a dd entry.
      *           CLEAR  dd_entry_to_get.
      *           CLEAR  entry_level.
      *           Return.
      *     }
      * } until the entire dd_internal has been read one time.
      *
      * if (dd_silent_flag = FALSE) {
      *     print "Name <dd_entry_to_get> not found in DD <gdd_id>".
      * }
      *
      * CLEAR  dd_entry_to_get.
      * CLEAR  entry_level.
      * Return.
      *
      ************************************************************************
      */
```

The hierarchical nature of the DFDs is lost in the code. A flat, single-level DFD can be reconstructed from the main program, but the result is messy (much like a flat, detailed structure chart).

As with all new techniques, people have to learn how to use it. Maintainers of the product developed with this technique must understand SA.

Another disadvantage is that not all of steps are automatic. Thus, changes are still made to the code rather than in the specification. If all steps were automated, changes could be made only in the specification and the code would simply get regenerated.

Future Work

Many paths can be followed from here to extend the advantages and to reduce the disadvantages of this scheme. Eliminating the manual steps from Figure 1 seems like an obvious next step. High level data flows could be removed from the *types* without too much trouble. Data declarations, if contained in the data dictionary, could be automatically extracted. If mini-specs were written in a more structured way, the translation of mini-specs to code would be easier. "Or" conditions could be placed directly into the DFD with a graphics editor permitting correct mini-spec invocation conditions to be generated the first time.

Work has also been started on animating a DFD to monitor the execution of a program.

Summary

A technique used to build programs from data flow diagrams has been presented. Some of the steps in the technique are automated while others are manual. The authors are currently working on automating some of the manual steps.

There are significant implications for using this approach to develop programs for computers with multiple central processors. It would be possible to have each module execute on a separate hardware processor. In this way, CPU intensive programs could execute much faster.

References

1. DeMarco, Tom, *Structured Analysis and System Specification*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.
2. Page-Jones, Meilir, *The Practical Guide to Structured Systems Design*, pp. 182-183, Yourdon Press, New York, 1980.
3. Babb, Robert G. II, "Parallel Processing with Large-Grain Data Flow Techniques," *Computer*, vol. 17, no. 7, pp. 55-61, July, 1984.
4. Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, pp. 18-21, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
5. Pugh, J. R., "Actors Set the Stage for Software Advances," *Computer Design*, vol. 23, no. 10, pp. 185-189, Sept., 1984.

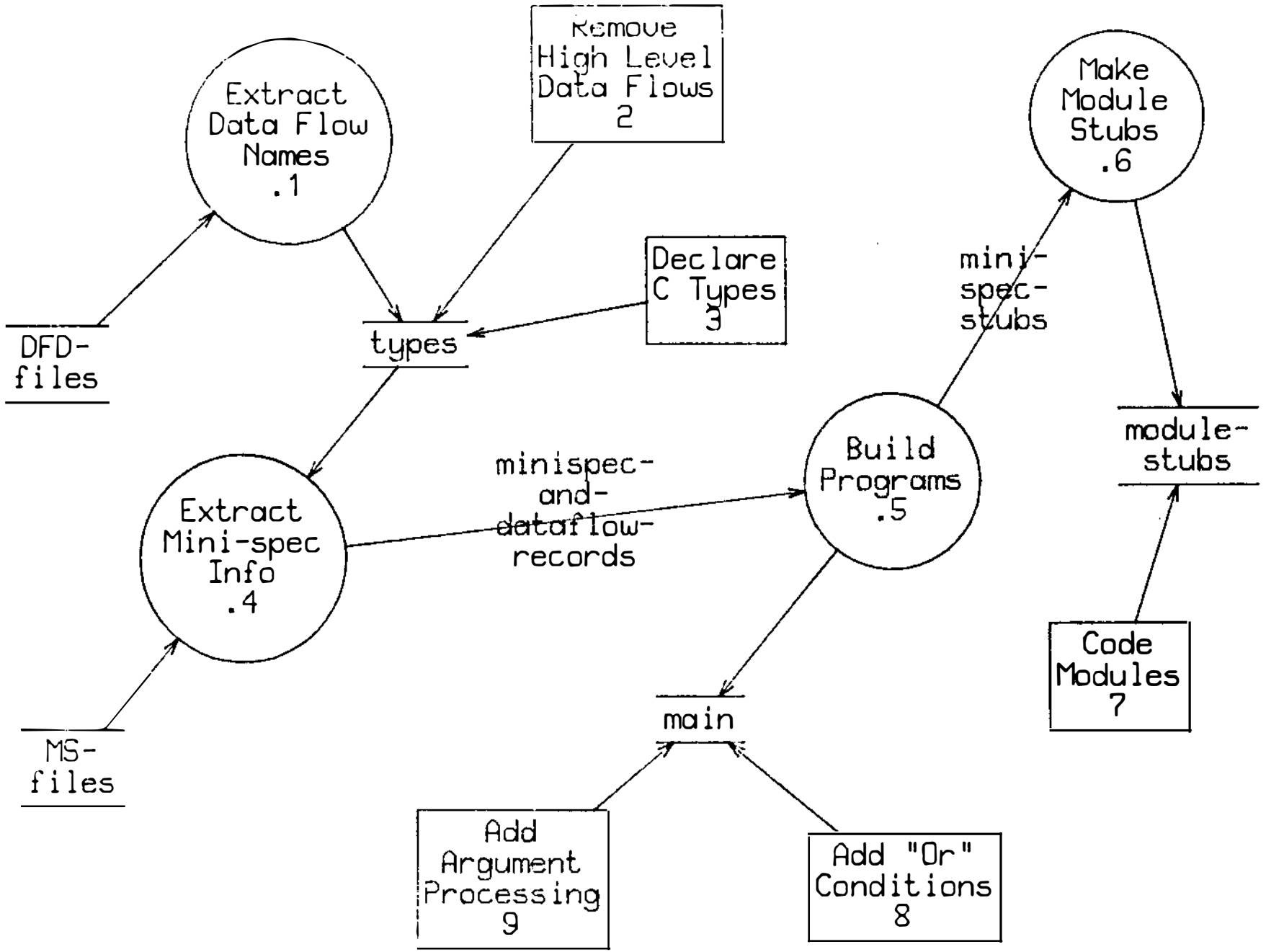
BIOGRAPHIES

Walter Webb, Rainer Wieland, and Chris Olson

Walter Webb is a software engineer manager in the Microprocessor Development Products Division at Tektronix. He has been employed at Tektronix since 1982. He is the project manager for the SA Tools Project. Previously he has worked for Federal Electric Corp., Aerospace Corp., AC Electronics, and Autonetics. He has an MS degree in systems management from the University of Southern California, an MS degree in electrical engineering from the University of California, and a BS degree in electrical engineering from the University of Santa Clara.

Rainer Wieland is a senior software engineer in the Microprocessor Development Products Division at Tektronix. He joined Tek in 1978. He is the project leader for the SA Tools Project. Earlier Mr. Wieland was a systems programmer with Motorola Microsystems. He has a BA degree in astronomy from Brown University.

Chris Olson is a software engineer in the Microprocessor Development Products Division at Tektronix. He started working with Tektronix immediately after receiving his bachelor's degree in computer science from Washington State University in 1983. For the past two years Mr. Olson has been a member of the team developing Structured Analysis Tools.



DFD 0 - Convert Data Flow Diagrams to Code

From DFDs to Code

- ☆ Make module stubs
- ☆ Code modules
- ☆ Add "or" conditions
- ☆ Add agrument processing

From DFDs to Code

- ☆ Extract data flow names
- ☆ Remove high level flows
- ☆ Declare C types
- ☆ Extract mini-spec info
- ☆ Build programs

New Model for Program Design

- ☆ State memory tracks queue usage
- ☆ MSs control own queues
- ☆ Program is done when no queues change state

New Model for Program Design

A module runs when:

- ☆ input queues full
- ☆ output queues empty

New Model for Program Design

The essential question:
When should a MS be run?

New Model for Program Design

- ☆ Based on the DFD
- ☆ Only MSs do work
- ☆ MSs run in parallel
- ☆ Data flows are 1-entry queues

IMPLEMENTING THE SOFTWARE REVIEW PROCESS

David J. Kerchner

Floating Point Systems, Inc.
P. O. Box 23489
Portland, OR 97223

ABSTRACT

Estimates as high as 40 to 60 percent of a software project's total lifecycle budget being spent for software maintenance are not uncommon. But effective implementation of the review process throughout the computer industry to help forestall the expense of this maintenance is not as widespread as one might believe.

The review process has been documented through many case studies, the results of which point to the fact that their effectiveness in helping reduce the number of software errors found during the pre-release software development phases cannot be ignored. The benefits to an organization far outweigh any possible negative aspects that software developers, from programmers to managers, perceive as being reasons for not using this readily available process.

Further, the review process is an invaluable tool for management to monitor the software development effort with a minimum amount of effort and cost to the organization (compared to today's software maintenance costs!). Moreover, it enhances interpersonal communication between developers, it's an excellent educational tool, and of course it helps produce better quality software.

The purpose of this paper is to examine the software review process and to outline the important steps that a QA group should take when implementing the software review process in order to reduce testing and post-release maintenance costs.

INTRODUCTION

A critical issue facing software developers is whether or not their products will meet the users' quality criteria (see BOE76, BOE78, POD85). In the past, software deficiencies could be covered up by the maintenance and support areas. Today's users are smarter. When offered a wide selection of available applications and systems, they invest their dollars in software that works.

Numerous QA standards may be written, but unless there's some mechanism to monitor their application and the developer's adherence to them, such standards are for all practical purposes useless. But if developers don't use quality

standards from the beginning, there's no assurance the final product will have any quality. So how does one ensure that quality, as predefined by the developer, is built into the software?

A proven and effective method for ensuring that quality objectives are addressed is the review process. QA should provide an objective viewpoint, through the review process, to bring into focus the predefined quality objectives.

THE REVIEW PROCESS

What is the review process?

The software development effort may be described as a multi-phased set of operations or functions, each operation or function resulting in a deliverable(s) for that phase. The review is that process by which each deliverable(s) is judged to be in conformity with a set of predetermined quality objectives.

Whether one uses the inspection, the structured walkthrough, the walkthrough, the review, and so forth, may be a matter of choice and practicality (or may be dictated by outside requirements). Each differs in its formality of approach and in the amount of quantitative data that can be extracted from the results. Generally speaking, the inspection method is the most formal, and most effective, method. But the others may be effective depending upon their implementation, application, or the deliverable under review. Each method can be tailored to a specific deliverable, or any one type may be applied to all project deliverables.

Excellent discussions regarding the definitions of the review process may be found in a number of works (e.g., YOU78 and FRE82), and a reader unfamiliar with the process is urged to do further research. These works cover the review process in detail with guidelines for its design and implementation, and just as importantly they consider the human element.

The characteristics which define the successful review program are organization and planning for each review conducted. Generally, all of the review processes incorporate the same ideas and procedures to some extent, but one, such as the inspection method, might stress them more explicitly than another, such as the walkthrough process. To illustrate this, Table 1 lists six explicitly defined steps which make up the inspection method.

When the inspection method is used, there is a strong dependency on role playing by the participants. On the other hand, a less formal peer review does not rely on such strict role playing for conducting the review session. In general, there are certain guidelines which should be followed to ensure that whichever type of review is held, it will be as successful as possible. We shall describe the more formal inspection process to illustrate the general procedure for conducting a review.

Table 1. Six Steps in the Inspection Process.

Step	Participant(s)	Objectives
1 Planning the Inspection	Moderator	Schedule review Distribute materials
2 Product overview	All participants	Familiarize inspection team with materials
3 Pre-inspection Preparation	Each participant	Examination of materials against checklists
4 Inspection/review	All participants	Error detection
5 Product rework	Product Designer	Correct errors in product
6 Inspection Follow-up	Moderator/Designer	Ensure defects corrected Feed-forward education Error analysis

Planning the inspection

Entrance criteria for the review must be defined; for example, a scheduled deliverable must be completed, such as a design specification. The completion of this deliverable serves as a trigger mechanism to initiate the review process. Other triggers might be a project manager's request to hold a review or a client's request.

Once triggered, the inspection moderator, ideally a member of the Quality Assurance group, assumes responsibility for scheduling and organizing the review and, with the designer's assistance, selects the other review participants. The number of reviewers may vary depending on circumstances such as the particular deliverables to be reviewed. Also, if any reviewer doesn't have the proper training in the review process, it's the responsibility of the QA group to provide such training.

The time and place for the inspection are set. It's important that the review be conducted in a room isolated from outside disturbances, that it be comfortable, and that it be adequately supplied with the proper blackboards, AV facilities, etc., necessary for the designer's overview presentation and for the inspection meeting itself. Also, it should be scheduled for a convenient time for all participants, and not, for instance, one hour before quitting time.

The moderator ensures that all deliverables to be reviewed are distributed to the inspection team several days before the review and that checklists and

reporting sheets are also distributed. It must be emphasized that any software code to be reviewed should already have been clean compiled; it's a waste of human resources to do the compiler's work. If a less formal review method is used, the moderator's responsibilities generally fall on the designer and/or other review participants.

Product overview

An overview of the deliverable(s) should be scheduled before the inspection meeting. This provides each reviewer with an understanding of the product and its intended function(s). It's essentially an educational session for the inspection team and it is usually conducted several days before the actual inspection. Less formal peer reviews may incorporate an overview into the actual review meeting to briefly familiarize the reviewers with the product, but its effectiveness at that time is questionable.

Pre-inspection preparation

Between the time of the overview and the inspection meeting, each reviewer examines the deliverables, evaluates them against the checklists provided, and makes notes regarding errors found so that these may be recorded in the inspection meeting. Reviewers should not attempt to provide solutions to errors discovered; that's the work of the designer and other software engineers. An advantage of checklists is that they provide objective criteria to evaluate the product. Otherwise, each reviewer is left to his or her own intuitive feeling as to what to look for in the product. If review participants are unable to prepare themselves ahead of time, they should excuse themselves from the review, and the review should be rescheduled, if necessary.

Inspection/review

At the beginning of the inspection the moderator should explicitly state the objectives for that particular inspection, and present the inspection agenda describing the sequence of events in the meeting. Since reviews/inspections may be tailored to a particular company's needs, there are variations to this process: a reader might be selected from among the reviewers to present the material rather than the designer, the designer might be the review moderator, checklists might not be used, and so forth. Generally, key guidelines to follow in conducting any type of review include:

- Optimum review duration is one hour, two hours maximum
- Unprepared reviewers should be excused and the review rescheduled, if necessary
- Focus on error detection, NOT error correction
- Review the product, not the person
- Record ALL errors/discrepancies
- Avoid discussions unrelated to the product under review
- Determine reviewers consensus for reinspection

Product rework

Whenever possible, the moderator should issue a report to the designer within

one working day of the inspection. This report lists the errors and discrepancies found during the review and their categorization with respect to the severity of impact on the reviewed product. The designer can then make the necessary changes to the product while the information is still fresh in his/her mind.

Inspection follow-up

In cases where a reinspection has not been scheduled, the moderator verifies that the proper changes have been made, and if satisfied, gives formal approval allowing the project development to continue. If the moderator feels the work should be reinspected, another inspection session may be scheduled. Continuing analysis of the error data can take place, and the results can be compiled to provide a future database of statistics regarding the software development process. Completion of all rework and follow-up activities define the end of one review cycle; the process is repeated when re-triggered.

In less formal reviews, the formal rework and follow-up work processes are generally omitted; it's assumed the designer will make the necessary corrections to the reviewed items. Also, the authority to schedule reinspections or to reject the rework does not usually exist within the less formal review processes.

Why the review process?

Project management requires there be some form of control over a software project's development effort (e.g., MET81). The review process gives management this needed control (FAG76, GLA84) over product quality much earlier in the project's lifecycle (Figure 1). Each scheduled review becomes a project milestone that must be passed, thus ensuring that the product's development is closely monitored and corrected, when necessary.

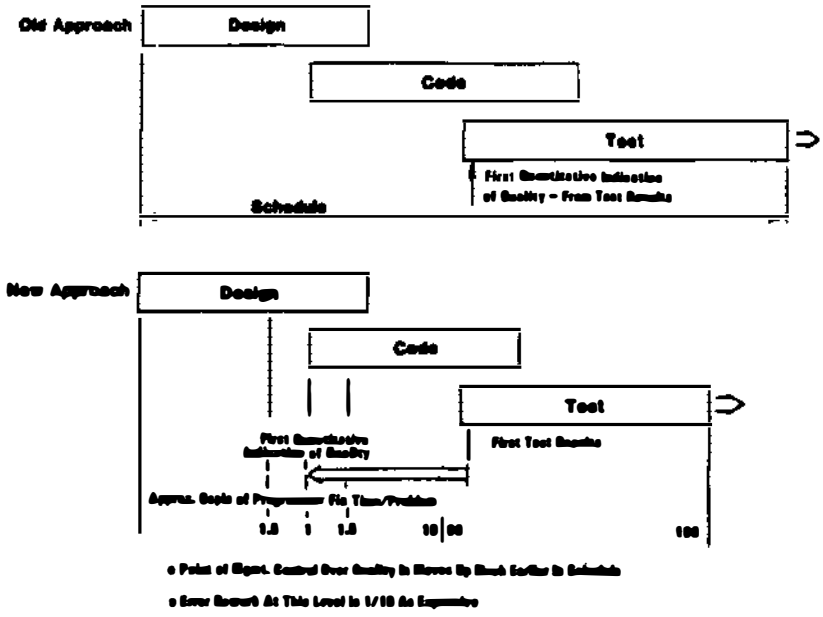


Figure 1. Enhanced Project Management Control (from FAG76).

The objective of the review is to find errors in the deliverable item(s) or product(s) being reviewed. Even after the review process, some errors may still be found. However, those errors detected at the earliest possible phase in the project's lifecycle will reduce the subsequent cost of testing the software before its release and the inevitable cost of its post-release maintenance. (It's not uncommon to hear numbers such as 40 to 60 percent of a project's total lifecycle budget being spent on maintenance.)

Many people still have the misconception that reviews are needed only for the actual software code, and programmers are too often blamed for the errors when the requirements and/or initial designs were really at fault. Yet problems arise in all phases of the development cycle (see HUG77). For example:

- o Immature, incomplete, or unvalidated requirements
- o Lack of traceability from requirements to operational software
- o Incomplete functional specifications, incomplete detail data dependencies not defined
- o Logic errors, unstated assumptions
- o Poor documentation
- o Changes in specifications

It is QA's responsibility to persuade management, project members et.al., of how effective reviews can be (CON85 and POD85). Logic, as well as simple economics, strongly suggest that a project be reviewed at all critical points and milestones in its lifecycle, not just during the coding phase (Figure 2). No matter what an organization/project defines its deliverables to be, (as commonly outlined in standard methodologies, DOD standards, IEEE standards, et. al.) each should be subject to review for conformance to the organization's quality standards. Otherwise, the proverbial wheel will continue to be reinvented!

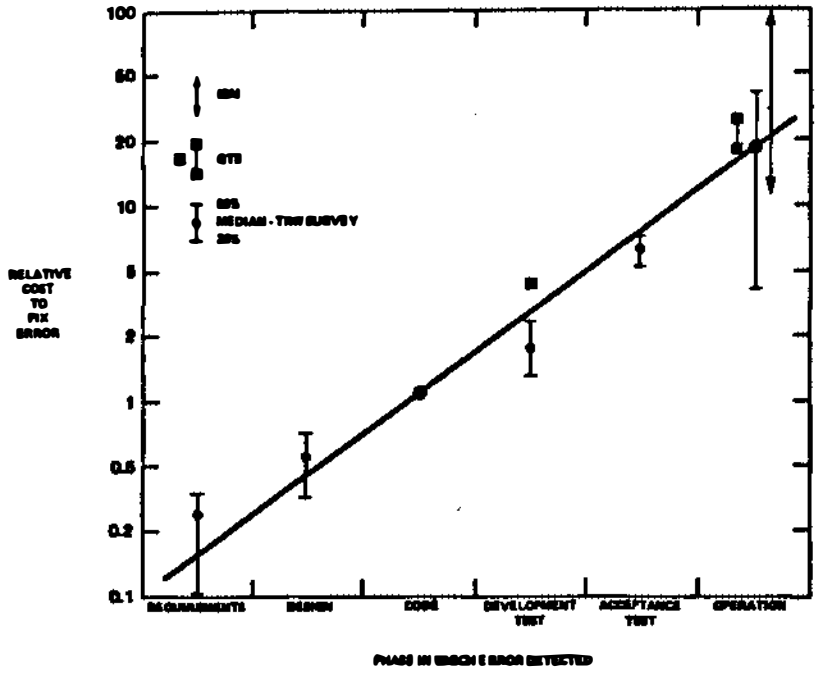


Figure 2. Relative Cost to Fix Defects by Project Phase.

Effectiveness studies

Studies done using the review process at a variety of companies point out some of the benefits of its use. For instance, at IBM (FAG76), one study project showed a 23 percent increase in productivity of the coding operation as compared to a control sample project. After conducting testing comparable to that done on a project of similar scope on which less formal walkthroughs had been used, the inspection project contained 38 percent fewer errors. The inspection method is in standard use there.

A case study conducted at Sperry-Univac recorded similar findings (HAR82). The project covered a period of one and a half years and involved writing 23,000 source statements in 165 code modules and 180 data modules. Strict adherence to one standard type of review was not followed; they used structured walkthroughs and later round-robin reviews as their schedule tightened. Yet the 90 percent of the product they reviewed accounted for only 25 percent of the significant error reports, whereas the 10 percent of the total (comparable) code not reviewed amassed 75 percent of the error reports.

IMPLEMENTATION OF THE REVIEW PROCESS

It's the responsibility of the Quality Assurance organization to oversee the review process implementation and to provide effective leadership for the review program (CON85). Commonly, the software development project teams are often in control of the review process. However, to provide that strong, yet objective voice to focus on quality issues during development, the QA group should ideally be in charge of this process. Therefore, to successfully establish a QA review process or to enhance an already existing one, the strategy should include certain key activities (ACK82, FRE82, YOU78).

- Management must decide to commit project resources in support of the review process and they must solidly support QA's efforts to implement the review process
- QA must plan the installation, execution, and evaluation of the review process
- QA must provide training to convey skills, technical information, and provide motivation for using the process
- The review process must be applied in a consistent, sustained effort

Although it's been said quality is free, every manager must allocate sufficient project resources for conducting the reviews that will ultimately further the program's success. If the commitment is there and reviews are scheduled as deliverable items or milestones themselves, then reviews will be respected.

Reviews become all the more effective as their visibility in the project increases.

Quality Assurance must decide which method, e.g., inspection, is most suitable for the company's purposes. The method selected must be clearly outlined as a company standard or guideline that has the full support of management as well as the software development teams.

Failure of the review process can frequently be attributed to a lack of understanding of the goals and methods of the process itself. The key to eliminating this problem is proper education of all project personnel. And although management doesn't normally participate in most reviews, they too should be educated as to the purpose and impact of the review process.

The objective of the review process is to enhance the development process itself and reduce the overall costs incurred during product development and maintenance. Furthermore, the review process allows management the opportunity to evaluate the quality of the product much earlier in its development cycle. In order to accomplish these objectives, the review process must be applied to the selected project(s) in a systematic, sustained fashion. There should be no special cases where deliverables are skipped over for review because of schedules, personal reasons, and so forth. To be truly effective, all deliverables must be reviewed.

The review process enhances and reinforces the idea in every software engineer's mind that quality is everybody's job. But when it comes to reviewing our own work, we tend to become overly protective, and therein lies an important source of resistance to the review process. Reviewing one's own work exposes it to others for constructive help, not destructive criticism. QA must ensure that reviews are 'egoless' and don't jeopardize the personal feelings of those whose work is under scrutiny. Once people perceive this, they tend to become more receptive to the review process and view it not as an outside imposition on them, but as a valuable tool.

To judge the effectiveness of the review program, QA must evaluate the efficiency and effectiveness of the review process itself. Feedback from project personnel and management will help to refine the process and correct any deficiencies. As statistics are gathered, they should be made available to the project and can serve as a basis for reviewing and evaluating the direction of future projects.

RESULTS

Preliminary results of using the review process at Floating Point Systems (FPS) are encouraging. Peer code walkthroughs have been the unofficial standard practice for some development groups and QA specification reviews have been conducted. Neither relied on checklists or other objective criteria to ascertain if quality objectives and issues were being addressed (the specifications were reviewed against existing standards).

On the project this writer was supporting, suggestions were made to improve the efficiency and effectiveness of the code reviews. An educational session was developed which would convey these ideas to all of the software engineers on the project and provide the necessary information needed to conduct effective reviews. (A number of these presentations have been given to other development groups as well).

Since most of the code had already been written for this project, the reviews conducted numbered only four. But 34 subroutines (approximately 4200 lines of code) were reviewed, and 237 items were recorded as possible discrepancies. Of these discrepancies, 156 were related to documentation problems and non-adherence to standards. The number of errors due to conflict with the existing programming standard emphasized the need to complete this standard's revision, which was already in progress. The remaining 81 discrepancies were addressed by the engineers and either corrected as errors or investigated further (not all turned out to be errors).

Some observations were made regarding these reviews. As one might expect, the more senior team members were far more knowledgeable and better prepared to participate in reviews, primarily because of their higher level of expertise; junior members were less likely to have review experience, and their contribution was perhaps less. In both cases, however, neither group, plus QA, could independently find all the defects (see an interesting study by Myers, MYE78). Checklists would help in this process. They can also help to define more specifically the types of errors to look for and to record and report the ones that were found. Yet even with their shortcomings, the reviews greatly enhanced each participant's understanding of both the product and the review process.

In addition, the manner in which software engineers design and develop software may be in conflict with the existing company standards (enforced by QA). In the case of the programming standard, what works best for the engineer was in conflict with the existing standard, so QA worked in harmony with the project to revise that standard. The point is that standards should be living documents; they too should be reviewed and updated as needed.

The review process at FPS is progressing and is currently in its second iteration. It's become apparent that a more formal approach should be taken in handling the review process, more along the lines of the inspection method. A review standard is being written, and in conjunction with this, checklists will be devised to make the review process more objective. It's felt this approach will give more quantitative results from the reviews and provide more precise definitions of the types of defects being reported. The educational process will also continue, helping to pave the way for general acceptance of the review process within the company. A real need, however, will be to have resources allocated from the beginning of a project for the purpose of conducting these reviews.

CONCLUSIONS

Because of the limited application of these reviews, the results achieved to

date have not been dramatic (considering the project's scope). On the other hand, any one of the errors detected could potentially have stopped the programs. Since only these few code reviews were held, they were not the most cost effective as far as the review program goes, but they set the future direction and hone the method. Greater benefits will result only from the systematic application of the review process over the complete development cycle of the software; management must realize this.

In summary, the review process should aid, not inhibit, the software development effort. An effective review program gives the Quality Assurance function a high degree of visibility that reinforces the concept of building quality into the product. Reviews serve as a positive standards reinforcement tool that will ensure uniformity during the development process (which later on eases maintenance problems). Further, reviews introduce the software engineers to the concept of software reviews, their importance in the development process, and the realization that quality is everyone's responsibility, not Quality Assurance's or one or two key project members. Finally, through an effective review program, management has a better assurance that quality issues have been addressed and that the final product will be as error free as possible.

"Quality doesn't happen, it's planned"

REFERENCES

ACK82 Ackerman, F., and Ackerman, A. "A software Inspection Training Program." Proceedings of COMPSAC, 1982

BOE76 Boehm, B., Brown, J., and Lipow, M. "Quantitative Evaluation of Software Quality." Proceedings, 2nd International Conference on Software Engineering, 1976

BOE78 Boehm, B., Brown, R., Kaspar, H., Lipow, M., MacLeod, G., and Merritt, M. "Characteristics of Software Quality." TRW Series of Software Quality, Vol 1, North Holland Publishing Co., 1978

CON85 Connell, J., and Brice, L. "Practical Quality Assurance." Datamation, March 1985

FAG76 Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." IBM Systems Journal, Vol 15, No 3, 1976

FRE82 Freedman, D., and Weinberg, G. "Handbook of Walkthroughs, Inspections, and Technical Reviews." Little, Brown and Company, Third Edition, 1982

GLA84 Glaser, G. "Managing Projects in the Computer Industry." Computer, October, 1984

HAR82 Hart, J. "The Effectiveness of Design and Code Walkthroughs." Proceedings of COMPSAC, 1982

HUG77 Hughes, J., and Michtom, J. "A Structured Approach to Programming." Prentice Hall, Inc., 1977

MET81 Metzger, P. "Managing a Programming Project." Prentice Hall, Inc., Second Edition, 1982

MYE78 Myers, G. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." Communications of the ACM, Vol 21, No 9, September 1978

POD85 Podolsky, J. "The Quest For Quality." Datamation, March 1985

YOU78 Yourdon, E. "Structured Walkthroughs." Prentice Hall, Inc., Second Edition, 1978

BIOGRAPHY

David Kerchner

David J. Kerchner is employed by Floating Point Systems of Beaverton, Oregon as a supervisor in the Methods, Standards, Quality Assurance department. He has earned both BA and MS degrees from Northwestern University in astronomy. He taught at two universities before continuing studies including computer science. He was employed by the McDonnell Douglas Automation Company of St. Louis, Missouri before coming to FPS.

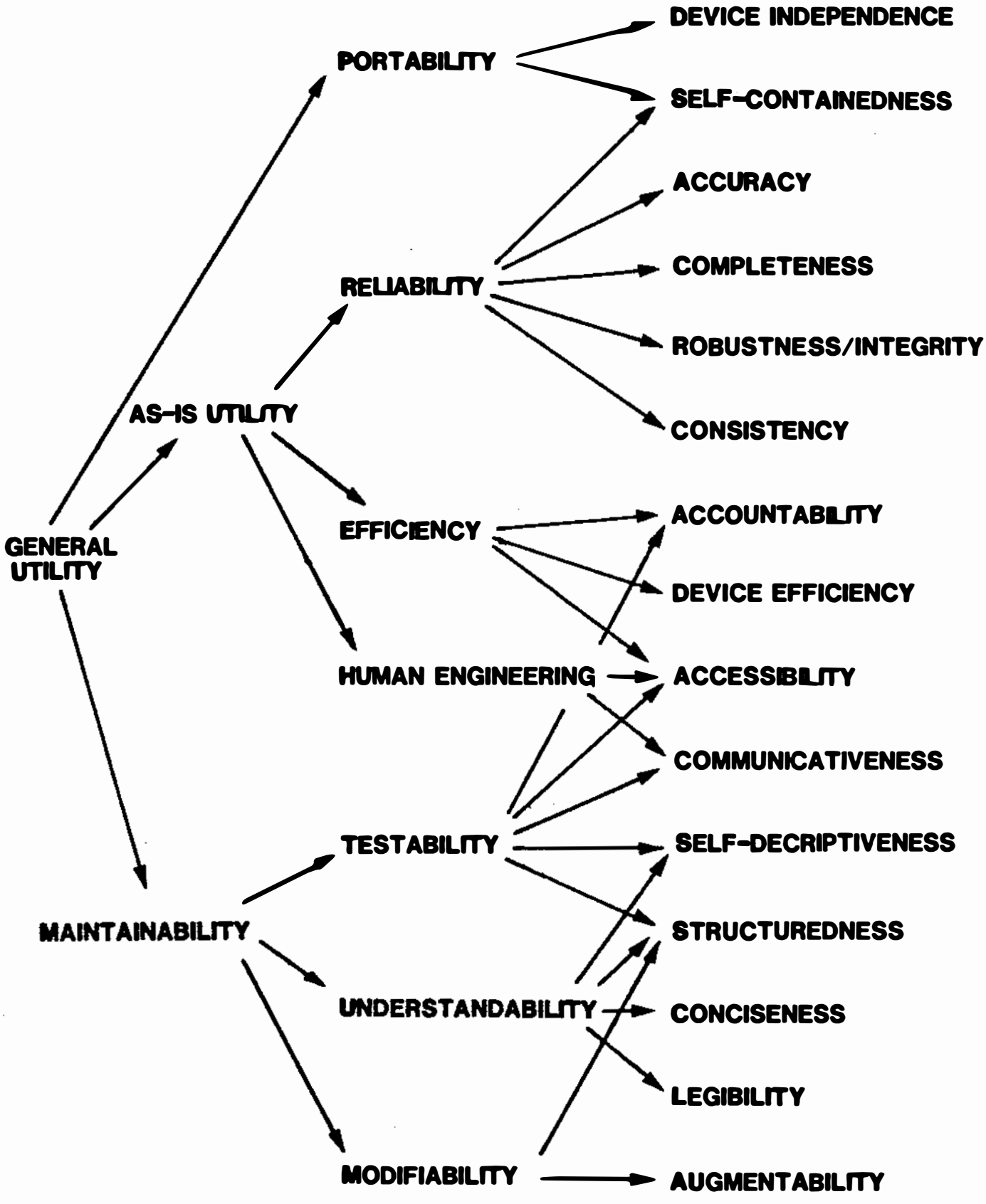
IMPLEMENTING THE SOFTWARE REVIEW PROCESS

BY

DAVID J. KERCHNER

FLOATING POINT SYSTEMS, INC.

SOFTWARE QUALITY CHARACTERISTICS TREE



REVIEW TYPE DIFFERENCES

- o FEEDBACK/FEEDFORWARD
- o CHECKLISTS
- o ERROR ANALYSIS
- o SIGNOFFS
- o QA/PROJECT DIRECTED
- o FORMALITY
- o ROLES
- o REPORTS

INSPECTION PROCESS

1. PLANNING	MODERATOR	SCHEDULE/DISTRIBUTE
2. OVERVIEW	ALL PARTICIPANTS	FAMILIARIZATION
3. PRE-INSPECTION	EACH PARTICIPANT	EXAMINE MATERIALS
4. INSPECTION	ALL PARTICIPANTS	ERROR DETECTION
5. PRODUCT REWORK	DESIGNER	CORRECT ERRORS
6. FOLLOW-UP	MODERATOR DESIGNER	ENSURE CORRECTIONS EDUCATION ERROR ANALYSIS

1. PLANNING

- o SCHEDULE OVERVIEW & INSPECTION
- o SELECT & NOTIFY PARTICIPANTS
- o ENSURE ACCEPTABLE ENVIRONMENT
- o DISTRIBUTE INSPECTION DELIVERABLES
- o DISTRIBUTE CHECKLISTS & FORMS

2. OVERVIEW

- OVERVIEW EDUCATIONAL SESSION
- GENERAL QUESTION & ANSWER

3. PRE-INSPECTION

- REVIEW DELIVERABLES
- RECORD DISCREPANCIES

4. INSPECTION

- STATE INSPECTION OBJECTIVES
- PRESENT AGENDA
- SELECT READER
- ENSURE PARTICIPANTS PREPARED
- ERROR DETECTION, NOT CORRECTION
- REVIEW PRODUCT, NOT PERSON
- RECORD ALL ERRORS
- AVOID EXTRANEOUS DISCUSSIONS
- DETERMINE SUBSEQUENT ACTION
- PARTICIPANTS SIGNOFF

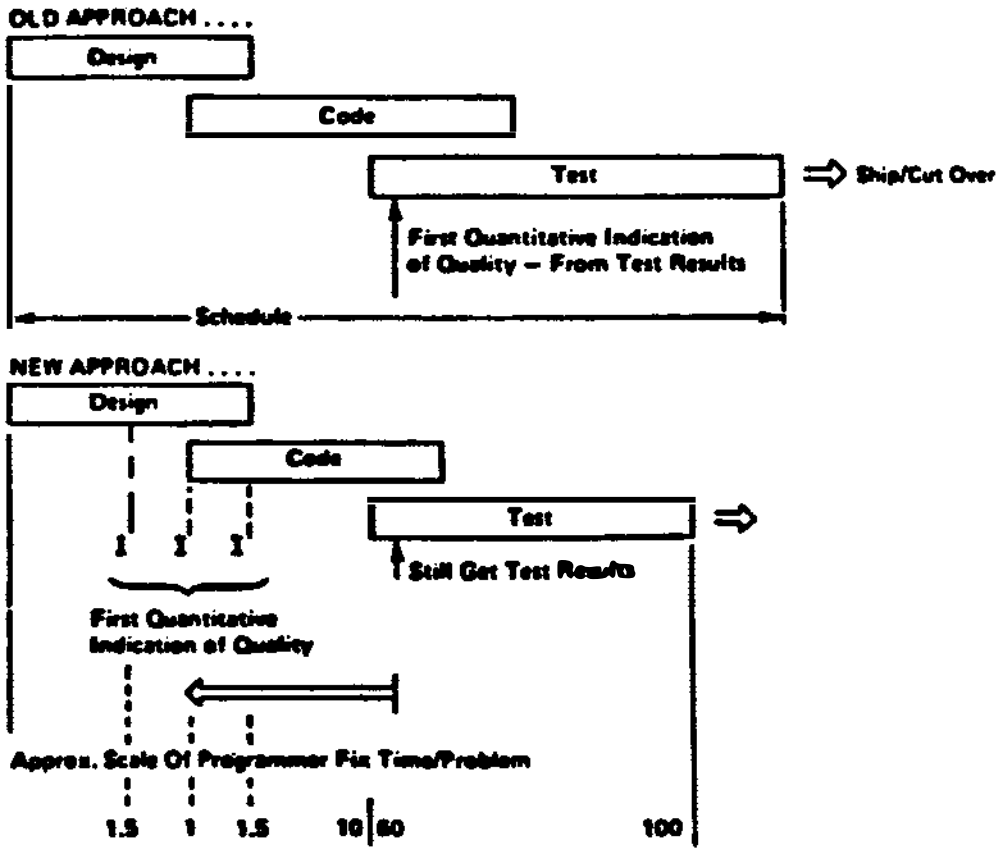
5. PRODUCT REWORK

- o CATAGORIZE ERRORS
- o INSPECTION REPORT
- o MONITOR REWORK

6. FOLLOW-UP

- o VERIFY CHANGES
- o SCHEDULE RE-INSPECTION
- o FEED FORWARD EDUCATION
- o ERROR ANALYSIS
- o STATISTICS
- o FINAL INSPECTION SIGNOFF

ENHANCED PROJECT MANAGEMENT CONTROL

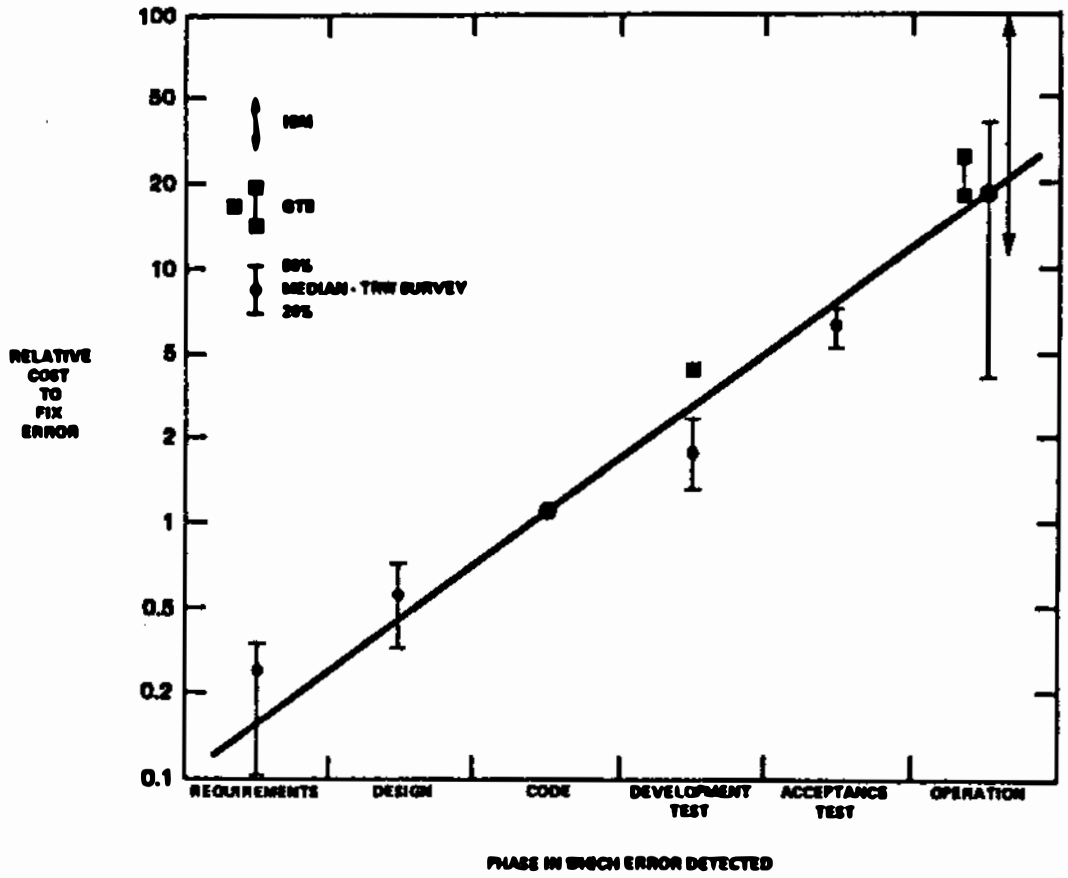


- Point of Mgmt. Control Over Quality Is Moved Up Much Earlier In Schedule.
- Error Rework At This Level Is 1/10 As Expensive.

DEVELOPMENT PROBLEMS

- o IMMATURE, INCOMPLETE, OR UNVALIDATED REQUIREMENTS
- o LACK OF TRACEABILITY
- o INCOMPLETE FUNCTIONAL SPECIFICATIONS
- o LOGIC ERRORS
- o POOR DOCUMENTATION
- o CHANGES IN SPECIFICATIONS

RELATIVE COST TO FIX DEFECTS BY PROJECT PHASE



EFFECTIVENESS OF THE PROCESS

- IBM
  - 23% CODING PRODUCTIVITY INCREASE
  - INSPECTIONS 38% FEWER ERRORS THAN WALKTHROUGHS
- SPERRY-UNIVAC
  - 165 CODE MODULES, 180 DATA MODULES
  - 23,000 SOURCE STATEMENTS (1 1/2 YR PROJECT)
  - 90% PRODUCT REVIEWED → 25% ERRORS
  - 10% PRODUCT NOT REVIEWED → 75% ERRORS

REVIEW IMPLEMENTATION

- **MANAGEMENT SUPPORT/PROJECT RESOURCES**
- **REVIEW PROCESS IMPLEMENTATION**
- **REVIEW PROCESS TRAINING**
- **CONSISTENT, SUSTAINED REVIEW APPLICATION**

FPS REVIEWS

- 34 SUBROUTINES (4200 LINES OF CODE)
- 237 POSSIBLE DISCREPANCIES
- 156 DOCUMENTATION ERRORS
- 81 OTHER ERRORS
- (OTHER REVIEWS NOT INCLUDED)

OBSERVATIONS

- SENIOR MEMBERS MORE EXPERIENCED
- TRAINING BENEFICIAL TO PARTICIPANTS
- NO CHECKLISTS REDUCED EFFECTIVENESS
- ERROR ANALYSIS DIFFICULT
- STANDARDS NOT ADHERED TO
- STANDARDS NEEDED REVISION
- QA ENHANCED REVIEW PROCESS

FUTURE DIRECTIONS

- FORMAL COMPANY REVIEW STANDARD
- USE OF CHECKLISTS
- CONTINUED EDUCATIONAL EFFORTS
- PROJECT RESOURCE ALLOCATION

SUMMARY

- REVIEW PROCESS AIDS DEVELOPMENT EFFORT
- QA HAS HIGH DEGREE OF VISIBILITY
- POSITIVE STANDARDS REINFORCEMENT TOOL
- QUALITY IS EVERYONE'S RESPONSIBILITY
- PROJECT MANAGEMENT HAS BETTER QUALITY CONTROL
- QUALITY PROJECTS/PRODUCTS

QUALITY DOESN'T HAPPEN, IT'S PLANNED

Pacific Northwest Software Quality Conference

Paper Abstract

George D. Tice, Jr.
Tektronix, Inc.
P.O. Box 4600 (M/S 92-525)
Beaverton, Oregon 97075
(503) 629-1310

Software Configuration Management: A Tool for Software Quality

Software configuration management is the software quality activity through which the products of the software development process are identified, labeled, controlled and accounted for. The four major functions of software configuration management involve:

- identifying the software product which includes the computer program, its documentation and associated data
- controlling changes to the software product
- reporting the status of the software product and all changes to it
- auditing the software product prior to release for production or delivery to a customer

This paper will present software configuration management as an important factor in the development of a quality software product. It will provide the reader with:

- an discussion of the need for software configuration management
- a definition of the functions of software configuration management
- an introduction to the techniques for performing software configuration management
- a summary of several standards for the practice of software configuration management

The underlying theme for the paper is that software configuration management as a major software quality activity is an important "tool" in the development of a software product within the management constraints of cost, schedule and performance.

Software Configuration Management: A Tool for Software Quality

George D. Tice, Jr.
Tektronix, Inc.
P.O. Box 4600 (M/S 92-525)
Beaverton, Oregon 97075
(503) 629-1310

Introduction

Software configuration management is the software quality activity through which the products of the software development process are identified, labeled, controlled and accounted for. The four major functions of software configuration management involve:

- identifying the software product
- controlling changes to the software product
- reporting the status of the software product and all changes to it
- auditing the software product prior to release for production or delivery to a customer

This paper presents software configuration management as an important factor in the development of a quality software product. It provides the reader with:

- a discussion of the need for software configuration management
- a definition of the functions of software configuration management
- an introduction to the techniques for performing software configuration management
- a summary of several standards for the practice of software configuration management

The underlying theme for the paper is that software configuration management as a major software quality activity is an important "tool" in the development of a software product within the project management constraints of cost, schedule and performance. (1)

The Need for SCM

The goal of software configuration management is to be able to reproduce the complete software configuration of a system for any specified version at any point in time starting with masters of all modifiable elements and specified rules of assembly. This goal reflects the needs of the various views of the software world depending on one's distance from the software development effort. This distance ranges from that of corporate or other upper level management through project management to the technical level of the actual software development team.

At the corporate or upper management level the primary concern is productivity. At this level software is considered a corporate asset to be kept, maintained and reused. Software configuration management is a tool which provides for the indexing, protection and availability of this software asset.

Project completion is the basic concern of project management. This involves the control of the products at each phase of the software life cycle. These products represent the what, the how, the answer, the evaluation and the changes of the software product. In other words, the products reflect the value added or resources expended at each phase of the software life cycle. Software configuration management is a tool for the protection of the value added to the software product.

At the technical level the software developer's concern is for the day-to-day job of creating the software product. This involves the generation of and changes to numerous software development documents and to the actual source code. It is at this level where the concerns for productivity and project completion are resolved. Software configuration management provides the software developer with the tools to achieve both software-engineering productivity improvement and software product integrity and quality. This is achieved through the control of the tapes, the disks, the listings and the documents produced by the software development team. This is accomplished on a day-to-day basis by the software development team. The success of the software configuration management effort is directly dependent on the ability to meet these real needs of the software development team.

Frequently these needs for software configuration management are unclear due to the transparency of the software itself as it passes through the software development process. An analogy may be made to the development of a more visible product - a new automobile. Consider the Ford Motor Company 1960 era development of the new Mustang moving from a concept to some 418,812 vehicles delivered to customers during the first year of production. Prior to the actual production the Mustang development team stated and restated the requirements for the new sports car. Once approved the requirements were expressed by Ford designers in a series of clay models which were reviewed by upper management. Selected designs were incorporated into a series of prototype vehicles which were subjected to both engineering and marketing tests. Only after the final design was selected and approved was the commitment made for the expensive retooling for the manufacturing capacity to mass produce the Mustang. It is at this point that the final configuration of the Mustang permitted creation of the necessary machine tools and assembly lines plus the training of people to enable profitable manufacture and marketing.

A software product's travel from concept to production is similar. Like the Mustang a software product will benefit from configuration management. This will permit the computer software reuse and increased productivity corporate management desires.

The Functions of SCM

Software configuration management is a four step process designed to meet the needs of corporate, project and technical managers. First the configuration items in a system must be identified and defined. This is followed by the control of the release and change of these items throughout the system life cycle. Concurrently, the status of configuration items and change requests must be recorded and reported. Finally the correctness and completeness of configuration items must be verified. Thus, the four functions of software configuration management are:

- configuration identification
- configuration control
- configuration status accounting
- configuration audit

Configuration identification is the process of designating the configuration items in a system and recording their characteristics. For the uninitiated a configuration item is a collection of hardware or software elements treated as a unit for the purpose of configuration management. Characteristics include the identification of the person responsible for the configuration item, its logical contents, its physical and control identification, and any special relationships. In addition to the code the configuration item must include the approved documentation that defines the configuration item and the approved technical documentation as set forth in specifications, drawings or associated lists.

Configuration control is the process of evaluating, approving and coordinating changes to configuration items after formal establishment of their configuration identification. This formal establishment of the configuration identification is typically accomplished by a configuration change board (CCB). The CCB normally is chaired by the program or project manager and has a membership which represents all the vested interests in the configuration item. At a minimum the CCB should include software development, quality assurance, maintenance, manuals, manufacturing and configuration management representatives. CCB approval or acceptance of the initial configuration and changes thereto establishes a baseline for the configuration item. This function is not meant to prevent or even inhibit change. Rather in recognition that there will always be change the intention is to provide for both orderly change and for integrity in the configuration item as value is added during each step of the software development process.

The configuration status accounting function provides for the recording and reporting of the information that is needed to effectively manage a configuration. This must include a listing of the approved configuration identification, the status of proposed changes and the implementation of approved changes. This information must be provided to every person with an interest in the configuration.

The configuration audit is the process of verifying that all required configuration items have been produced, that the current version agrees with specified requirements, that the technical documentation completely and accurately describes the configuration items and that all change requests have been resolved. In-process audits may be conducted during the software development process. At project completion both functional and physical audits are conducted. These check that the software product meets requirements and has all of its physical elements present.

The Activities of SCM

Software configuration management is performed throughout the software life cycle from project preparation through software maintenance. These activities involve product management - the control of the software product through its evolution and maintenance. They are not project management - the control of the organization which develops the software product. They are not support management - the control of the process which is used to develop the software product. These activities can be viewed from either the software configuration management function or a software life cycles perspective.

From the perspective of each of the software configuration management functions, activities are either planned or conducted. The following list includes both the planning and conducting activities for each of the four software configuration management functions:

Configuration Identification

Plan

- establish rules for titling, labeling, numbering, cataloging
- define baselines to be established and their documentation
- establish procedures for preparation, approval, control and maintenance of all software code and documentation

Conduct

- label all SCI documentation and code
- define and update the set of SCI's
- identify and record all SCI dependencies
- list SCI's in a baseline
- list current software configuration

Configuration Control

Plan

- describe level of authority for change approval in each life cycle phase
- define methods and procedures for processing change proposals
- define role for each CCB and other change management bodies
- state methods to be used for configuration control of interfaces with programs and projects
- state the control procedures for associated special software products in support and/or vendor software

Conduct

- evaluate and record changes
- assure all required changes are implemented
- propagate changes throughout the software configuration

Configuration Status Accounting

Plan

- delineate how information on status is to be collected, verified, stored, processed and reported
- identify the periodic reports to be provided
- state what or establish query capabilities
- describe any special status accounting requirements specified

Conduct

- record the establishment of each SCI
- record the establishment of each baseline
- record changes to SCI's and baselines
- track and report change request processing
- track and report the status of all SCI's

Configuration Audit

Plan

- identify the review (internal/formal) and audits to be held during the life cycle
- define SCM's role in reviews and audits
- identify the configuration items and associated documentation and/or software to be covered in each identified audit
- establish the rules for audit and review agenda, action item reporting and follow-up by SCM

Conduct

- approve baseline composition and functionality
- determine differences between baselines

The software life cycle permits a somewhat different view of software configuration management activities. This list includes the activities performed during each phase of the software life cycle:

Concept Exploration (Project Preparation)

- determine scope of SCM for project
- establish project documentation scheme
- create SCM plan

Requirements Phase

- establish control procedures and organization
- prepare the project master data base and tools
- acquire project plans
- acquire systems requirement documentation

Design Phase

- save and distribute system requirements documentation
- control updates
- acquire system architecture documentation including interface and data base specifications
- begin project system structuring
- acquire system test documentation

Implementation Phase

- save and distribute system architecture documentation
- control updates
- acquire module design documents
- complete product system structuring
- acquire code and unit test documentation

Test Phase

- save and distribute code for product system versions
- control updates
- acquire system test results

Maintenance Phase

- save and distribute product system versions
- control updates
- acquire system enhancements in documents, code and data

The desired result of the total software configuration management effort as represented by the above lists of activities is an environment which makes the overall software development process more stable by establishing baselines and controlling change. This will enhance the probability of delivering a quality software product on time and within budget.

The Implementation of SCM

In those instances where software professionals and managers must initiate software configuration management an implementation approach must be selected. Organizations which have specific standards and/or guidelines specified in contracts or similar requirements have the relatively easy task. They can, and most likely must, follow the process and procedures stated in the contract and the specified standards and/or guidelines. For those organizations that lack this luxury the following is a suggested approach to implement software configuration management:

- define a long term goal
- define the environment
- select a project for trial
- train a team
- implement the trial project
- measure and evaluate the results

Mandated or not, software configuration management should be able to stand the test of making a positive contribution to both the software project and the software product. When there is no mandate it is critical that a long term goal for software configuration management be established. This goal should reflect both project management concerns for cost, schedule and performance and upper management concern for the bottom line. Therefore a software configuration management goal should be stated in terms of specific contributions to both software projects and products. This goal should be measurable in terms of cost (budget), schedule, performance (functionality and quality) and return on investment (ROI).

It is essential that the environment for software development be defined. Caution must be taken to avoid situations where software configuration management is being looked upon as a miracle cure for an otherwise unenlightened software development environment. In such a situation even the best implementation of SCM is unlikely to succeed. Should such an environment be discovered every effort should be made to upgrade the overall environment. Otherwise it is best to seek a better situation for the initial SCM effort. If one must continue to work in an unenlightened environment the situation should be reflected by stating very conservative long term goals.

The next step is to select the candidate software project for trial. The selected project will determine both the schedule for the project and for implementing SCM. It will also determine the staff who will participate in the SCM trial and therefor must be trained. If there is an option the selected project should be one with good opportunity for successful completion within a year. It should also be staffed with software development personnel who are ready to accept software configuration management contributions to their project effort. In any event project selection is a major factor in the success or failure of the SCM trial and should be handled with due care and consideration.

Once the project is selected the next task is to train the team with the software configuration management background appropriate to each individual job. In addition to sharing SCM knowledge this training must get the entire team involved to the extent that each person knows what is expected, why it is expected and what is in it for him/her. This should result in the team's "buy in" to the SCM trial. The training should be selective in timing, subject, detail and audience.

The actual implementation one the selected project should come just after the selective training and with consulting support to the project team. Remember, it is the project schedule that will drive the SCM trial not vice versa.

In order that the SCM trial can be adequately evaluated statistics on effort, dates and benefits must be kept throughout the trial. At appropriate times during and on completion of the trial these statistics should be evaluated to determine if the trial has been successful. To be successful the SCM trial must indicate a measurable improvement to the process - that is: SCM's contribution to the product and the project must be quantified and should result in savings greater than cost.

Standards for SCM

At this writing that are two sources of general standards for software configuration management. They are the numerous directives and standards issued by the U. S. Department of Defense (DoD) and the IEEE Standard 828-1983, IEEE Standard for Configuration Management Plans.

The recently issued DoD Software Development Standard (DoD-STD-2167) deals with specific software configuration management issues. By reference, it also invokes several other general configuration management directives and standards. These include:

DoD-D 5010.19 - Configuration Management: directs the implementation of CM in the DoD and DoD agencies Configuration Management - Joint Regulation Navy NAVMATINST 4130.1A, AF-AR 65-3, ARMY AR 70-37, NSA CSS 80-14, MCO-4130.1A, DCAC 100-50-2, DNAINST 5010.18, DSAR 8250.4: a joint policy on configuration management that defines CM and its role throughout contractual phases and requires implementation of CM by DoD services and components on contracts.

DoD-STD-480 - Engineering Changes, Deviations and Waviers: provides direction for dealing with engineering change proposals (ECP), waviers and deviations.

MIL-STD-483 (USAF) - Configuration Management Practices for Systems, Equipment, Munitions and Computer Programs: provides for a configuration management plan which is adaptable to hardware and software application and contains guidance relative to configuration items (CI) and computer program configuration items (CPCI). (This standard is being updated to incorporate changes necessary to support DoD-STD-2167.)

MIL-STD-1521 (USAF) - Technical Reviews and Audits for Systems, Equipment and Computer Programs: provides detailed guidance for the conduct of reviews and audits (This standard is being updated to incorporate changes necessary to support DoD-STD-2167.)

The IEEE Standard 828-1983, IEEE Standard for Software Configuration Management Plans is one of the series of IEEE Software Engineering Standards being developed by volunteers under the auspices of the Technical Committee for Software Engineering in the IEEE Computer Society. As with all IEEE standards this standard is voluntary and is invoked only as desired by the organization doing the software development. However at such time that compliance with the standard is declared the directives of the standard become mandatory. IEEE Standard 828-1983 provides for the development of a software configuration management plan with the following outline:

Introduction

- Purpose
- Scope
- Definitions and acronyms
- References

Management

- Organization
- SCM responsibilities
- Interface control
- SCMP implementation
- Applicable policies, directives and procedures

SCM activities

- Configuration identification
- Configuration control
- Configuration status accounting
- Audits and reviews

Tools, techniques and methodologies

Supplier control

Records collection and retention

Conclusion

Software configuration management is not merely a collection of specified tasks assigned to a few clerks. Rather, it is a way - a tool - for all project team members to develop a software product that is complete in its parts, and consistent in and traceable through all representations. To this end, software configuration management provides visibility of the evolving software product to both the entire software development team and the customer.

References

-----, IEEE Standard 828-1983, IEEE Standard for Software Configuration Plans, IEEE

Buckle, J. K. Software Configuration Management, MacMillan Press Ltd, 1982

Bershoff, Edward H., Vilas D. Henderson & Stanley G. Seigel Software Configuration Management, Prentice-Hall, 1980

(1) This paper is an overview of a portion of a book entitled "Software Quality Control - Practices and Procedures" in preparation for publication by Prentice-Hall.

BIOGRAPHY

George Tice, Jr.

George D. Tice, Jr. is a senior software engineer in the Micro-Computer Development Products Division of Tektronix. He is preparing a software development methodology to support MDP's software and microcomputer development products, and is involved in numerous software quality and productivity improvement projects. Prior to joining Tektronix he managed software quality control projects at the Naval Ocean Systems Center in San Diego, California. Mr. Tice is Chair of the IEEE Computer Society Software Engineering Standards Subcommittee. He received his BS degree from Pennsylvania State University and an MPA degree from San Diego State University.

GOAL OF SCM

To reproduce the complete SOFTWARE CONFIGURATION of a system

- for *any* specified *version*
- at *any* point in *time*

Starting with

- *masters* of all *modifiable elements*
- *methods* of *construction*

THE NEED FOR SCM

Corporate Management

→ Productivity

Project Management

→ Project Completion

Technical

→ Day-to-Day Job

WHY SCM — Corporate Management

Software is a corporate asset to be

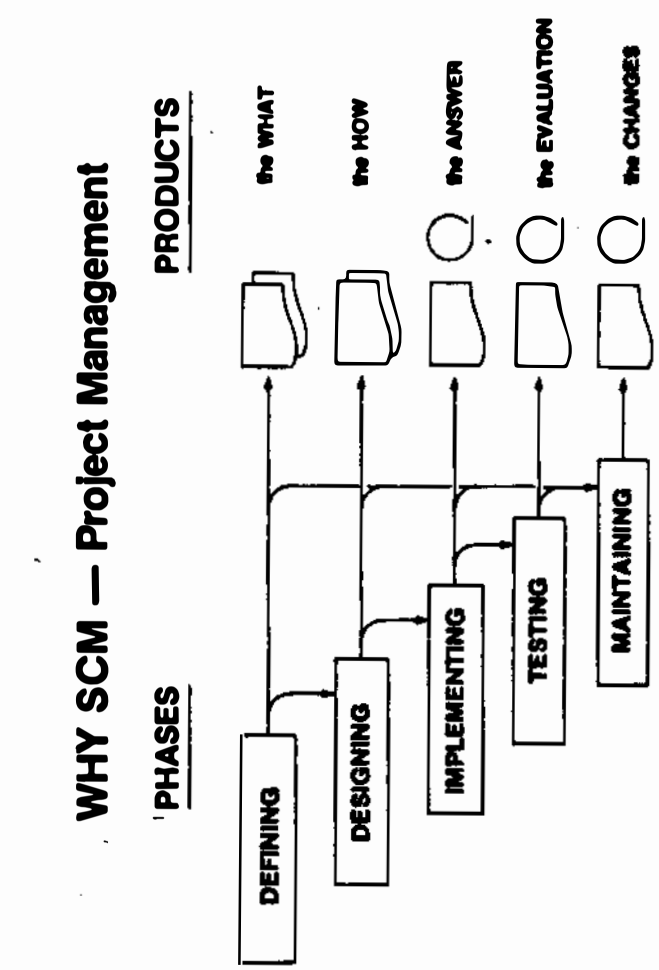
- kept
- maintained
- REUSED

The software assets must be

- indexed
- protected
- AVAILABLE

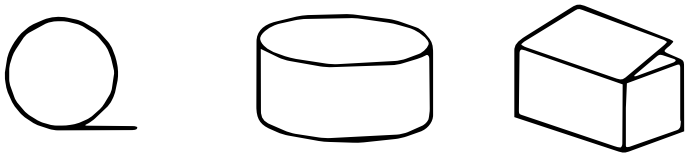
→ SCM

WHY SCM — Project Management



What and where are all of the product pieces on which we spent our resources?

WHY SCM — Technical



TAPES DISKS LISTINGS

Where's the source?
Is this the right listing for that file?
Did you have a backup copy of that program somewhere?
Where are the build instructions?
Why are all the function library routines marked as "missing externals"?

THE FOUR FUNCTIONS OF SCM

- Configuration Identification
- Configuration Control
- Configuration Status Accounting
- Configuration Audit

CONFIGURATION ITEM

A collection of hardware or software elements treated as a unit for the purpse of configuration management.

CONFIGURATION IDENTIFICATION

(1) The process of designating the configuration items in a system and recording their characteristics.

(2) The approved documentation that defines a configuration item

(3) The current approved or conditionally approved technical documentation for a configuration item as set forth in specifications, drawings, or associated lists, and documents referenced therein.

CONFIGURATION CONTROL

(1) The process of evaluating, approving or disapproving, and coordinating changes to configuration items after formal establishment of their configuration identification

(2) The systematic evaluation, coordination, approval or disapproval, and implementation of all approved changes in the configuration of a configuration item after formal establishment of its configuration identification. (DOD STD 480 A)

CONFIGURATION STATUS ACCOUNTING

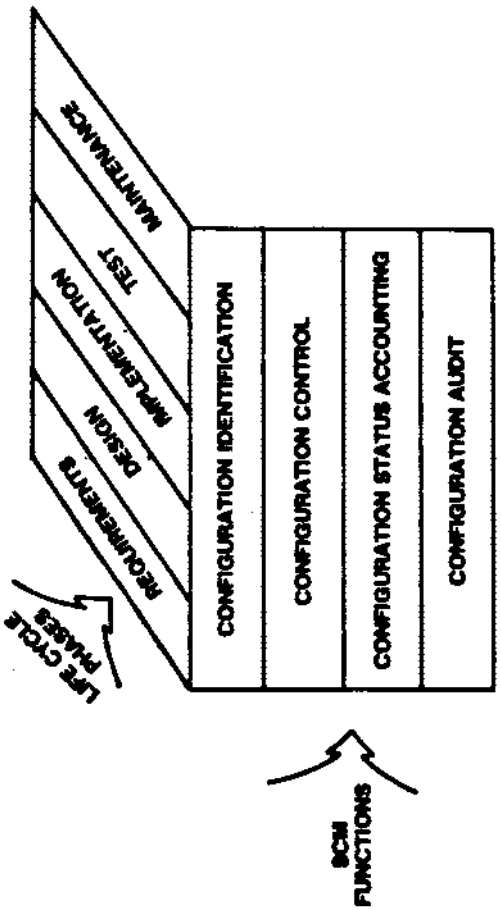
The recording and reporting of the information that is needed to manage a configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of approved changes.

(DOD STD 480 A)

CONFIGURATION AUDIT

The process of verifying that all required configuration items have been produced, that the current version agrees with specified requirements, that the technical documentation completely and accurately describes the configuration items, and that all change requests have been resolved.

SCM ACTIVITIES THROUGH THE LIFE CYCLE



SCM Is

product management -	controlling the software product through its evolution and maintainance
not project management -	controlling the organization which develops the software product
not support management -	controlling the process which is used to develop the software product

IMPLEMENTING SCM

- A change in how the Organization works affects everyone and everyone needs to "Buy In" to the change
- Change involves
  - Defining long term goal
  - Defining present or should be environment
  - Select trial project
  - Train team
  - Implement trial
  - Evaluate SCM Program

MILITARY STANDARDS

DOD-D 5010.19 — CONFIGURATION MANAGEMENT

CONFIGURATION MANAGEMENT — JOINT REGULATION NAVY—NAVMATINST 4130. 1A, AF-AR 65-3, ARMY AR 70-37, NSA CSS 80-14 MCO-4130.1A, DCAC 100-50-2, DNAINST 5010.18, DSAR 8250.4

DOD STD 480 — ENGINEERING CHANGES DEVIATIONS AND WAIVERS

MIL STD 483 (USAF) (NOTICE-2) — CONFIGURATION MANAGEMENT PRACTICES FOR SYSTEMS, EQUIPMENT, MUNITIONS AND COMPUTER PROGRAMS.

MIL STD 1521 (USAF) — TECHNICAL REVIEWS AND AUDITS FOR SYSTEMS EQUIPMENT & COMPUTER PROGRAMS.

IEEE Standard for Software Configuration Management Plans

OUTLINE

(1) Introduction
- (a) Purpose
- (b) Scope
- (c) Definitions and acronyms
- (d) References

(2) Management
- (a) Organization
- (b) SCM responsibilities
- (c) Interface control
- (d) SCMP implementation
- (e) Applicable policies, directives and procedures

(3) SCM activities
- (a) Configuration identification
- (b) Configuration control
- (c) Configuration status accounting
- (d) Audits and reviews

(4) Tools, techniques, and methodologies

(5) Supplier control

(6) Records collection and retention

SOFTWARE CONFIGURATION MANAGEMENT

- provides visibility of the evolving software product
  - to the entire project development team
  - to the customer

- assures that each successive refinement of the software product is
  - complete with all of its configuration items
  - consistent between all of its configuration items

Session 2

METATOOLS

Titles and Speakers:

"Reduced Form for Sharing Software Complexity Data"
Warren Harrison, University of Portland, and Curtis Cook, Oregon State University

"A Practical Guide to Acquiring Software Engineering Tools"
Tom Milligan, Tektronix, Inc.

"The Use of Software Metrics to Improve Project Estimation"
Bob Grady and Debbie Caswell, Hewlett Packard Co.

REDUCED FORM FOR SHARING SOFTWARE COMPLEXITY DATA

Warren Harrison

Curtis Cook

One of the most important aspects of program quality is how easy it is for a programmer to understand a program. Software complexity metrics are a method of quantifying the understandability (of lack thereof) of a program. The goal of researchers in this area is to develop measures that can assist in estimating the difficulty of a programmer performing a task on the software such testing or maintenance.

In order to study and compare the performance of measures, researchers need data from "real world" software systems. However, industrial and business organizations are often reluctant to provide the needed data. They are especially reluctant to provide researchers with copies of their source code because they have a considerable investment in the code and the obvious security problems. Unfortunately, this data is essential to the work of the software complexity researcher.

The Reduced Form of a source program provides the researcher with information about the characteristics of the code without disclosing the code. For each subprogram, the Reduced Form provides a list of the program characteristics that are of interest to complexity researchers. The actual program cannot be reconstructed from this information because the operands and operators in each statement and the order of the statements cannot be inferred from the information.

Programs that automatically generate the Reduced Form for several high level languages have been developed. In addition to providing a relatively secure method of sharing data, the format of the Reduced Form makes it trivial to compute most of the common metrics such as McCabe's V(g) and Halstead's E. We hope that the use of a convenient tool such as the Reduced Form will contribute greatly to the development and encourage the use of software complexity metrics.

SOME RESULTS FROM USING A REDUCED FORM FOR SHARING SOFTWARE COMPLEXITY DATA

Warren Harrison
University of Portland
Portland, OR 97203

Curtis Cook
Oregon State University
Corvallis, OR 97331

1. Introduction

Only recently, has the importance of writing understandable software been acknowledged to be as important as program efficiency. This importance stems from the high cost of software maintenance (estimated to consume up to 70% of the total amount spent on software) and the great amount of time spent on testing (estimated to be up to 50% of software development time) [1].

Software complexity metrics are one approach to an objective measure of the understandability of a piece of software. These metrics are based on the hypothesis that the difficulty of understanding a piece of software depends on a set of characteristics of the software, and the degree to which these characteristics are present. For example, it is widely believed that a large number of IF statements make a program more difficult to understand than a similar program with fewer IF statements. Unfortunately, there is no consensus as to exactly which characteristics contribute most to software complexity. As a result, many software complexity metrics have been proposed over the last several years.

Some of the more popular metrics include the Cyclomatic complexity of McCabe [2] and Halstead's software science [3].

In order to validate a complexity metric (ie, find out if it really "works") one must determine if a particular characteristic or set of characteristics which the metric measures actually has an effect on program understandability. Typically, one of two approaches are taken in the validation of a metric:

(1) Controlled Experimentation

(2) Field Studies

In controlled experimentation, two or more versions of the same program are prepared, each with differing degrees of

the characteristic being studied. For example, one version might use detailed comments while the other version may contain only high-level comments. A number of subjects are recruited and asked to perform some programming task that is thought to be affected by understandability, such as correcting an error or answering some questions about what the program does. The subjects' performance on the task (eg, time required to fix the error, or number of correct answers) is then analyzed to assess which version was easier to work with. If the metric is a true measure of software complexity, it should agree with the observed outcome of the experiment.

Controlled experimentation possesses some major weaknesses. To allow the subjects to complete the experiment within a reasonable amount of time, only small programs can be studied. Often, the programs used contain less than 50 lines, and usually perform fairly trivial operations (eg, finding a mean). More importantly, since most experiments are performed at universities, students are most often used as subjects. It is not clear that the results of such experiments can be generalized to large software systems written by professional programmers.

In field studies, data is collected from one or more "real world" systems and analyzed. The data includes software characteristics and the degree to which they occur, as well as performance measures of programmers doing typical tasks such as debugging, testing or maintenance. The analysis attempts to determine significant relations between the software characteristics and the performance measures.

While field studies have a few weaknesses, such as difficulty in finely controlling the variables being studied, results tend to be more generalizable to industrial applications, and more credible to programming managers than small, academic experiments. Unfortunately, the major difficulty in field studies is obtaining accurate data - both program characteristics and performance measures.

2. Acquiring Field Data to Validate Metrics

Many organizations are reluctant to allow access to their code systems by "outsiders". This is understandable since it would entail circulating copies of source code which may have taken them thousands of man-hours to develop. Even though the researcher may not provide the source code to others, the mere distribution of the code to researchers outside the organization could jeopordize any "trade secret" protection it, or algorithms and formulas it contains may

possess [4].

In a recent survey of approximately 40 industrial organizations, only 35% of the respondents felt their organization would share actual source code with researchers (see Table I). Obviously, this would make obtaining industrial data quite difficult for academic researchers who are not affiliated with any industrial organization.

To overcome this problem, we have recently proposed a Reduced Form which provides information on the software characteristics of interest to metric researchers, but which prevents the reconstruction of the original source program [5]. A example of this Reduced Form for C is shown in Figures 1 and 2.

In [5], we present evidence which suggests that while most current metrics can be obtained from the Reduced Form, it is all but impossible to reconstruct the original source code from the Reduced Form. Seventy percent of the respondents in the previously mentioned survey would be willing to share field data in its Reduced Form, or double the number who would be willing to provide the actual code (see Table I).

The Reduced Form we have developed should be viewed as a prototype of a more refined version which will be developed in concert with other researchers. In addition to developing a more refined version of the Reduced Form, we must also address the need for a similar device to allow programmer performance data to be distributed to researchers. We plan to focus our efforts on this, once we have developed a more refined version of the Reduced Form.

3. Results of a Study Using the Reduced Form

In early 1984, we implemented a prototype version of a Reduced Form generation tool which worked for programs written in C. A number of organizations expressed interest in participating in our study, but for various reasons, we decided to limit our initial study to a single project within a single organization. The project we decided to study involved a major compiler development effort involving about 30,000 lines of C code, and approximately 20 logically identifiable modules.

In addition to providing us with the Reduced Form data for the 20 modules, the organization also agreed to provide performance data in the form of error reports identifying the number and type of errors associated with each module. Approximately 275 errors were logged for the 20 modules in

our study, with from 1 to 35 errors associated with each module.

The Reduced Form data was used to calculate six metrics:

(1) DSL - Simply the total number of lines in the module. This is the most easily obtained metric in use, which is perhaps why it is the favorite of both researchers and practitioners.

(2) PRC - The number of function definitions within each module. This is almost as easy to obtain as total lines of code, and provides an alternate measure of software "size".

(3) E - Halstead's Effort measure [3]. Calculation of E involves obtaining the software science measure of "Program Volume":

$$V = N(\log_2 n)$$

where N is the total number of tokens used in the program, and n is the number of unique tokens used in the program (the calculations we used, assumed that a particular token was unique only within its module definition, and the use of that token, or a token with the same identifier in another function definition was yet another unique token). As well as the software science "Program Level" measure:

$$L = V^* / V$$

where V is the software science Program Volume and V* is the software science "Potential Volume" (the volume the program would possess if it were implemented as a simple procedure call, calculated as: (2+n2*)log2(2+n2*), where n2* is the number of Input/Output variables to the program). Effort is then calculated as:

$$E = V/L$$

(4) VG - Which is a measure of the "control flow complexity" of a piece of software. VG is calculated by summing the number of decision points in the program and adding one. We considered the following to represent decision points in the code: FOR, IF, ELSEIF, WHILE, CASE, BREAK and EXIT.

(5) HARR - A new metric developed in [6] which measures the "macro-complexity" of a piece of software (ie, entire system complexity as opposed to the complexity of a single piece of software in isolation). HARR is calculated as:

System Complexity * average(Module Complexity)

Where the average module complexity is the average VG measure for all the function definitions within the module, and System Complexity is:

$$\sum_{i=1}^{\#modules} [Glob(i)*(\#modules-1)]+[Param(i)*(1-DI(i))]$$

Where:

Glob(i) - number of times global variables are used in function definition i

Param(i) - number of times parameters are used in function definition i

DI(i) - a "documentation index" for module i, which is a measure of the quality of internal documentation within a function definition. We calculated it as:

$$DI(i) = (DSL(i)-NCSL(i))/DSL(i)$$

where DSL(i) is the total number of lines in function definition i, and NCSL is the number of non-comment lines in function definition i. In essence, our calculation provides the percentage of comment lines in function definition i.

(6) HNK - The macro-complexity measure by Henry and Kafura [7]. Limitations of the Reduced Form would not permit the exact calculation of the HNK metric suggested by Henry and Kafura (it was not clear which parameters are simply used and which ones were actually changed - this is necessary to prevent reconstruction of the code - thus, it is not clear if an item is a FanIn or a FanOut). The calculation we used was:

(FanIn + FanOut)**2 * DSL

where FanIn and FanOut are the number of information flows into and out of each procedure (Henry and Kafura suggested FanIn and FanOut be multiplied). This was obtained by simply summing the number of unique parameter usages, global variable usages and function calls over all the function definitions and multiplying by the total number of lines of code in the module.

In order to see if any of the metrics were related to the number of errors observed in each module, we performed a simple correlation analysis, the results of which are shown in Table II. As can be seen, the HARR metric and total lines of code were most closely related to number of bugs attributed to each module, followed closely by VG. These three correlations were significant at the .ØØ1 level (ie, there is a .ØØ1 chance that the correlation observed was due to chance and that the true correlation is actually Ø.ØØ).

4. Conclusions

The major goal of this paper was to illustrate the usefulness of the Reduced Form. While the Reduced Form described in this paper can hardly be considered anything more than a prototype (more input from other metric researchers will be needed before the final version of the Reduced Form can be established), it does show that:

(1) A Reduced Form can aid in obtaining data for metric studies

(2) Many current metrics can be easily calculated using the Reduced Form

(3) The Reduced Form can be used to develop and study new metrics

The results of the study suggest that perhaps the most reasonable metric to use in assessing software complexity is simply number of lines of code in the program. While some other metrics may work just as well, or even better in some cases, lines of code is almost trivial to obtain, in relation to some of the other metrics (eg, E, HARR, and HNK).

However, one must be careful about drawing sweeping conclusions about which is the best software complexity metric from only one study. It is difficult to estimate the influence of the programming language (C in our study) or the type of software project (a compiler in our study). Thus, the conclusions reached from this work are highly tentative. One battle does not win a war, and one study does not settle the software metric controversy. However, it is a step in the right direction. Before more definite conclusions can be reached, software written in other languages and for other applications must be analyzed.

We hope to be able to continue our work in this area, and encourage others, both academic and practitioner, to become involved by developing new metrics, investigating proposed metrics, and most important of all, providing data for researchers.

5. Acknowledgements

We would like to express our appreciation to Nancy Currans for her assistance during this work.

6. References

[1] Zelkowitz, M., A. Shaw and J. Gannon, Principles of Software Engineering and Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[2] McCabe, T., "A Complexity Measure", IEEE Transactions on Software Engineering, December 1976, pp 308-320.

[3] Halstead, M., Elements of Software Science, Elsevier, New York, 1977.

[4] Graham, R., "The Legal Protection of Computer Systems", Communications of the ACM, May 1984, pp 422-426.

[5] Harrison, W. and C. Cook, "A Method of Sharing Industrial Software Complexity Data", ACM SIGPLAN Notices, February, 1985, pp 42-51.

[6] Harrison, W., "A Study of Macro Level Complexity Metrics", PhD Dissertation, Department of Computer Science, Oregon State University, July, 1985.

[7] Henry, S. and D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, September 1981, pp 510-518.

Percent of respondents whose organizations would share source code with researchers	35%
Percent of Respondents whose organizations would not share their source code,but would share the Reduced Form with researchers	35%
Percent of respondents whose organization would share performance data with researchers	57%
Percent of respondents whose organization would not share any data describing their code systems with researchers	28%

Table I. Major results of the survey.

Metric	Bugs
HARR	.7538**
HNK	.6231*
DSL	.76ØØ**
VG	.739Ø**
E	.6919**
PRC	.6493*

* Significance < .Ø1 ** Significance < .ØØ1

Table II. Correlation of metrics with bugs.

```
readfile (fname)
char *fname; {
    register FILE *f = fopen (fname, "r");
    if (f==Ø) {
        error ("Can't read %s", fname);
        return;
    }
    erasedb ();
    while (fgets(line,sizeof line,f)) {
        linelim = Ø;
        if (line[Ø] != '#') yyparse ();
    }
    fclose (f);
    DBchanged = Ø;
    linelim = -1;
}
```

Figure 1. Sample C Program.

```
PROCEOURE readfile()
DCLS
FILE                1
char                1
register                 1
CONSTANTS
CONØØØØ2Ø               4
CONØØØØ21               1
VARIABLES
VARØØØ131               1   unknown  unknown
VARØØØ128               4   FILE  local
VARØØØ127               4   FILE  formal parameter
VARØØØ129               3   unknown  unknown
VARØØØØØ8               2   int  global
STRINGS
STRØØØØ47               1
STRØØØØ46               1
STRØØØØ48               1
FNCALLS
erasedb()                 1
error()                 1
fclose()                  1
fgets()                 1
fopen()                 1
yyparse()                   1
OPERATORS
!=                   1
""                   2
''                   1
*                   2
,                   4
-                   1
;                  1Ø
=                   4
==                    1
[                   1
if()                     2
return                      1
sizeof                      1
while()                      1
{                   3
LENGTH               16              16
```

Figure 2. Reduced Form for C subprogram in Figure 1.

BIOGRAPHY

Warren Harrison and Curtis Cook

Warren Harrison is an Assistant Professor of Business Administration at the University of Portland. He holds a BS in accounting from the University of Nevada, an MS in computer science from the University of Missouri, and is a PhD candidate in computer science at Oregon State University. He has worked as a computer scientist at Bell Telephone Laboratories in New Jersey and Lawrence Livermore National Laboratory in California. His research interests include software metrics, decision support systems, and software project management and estimation.

Curtis Cook is Professor of Computer Science and Acting Chairman at Oregon State University. He earned a BA in mathematics from Augustana College, and an MS and PhD in computer science from the University of Iowa. His research interests are software complexity metrics, graph theory applications in computer science, minimal perfect hashing functions, and formal languages.

SOME RESULTS FROM USING A REDUCED FORM FOR SHARING SOFTWARE COMPLEXITY DATA

Warren Harrison
The University of Portland
Portland, OR 97203

Curtis Cook
Oregon State University
Corvallis, OR 97331

PROGRAM MAINTENANCE

- MODIFICATIONS MADE TO SOFTWARE AFTER COMPLETION - VERY EXPENSIVE

- THREE PHASES:

  1. UNDERSTANDING THE SOFTWARE

  2. MODIFYING THE SOFTWARE

  3. RETESTING THE SOFTWARE

- PROGRAM UNDERSTANDABILITY HAS AN EFFECT ON PROGRAM QUALITY

- USEFUL TO BE ABLE TO ASSESS PROGRAM UNDERSTANDABILITY

SOFTWARE COMPLEXITY METRICS

- MEASURE DEGREE TO WHICH PROGRAM CHARACTERISTICS THAT DETRACT FROM UNDERSTANDABILITY ("COMPLEXITY CHARACTERISTICS") EXIST IN CODE.

- DEVELOP SET OF CONSISTENT, OBJECTIVE RULES TO ASSESS DEGREE TO WHICH COMPLEXITY CHARACTERISTICS EXIST IN SOFTWARE, AND WEIGHT THEIR PRESENCE.

- ALLOW CONSISTENT RANKING OF PROGRAMS BASED ON THEIR COMPLEXITY

- COULD BE USED AS A FEEDBACK TOOL FOR PROGRAMMERS, PERSONNEL SCHEDULING TOOL FOR MANAGERS

- DIFFERENT IDEAS ON SET OF VARIABLES TO CONSIDER AS COMPLEXITY CHARACTERISTICS AND THEIR WEIGHTING

- METRICS INCORPORATING VIRTUALLY EVERY MEASURABLE CHARACTERISTIC - WHICH ONE(S) WORK?

VALIDATION OF METRICS

- FIND OUT IF THEY "WORK"
- TWO APPROACHES:
  1. CONTROLLED EXPERIMENTATION
     - o BUILD TWO VERSIONS OF SAME PROGRAM
     - o RECRUIT LARGE NUMBER OF SUBJECTS AND HAVE HALF PERFORM SAME PROGRAMMING TASK ON ONE VERSION OF THE PROGRAM, AND THE OTHER HALF PERFORM THE SAME TASK ON THE OTHER VERSION
     - o COMPARE THE PERFORMANCE OF THE TWO GROUPS
     - o ATTRIBUTE THE DIFFERENCES IN PERFORMANCE TO THE DIFFERENCE IN COMPLEXITY CHARACTERISTICS OF THE TWO PROGRAM VERSIONS
  2. FIELD STUDIES
     - o COLLECT DATA FROM "REAL WORLD" PROJECTS
     - o MEASURE PERFORMANCE OF PROGRAMMERS CARRYING OUT CERTAIN TASKS ON PROGRAMS
     - o COMPARE PERFORMANCE OF PROGRAMMERS ON DIFFERENT PARTS OF THE PROJECT
     - o ATTRIBUTE DIFFERENCES IN PROGRAMMER PERFORMANCE TO DIFFERENCES IN PROGRAM CHARACTERISTICS

REDUCED FORM

PROBLEM: RESEARCHERS NEED DATA FROM ACTUAL PROJECTS, BUT INDUSTRY FEARS TRADE SECRETS WILL BE COMPROMISED.

SOLUTION: EXTRACT IMPORTANT CHARACTERISTICS OF THE CODE, WITHOUT PROVIDING ENOUGH INFORMATION TO RECONSTRUCT THE PROGRAM AND/OR FORMULAS.

ANALYSIS OF A PROJECT

- 30,000 LINES OF 'C' CODE AND 20 LOGICAL MODULES
- 275 ERROR REPORTS
- METRICS CALCULATED:
  1. LINES OF CODE
  2. NUMBER OF PROCEDURES
  3. SOFTWARE SCIENCE 'E'
  4. CYCLOMATIC COMPLEXITY, VG
  5. HENRY AND KAFURA'S INFORMATION FLOW METRIC
  6. HARR, A MEASURE OF GLOBAL COMPLEXITY
- RESULTS:

METRIC	CORRELATION WITH BUGS
HARR	.7538
HNK	.6231
LOC	.7600
VG	.7390
E	.6919
PRC	.6493

CONCLUSIONS

1. REDUCED FORM CAN SOLVE DATA COLLECTION PROBLEMS:

   - MAKE ORGANIZATIONS LESS RELUCTANT TO SHARE DATA

   - CAN CALCULATE MANY CURRENT METRICS USING REDUCED FORM

   - REDUCED FORM CAN BE USED TO DEVELOP AND STUDY NEW METRICS

2. MANY METRICS ARE HIGHLY RELATED TO PROGRAMMER PERFORMANCE MEASURES (EG, ERRORS), BUT LINES OF CODE SEEM 'BEST'

3. NEED ADDITIONAL DATA FOR FOLLOWING STUDIES

   - REDUCED FORM DATA

   - PROGRAMMER PERFORMANCE DATA

A Practical Guide to Acquiring Software Engineering Tools

Tom Milligan

Software Center Tools Support Group
Software Center
Tektronix, Inc.

Abstract

In the last few years an increasing number of software vendors are providing tools to address the needs encountered in the Software Engineering Process. Unfortunately, the targeted audience for these tools (engineers) are not traditionally educated nor experienced in the techniques for acquiring software tools. This paper will present a method for identifying and then acquiring useful software engineering tools from third-party vendors. This method has been developed and is in use at Tektronix, Inc. by a corporate group of software engineers who are acquiring an integrated set of software engineering tools for use throughout Tektronix. While the method was derived and is tuned for use in a central group doing corporate tools acquisitions, the sub-methodologies described are discrete. Parts not appropriate for other types of acquisitions can easily be deleted from the overall method without threatening the overall structure of the process. The outlined method is straightforward, thorough, and tested. It addresses the following topics:

1 Assessing needs for tools.

2 Finding tools to meet the defined need.

3 Evaluating a prospective tool.

4 Selecting a vendor.

5 Purchasing a tool.

6 Supporting a tool.

A Practical Guide to Acquiring Software Engineering Tools

Tom Milligan

Software Center
Tektronix, Inc.

Most of us readily admit that significant productivity gains can be achieved through the prudent introduction of software tools into a software engineering environment. What most of us don't know is how to approach that word "prudent". This paper will present a practical set of methods for intelligently selecting and introducing software tools into an engineering environment. It will deal with how to determine what tools are needed, how to find needed tools, how to evaluate prospective tools, how to select a vendor for tools, then how to purchase tools, and finally how to go about supporting tools.

Assessing Needs for Tools

While we may all agree that we need tools, we should also understand that not just any tools will do. We don't want to solve nonexistent problems, nor do we want to let a critical need go unanswered because we are off dealing with a not-so-critical one. So, how do we determine where we are to exert our efforts in acquiring tools? One method is to ask the people who would be using them. In particular, ask them to describe what they do, how they do it, and finally how they would like to do it. This will give you an idea of the problems you are trying to solve, and will give the would-be users a chance to define their own problem. A word of caution is in order here: when you ask these questions, beware of the answers. Typically answers like "I need more computing power." or "I need a faster something else." abound. This is not what you are looking for. The answer lies not in making machines or even tools "faster", the answer lies in making people faster.

Another technique for determining which tools to pursue lies in your imagination, use it. Put yourself in the shoes of those you are trying to help. If you have worked in that environment before you may find this task easier, but beware of the limitations this "advantage" puts on you. Specifically you may find yourself bound to what you believe is current technology. If you are going to use your imagination, then don't bind it. Imagine the ideal tools for the job, then look for them.

One of the most important aspects of assessing tools needs is to be able to distinguish between "tools" and "toys". At the most abstract level, a tool is a useful instrument in doing a particular job, while a toy is something to play with. It should be clear that we desire to find tools, not toys. Some distinguishing characteristics of tools and toys are give below.

Distinguishing Characteristics of Tools

A tool typically aids in doing a discreet part of of a larger process.
A tool typically does only one or two functions, but does them well.
It is easy to quantify time savings that will result from the use of a tool.

Distinguishing Characteristics of Toys

A toy may replace an already satisfactory tool, but not deliver any significant productivity gain.
The driving factor for wanting a toy will typically be personal preference.
Toys are typically touted as "more convenient" than an existing tool.
It is difficult to quantify time savings that will result from the use of a toy.
People who want toys are upset when told that they can't have them.

After you have assessed what tools are needed in a particular environment, the task becomes one of finding tools to fit those needs.

Finding Tools

A number of sources are available in which to look for software tools. Trade Publications are rife with reports and advertisements for all kinds of tools. Some of the most useful are Computerworld, Electrical Engineering Times, Electronics Week, InfoWorld, and if the tool is to run on a Personal Computer, one of the myriad of magazines dedicated to that particular PC.

Another good source of information on tools is the trade conference. For UNIX-based tools, Uniforum and Usenix are the primary trade conferences. Typically Uniforum has a wide variety of software tool vendors displaying their wares, while Usenix may have more intense technical sessions relating to new tools development.

Also, many catalogs exist that list and summarize features provided by a wide variety of software tools. These catalogs are published by the federal government, by operating system vendors, by computer vendors, and by independent organizations. Some of these are even available in machine-readable format, allowing a computerized database of software tools to be compiled. A list of software tools catalogs, as well as their publishers are given in Appendix A of this document.

Evaluating Prospective Tools

After determining what tools exist that address a defined need, you will need to evaluate whether the tools address the problems correctly, and possibly which tool among many appears to be the best for your particular environment. In addition, at the same time, you will be evaluating prospective vendors for a tool. A number of approaches are possible, depending upon time and other resources available for the evaluation.

One of the fastest ways to evaluate a prospective tool is to talk to current users of that tool. Most vendors are happy to give you company names as well as the names of individuals within those companies who are using a particular tool. Call these people, and ask them the following questions:

1. "What do you like about the tool?"
2. "What don't you like about the tool?"
3. "What would you change about the tool?"
4. "How much time does it save you?"
5. "How well does the vendor respond to your problems?"

The answers to these questions will tell you a lot about both the tool and about how well the vendor responds to its customers. This will aid you in both in selecting a tool and in selecting a vendor.

After the contacts with current users of the tool, you will probably want to evaluate the tool for yourself. Contact an appropriate vendor and tell them that you are interested in the tool. Then ask if you can have an evaluation copy of the tool for a specific period of time. Typically 2 weeks to a month is an reasonable period of time for an in-depth evaluation. Most vendors are prepared to honor this request. Some vendors may want to offer you a demonstration of the tool on one of their machines through the use of a modem and phone lines. In general this is not an acceptable way to evaluate a tool. Don't buy a tool unless you have the opportunity to try it out in your environment to determine its usefulness to you.

Once the tool is in-house, you will want to give it a cursory acceptance test. Factors to consider in this initial evaluation include ease of installation, simple invocation without traumatic side-effects, and general good behavior. After this initial test move on to more in-depth testing.

There is no better in-depth test for a software tool than to "drop" it into the type of environment that the tool will eventually be used. When placed into these environments, the tools are used in exactly the manner that is appropriate for your organization. Any deficiencies in the tool that relates to the way your software development environment operates are readily apparent. Another useful piece of information available from this type of evaluation relates to how readily the tool will be accepted into your software development environment. If you are unable to find a software development group into which to place the tool, then maybe that tool is not appropriate, or perhaps its time has not yet come. After your evaluation is complete, give the information on the tool, its weaknesses and its strengths, to the vendor. Be candid but fair, and give the vendor a chance to fix the problems.

A note is in order regarding relations with vendors. The principles are honesty and fairness. Be straightforward with a vendor, telling them your concerns about the tool, and then let them answer. Assure them that you will not steal their software and then make sure you and everyone else in your company adheres to that promise. Don't adopt an "us against them" mindset. Be prepared to pay the vendor a fair sum for the tool. If the tools is a good one, it is worth it, and

usually the vendor doesn't "owe" you a bargain.

Selecting a Vendor

Beyond simply evaluating how the tool works, you also need to evaluate and select the tool vendor. There are many types of software vendors, value added resellers (VARs), simple distributors, developers, etc. The type to select depends on the applicability of the tool across different environments, and your needs for support.

Value added resellers buy a tool from a developer, add some value to it, and then resell it. This added value can be in the area of support, or in added functionality. The disadvantages to VARs lies in their distance and independence from the original developer of the software. Their changes or enhancements to the software may not track subsequent releases from the developer. If the developer stops supporting the software however, this independence can be a positive attribute.

Simple distributors are "front ends" to developers, and are typically better able to deal with customers than are the developers. Distributers, however, are typically not prepared to support or enhance the software on their own, depending on the developer for those functions. This could mean some delays in getting bugs fixed or in an inability to get answers to highly technical questions regarding the tool.

Developers of software are able to respond quickly to requests for bug fixes or enhancements, but are typically not able to deal effectively with customers wanting only one or two copies of a tool, preferring instead to OEM their software to another distribution agent.

Things to consider in selecting a vendor are how many copies of software you anticipate needing, how much money you have to spend for an acquisition, and how well the vendor can respond you your request for support, enhancements and bug fixes. An important, often overlooked aspect of acquiring software is ongoing support for that software. Just because you have purchased a license for a particular software tool does not mean you are entitled to help, consultation, bug fixes, enhancements, or automatic updates to the tool. These considerations are usually negotiated and purchased seperately in a support contract with the tool vendor.

Purchasing a Tool

The most important aspect of purchasing a tool is to clearly define your needs. Define them in terms of how many copies of the tool you need, what kind of support you need, and which vendor seems best able to fulfill those needs. When these aspects of acquiring a tool are answered, you are ready to meet with your corporate purchasing agent and contracts administrator. Meet with each of these people and clearly outline your needs and give them whatever information you have that is relevant to the purchase of the tool. Included in this information should be the following:

1. A document clearly outlining how many copies of the tool you want to but, along with where the tool will reside, and who will be responsible for them

2. The name, address and phone number of the vendor.
3. A copy of the vendor's pricing policy.
4. A copy of the vendors appropriate software purchase contract.
5. A copy of the vendors appropriate software support contract.

If they need further information, they will tell you. When you meet with the contracts administrator, you should plan to go over the contract item-by-item to determine whether changes are necessary. Your contracts administrator should be an attorney or some other person versed in legal terminology and, hopefully, software law. You can use the contracts administrator to translate the legal jargon into english. Be aware that terms of contracts and prices are negotiable, as was mentioned above however, work with and be fair to the vendor. When you have received this information from the vendor and have passed it on to the purchasing agent and the contracts administrator, it is time to turn all of these people loose on each other.

Supporting a Tool

After you have successfully purchased a tool there are only few other details to attend to in supporting that tool. First, you must install the software. If the software is meant for one machine, this should be straightforward. If, however, the software is destined for more than one machine, you will have to decide on an appropriate mechanism for the installation. One option is to simply follow the installation procedure for a single machine on each separate machine that the software is to reside. At Tektronix, we have a very efficient computer network whereby every engineering computer in the company is directly accessible by every other engineering computer. Thus, it is possible to install the software once on one computer, and then automatically ship the software in the correctly installed configuration to all of the other computers who are to receive it. This has the advantage of reducing a 30 minute installation procedure to about 6 minutes.

After the software is installed, you should have some mechanism set up to answer user questions about the software. These questions can range from highly detailed technical questions, to very simple invocation inquiries. Ideally, one person can be designated as the contact for a particular tool, and this person would process most inquiries. An important factor to remember is that the contact person for a particular tool does not need to know everything about the tool, but rather, they need to know where they can find the answers.

Invariably some bugs will be found in the software. You should have a mechanism for accepting bug reports on the tool, for analyzing them as to their validity, for organizing bug reports so that you can tell if a given bug has been reported before, and finally for reporting bugs back to the vendor. The contact person for the tool is usually the focal point for this bug activity. Tektronix has developed a fairly thorough and extensible bug-tracking system for system software bug reports. This extensibility has made it possible to include bug reporting for software tools into the tracking system.

Finally, you will need a mechanism for distributing subsequent software releases to users of the tool. Most often, this can be accomplished through the same process as the initial installation.

At Tektronix, we maintain a database relating software tools to machines. Tektronix has developed a software distribution system that queries the database whenever a new release of some software is available, and which then distributes it to the appropriate places automatically. This approach, however, has met with some resistance from local system administrators who object to having their machines changed without their knowledge.

Conclusion

Thus we have seen a practical, tested methods for identifying needs for tools, for finding needed tools, for evaluating tools, for selecting vendors for tools, for purchasing tools, and finally for supporting software tools. This methodology has been tested in practice and works. In addition, the outlined sub-methodologies are discrete and inappropriate segments can be deleted as available resources dictate.

Appendix A
Software Tool Guides and Catalogs

Publications from the Federal Government

COSMIC Software Catalog
NASA's Computer Software Management and Information Center
112 Barrow Hall
The University of Georgia
Athens, GA 30602

Computer Science and Technology, NBS Special Publication 500-88, Software Development Tools
Raymond C Houghton, Jr. - Author
US Government Printing Office
Washington, DC 20402

Office of Software Development, Federal Software Testing Center, Software Tools Survey
Federal Software Testing Center
Office of Software Development
Two Skyline Place, Suite 1100
5203 Leesburg Pike
Falls Church, VA 22041

Vendor Publications

CAEM Software Referral Catalog
Digital Equipment Corporation
Computer Aided Engineering and Manufacturing
Two Iron Way
MR03-1/E8, Box 1003
Marlboro, MA 01752

Engineering Applications Graphics Referral Catalog
Digital Equipment Corporation
Engineering Systems Group
Marlboro, MA 01752

Engineering Applications Software Referral Catalog
Digital Equipment Corporation
Publishing and Circulation Services
10 Forbes Road
Northboro, MA 01532

Intel Yellow Pages Software Directory
Intel Literature Department
3065 Bowers Ave.
Santa Clara, CA 95051

US Chapter DECUS Program Library Software Abstracts
Digital Equipment Corporation
Marlboro, MA 01752

Independent Catalogs

The Software Catalog, Science and Engineering
Elsevier Science Publishing Co., Inc.
52 Vanderbilt Ave.
New York, NY 10017

Unix* Applications Software Directory
Onager Publishing
6451 Standridge Court
San Jose, CA 95123

Unix* Software Directory
Onager Publishing
6451 Standridge Court
San Jose, CA 95123

Unix* Software Tools Directory
Reifer Consultants, Inc.
2550 Hawthorne Blvd., Suite 208
Torrance, CA 90505

The Unix* System V Software Catalog
Reston Documentation Group
Reston Publishing Co., Inc.
Reston, VA 22090

*Unix is a trademark of ATT Bell Laboratories

BIOGRAPHY

Tom Milligan

Tom Milligan graduated from the University of Oregon in 1978 with a BA in computer science, secondary emphasis in mathematics. He has worked as a software engineer developing embedded systems, as a technical writer, as a software evaluation engineer, and most recently has been project leader for Tektronix Software Center Tools Support Group. The Tools Group is a corporate entity, composed of software engineers, whose purpose is to identify, acquire, and support software engineering tools, as well as to put those tools together into an integrated development environment for use by software engineers at Tektronix.

THE USE OF SOFTWARE METRICS TO IMPROVE PROJECT ESTIMATION

Bob Grady and Debbie Caswell
Hewlett-Packard Company
Software Engineering Lab
Corporate Engineering

ABSTRACT

In 1983, a company-wide program was initiated to measure and improve the process of developing software at Hewlett-Packard. One of the objectives of this program was to use measurements to achieve immediate short-term improvements in productivity and quality. This paper reviews various efforts during the first year of measurements which led to significant development process changes and a greater awareness of which elements to monitor.

BACKGROUND

Hewlett-Packard designs and manufactures scientific instruments, small to medium-size computers, and medical and analytical instruments. During the past fifteen years these components have been increasingly designed for and used in systems which solve complex problems. In the forty-six years since its founding, HP has grown until today its annual sales are in excess of six billion dollars and research and development for new products is carried on in twenty-five decentralized laboratories scattered throughout the U.S., Europe, and Japan.

The first HP computers were introduced in 1966 and 1967. The HP2116A computer and the HP9100A desktop computer (or calculator as it was initially referred to) were designed for totally different markets and produced by two geographically separate divisions. Each contained HP's first substantial efforts in the software engineering field and characterize how rapidly the breadth of HP's software production developed. Today, the majority of the software produced in HP is only loosely coupled among any set of divisions, even though the systems nature of HP's products suggests the need for tight coupling.

Types of Software

Software at HP is created for a wide spectrum of applications and customer types. For the sake of convenience, though, the applications can be reduced to four major types: firmware, systems, applications, and end user. Firmware consists of software generally designed to execute from ROM (read only memory) under control of a microprocessor. Examples of divisions designing firmware are those producing instruments and computer peripherals. Systems software consists of software generally designed to execute from the memory of mini-computers. It functions as the framework for developing and executing other software. Examples of divisions designing systems software include divisions directly involved in producing computers, network software, languages, and data bases. Applications consist of software that operates on top of and using systems software. Applications software also generally solves a generic class of problems for a narrow set of customers and needs. Examples of divisions designing applications software include those dealing with manufacturing, medical, and financial customer solutions. End-user software at HP consists of software which generally doesn't fit the other three categories. In many cases end user software as defined here operates on top of or, in addition to,

applications software, instruments or systems. Examples of end-user software include electronic data processing and software done by such groups as Production Engineering and Quality Engineering.

Table 1 illustrates some characteristics of these four categories of software. Because each of these types has different driving characteristics, discussions among the many R&D groups concerning software encounter difficulties when people try to compare methods, tools, priorities, and estimates.

HP SOFTWARE DEVELOPMENT ENVIRONMENTS

INFLUENCING FACTORS	MICROPROCESSOR	SOFTWARE SYSTEMS	APPLICATIONS	END-USER
TEAM SIZE	Small	Large *	Large *	Small
MARKET SIZE +	Small → Large	Large	Large	Small
LANGUAGE	Asmb.Pascal	C/Pascal/SPL	High-level	All
USER	Single	Multiple	Heavy, multiple	Single
TIMING	Important, sometimes critical	Critical	Mild importance	Varies in importance
METHODOLOGY	Few standards	Control-oriented	Data-oriented	Varies
COST OF CHANGE AFTER RELEASE	Large → Huge	Large	Moderate	Small
MAJOR APPLICATION CONCERN	Timing of ext. processes	Process interaction peripheral generality, recovery	Data integrity, user interface, portability	Single problem oriented

* Project sizes not large, but generally aggregates of projects are large.

\+ As measured in number of customer sites.

Table 1

Focus on Customer Satisfaction

The one topic which all HP divisions can agree upon is that the final test of a product's worth is measured by customer satisfaction. This can be characterized in a number of ways, but one early method which was established at HP in 1979, was to methodically record and analyze reports of defects and enhancement requests from customers. A customer with a software problem contacts the field service organization which verifies that the problem is indeed a defect. The field submits a service request to the factory via a system called STARS (Software Tracking And Reporting System).

In the factory, the marketing organization assigns a priority to fixing it. Next, the lab diagnoses the problem. Diagnosing and fixing the problem are two distinct steps and might or might not occur at the same time. After a fix is produced, it must be integrated into a product update and tested before it is released to customers.

Each month, a centralized support division publishes graphs by product line showing the number of defects reported but not diagnosed, the average amount of time a defect waits to be diagnosed, the number of critical and serious unresolved defects, and the mean time to fix a critical or serious defect (refer to Figure 1).

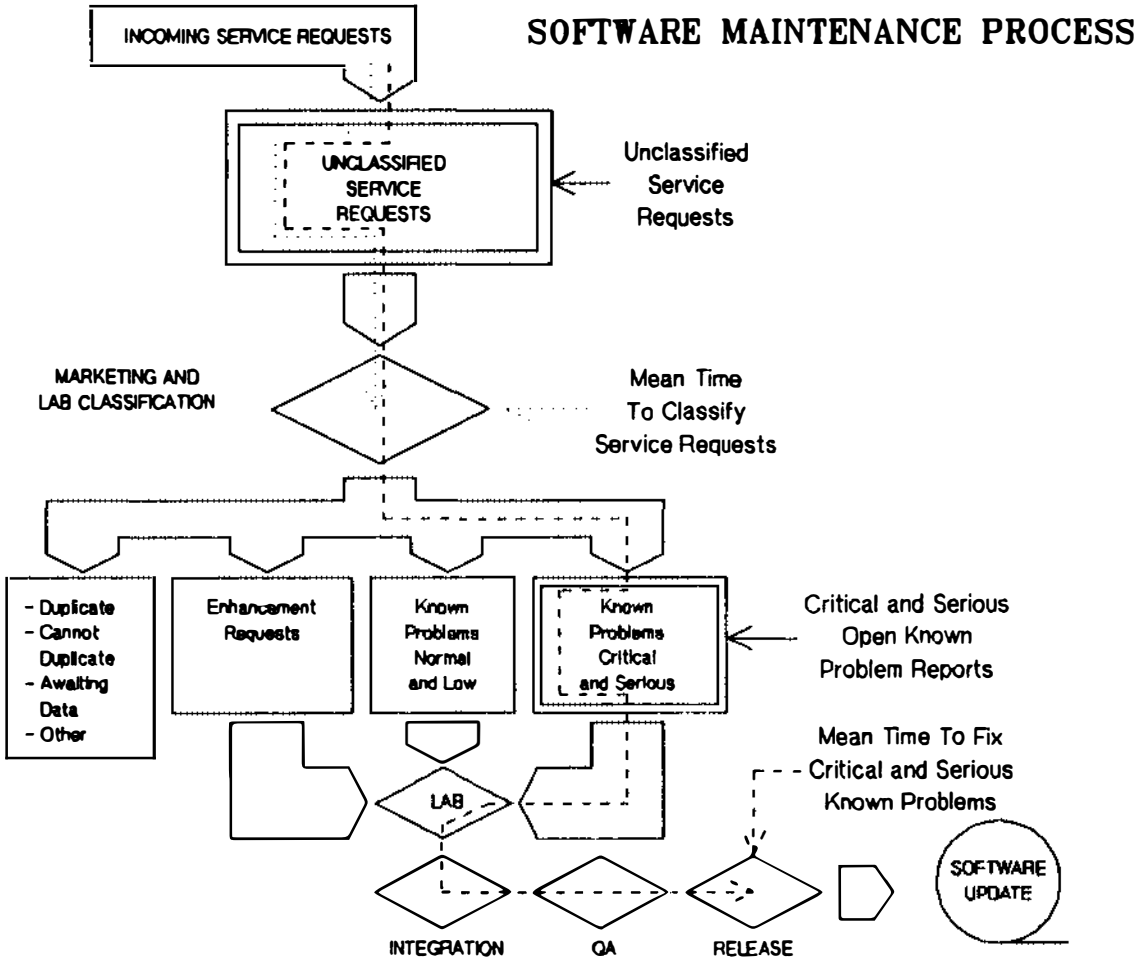


Figure 1

The monthly reports give a written analysis of trends indicated by the graphs. Their intent is to raise awareness of the amount of time it takes to get a software problem resolved from the customer's point of view. They also attempt to give information about the responsiveness of the factory maintenance teams.

These graphs have been very successful at focusing top management's attention on the customer satisfaction issue. Since managers know that every month the whole company will know their maintenance status, they make an effort to bring their defect backlog under control.

ESTABLISHING PROCESS METRICS

While the consistent reporting of defects and enhancement requests provided HP with a measure of its success, it fell short of providing an effective method for understanding the development process and accurately predicting results. What was needed was a common set of terminology and measures for the process of software development that could be used throughout HP early enough in the development process to affect change. A group of twenty software managers and developers from thirteen divisions were invited to establish an HP Software Metrics Council. These representatives were chosen on the basis of software experience, software management experience, interest, and prior work in software measurement and/or influence within their organizational entity to implement the council's decisions. Personal commitment and enthusiasm were also important. In addition, developers of all the various classes of software were represented.

The objective of the first meeting of the council was:

> To gain agreement on a set of software measurement criteria which managers feel are meaningful, reasonable to collect, and can be used to measure progress and predict results.

Explanation of HP Metrics

The result of the first meeting was agreement to collect metrics for five categories of information. Forms were created to ensure consistency and to facilitate collection of the data. They are reviewed and updated at the end of each phase, and the completed forms are collected at a central point upon product release. The data is then added to a database and used to compare data at a high level. Within a year of the initial agreement to metrics, over 100 projects had measured or were in the process of measuring these metrics. The standard metrics are explained below.

SIZE - The standard metric for size is NCSS (non-commented source statements). This means that the source code, not the object code, is used. Compiler directives, data declarations, and executable lines are counted, but not blank lines or whole comment lines.

In keeping with our "reasonable to collect" objective, it is assumed that an automatic line counter is used. In the absence of such a counter, the size is approximated. An educated guess is better than nothing.

PEOPLE/TIME/COST - The standard metric for cost is the engineering month. It is important to notice that it is defined as "40-50 hours per week with no compensation for vacation or sick time." Therefore, every engineer who works 80 hours a week for one month has contributed 2 engineering months in one calendar month. Not compensating for vacation or sick days is in line with our "reasonable to collect" objective. Also, time project managers spend managing is not included

DEFECTS - A defect is a problem or an error: anything that appears in the output of the software process which would not appear if it were perfect. Defects can occur at any life cycle stage. Right now, there is no attempt to distinguish severity. All defects are equal.

DIFFICULTY - The standard metric for difficulty is a number between 35 and 165 with 165 as the most difficult. The number is determined by filling in a questionnaire and inputting the responses to a program called SOFTCOST. The questionnaire asks about stability of requirements, experience of personnel on the project, familiarity with the type of software and development environment, access to needed hardware, and many other general project questions. In addition to generating the difficulty factor, the questionnaire helps to qualify productivity numbers which are computed.

COMMUNICATIONS - The number of interfaces that the lab project team has is the standard metric. The intent is to quantify constraints on the project team due to dependencies with entities politically and physically distant. If this metric were thought out at the beginning of the project it:

1. Could influence the partitioning of the task to minimize necessary interfaces.

2. Would raise awareness of who the suppliers and customers are for the project.

PROCESS IMPROVEMENTS

One of the most important results of the use of standard metrics was that many divisions went beyond the standard metrics to understand why certain results were occurring. The results of these more detailed studies encouraged other groups to leverage off these experiences and extend them in ways appropriate to their own development needs. The remainder of this paper reviews some of these studies, including how they have led to better understanding of the tasks being done and how long the tasks should take.

An Example of Statistical Quality Control

One experiment, which actually began before the definition of the HP metrics, used the techniques of statistical quality control (SQC) which HP has used effectively for several years throughout our manufacturing areas. This entity believed that by focusing on defects, the causes of the defects could be discovered and permanently removed.

The software studied in this case was a series of applications packages designed for internal company use in support of purchasing and vendor analysis. This type of package is ultimately implemented in over fifty divisions which operate in a relatively consistent fashion, so development of such systems is typically done in partnership with several divisions. A prototyping approach was chosen to maximize the feedback from the customer divisions and avoid some types of problems which had typically appeared in the past. It was also believed that analysis of defects which appeared in each prototype could lead to elimination of those defects in subsequent prototypes.

The first step was to prepare a list of defects which applied to the type of software they were producing. Figure 2 shows that they grouped defects into three principal categories [1]. It is important to note that these definitions and categories are relatively unique to this particular type of application and development environment. In a later discussion in this paper, we will see a similar approach taken with quite different prevalent defects.

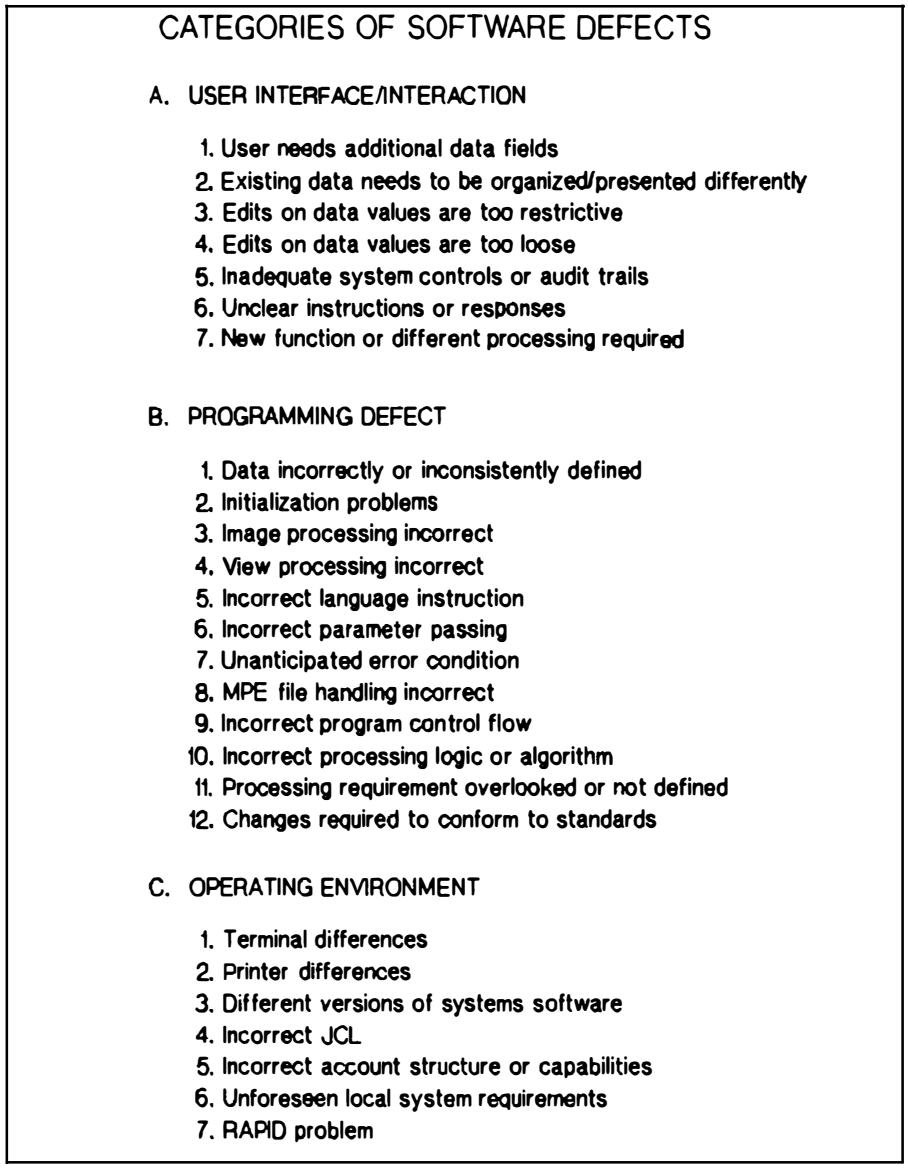


Figure 2

A Pareto analysis was then done to identify the most frequently occurring defects. In this case over one-third of the defects corresponded to categories A7, A2, and A1 from Figure 2. The probable causes of these defects were then determined using SQC, and changes were instituted into the development process.

The second series of software was completed using the modified prototyping development process. As was desired, the results showed that instead of these major defect categories appearing after release to the internal customers, they now appeared much earlier in the process during the several prototyping stages. In fact, categories A2 and A7 accounted for over fifty percent of the pre-release defects recorded.

Predicting the Testing Process

Another division develops firmware used in communications applications. Their projects are typically short (less than six months) but the type of application and the number of installations is such that the final quality of their product is very critical. Because their product line is reasonably repeatable and their development cycle short, they were able to characterize parts of their process relatively quickly. They determined that their average coding rate was 670 NCSS/programmer month (NCSS is non-commented source statements) and that their average pre-release defect density was 9.6 defects/1000 NCSS. (Note that any defect rate is entirely dependent upon how a given organization defines defects. Our early experience shows variation of up to a factor of 200 in defect density among different entities depending upon how defects are defined and recorded.) Using these averages they were able to make their process more predictable.

They focused their attention particularly on the testing cycle. Using the model defined in Figure 3, they started predicting how long the testing phase should take as well as recording and categorizing defects in detail.

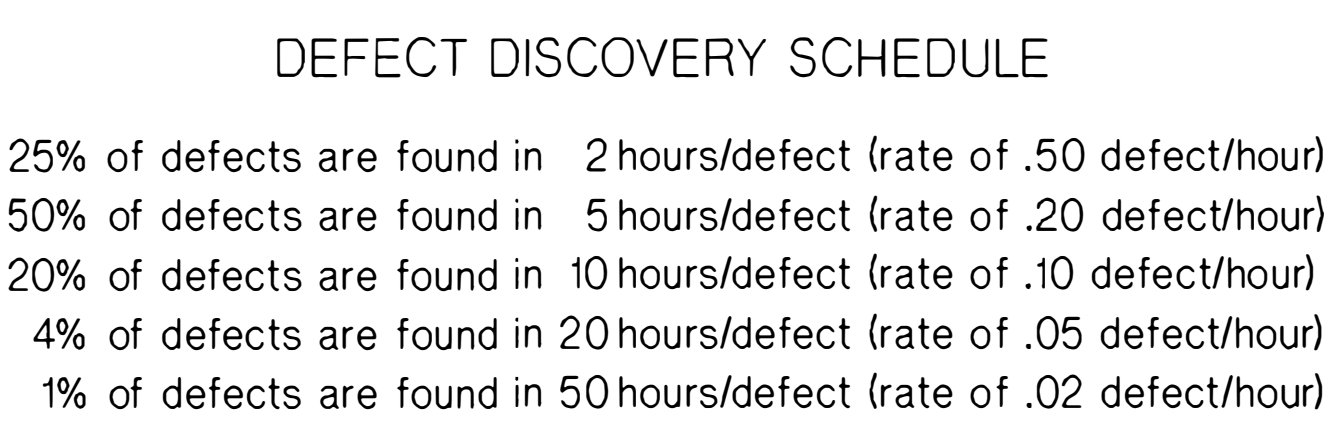


Figure 3

Figure 4 shows the predictive model of a typical project and the actual rate of defect discovery and completion. As can be seen from this example, the amount of testing required to achieve a desired level of quality can be predicted reasonably well.

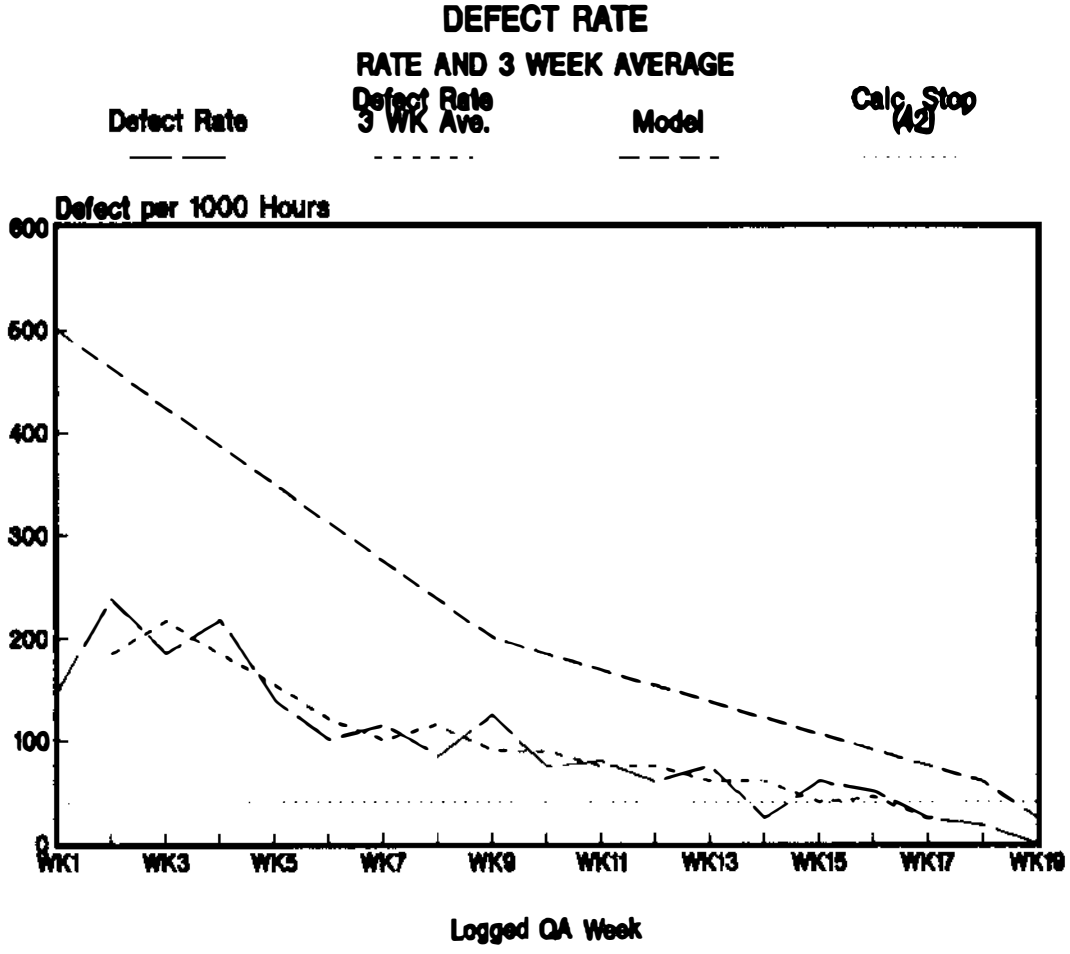


Figure 4

In addition to predicting and monitoring defects during the testing process, this division also made an effort to categorize the defects by severity to use as an aid in project tracking. They used the same severity categories described earlier that HP uses for reporting defects after a product is released. Displaying these defects in the form of a stacked bar chart (Figure 5 shows defects for the same project displayed by Figure 4) on a weekly basis then shows not only the downward trend of defects toward project completion, but also flags the presence of major problems past the point when they might be expected.

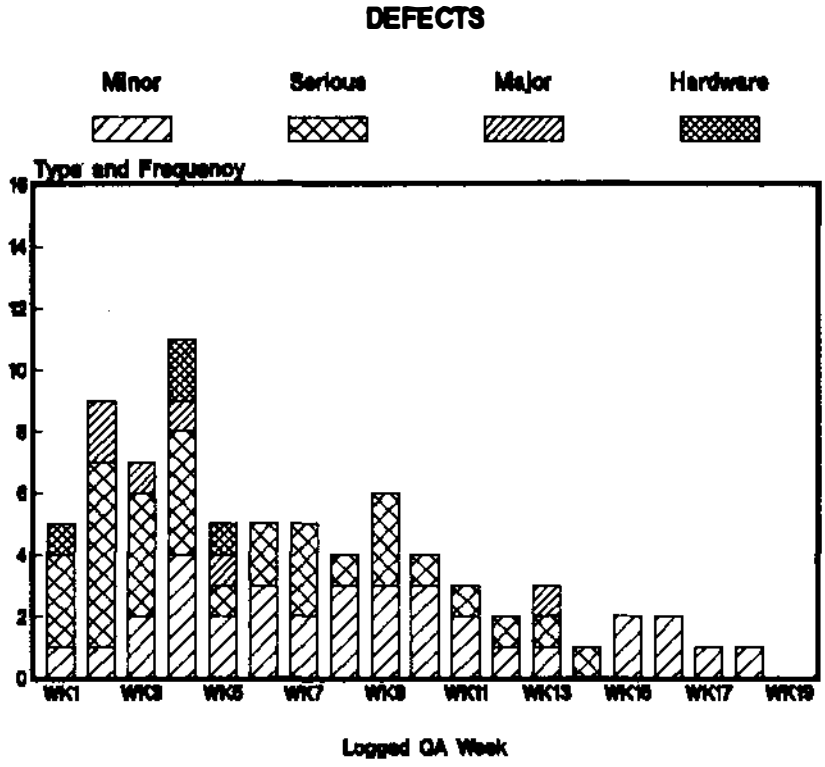


Figure 5

Unlike the applications environment discussed earlier where the primary source of defects was in the user interface specifications, this division found their major source of defects was in the implementation of algorithms. Figures 6 and 7 show the breakdown of defects by project phase and classification. These measurements and analyses have not only made their process more predictable, but they have pointed out the primary areas where effort can be focussed to improve the process.

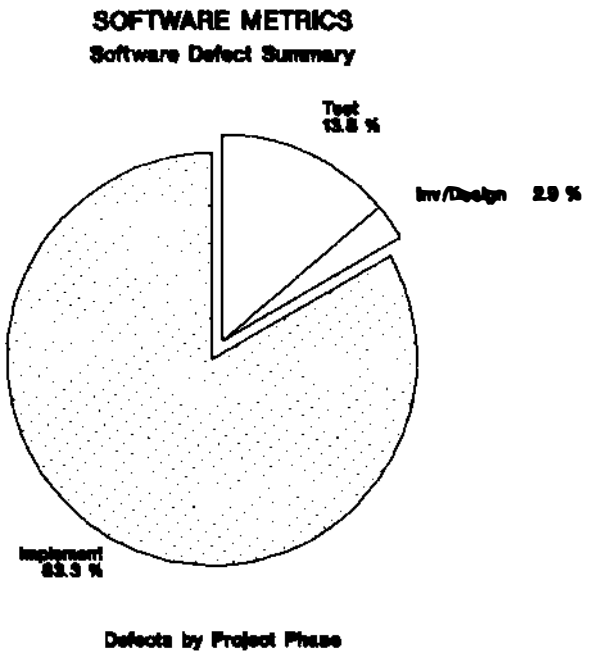


Figure 6

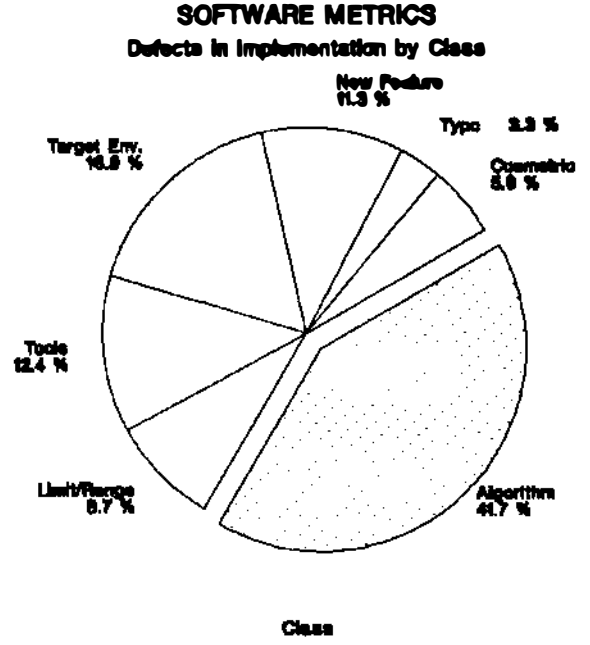


Figure 7

Using their technique to predict testing time and effort necessary, they are routinely predicting the testing phase within ten percent of the actual times spent now. For three products which have been released long enough to accurately draw conclusions, they have seen a total of only one defect after this testing process has been completed.

Project Prediction and SQC at a Systems Division

A third division produces systems and software used to develop firmware applications. They have a large team of software developers with projects of varying size which primarily fall into operating system and compiler software, but also include firmware and applications as well. Their productivity has been quite respectable, but they felt that their ability to predict project completions was poor and that they really didn't have good understanding or control over defects in their process. Their pre-release defect densities have varied from .4 to 6 defects/1000 NCSS. Figure 8 shows a graph of the accuracy of their project estimates.

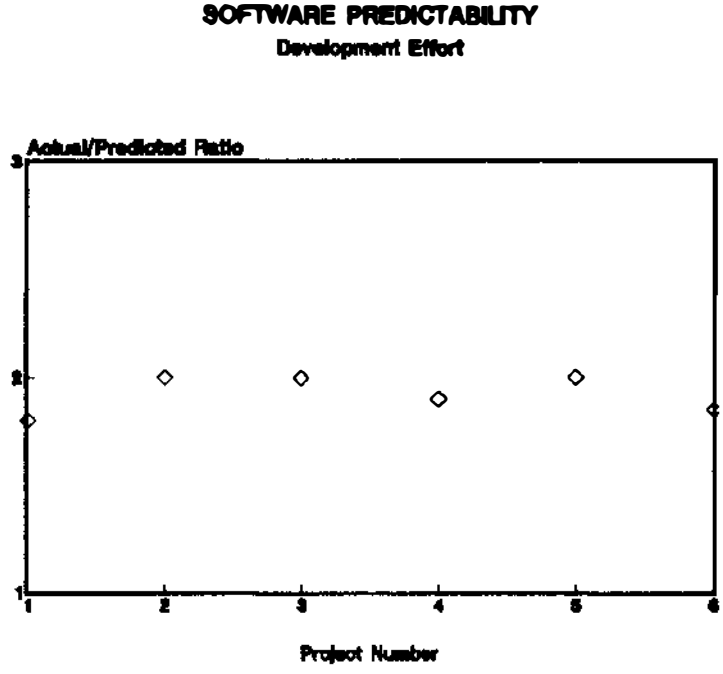


Figure 8

By initiating measurements in all areas of development, they hoped to improve their ability to estimate projects. In addition, by focusing heavily on defect analysis they felt they also had the best chance of improving their process. They used techniques similar to those described in the previous two divisions. Figures 9 and 10 show categorization of defects for one of their development areas. One of the most interesting results of these measurements was that in the category of detailed design defects, the largest category, over half of the reported errors occurred during redesigns. During redesigns they typically did not have formal review mechanisms in place to ensure top quality. These measurements, of course, led to the introduction of such reviews.

COMPILER DEFECTS

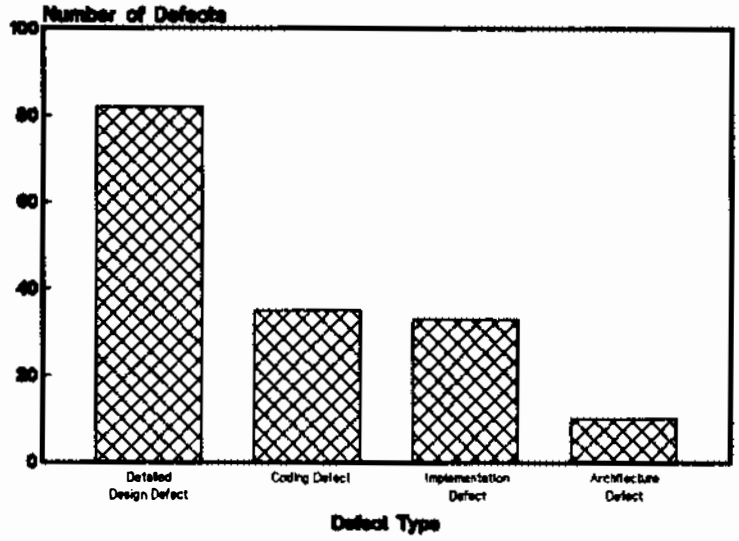


Figure 9

COMPILER DESIGN DEFECTS

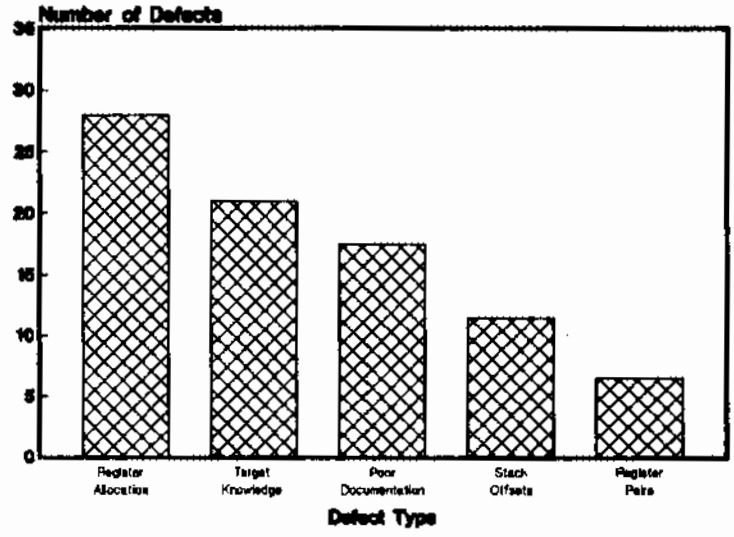


Figure 10

As was pointed out earlier, it can be seen from all three examples that the detailed definition of defects was not consistent from division to division, yet in each case significant understanding and progress was made in eliminating the causes of defects from the development process.

One tool which was used to identify problem causes was the "fishbone" diagram. A high-level analysis of the primary defect category, detailed design defects, is illustrated in Figure 11. This type of analysis is suitable for creating change at the lab level. A similar diagram was done for register allocation defects and some of the others. These led to actions within smaller areas of the lab, since they represented subsets of the overall problem category.

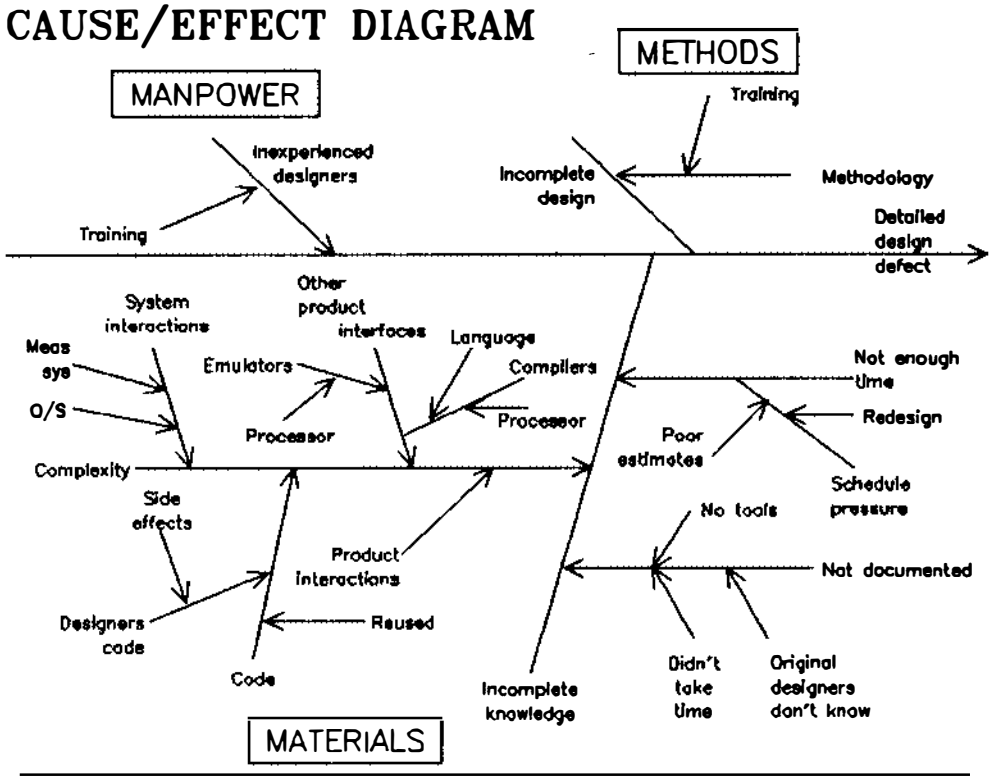


Figure 11

We have seen that for different software development environments, the primary defect categories are significantly different. The techniques for identifying defects involve discipline in recording defects during the software development process. Once the primary defect categories are identified, the causes of defects can be determined and permanently removed, and defects are one of the primary factors which contribute to our inability to accurately estimate.

A Tool for Project Estimation

A fourth division develops communications software which operates very closely with the operating systems software used by all of HP's computers. Their interest in metrics is driven by the need for project control, which includes predictability of schedules and staffing.

This division's first attempt at cost estimation modeling was to lease a software package which implements a model discussed much in the metrics literature. This model was studied in terms of its accuracy and assumptions concerning the development process. A major problem with it was the ease with which managers could manipulate the inputs to the model to get virtually any answer, realistic or unrealistic. Also, the high leasing price made the prospect of developing an in-house tool cost-effective for this division.

The in-house tool, SOFTCOST, was based on a paper written by Robert Tausworth of the Jet Propulsion Laboratory [2]. Here is a description of SOFTCOST's functionality:

1. Estimates project size and difficulty. The Difficulty Factor provided by SOFTCOST is based on various aspects of the project environment, such as product complexity, staff experience, support of the programming environment, etc.

2. Estimates development resources. SOFTCOST approximates the total amount of engineering effort, time, and staffing required for development of the project (from Internal Design through Manufacturing Release).

3. Allows arbitrary resource budgets and performs tradeoffs between time and effort. SOFTCOST allows the user to specify certain budget constraints, and shows what the time/effort/staffing tradeoffs are.

4. Generates a staffing schedule. For large projects, effort is applied in a predictable way, following what is known as a Rayleigh Curve.

SOFTCOST's goal is to provide Project Managers with a valuable comparison between their expectations of a project's behavior and industry-based statistical expectations of that project's behavior. It provides an additional basis for budgeting project time and effort to a project based on estimated confidence limits for the project's successful completion. Further, continued use of this estimation tool can aid in developing an information base of productivity factors which are candidates for improvements.

The model uses some very complicated mathematics. An HP engineer ported the public-domain BASIC implementation into HP-portable PASCAL. The submodels that compose SOFTCOST are each calibrated to certain non-HP data, and the sum total of the models does not reflect a single set of industry data. HP had no data and no instructions for customizing the data file.

After a year of using the first version of SOFTCOST, enough information had been gathered concerning its usability and functionality. HP then created the second major revision, which included a total rewrite of the user manual.

The metrics data collected has shown that for a small number of the division's projects, SOFTCOST predicted the duration within 20 percent and the effort within 30 percent when correction factors were used. These results are shown in Figures 12 and 13.

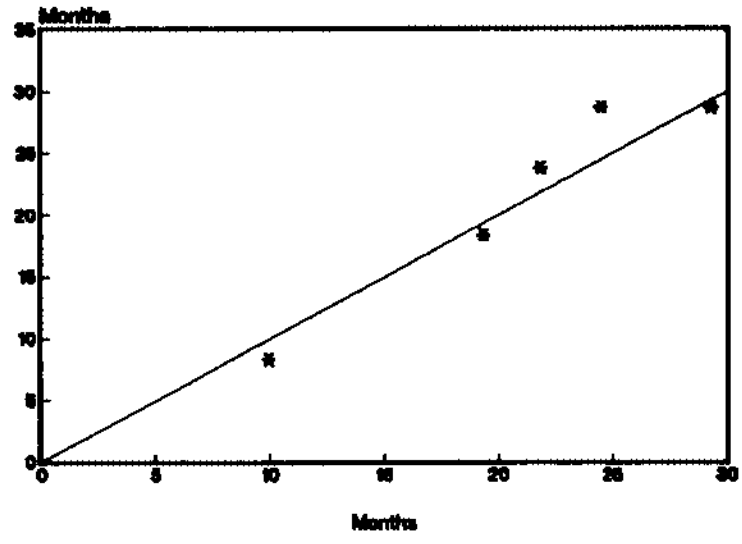


Figure 12

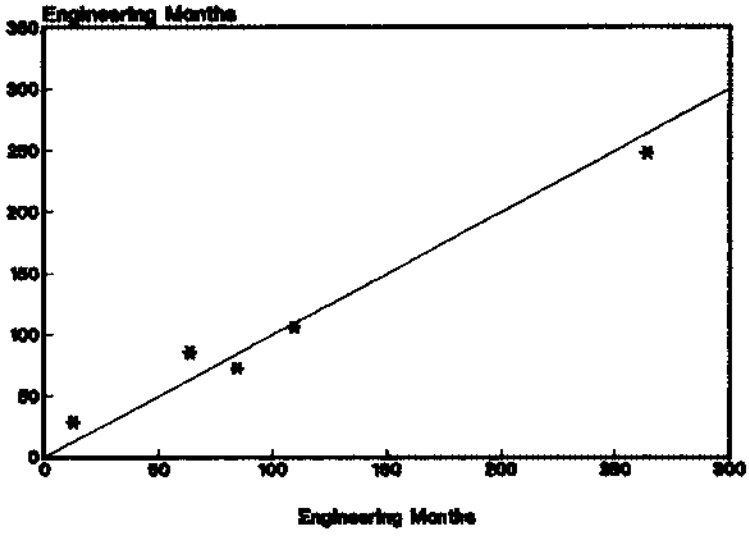


Figure 13

The use of SOFTCOST has spread. Another division doing firmware development found SOFTCOST's estimates to be far too optimistic. However, it was consistently wrong by the same relative amount such that modified SOFTCOST estimates are good predictors. (Again, a limited number of projects have been used. For four projects, an offset factor of 2.5 appeared good.) The need to calibrate the model for a specific development environment gives projects an incentive to collect accurate data for local calibration. It is used as a check against a manager's own expert judgement. In at least two cases, schedules have been revised as a result of the large discrepancy between the managers' initial estimates and the estimates produced by SOFTCOST. SOFTCOST's biggest advantage, however, is reminding the project manager in the investigation phase of most of the factors that affect a project's schedule.

The next step is to study the model itself and try to understand how to make it more responsive to factors which have a big impact on project schedules in the HP environments. As data is collected on projects producing different software types, the model will be calibrated to give more accurate estimates in each software environment.

CONCLUSION

Probably the most remarkable aspect of the Software Metrics Program at HP has been how quickly measurable results have been attained. Some aspects of measurements have spread to virtually all software development labs within the company, and from the examples included in this paper it can be seen that significant changes have been achieved in a relatively short time, particularly in understanding defects in all of the major development categories. In some cases measurements are limited to individual projects in a lab, but in many cases the process is now virtually across entire labs.

The original metrics accepted by the HP Software Metrics Council are internal standards now, subject to growth and change over time as various experiments define new needs. The paper forms originally created over a year ago have been supplemented by some tools which meet collection and presentation needs. In addition, A set of three high-level management graphs, based upon data from the standard metrics, have been accepted as the basis for evaluating software quality and productivity throughout HP at the division level. These graphs (scattergrams) portray productivity, pre-release quality, and post-release quality.

Finally, the major issue of predicting software development costs and schedules is being addressed by both measurements to help calibrate our ability to estimate, as well as tools, to help standardize and ensure completeness. In some HP environments, the time necessary to achieve desired quality goals can be computed today so that the necessary resources can be allocated. This predictive ability must be extended to other parts of the development process and the accuracies of prediction must continue to improve until software development is really a predictable engineering discipline

BIBLIOGRAPHY

1. C. Sieloff, "Software TQC: Improving the Software Development Process Through Statistical Quality Control," HP Software Productivity Conference Proceedings, (April, 1984).

2. R. C. Tausworth, "Software Specifications Document, DSN Software Cost Model," Jet Propulsion Laboratory, Pasadena, CA, 1981.

BIOGRAPHIES

Bob Grady and Deborah Caswell

Bob Grady manages the Software Engineering Lab of Hewlett Packard Company. SEL is a Corporate Engineering function responsible for software tools development environments, and metrics. During his 15 years with HP, he has managed software development projects in the areas of compilers, measurement and control systems, firmware development and manufacturing automation. Mr. Grady holds a BSEE from MIT and MSEE from Stanford University.

Deborah Caswell is a software development engineer for the Software Engineering Lab of Hewlett Packard Company. She was instrumental in initiating and coordinating the software metrics effort at HP. During the three years she has been with HP, Ms. Caswell has developed automated testing programs and other software engineering tools. She has a BA in computer science from Dartmouth College and is pursuing an MSCS at Stanford University.

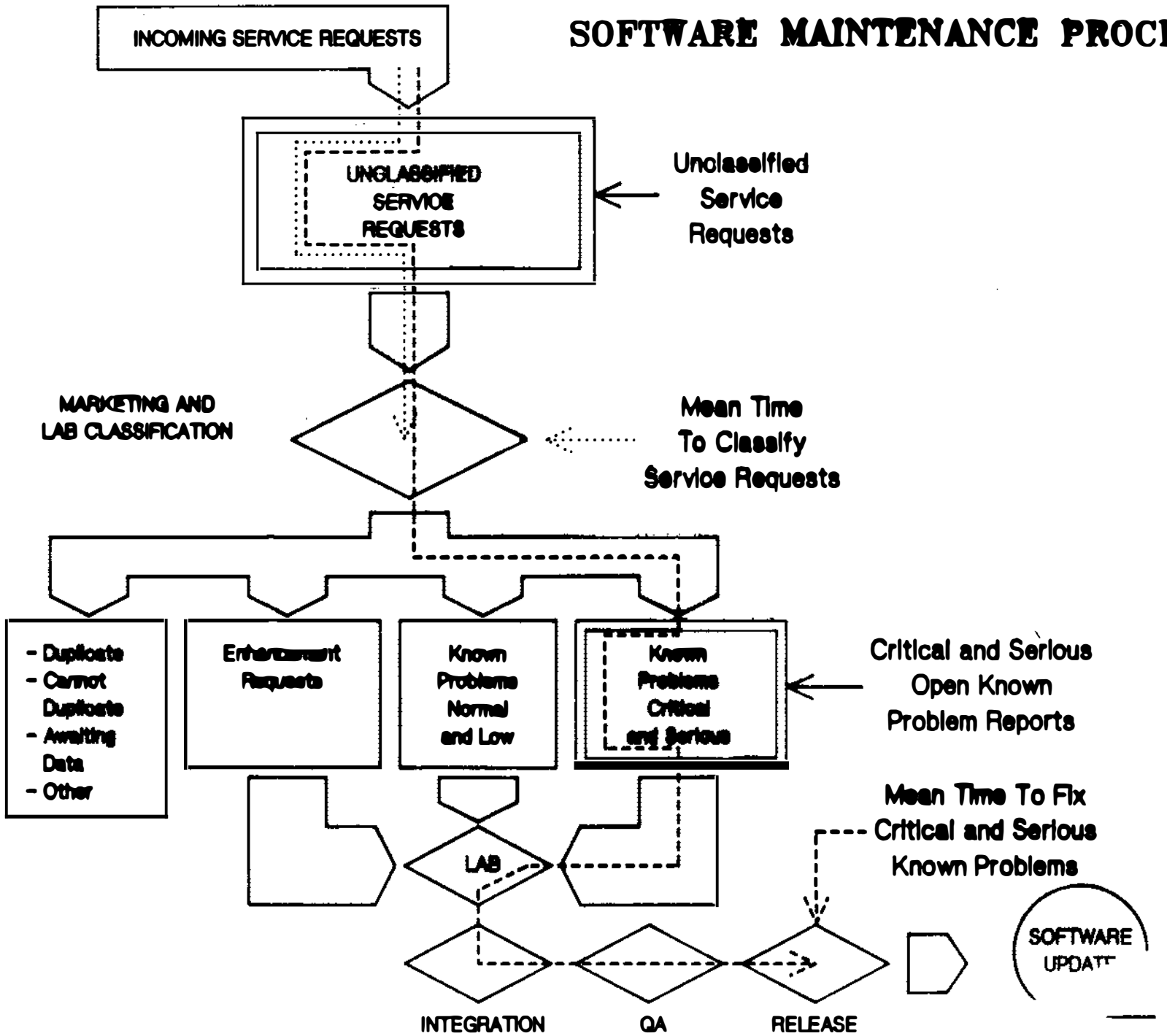
HP SOFTWARE DEVELOPMENT ENVIRONMENTS

INFLUENCING FACTORS	MICROPROCESSOR	SOFTWARE SYSTEMS	APPLICATIONS	END-USER
TEAM SIZE	Small	Large *	Large *	Small
MARKET SIZE +	Small → Large	Large	Large	Small
LANGUAGE	Asmb.Pascal	C/Pascal/SPL	High-level	All
USER	Single	Multiple	Heavy, multiple	Single
TIMING	Important, sometimes critical	Critical	Mild importance	Varies in importance
METHODOLOGY	Few standards	Control-oriented	Data-oriented	Varies
COST OF CHANGE AFTER RELEASE	Large → Huge	Large	Moderate	Small
MAJOR APPLICATION CONCERN	Timing of ext. processes	Process interaction peripheral generality, recovery	Data integrity, user interface, portability	Single problem oriented

* Project sizes not large, but generally aggregates of projects are large.

\+ As measured in number of customer sites.

SOFTWARE MAINTENANCE PROCESS



CATEGORIES OF SOFTWARE DEFECTS

A. USER INTERFACE/INTERACTION

1. User needs additional data fields
2. Existing data needs to be organized/presented differently
3. Edits on data values are too restrictive
4. Edits on data values are too loose
5. Inadequate system controls or audit trails
6. Unclear instructions or responses
7. New function or different processing required

B. PROGRAMMING DEFECT

1. Data incorrectly or inconsistently defined
2. Initialization problems
3. Image processing incorrect
4. View processing incorrect
5. Incorrect language instruction
6. Incorrect parameter passing
7. Unanticipated error condition
8. MPE file handling incorrect
9. Incorrect program control flow
10. Incorrect processing logic or algorithm
11. Processing requirement overlooked or not defined
12. Changes required to conform to standards

C. OPERATING ENVIRONMENT

1. Terminal differences
2. Printer differences
3. Different versions of systems software
4. Incorrect JCL
5. Incorrect account structure or capabilities
6. Unforeseen local system requirements
7. RAPID problem

DEFECT DISCOVERY SCHEDULE

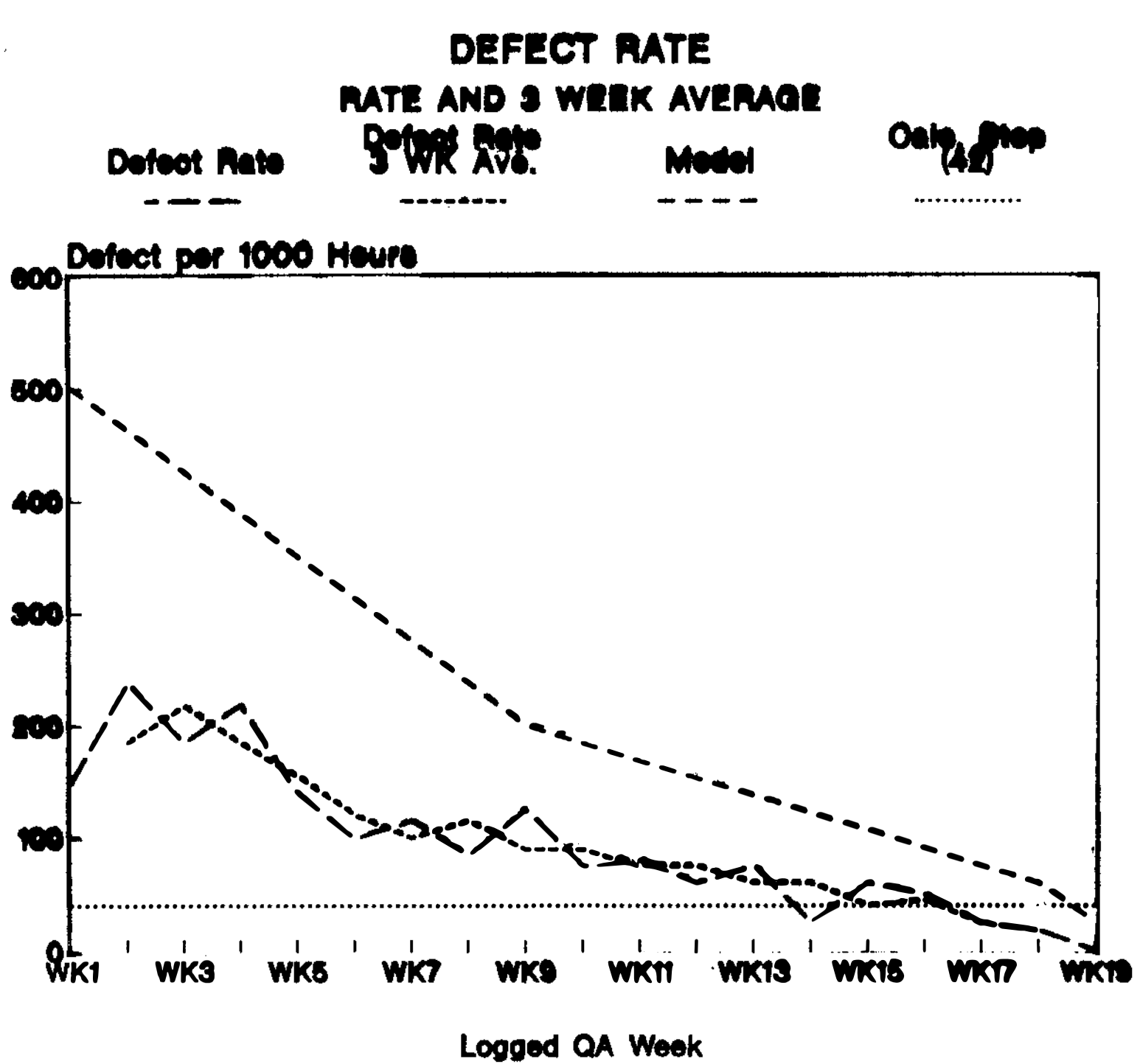
25% of defects are found in 2 hours/defect (rate of .50 defect/hour)

50% of defects are found in 5 hours/defect (rate of .20 defect/hour)

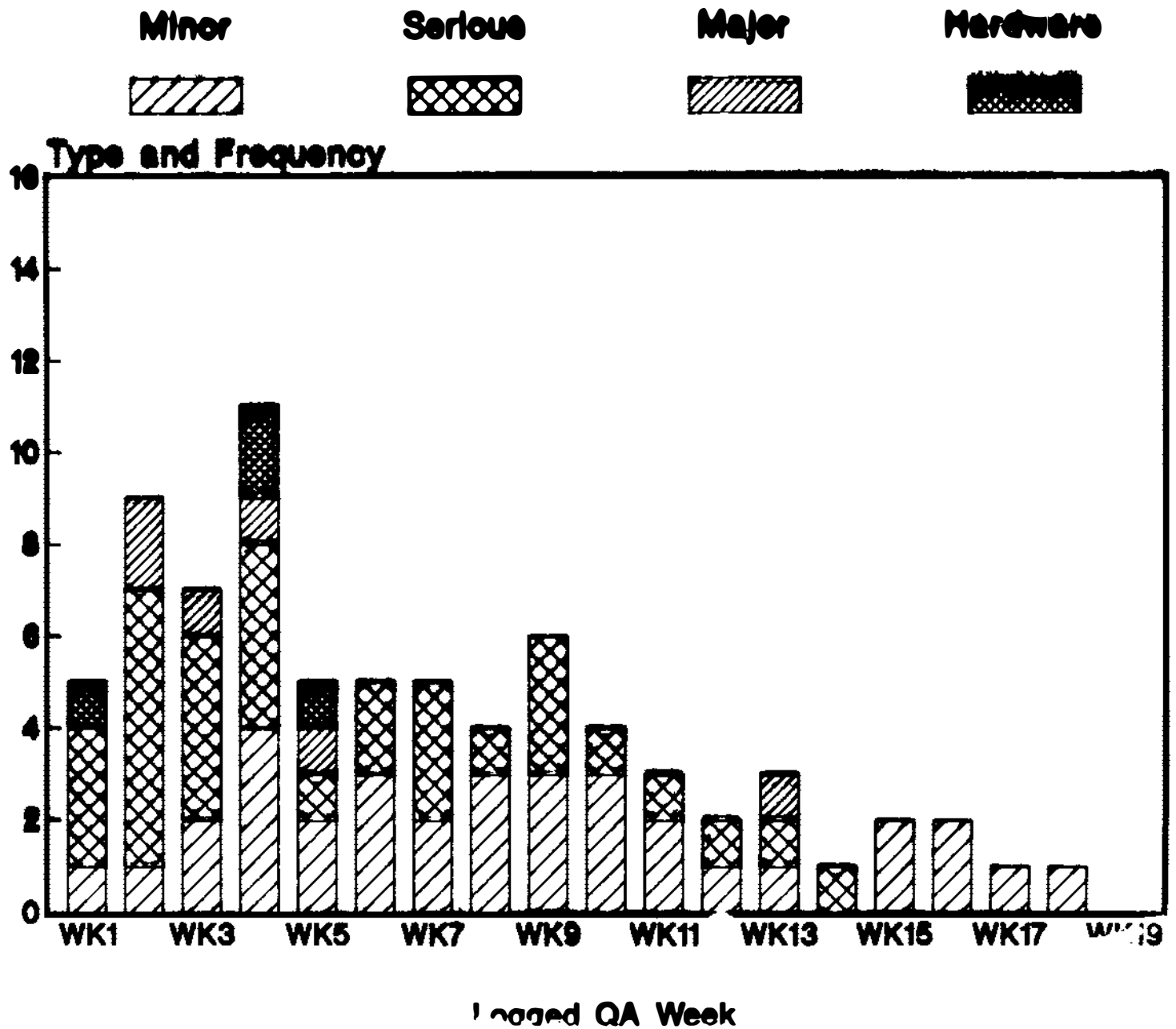
20% of defects are found in 10 hours/defect (rate of .10 defect/hour)

4% of defects are found in 20 hours/defect (rate of .05 defect/hour)

1% of defects are found in 50 hours/defect (rate of .02 defect/hour)

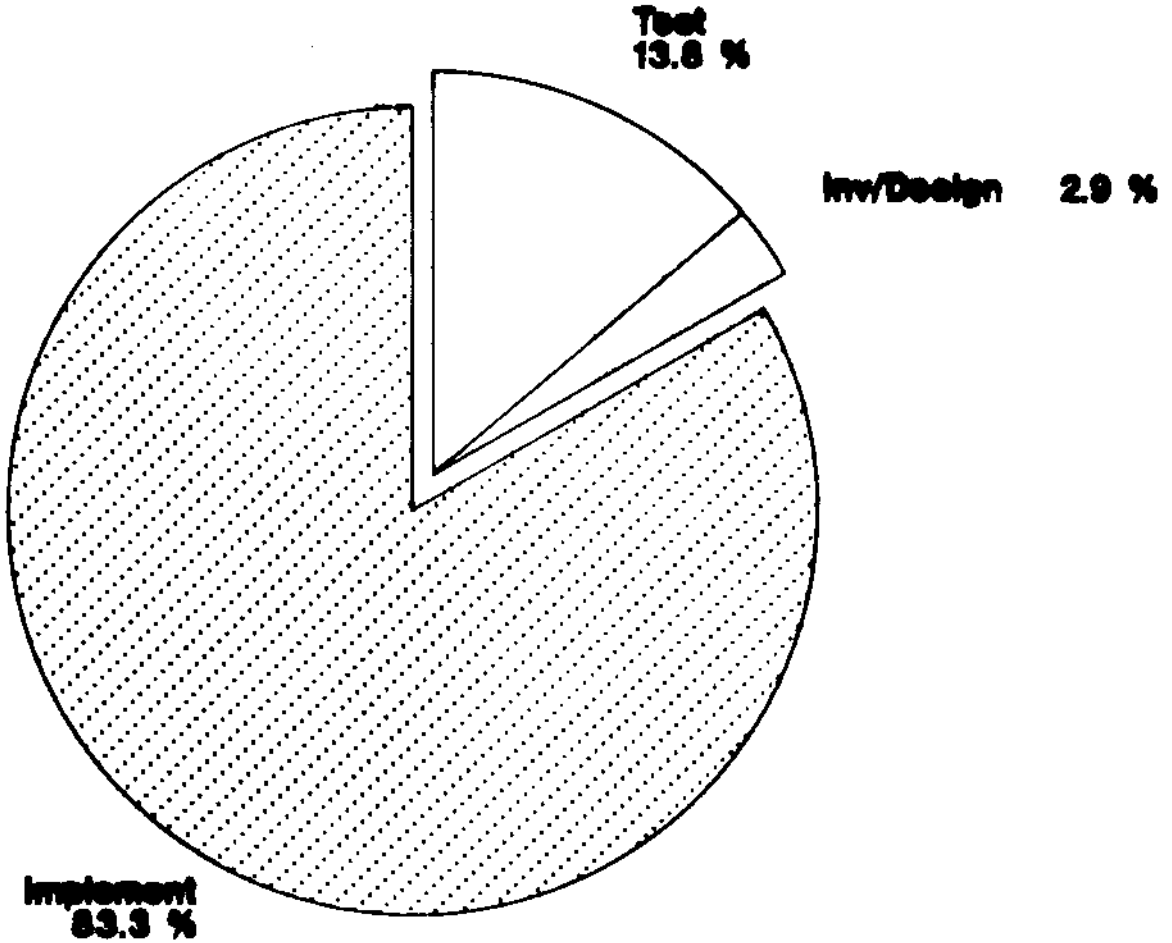

DEFECT RATE
RATE AND 3 WEEK AVERAGE
Defect Rate
3 WK Ave.
Model
Defect per 1000 Hours
600
500
400
300
200
100
0
WK1
WK3
WK5
WK7
WK9
WK11
WK13
WK15
WK17
WK19
Logged QA Week

DEFECTS

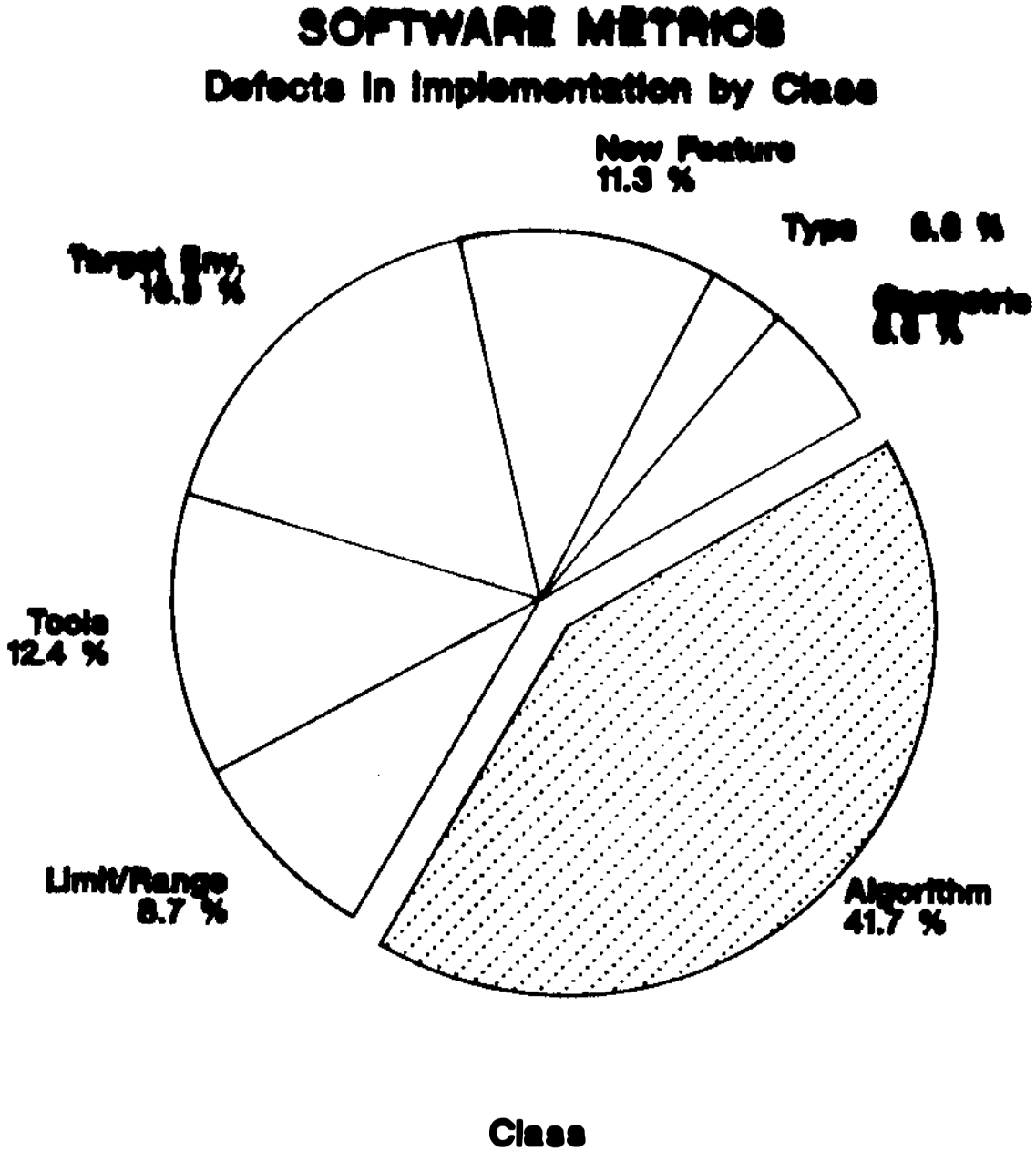

Minor
Serious
Major
Hardware
Type and Frequency
16
14
12
10
8
6
4
2
0
WK1
WK3
WK5
WK7
WK9
WK11
WK13
WK15
WK17
Logged QA Week

SOFTWARE METRICS

Software Defect Summary

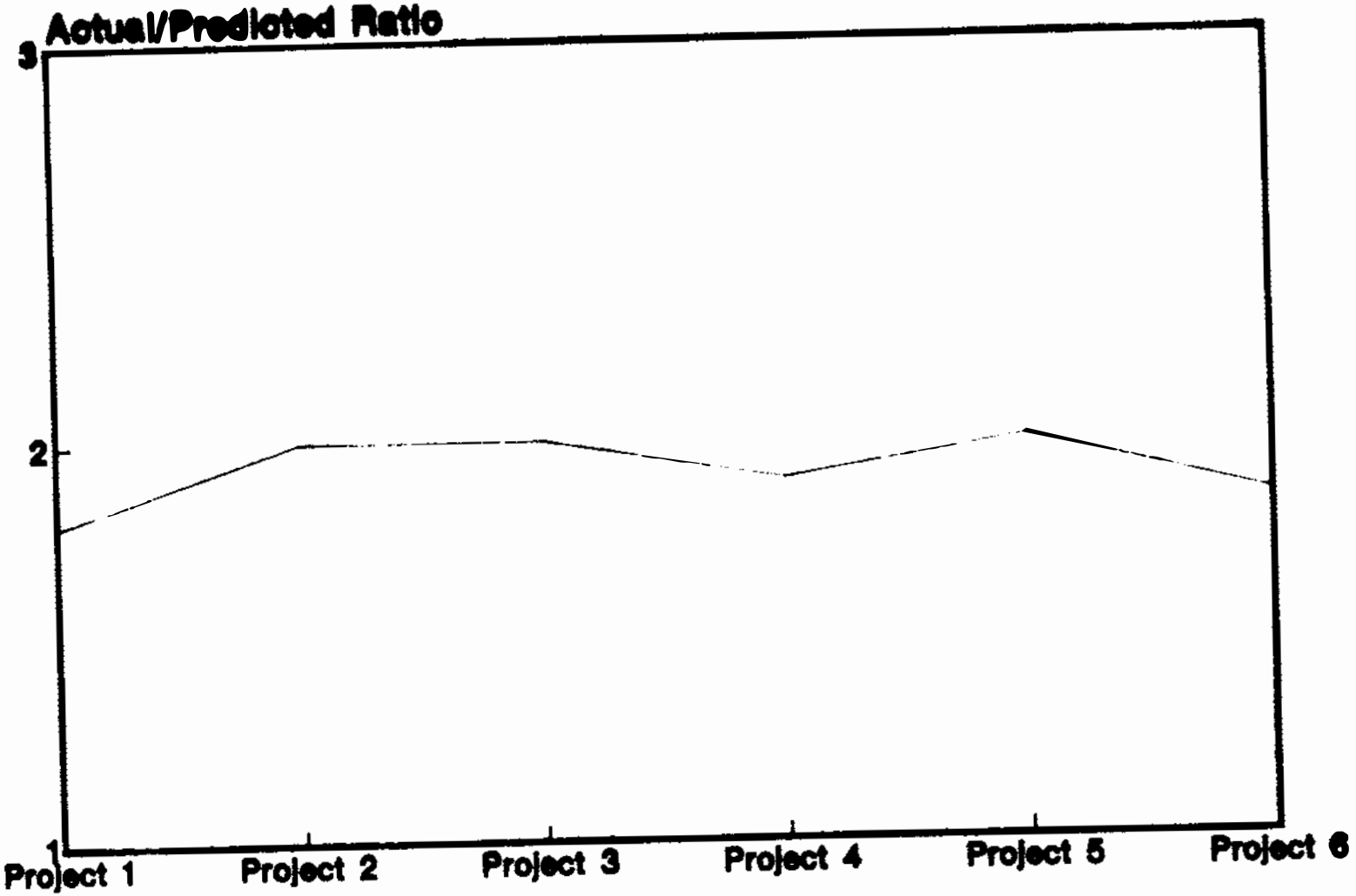


Defects by Project Phase

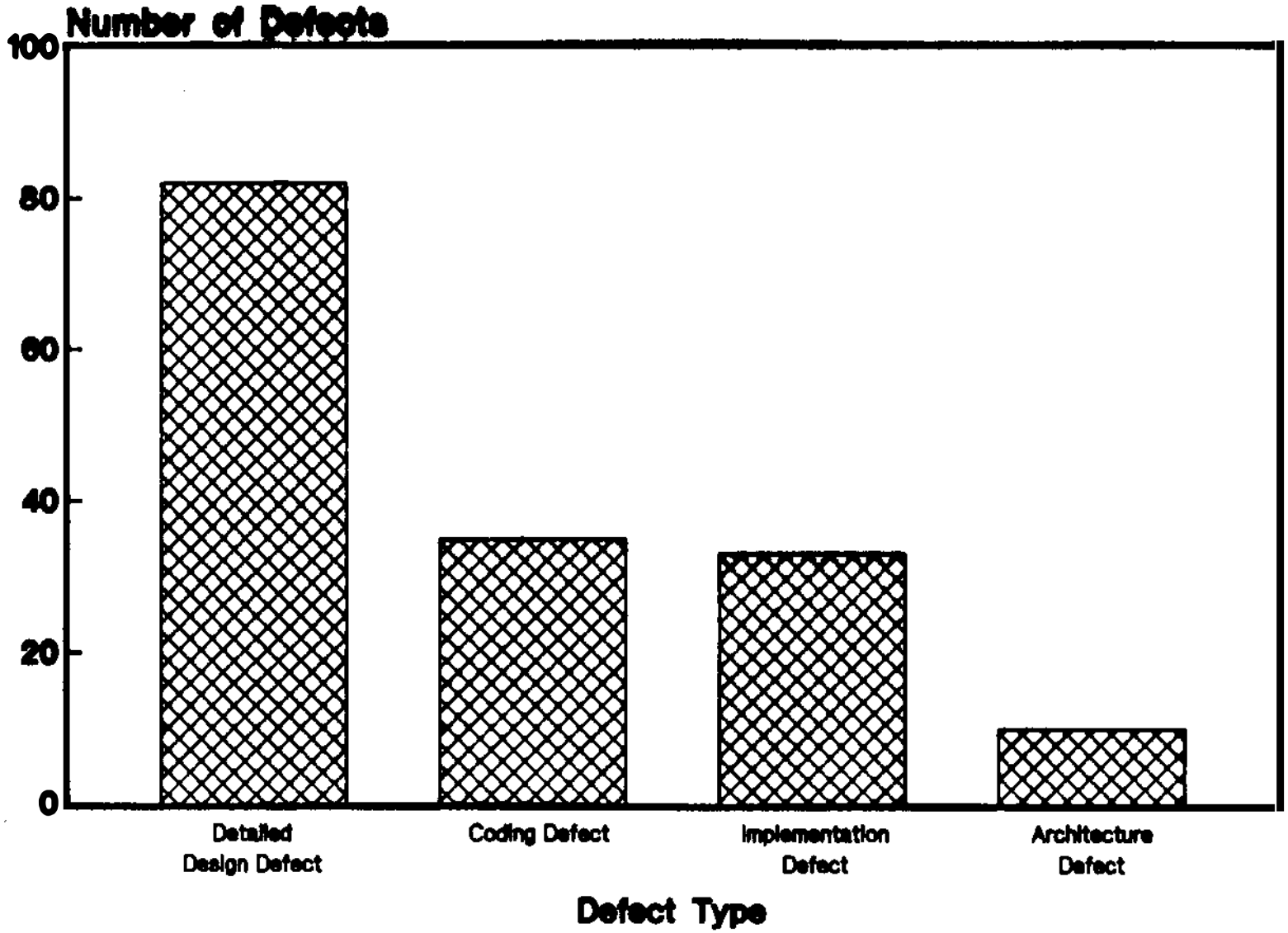

SOFTWARE METRICS
Defects in Implementation by Class
New Feature
11.3 %
Type 6.8 %
Tools
12.4 %
Limit/Range
8.7 %
Algorithm
41.7 %
Class

SOFTWARE PREDICTABILITY

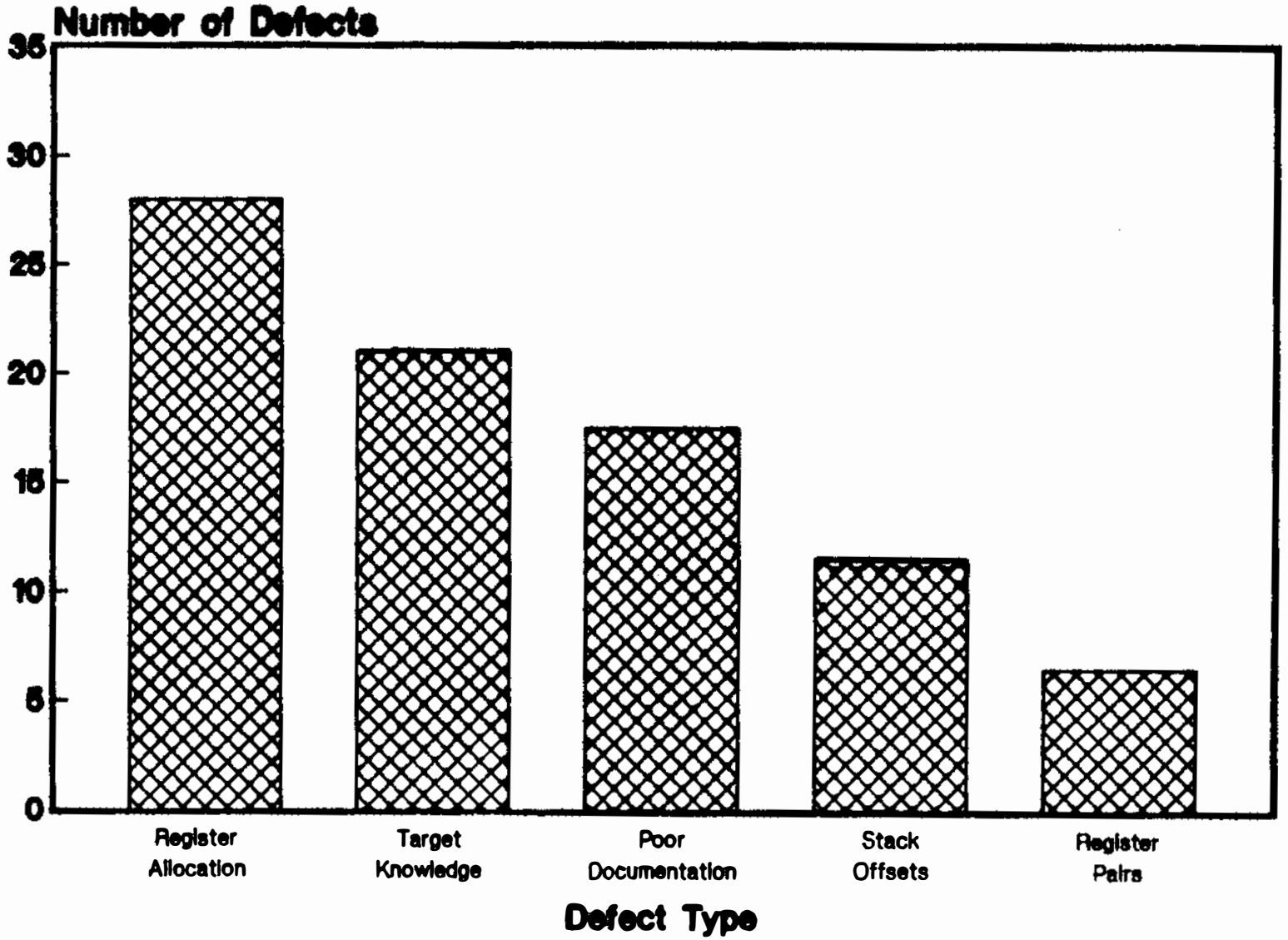
Development Effort



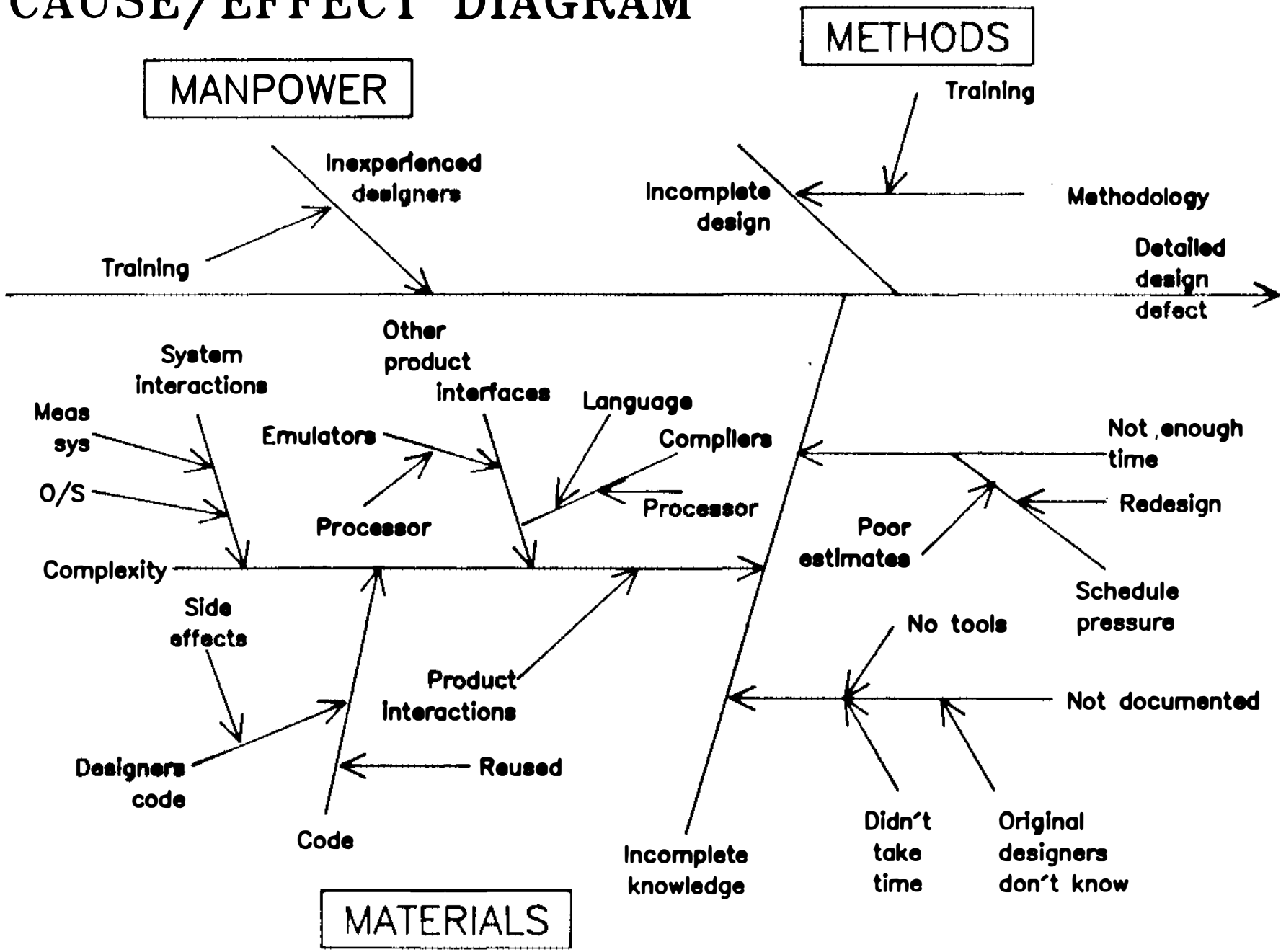
COMPILER DEFECTS



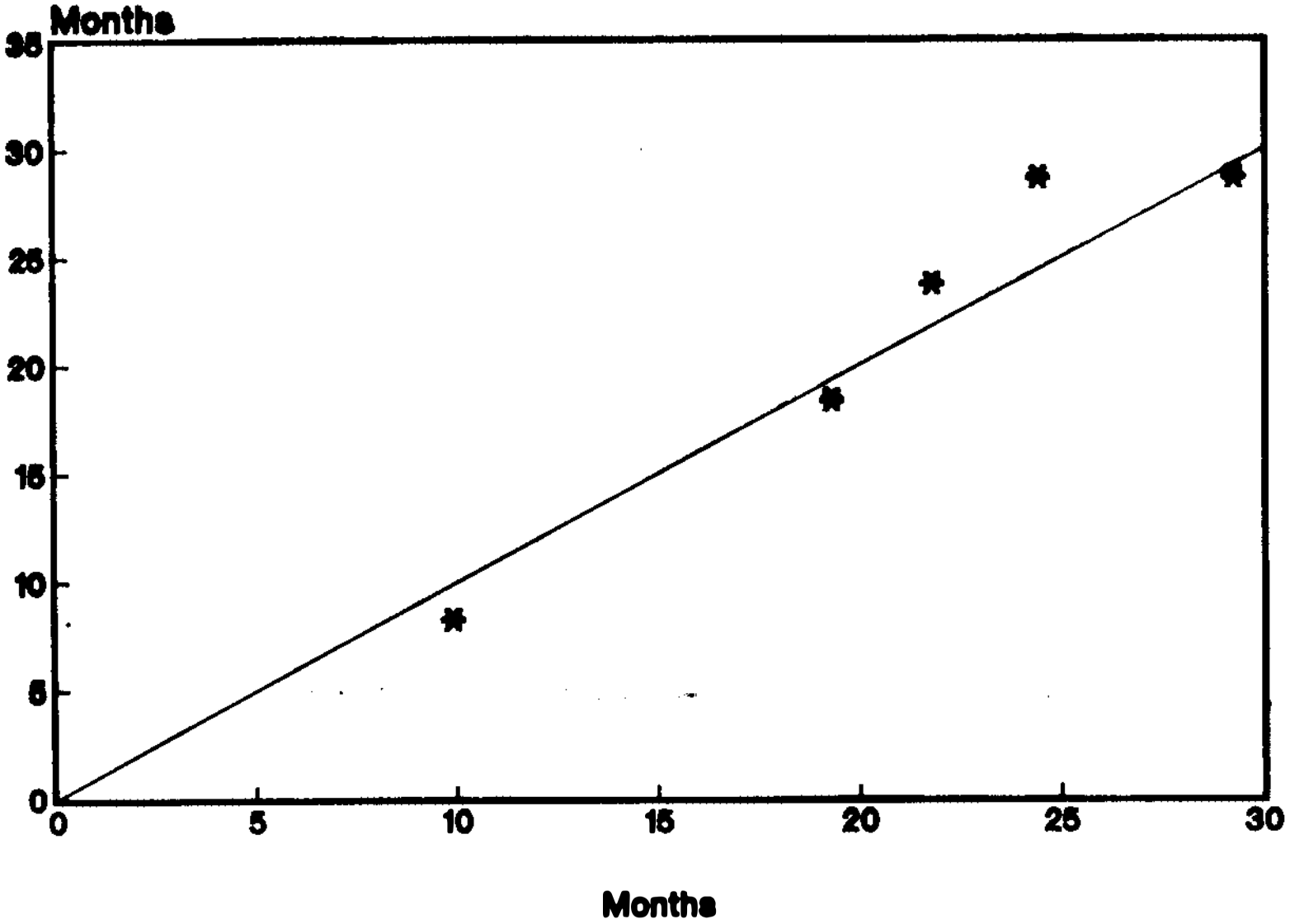
COMPILER DESIGN DEFECTS



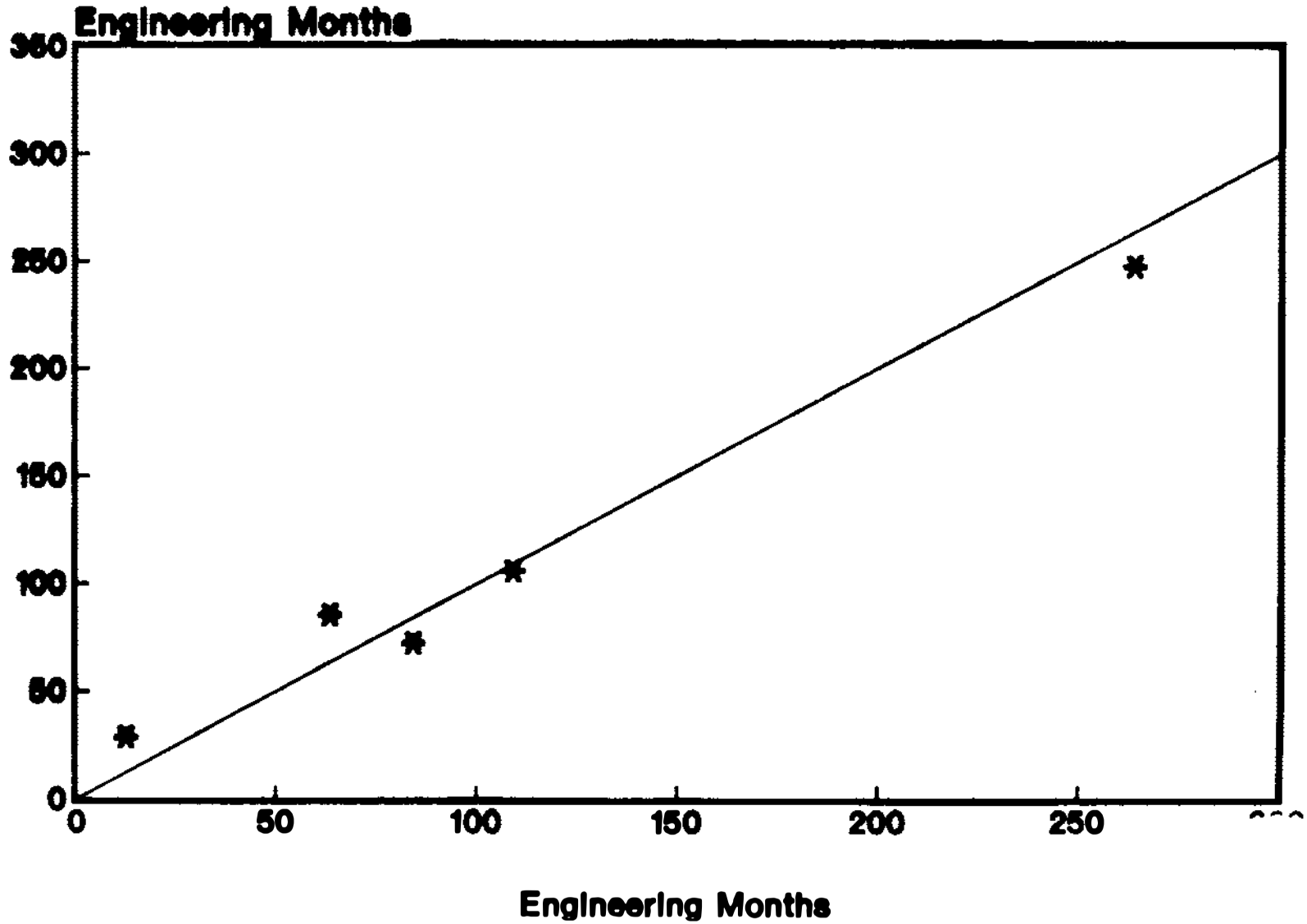
CAUSE/EFFECT DIAGRAM



SOFTCOST ESTIMATE OF PROJECT DURATION



SOFTCOST ESTIMATE (CORRECTED)
DEVELOPMENT EFFORT



Session 3

PANEL SESSION

"The Pros and Cons of Rapid Prototyping"

Panelists:

Rick Samco, Mentor Graphics
Robert Babb, Oregon Graduate Center
Will Clinger, Tektronix, Inc.
Dave Kerchner, Floating Point Systems, Inc.

Moderator:

LeRoy Nollette, Tektronix, Inc.

Overview Prepared by Will Clinger, Tektronix, Inc.

OVERVIEW OF RAPID PROTOTYPING

by William Clinger

TEKTRONIX, INC.

Rapid prototyping is a technique used in the early stages of software development. The prototype is an executable software specification. In many cases the prototype is obtained by translating an existing specification into a programming language, but in some cases the prototype is itself the first specification. The prototype is developed relatively rapidly and cheaply by using a high level programming language, by using existing code where possible, by using simple but inefficient data structures and algorithms, and by ignoring frills.

Rapid prototypes help to catch specification errors early, before they waste much programming effort. For example, the specification for a numerical calculation can be tested by translating it into APL. Specifications expressed in a functional language such as lambda calculus can be translated into Lisp. Specifications expressed using first order logic can often be translated into Prolog.

Unfortunately, most specifications are informal and imprecise. In such cases the rapid prototype serves as the first formal specification of the software to be built. The prototype can even be used to develop requirements. Though it is difficult to specify an interactive user interface that makes significant use of graphics, for example, a prototype written in Smalltalk can be used to explore the possibilities.

Given enough care, rapid prototypes can also be used to explore the feasibility of novel implementation strategies. Such prototypes can be thought of both as specifications and as simulators for the final software product.

Once the rapid prototype is complete, it should remain as an important component of the design documentation. To be most useful, rapid prototypes should be written and commented as carefully as any other software.

Rapid prototypes are rapid and cheap only by comparison with the software development process they support. The final product is more expensive to build than the prototype because it must have better performance, more extensive features, and better documentation for its intended users.

Rapid prototyping should not be confused with sloppy programming, poor internal documentation, and buggy code. Sloppy programming, poor internal documentation, and buggy code should be confined to the later stages of software development, where they are cheaper

to fix. The purpose of rapid prototyping is to remove sloppy thinking and buggy specifications from the early stages of software development, where mistakes are most expensive.

Rapid prototyping is not always cost-effective. A software project that begins with a detailed formal specification that is known to be correct does not need a rapid prototype, and it would be a waste of time and money to construct one. Most projects, of course, begin with a detailed informal specification that is believed to be correct. In such cases a rapid prototype can increase confidence, but the prototype must be weighed against its cost. The worst possible thing to do in such a situation would be to construct a prototype, but to construct it hurriedly and sloppily in order to hold down costs. A sloppy and hurried prototype is all cost and no benefit.

To make best use of rapid prototyping, programmers need to be trained in the use of formal specifications and should understand the principles of programming language semantics and program verification. Programmers must understand and use abstract data types to separate the objects that appear in the specification from their inefficient implementations in the prototype. Finally, programmers need to learn about the software development process, lest they view the prototype as a quick and dirty throwaway implementation, undertaken perhaps for practice; many programmers will think enough of their talents to believe they can get it right the first time.

Rapid prototyping should be supported by an excellent interactive programming environment so the prototype can be developed as quickly as possible. The basic tool is an executable specification language or a programming language whose semantics is clean enough to be used as a specification language. The language should supply a convenient means of synthesizing new abstract data types, predefined modules for the most common data types and operations, facilities for reading and printing objects of all types, automatic storage management, and a convenient I/O package that includes support for graphics displays. In general, the programming environment should minimize the amount of new code that must be written to develop the prototype, and should make it easy to debug whatever new code is written.

Among well-known programming languages, APL, Lisp, and Smalltalk are the best for rapid prototyping. Prolog is also good, but the currently available Prolog programming environments are primitive.

The hardware required to test rapid prototypes may have to be faster than hardware used to run production software, both because rapid prototypes are very slow and because the prototype testing time is usually on the software project's critical path.

Session 4

TESTING AND PROBLEM REPORTING, I

Titles and Speakers:

"A Tool for Analyzing the Logic Coverage of Source Programs"
Arun Jagota, Intel Corp.

"TCAT/C: A Tool for Testing C Software"
Edward Miller, Software Research Associates

"A Unix Based Software Development Problem Tracking System"
Gordon Staley, Hewlett Packard Co.

A Tool For Analyzing The Logic Coverage Of Source Programs

Arun Jagota

Oregon Micro-proccessor systems
Intel Corporation

This paper describes a software tool which can aid in analyzing the logic coverage of source programs. The first section explains what logic coverage is and how it can be done. The next section gives an overview of the tool and shows how it can be used. The final section presents a general strategy for its use and a summary of results and observations from my experience in using the tool.

---

1. What Do We Mean By Logic Coverage?

It is a measure of how well the internal control flow logic of a source program has been exercised. The primary goal is to find errors in the program's logic. One simple form of logic coverage is to check if all statements in a program can be executed at least once. Consider the following example.

```
X = Y;
if X > Y then S1
else S2;
```

It is obvious that S1 can never be executed. This form of coverage is called statement coverage. Now, is statement coverage sufficient for detecting all kinds of logical errors? No, and the following example should show why not.

```
X = Y;
if X = Y then S1;
S2;
```

There is an obvious error in the program due to the fact that the conditional expression in the IF statement can never take the false value. Yet, statement coverage would not detect this error because both statements are executed. Hence, to detect such errors, we would need to cover both the branches of IF statements (and other two-way decisions). This kind of coverage is known as branch coverage. Again, is branch coverage sufficient for detecting all kinds of logical errors? No, and the following example should again show why not.

```
X = Y;
if (X = Y) and (Y > 2) then S1
else S2;
```

Again, there is an obvious error in the progam which is due to the fact that the first condition in the IF statement is always true. But, this error cannot be detected by branch coverage because we can cover both branches of the IF statement by running the program twice with Y = 2 and Y = 3. How do we detect such errors then? We need to cover both values (TRUE,FALSE) of each condition in the IF statement. But, simply doing this does not guarantee branch coverage. Consider the following IF statement.

```
IF C1 and C2 THEN —
```

Suppose that it's execution history shows the following coverage.

(1) C1 is false and C2 is true.
(2) C1 is true and C2 is False.

Each condition (C1 and C2) individually takes on both the values (TRUE,FALSE) at least once. Yet the THEN part of the IF statement is never executed.

So, what we really need to do is to cover all combinations of outcomes of each condition in an IF statement, and in two-way decisions, in general. This is known as multi-condition coverage [1]. A condition is defined as a relational expression separated from other conditions by the logical operators AND, OR or XOR. NOT is the only logical operator allowed as part of a condition.

1.1. What about multi-way decisions?

So far, we have confined our decision to two-way decisions, in particular IF statements. In addition to such decisions, most programming languages allow multi-way decisions, for example Pascal case statements. For such decisions, we need to ensure that each branch (case alternative) is executed at least once. This is known as case coverage.

1.2. How can we automate the process of multi-condition and case coverage?

For each two way decision in the program, we need to insert hooks for monitoring the run time values of all its conditions. For each case statement in the program, we need to insert a hook to monitor the values of the associated case expression. The range of values that needs to be monitored depends on the number of case alternatives in a case statement. The number of hooks that need to be inserted is not astronomically large since the number of decisions in a program is bounded by its size.

2. Overview of the Logic Coverage Tool

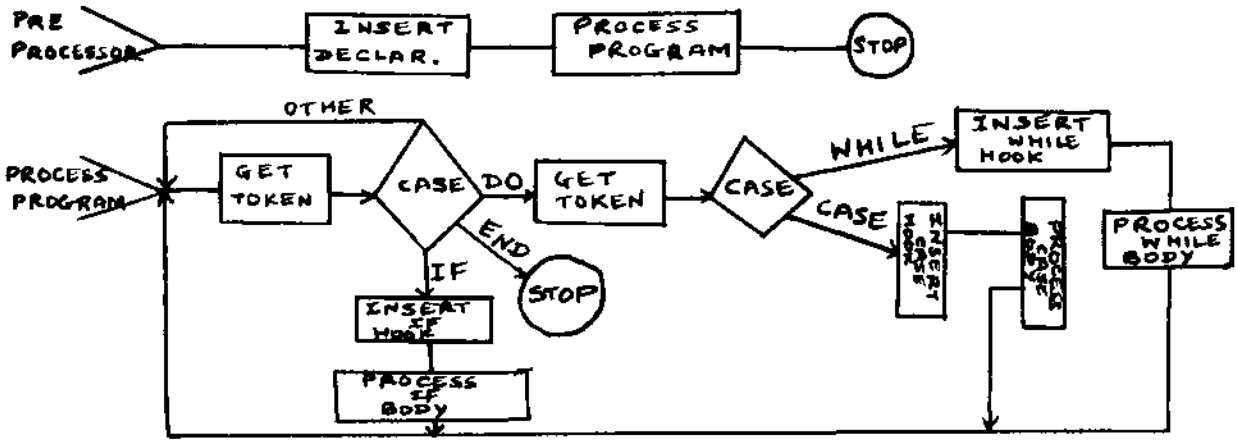
The logic coverage analyzer is targetted for PL/M source programs. PL/M is an Intel developed language which supports three kinds of Pascal like control flow statements which are IF .. THEN, DO WHILE .. and DO CASE .. . The analyzer performs multi-condition coverage on IF and DO WHILE statements. A DO WHILE statement is interpreted as having two branches-execute the loop or skip it. It performs case coverage on DO CASE statements. It can handle the most complicated forms of nesting of IF, DO WHILE and DO CASE statements. It can also handle very complex boolean expressions in IF and DO WHILE statements.

The Analyzer is partitioned into three parts-Preprocessor, Monitor and Reporter. The preprocessor inserts the hooks for IF, DO WHILE and DO CASE statements. The modified program can then be linked in with the second part, the MONITOR. It can then be executed with any set of test data. The monitor uses the run time values supplied by the hooks to perform the multi-condition and case coverage. At the end of a session, the monitor writes its coverage status onto a file. This makes it possible to run the subject program in multiple sessions and use the MONITOR to accumulate coverage data. The third part, the reporter, interprets the contents of the coverage file to produce a coverage report.

2.1. Implementation
The Preprocessor

The preprocessor is coded in standard Pascal. It uses a recursive descent LL1 grammar for parsing IF, DO WHILE and DO CASE statements. The basic processing algorithm is shown below. There is a look ahead of one symbol everywhere except for DO WHILE and DO CASE statements.

The algorithm is fully recursive.



The preprocessor can process multiple PL/M modules together. The only condition is that the first one of the multiple modules should be the main module. This is because the main module is treated differently as shown below.

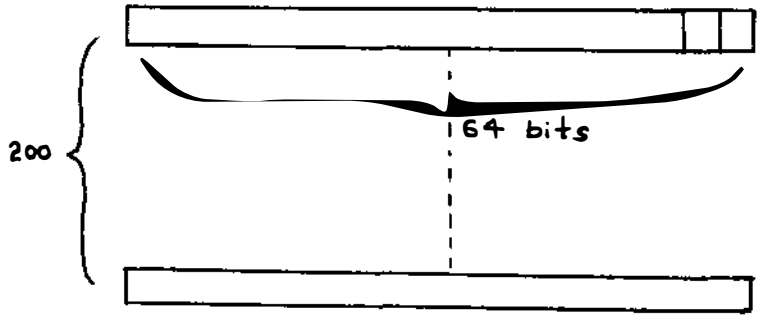
```
        Main Module
Main: do;
  .
  .
  Call Init; --Initialises the MONITOR
  .
  .
  Call SaveInfo; --Saves the Coverage into a file
End Main;
```

The first executable statement should be a call to a MONITOR routine which initialises the MONITOR. The last executable statement should be a call to a MONITOR routine which saves the coverage results in a file. The preprocessor inserts these calls into the main module at the appropriate places.

The Monitor

The monitor is written in PL/M. It can process a maximum of 200 statements for multi-condition coverage (IF + DO WHILE statements). If there are more such statements, then their coverage is ignored. The maximum number of conditions in each statement can be six. Any more conditions are ignored. The monitor can also process a maximum of 200 case statements. There can be a maximum of 32 case alternatives in each case statements. Anything exceeding these limits is ignored similarly.

The information that needs to be recorded, especially for multi-condition coverage can be very large. In the maximum case, for instance, we need to record 12,800 boolean values ($200 * 2^6$). A unique scheme is used to represent this much information in just 1600 bytes.



The maximum number of possible combinations for doing multi-condition coverage of one statement is 64 (2^6). Hence, we use 64 bits for the multi-condition coverage of each statement. Each combination is associated with a particular bit. A 0 for that bit indicates that the combination hasn't been covered. A 1 indicates that the combination has been covered.

The Reporter

It is also coded in PL/M. It interprets the logic coverage data from the coverage file and creates a report in the following format.

Multi-Condition Coverage

Statement Type	Line No	No Of Conditions	Condition Combinations Not Covered
IF	-	-	- - - - - - - - -
-			
WHILE	-	-	- - - -

Case Coverage

Line No	No Of Cases	Cases Not Covered
-	-	-
		-
		-

3. Example

The following example illustrates how the three parts of the logic coverage analyzer can be used together.

The original program is:

```
read(a,b);
IF (a>2) AND (b<1) then <S0>;
```

After preprocessing, the subject program looks like this:

```
DECLARE c array(6) BYTE EXTERNAL;
--c is used to pass condition values to if probe.
read(a,b);
DO;
  c[1] = (a>2);
  c[2] = (b<1);
  CALL ifprobe(1,2);
  IF c[1] and c[2] then <S0>;
END;
```

The array c is declared in the MONITOR and is used to pass the condition values to it. "ifprobe" is the MONITOR hook which processes these values. Its first parameter indicates the index of the "if" statement being processed. The second parameter indicates how many conditions the "if" statement contains. Now, let us run the altered subject program with the following test data.

a	b
3	3
3	0

We can now invoke the reporter to show us the current level of coverage.

Statement Type	Line No	No Of Conditions	Combinations Not Covered
IF	2	2	False False False True

The report clearly shows that we have not completely exercised the logic of the IF statement. Specifically, we can see that the following situations have not been covered.

(1) a<=2 and b>=1
(2) a<=2 and b<1

4. Summary of usage

Originally, this tool was implemented for analyzing single PL/M modules. It was tested with quite a few single module PL/M programs (less than 100 lines each). But once it was ready, and I decided to test a large PL/M program, I realised that the preprocessor had to be modified to process multiple modules. Once this was done, the PL/M program was preprocessed. There were 56 IF and DO WHILE statements and 18 DO CASE statements (The preprocessor gathered this information). The program spanned two modules. But, when I tried to execute it I ran into a problem. I had assumed that the program had a single exit point (the last statement of the main module) and this is where I inserted CALL SAVEINFO. But, evidently, PL/M allowed the program to exit from any point under certain conditions. Since my preprocessor had not made such allowances, I had to manually insert CALL SAVEINFO 's at all such points.

Once this was done, I executed the program and it worked perfectly. Test data was fed interactively and the coverage was seen to correspond to it. In fact, even with very limited test data, I succeeded in detecting a program error- a WHILE loop which would never be entered.

5. A testing strategy based on analyzing logic coverage

The source program should be driven with test data derived solely through its functional specifications, in other words with Black box test data. This will make it easier to correlate the internal control flow logic of the program to its specifications. The analyzer will detect which logic has not been excercised so far. There could be three reasons why the logic wasn't covered.

(1) The test data was insufficient.
(2) There were logical errors in the program.
(3) Some combinations in multi-condition coverage were not intended to be covered. An excellent example of this is

    if (X = 1) and (X = 2) then S1;

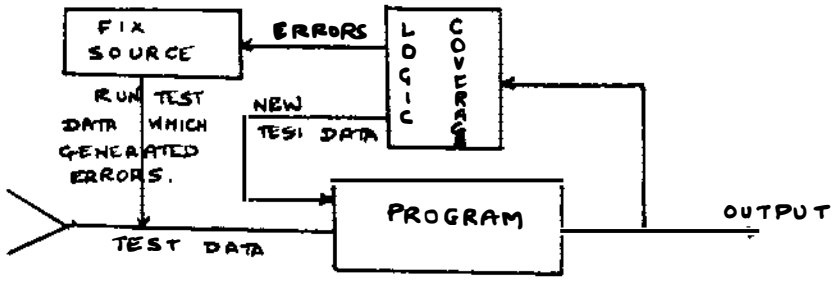
    It should be obvious that both conditions cannot be true at the same time.

Close inspection of the source code will usually give us a clue as to which reason applies to individual cases of incomplete logic coverage.

Let us examine the role of an analyzer in selecting additional test data. Analyzing logic coverage is one of

the best ways of receiving feedback on how exhaustive test coverage has been. The primary reason for this is that the functional specifications of a program are usually not detailed enough to cover all the program's logic and hence test the program under all possible situations. This fact is especially true for "memory" programs - that is programs whose output is dependent not only on its input, but also on the state of the environment at that time. Such programs have control flow logic which takes care of environmental factors. The functional specifications, usually do not cover such logic too well and hence monitoring this logic through a logic analyser provides a very useful insight into how well it has been exercised, and hence, how well this part of the program has been tested.

In conclusion, then, logical coverage analyzers can serve two functions. They can detect logical errors and they can aid us in estimating how complete our coverage has been. To do these optimally, we should use a logic coverage analyzer as a feedback element in a testing loop (as shown below).



6. Additional uses-Measuring control flow complexity

The preprocessor, in addition to its normal function, gathers the following statistics. It indicates the total lines of code, the number of IF statements, the number of WHILE statements, the number of CASE statements and the average number of conditions in IF and WHILE statements and the average number of alternatives in CASE statements.

7. References

1. Myers Glenford J, 1979. The Art Of Software Testing. John Wiley & Sons.

BIOGRAPHY

Arun Jagota

Arun K. Jagota is a software engineer at Intel Corporation in Hillsboro, Oregon. He holds an MS in computer science from the University of Kansas and a BTech electrical engineering degree from the Indian Institute of Technology in Delhi.

A Logic Coverage Analyzer of Source Programs

What is Logic Coverage

- A measure of how well the internal control flow logic of a program has been exercised
- The goal is to find logical errors
- Statement Coverage - A Form of Logic Coverage

```
***Example***
X=Y;
if X > Y then S1;
else  S2;
```

- S1 is never executed

-contd-

- Is statement coverage enough?
  No. Why not?
  Example
  X=Y;
  if X=Y then S1;
  S2;
- Statement coverage is complete but there is still a logical error
- What is the solution? Cover both the branches of the decision. This is known as branch coverage.

-contd-

- Is branch coverage enough?
  No. Why not?

  Example

  A=B;

  if (A=B) and (B > 2) then S1;

  else S2;
- B=2 and B=3 will guarantee that both branches are covered. Yet we see a logic error in the program
- What is the solution?
- Cover all combinations of outcomes of each condition
- This is known as multi-condition coverage

What about multi way decisions?

- Excercise all possible branches of the decision
- This is called case coverage

Is it easy to automate multi-condition and case coverage?

- Yes
- For every two-way decision, we need to insert hooks for monitoring the boolean values of all its conditions
- For every multi-way decision, we need to insert hooks for monitoring the values of the case expression
- The number of decisions in a program is bounded by its size

Overview of the Logic Coverage Analyser - Features

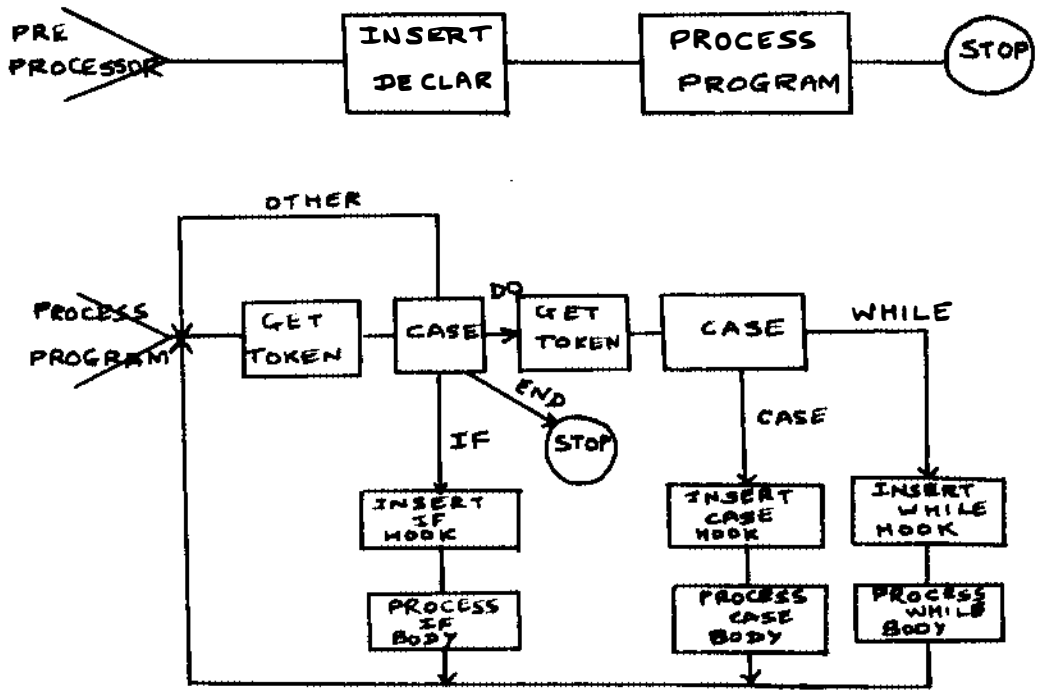
- It is targetted for PL/M source progams
- It performs multi-condition coverage on two-way decisions (IF and WHILE statements)
- A WHILE statement has two branches-execute the loop or skip it
- It performs branch coverage on CASE statements
- It can handle very complex nestings

Overview - Implementation

- It is subdivided into three parts
- Preprocessor, Monitor and Reporter
  - The preprocessor inserts hooks into the source program to monitor condition and case expression values
  - The modified program is linked and executed with the monitor which keeps track of the actual logic coverage
  - The monitor stores the results in a permanent file
  - The reporter interprets the data in the file to produce a coverage report

Implementation- Preprocessor

- It is coded in standard Pascal and runs on RMX 86
- Uses a recursive descent LL1 grammar
- Can process more than one PL/M module
- The source program should be error free. It cannot recover from syntax errors in the source
- The basic processing algorithm is shown below



-contd-

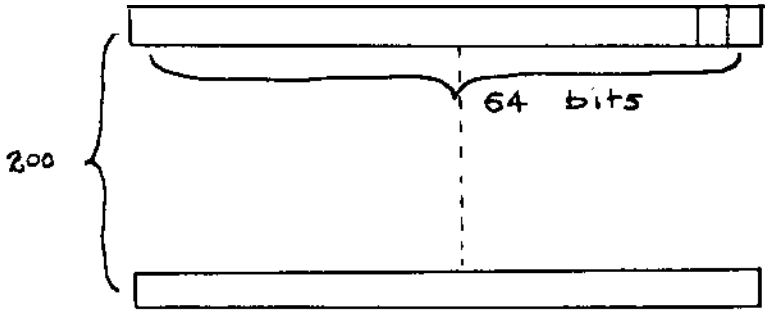
- The main module requires special treatment

```
Main: do;
  ..
  CALL INIT; --Initialises the monitor
  ..
  ..
  CALL SAVEINFO; --Saves the coverage into a file
end Main;
```

The Monitor

- It is coded in PL/M
- Maximum number of (IF + WHILE) statements that can be covered = 200
- Maximum number of allowable conditions in each statement = 6
- In the maximum case, we would need to record 12,800 condition values (200 * 2)
- A special scheme allows the monitor to use only 1600 bytes to represent all of them
- Maximum number of Case statements that can be covered = 200
- Maximum number of allowable cases in each case statement = 32

How it represents a Max of 76,800 condition values



- Each condition combination is represented by a particular bit
- A 0 for that bit indicates that the combination hasn't been covered
- A 1 indicates that it has been covered

The Reporter

- It is coded in PL/M 86
- The coverage data in the file is stored in the internal format
- The Reporter translates the data into a report showing logic coverage

MULTI CONDITION COVERAGE

STATEMENT TYPE	LINE NO	NO OF CONDITIONS	CONDITION COMBINATIONS NOT COVERED
IF	—	—	— — — — — — — — —
WHILE			

CASE COVERAGE

LINE NO	NO OF CASES	CASES NOT COVERED
—	—	—

Testing tool

A strategy for optimally using a logic coverage analyser

- The source program should be driven with test data derived solely through it's specifications (Black box)
- The analyser will detect which logic hasn't been excercised
- There are three possibilities here
    - Testing was incomplete. This helps in selecting more tests
    - There are logic errors in the program
    - The program logic was designed to be incomplete
    - An example of this is -- if (X=1) or (X=2) then S1;
- Logic coverage serves a dual purpose.
    - Find errors
    - Select additional test data

Example

The Original Program:

```
read(a,b);
IF (a>2) AND (b<1) then <S0>;
```

After preprocessing:

```
**DECLARE c array(10) BYTE EXTERNAL;**
--c is used to pass condition values to if probe.
  read(a,b);
* DO; *
    * c[1] = (a>2); *
    * c[2] = (b<1); *
    * CALL ifprobe(1,2); *
      IF c[1] and c[2] then <S0>;
* END; *
```

Example -contd-

Run the altered program with the following test data.

a = 3, b = 3

a = 3, b = 0

Call the reporter. It prints the following report.

Type	Line No	Num Conditions	Combinations Not Covered
IF	2	2	False False
			False True

Other uses - Measuring control flow complexity

- The following statistics are gathered
- Total Lines of code
- Number of if statements
- Number of while statements
- Number of case statements
- Average number of conditions in if statements
- Average number of conditions in while statements
- Average number of alternatives in case statements

TCAT/C: A Tool for Testing C Software

Edward Miller
Technical Director

July 1985

TN-1183/1

Software Research Associates
P. O. Box 2432
San Francisco, CA 94126 USA

Phone: (415) 957-1441 -- Telex: 340-235 (SRA SFO)

TCAT/C: A Tool for Testing C Software

Dr. Edward Miller
Technical Director
Software Research Associates
P. O. Box 2432
San Francisco, CA 94126

(415) 957-1441

ABSTRACT

SRA has developed a sophisticated test coverage analysis tool for software written in "C", TCAT/C. The TCAT/C system operates under VAX/Unix and supports automatic instrumentation, runtime support, and coverage analysis.

TCAT/C applies to unit-testing, sub-system testing and to system testing. In operation, TCAT/C introduces minimum system overhead and provides for a high level of convenience in use of the tool.

Reports produced by TCAT/C show the impact of testing on a system that has been processed by the TCAT/C instrumenter in two ways: (1) by identifying the complete extent of exercise of the program, and (2) by identifying the set of logical elements in the code that are NOT yet exercised by the current set of tests.

In practice, the TCAT/C system lends itself very easily to systematic testing. In several SRA projects TCAT/C has been used as the basis for completeness testing, with very good effect. SRA estimates that, with TCAT/C in use and with appropriate levels of test coverage obtained, the error rates in treated software drop by a factor of at least 10:1. Such improvement values easily justify TCAT's moderate cost and use overheads.

The TCAT/C product has been developed as part of SRA's long term strategy for developing an integrated collection of test support tools. TCAT's are already implemented for PASCAL, BASIC, COBOL and several types of assembly language.

BIOGRAPHY

Edward Miller

Dr. Edward Miller is technical director of Software Research Associates of San Francisco. He specializes in advanced technology for software engineering management, software testing, software maintenance, and automated tool design. Previously Dr. Miller was Director of the Software Technology Centre, Science Applications, Inc., and Director of the Program Validation Project at General Research Corporation. He has lectured at the University of California at Santa Barbara and at the University of Maryland, where he received his PhD.

AVAILABLE SOFTWARE TESTING TOOLS AND TECHNIQUES

Requirements based testing

- Black-box test planning
- Requirements linking
- Abstraction approaches

Inspection and review methods

- Design reviews
- Code reviews
- Test plan reviews

Static analysis

- Control flow analysis
- Data flow analysis
- Interface analysis

Unit (development, module) testing

- White-box (structural) testing
- Interactive test bed systems

Subsystem testing

- Automated test scenarios
- Automated test data generation schemes
- Interface testing

Interface & integration testing

- Interface checking
- Compiler-assisted testing

System (function) testing

- Black-box functional testing
- Coverage analysis
- Gray box testing
- FSM-based testing

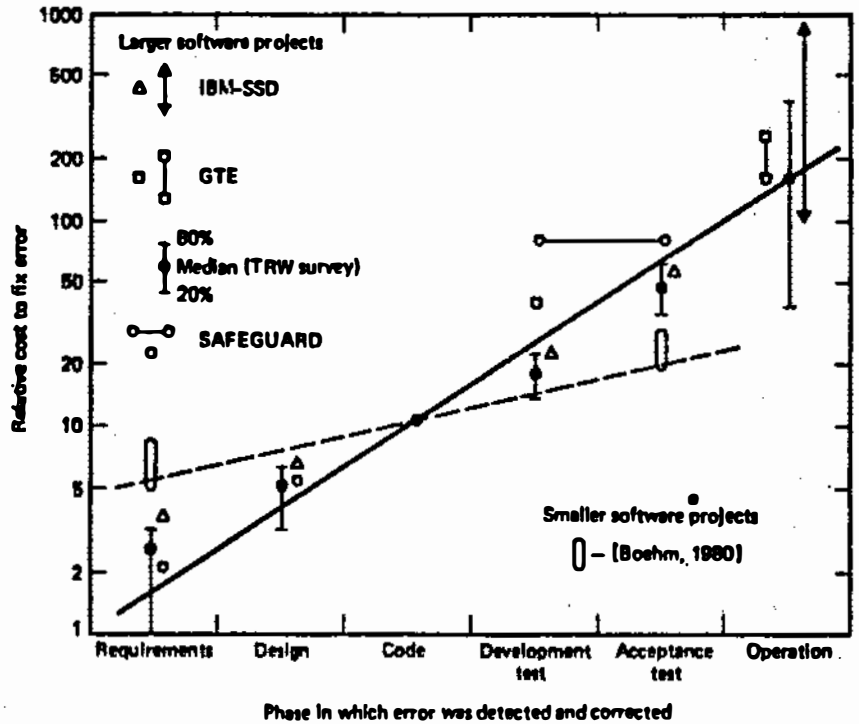
Regression testing

- Change control
- Coverage analysis
- Modification analysis

QA-SRA-0.1

SOFTWARE RESEARCH

COCOMO DATABASE REPRESENTATION OF COST-TO-FIX OR CHANGE SOFTWARE THROUGHOUT LIFE CYCLE



SOURCE: Boehm, Software Engineering Economics, Prentice-Hall, 1981.

QA-2-18.1-a

SOFTWARE RESEARCH

RANGE OF SOFTWARE QUALITY LEVELS

METRICS USED

1000's OF LINES OF CODE
DEFECTS PER 1000 LINES OF CODE (KLOC)

NORMAL QUALITY

DEFECTS LESS THAN 60 (+30 -20) PER KLOC

NORMAL PROGRAMMING PROCESS, NO SPECIAL QUALITY MANAGEMENT METHODS

GOOD QUALITY

DEFECTS LESS THAN 10 PER KLOC

BASIC QUALITY MANAGEMENT ACTIVITY:
INSPECTION/REVIEWS
DEFECT TRACKING
SIMPLE COVERAGE ANALYSIS

HIGH QUALITY

DEFECTS LESS THAN 1 PER KLOC

INTERMEDIATE QUALITY MANAGEMENT ACTIVITY:

FORMAL TES TPLANNING
INSPECTION/REVIEWS
C1 COVERAGE ANALYSIS

HIGHEST QUALITY

DEFECTS LESS THAN 0.1 PER KLOC

ADVANCED QUALITY MANAGEMENT ACTIVITY:

FORMAL TEST PLANNING
INSPECTION/REVIEWS
CT COVERAGE ANALYSIS
SYMBOLIC EVALUATION

QA-SRA-0.2

SOFTWARE RESEARCH

```
/*** Reference listing for SRA C instrumentor
     instr. version 1.9 -e; 1.10 statistics ***/
/* Copyright (c) 1984 by Software Research Associates.
All Rights Reserved.  */

          int c;        /* c is column count to skip empty columns */
          GetName(line, name)
          char line[], name[];
          {
                char token[20], buf[80];
                static char affixm[MAX] =  "a" ;
/** Begin module GetName: segment 1 **/


1               GetToken(line, token);  /* Returns token from line */

                if (strcmp(token, "SUBROUTINE") == 0 ||
2                   strcmp(token, "FUNCTION")  ==  0  ) /**2 if**/  {
                        GetToken(line, name);
                        strcpy(name, buf);
                }
3    4          else /**3 else**/  if (strcmp(token, "BLOCK") == 0)
                        /**4 if**/  {
                        catstr("blkdat.", affixb, buf);
                        affixb[0] = affixb[0]+1;
                }
5    6          else /**5 else**/  if (strcmp((GetToken(line, token)),
                                "FUNCTION") == 0) /**6 if**/  {
                        GetToken(line, name);
                        strcpy(name, buf);
                }
7               else /**7 else**/  {
                        strcpy(name, "main.f");
                        printf(": %s : main program0, name);
                }

 ...

/*      Total of 18 statements and 12 segments  */
/*      Total of 263 tokens in 39 lines.        */
```

Coverage Analyzer, Version 1.8 (80 Column)

	Module Name	Number Of Segments:	(Archived) Past Tests: Number Of Invocations	Number Of Segments Hit	Percent Coverage
1:	SCN_BUFI	1	4	1	100.00
2:	get_cell_data	11	19	7	63.64
3:	do_parm_type_chk	15	2	7	46.67
4:	set_source_ptrs	7	2	6	85.71
5:	TEST_BREAK	9	5	2	22.22
6:	POINTER_ON	23	22	12	52.17
7:	look_up	7	2	6	85.71
8:	UPDATE_ROWS	7	3	4	57.14
9:	SET_RULER	3	1	2	66.67
10:	NEXT_ROW	13	8	3	23.08
11:	Get_mem_blk	1	4	1	100.00
12:	Dl_CTL_PAGE_DOWN	7	1	3	42.86
13:	DET_FORMAT	15	5	12	80.00
14:	eval	17	5	12	70.59
15:	decide_exe_mode_for_E	4	5	3	75.00
16:	CHANGE_KBD	4	1	3	75.00
17:	RULER	11	1	6	54.55
18:	do_asg	30	5	17	56.67
19:	SET_STATUS_LINE	31	22	19	61.29
20:	RESET_GLOB_VARS	7	1	5	71.43
21:	RESET_DATA_WDS	1	1	1	100.00
22:	ROW_STATUS	5	17	4	80.00
23:	analyze_source	29	2	9	31.03
24:	DET_DIRECTION	107	11	16	14.95
25:	perform	11	9	7	63.64
26:	RESET_PDATA_AREA	1	1	1	100.00
27:	find_element	7	8	2	28.57
28:	POINTER_OFF	5	33	3	60.00
29:	Initialize_mx	9	1	7	77.78
30:	NEXT_RIGHT	29	1	2	6.90
31:	get_nxt_row_	5	7	3	60.00
32:	parse	30	5	15	50.00
33:	Dl_HOME	7	1	3	42.86
34:	do_eval	27	5	10	37.04
	Totals	1192	397	454	38.09

Coverage Analyzer, Version 1.8 (80 Column)

Module Name:	Number Of Segments:	This Test: No. Of Invokes	This Test: No. Of Segments Hit	This Test: Cl% Cover	Cumulative Summary: No. Of Invokes	Cumulative Summary: No. Of Segments Hit	Cumulative Summary: Cl% Cover
SCN_BUFI	1	0	0	0.00	4	1	100.00
get_cell_data	11	0	0	0.00	19	7	63.64
do_parm_type_chk	15	0	0	0.00	2	7	46.67
set_source_ptrs	7	0	0	0.00	2	6	85.71
TEST_BREAK	9	0	0	0.00	5	2	22.22
POINTER_ON	23	0	0	0.00	22	12	52.17
look_up	7	0	0	0.00	2	6	85.71
UPDATE_ROWS	7	0	0	0.00	3	4	57.14
SET_RULER	3	0	0	0.00	1	2	66.67
NEXT_ROW	13	0	0	0.00	8	3	23.08
Get_mem_blk	1	0	0	0.00	4	1	100.00
Dl_CTL_PAGE_DOWN	7	0	0	0.00	1	3	42.86
DET_FORMAT	15	0	0	0.00	5	12	80.00
eval	17	0	0	0.00	5	12	70.59
decide_exe_mode_for_E	4	0	0	0.00	5	3	75.00
CHANGE_KBD	4	0	0	0.00	1	3	75.00
RULER	11	0	0	0.00	1	6	54.55
do_asg	30	0	0	0.00	5	17	56.67
SET_STATUS_LINE	31	0	0	0.00	22	19	61.29
RESET_GLOB_VARS	7	0	0	0.00	1	5	71.43
RESET_DATA_WDS	1	0	0	0.00	1	1	100.00
ROW_STATUS	5	0	0	0.00	17	4	80.00
analyze_source	29	0	0	0.00	2	9	31.03
DET_DIRECTION	107	0	0	0.00	11	16	14.95
perform	11	0	0	0.00	9	7	63.64
RESET_PDATA_AREA	1	0	0	0.00	1	1	100.00
find_element	7	0	0	0.00	8	2	28.57
POINTER_OFF	5	0	0	0.00	33	3	60.00
Initialize_mx	9	0	0	0.00	1	7	77.78
NEXT_RIGHT	29	0	0	0.00	1	2	6.90
get_nxt_row_	5	0	0	0.00	7	3	60.00
parse	30	0	0	0.00	5	15	50.00
Dl_HOME	7	0	0	0.00	1	3	42.86
do_eval	27	0	0	0.00	5	10	37.04
Totals	1192	0	0	0.00	397	454	38.09

```
Coverage Analyzer,  Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

C1 Not Hit Report.

Module: SCN_BUFI                -- All Segments Hit.  C1 = 100%

Module: get_cell_data           -- Segments Not Hit:

          2    4    6    9

Module: do_parm_type_chk        -- Segments Not Hit:

          4    5    6    8    9   10   11   12

Module: set_source_ptrs         -- Segments Not Hit:

          4

Module: TEST_BREAK              -- Segments Not Hit:

          2    3    4    5    6    7    8

Module: POINTER_ON              -- Segments Not Hit:

          2    5   10   12   14   15   16   17   18   22
         23

Module: look_up                 -- Segments Not Hit:

          7

Module: UPDATE_ROWS             -- Segments Not Hit:

          2    3    4

Module: SET_RULER               -- Segments Not Hit:

          2

Module: NEXT_ROW                -- Segments Not Hit:

          4    5    6    7    8    9   10   11   12   13

Module: Get_mem_blk             -- All Segments Hit.  C1 = 100%

Module: D1_CTL_PAGE_DOWN        -- Segments Not Hit:

          3    4    5    7
```

```
Coverage Analyzer,  Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

Segment Level Histogram for Module:  animal


                      I Logarithm of Executions,  Normalized to Maximum
                      I (Maximum =    296 Hits)
Segment  Number Of    I
Number   Executions   I----------1----------10-----20----30---40--80-100
--------------------+-------------------------------------------------
                      I
    1              2 I XXXXXXXXXXXXXXXXX
    2              2 I XXXXXXXXXXXXXXXXX
    3   *            I
    4              2 I XXXXXXXXXXXXXXXXX
    5   *            I
    6             14 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    7              2 I XXXXXXXXXXXXXXXXX
    8              2 I XXXXXXXXXXXXXXXXX
    9   *            I
   10   *            I
   11              2 I XXXXXXXXXXXXXXXXX
   12   *            I
   13              2 I XXXXXXXXXXXXXXXXX
   14             20 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   15   *            I
   16             44 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   17   *            I
   18             20 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   19   *            I
   20             44 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   21   *            I
   22   *            I
   23              8 I XXXXXXXXXXXXXXXXXXXXXXXX
   24              4 I XXXXXXXXXXXXXXXXXXXXXX
   25              4 I XXXXXXXXXXXXXXXXXXXXXX

   ...

   38              8 I XXXXXXXXXXXXXXXXXXXXXXXX
   39              8 I XXXXXXXXXXXXXXXXXXXXXXXX
   40             16 I XXXXXXXXXXXXXXXXXXXXXXXXXXXX
                      I
--------------------+-------------------------------------------------
       (* = Zero Hits)

Average Hits Per Executed Segment:       18.3860
        C1 Value for This Module:        65.7895 %
```

Coverage Analyzer, Version 1.8 (80 Column)

Segment Level Histogram for Module: animal

```
                        I Number of Executions, Normalized to Maximum
                        I (Maximum =     296 Hits)
                        I (Scale:     .338 Xs = One Hit;  Each X =    5.920 Hits)
Segment  Number Of      I
Number    Executions    I 1-------20--------40--------60--------80-------100
--------------------+-------------------------------------------------------
                        I
    1                 2 I X
    2                 2 I X
    3    *              I
    4                 2 I X
    5    *              I
    6                14 I XX
    7                 2 I X
    8                 2 I X
    9    *              I
   10    *              I
   11                 2 I X
   12    *              I
   13                 2 I X
   14                20 I XXX
   15    *              I
   16                44 I XXXXXXX
   17    *              I
   18                20 I XXX
   19    *              I
   20                44 I XXXXXXX
   21    *              I
   22    *              I
   23                 8 I X
   24                 4 I X
   25                 4 I X

   ...

   38                 8 I X
   39                 8 I X
   40                16 I XX
                        I
--------------------+-------------------------------------------------------
       (* = Zero Hits)
```

Average Hits Per Executed Segment: 18.3860
Cl Value for This Module: 65.7895 %

S-TCAT/C FEATURES

SYSTEM TEST VERSION OF TCAT/C -- S-TCAT/C

- SIMILAR TO TCAT/C FOR SINGLE/MULTIPLE MODULE
- SLANTED TO NEEDS OF INTEGRATION/SYSTEM TESTING

S1 METRIC

- ALL CALLER-CALLEE PAIRS EXERCISED
- STRONGER THAN "EVERY MODULE CALLED"

MEASUREMENT TECHNIQUE

- SEMI-INVASIVE INSTRUMENTATION
- RUN-TIME PACKAGE
- TRACEFILES
- "STANDARD" COVER ANALYZER
- SPECIAL INTERACTIVE UTILITIES

ADDITIONAL REPORTS

- FULL CALL-PAIR ANALYSIS
- COMPLETE CALLING TREE
- SYSTEM STRUCTURE STATISTICS

IMPLEMENTATION BASE

- UNIX ENVIRONMENTS
  - Berkeley Unix
  - AT&T Unix
  - Xenix
- PC-DOS

STRUCTURAL TEST PLANNING -- ADVANCED TECHNIQUES

- BASIC APPROACH
  - PROGRAM FLOWCHART -- DIGRAPH
  - PROGRAM SECTOR GRAPH
  - HIERARCHICAL DECOMPOSITION
    - SUCCESSION
    - IF ELSE END
    - WHILE END WHILE
  - LOOP ENUMERATION
  - PATH REPRESENTATION
- TECHNICAL LIMITS
  - NON-PURE-SP PROGRAMS
  - COMBINATORICS
  - PATH COMPLEXITY
- OPPORTUNITIES FOR AUTOMATION
  - FINDING NEXT USEFUL TEST
  - SELECTING "RELIABLE" TEST VALUES
  - INTERACTION WITH DEBUGGER
    - CONVENTIONAL LEVEL
    - ADVANCED LEVEL

SRA APPROACH TO SOFTWARE QUALITY MANAGEMENT

- Recognition of realities of problem
  - Knowledge of technology
  - Availability of information
  - Research and development capacity
- Cognizance of client needs and wants
  - Budget constraints
  - Personnel issues
  - Trust
  - Confidentiality
- Productivity gain through automation
  - Recordkeeping
  - Test planning
  - Coverage analysis
  - Re-testing (regression)
  - Electronic communication
- Quality gain through automation
  - Interactive inspections
  - Automated static analysis
  - Automated unit testing
  - Coverage analysis

QA-SRA-0.6

SOFTWARE RESEARCH

```
/*** Reference listing for SRA C instrumentor
     instr. version 1.9 -e; 1.10 statistics ***/
/* Copyright (c) 1984 by Software Research Associates.
All Rights Reserved.  */

          int c;        /* c is column count to skip empty columns */
          GetName(line, name)
          char line[], name[];
          {
                char token[20], buf[80];
                static char affixm[MAX] =  "a" ;
/** Begin module GetName: segment 1 **/


1               GetToken(line, token);  /* Returns token from line */

                if (strcmp(token, "SUBROUTINE") == 0 ||
2                   strcmp(token, "FUNCTION")  ==  0  ) /**2 if**/  {
                        GetToken(line, name);
                        strcpy(name, buf);
                }
3    4          else /**3 else**/  if (strcmp(token, "BLOCK") == 0)
                        /**4 if**/  {
                        catstr("blkdat.", affixb, buf);
                        affixb[0] = affixb[0]+1;
                }
5    6          else /**5 else**/  if (strcmp((GetToken(line, token)),
                                "FUNCTION") == 0) /**6 if**/  {
                        GetToken(line, name);
                        strcpy(name, buf);
                }
7               else /**7 else**/  {
                        strcpy(name, "main.f");
                        printf(": %s : main program0, name);
                }

 ...

/*      Total of 18 statements and 12 segments  */
/*      Total of 263 tokens in 39 lines.        */
```

Coverage Analyzer, Version 1.8 (80 Column)

			(Archived) Past Tests		
No.	Module Name	Number Of Segments:	Number Of Invocations	Number Of Segments Hit	Percent Coverage
1:	SCN_BUFI	1	4	1	100.00
2:	get_cell_data	11	19	7	63.64
3:	do_parm_type_chk	15	2	7	46.67
4:	set_source_ptrs	7	2	6	85.71
5:	TEST_BREAK	9	5	2	22.22
6:	POINTER_ON	23	22	12	52.17
7:	look_up	7	2	6	85.71
8:	UPDATE_ROWS	7	3	4	57.14
9:	SET_RULER	3	1	2	66.67
10:	NEXT_ROW	13	8	3	23.08
11:	Get_mem_blk	1	4	1	100.00
12:	Dl_CTL_PAGE_DOWN	7	1	3	42.86
13:	DET_FORMAT	15	5	12	80.00
14:	eval	17	5	12	70.59
15:	decide_exe_mode_for_E	4	5	3	75.00
16:	CHANGE_KBD	4	1	3	75.00
17:	RULER	11	1	6	54.55
18:	do_asg	30	5	17	56.67
19:	SET_STATUS_LINE	31	22	19	61.29
20:	RESET_GLOB_VARS	7	1	5	71.43
21:	RESET_DATA_WDS	1	1	1	100.00
22:	ROW_STATUS	5	17	4	80.00
23:	analyze_source	29	2	9	31.03
24:	DET_DIRECTION	107	11	16	14.95
25:	perform	11	9	7	63.64
26:	RESET_PDATA_AREA	1	1	1	100.00
27:	find_element	7	8	2	28.57
28:	POINTER_OFF	5	33	3	60.00
29:	Initialize_mx	9	1	7	77.78
30:	NEXT_RIGHT	29	1	2	6.90
31:	get_nxt_row_	5	7	3	60.00
32:	parse	30	5	15	50.00
33:	Dl_HOME	7	1	3	42.86
34:	do_eval	27	5	10	37.04
	Totals	1192	397	454	38.09

Coverage Analyzer, Version 1.8 (80 Column)

Module Name:	Number Of Segments:	This Test: No. Of Invokes	This Test: No. Of Segments Hit	This Test: C1% Cover	Cumulative Summary: No. Of Invokes	Cumulative Summary: No. Of Segments Hit	Cumulative Summary: C1% Cover
SCN_BUFI	1	0	0	0.00	4	1	100.00
get_cell_data	11	0	0	0.00	19	7	63.64
do_parm_type_chk	15	0	0	0.00	2	7	46.67
set_source_ptrs	7	0	0	0.00	2	6	85.71
TEST_BREAK	9	0	0	0.00	5	2	22.22
POINTER_ON	23	0	0	0.00	22	12	52.17
look_up	7	0	0	0.00	2	6	85.71
UPDATE_ROWS	7	0	0	0.00	3	4	57.14
SET_RULER	3	0	0	0.00	1	2	66.67
NEXT_ROW	13	0	0	0.00	8	3	23.08
Get_mem_blk	1	0	0	0.00	4	1	100.00
Dl_CTL_PAGE_DOWN	7	0	0	0.00	1	3	42.86
DET_FORMAT	15	0	0	0.00	5	12	80.00
eval	17	0	0	0.00	5	12	70.59
decide_exe_mode_for_E	4	0	0	0.00	5	3	75.00
CHANGE_KBD	4	0	0	0.00	1	3	75.00
RULER	11	0	0	0.00	1	6	54.55
do_asg	30	0	0	0.00	5	17	56.67
SET_STATUS_LINE	31	0	0	0.00	22	19	61.29
RESET_GLOB_VARS	7	0	0	0.00	1	5	71.43
RESET_DATA_WDS	1	0	0	0.00	1	1	100.00
ROW_STATUS	5	0	0	0.00	17	4	80.00
analyze_source	29	0	0	0.00	2	9	31.03
DET_DIRECTION	107	0	0	0.00	11	16	14.95
perform	11	0	0	0.00	9	7	63.64
RESET_PDATA_AREA	1	0	0	0.00	1	1	100.00
find_element	7	0	0	0.00	8	2	28.57
POINTER_OFF	5	0	0	0.00	33	3	60.00
Initialize_mx	9	0	0	0.00	1	7	77.78
NEXT_RIGHT	29	0	0	0.00	1	2	6.90
get_nxt_row_	5	0	0	0.00	7	3	60.00
parse	30	0	0	0.00	5	15	50.00
Dl_HOME	7	0	0	0.00	1	3	42.86
do_eval	27	0	0	0.00	5	10	37.04
Totals	1192	0	0	0.00	397	454	38.09

```
Coverage Analyzer,  Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

C1 Not Hit Report.

Module: SCN_BUFI              -- All Segments Hit.  C1 = 100%

Module: get_cell_data         -- Segments Not Hit:

          2    4    6    9

Module: do_parm_type_chk      -- Segments Not Hit:

          4    5    6    8    9   10   11   12

Module: set_source_ptrs       -- Segments Not Hit:

          4

Module: TEST_BREAK            -- Segments Not Hit:

          2    3    4    5    6    7    8

Module: POINTER_ON            -- Segments Not Hit:

          2    5   10   12   14   15   16   17   18   22
         23

Module: look_up               -- Segments Not Hit:

          7

Module: UPDATE_ROWS           -- Segments Not Hit:

          2    3    4

Module: SET_RULER             -- Segments Not Hit:

          2

Module: NEXT_ROW              -- Segments Not Hit:

          4    5    6    7    8    9   10   11   12   13

Module: Get_mem_blk           -- All Segments Hit.  C1 = 100%

Module: D1_CTL_PAGE_DOWN      -- Segments Not Hit:

          3    4    5    7
```

```
Coverage Analyzer,  Version 1.8 (80 Column)
(c) Copyright 1984 by Software Research Associates

Segment Level Histogram for Module:  animal


                      I Logarithm of Executions,  Normalized to Maximum
                      I (Maximum =    296 Hits)
Segment  Number Of    I
Number   Executions   I----------1----------10-----20----30---40--80-100
---------------------+-----------------------------------------------------
                      I
    1                2 I XXXXXXXXXXXXXXXXX
    2                2 I XXXXXXXXXXXXXXXXX
    3   *              I
    4                2 I XXXXXXXXXXXXXXXXX
    5   *              I
    6               14 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    7                2 I XXXXXXXXXXXXXXXXX
    8                2 I XXXXXXXXXXXXXXXXX
    9   *              I
   10   *              I
   11                2 I XXXXXXXXXXXXXXXXX
   12   *              I
   13                2 I XXXXXXXXXXXXXXXXX
   14               20 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   15   *              I
   16               44 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   17   *              I
   18               20 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   19   *              I
   20               44 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   21   *              I
   22   *              I
   23                8 I XXXXXXXXXXXXXXXXXXXXXXXXX
   24                4 I XXXXXXXXXXXXXXXXXXXXXX
   25                4 I XXXXXXXXXXXXXXXXXXXXXX

   ...

   38                8 I XXXXXXXXXXXXXXXXXXXXXXXXX
   39                8 I XXXXXXXXXXXXXXXXXXXXXXXXX
   40               16 I XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
                      I
---------------------+-----------------------------------------------------
       (* = Zero Hits)

Average Hits Per Executed Segment:       18.3860
        C1 Value for This Module:        65.7895 %
```

Coverage Analyzer, Version 1.8 (80 Column)

Segment Level Histogram for Module: animal

```
                      I Number of Executions, Normalized to Maximum
                      I (Maximum =    296 Hits)
                      I (Scale:    .338 Xs = One Hit;  Each X =   5.920 Hits)
Segment  Number Of    I
Number   Executions   I 1-------20--------40--------60--------80-------100
---------------------+-------------------------------------------------
                      I
    1              2 I X
    2              2 I X
    3   *            I
    4              2 I X
    5   *            I
    6             14 I XX
    7              2 I X
    8              2 I X
    9   *            I
   10   *            I
   11              2 I X
   12   *            I
   13              2 I X
   14             20 I XXX
   15   *            I
   16             44 I XXXXXXX
   17   *            I
   18             20 I XXX
   19   *            I
   20             44 I XXXXXXX
   21   *            I
   22   *            I
   23              8 I X
   24              4 I X
   25              4 I X

   ...

   38              8 I X
   39              8 I X
   40             16 I XX
                      I
---------------------+-------------------------------------------------
       (* = Zero Hits)
```

Average Hits Per Executed Segment: 18.3860
C1 Value for This Module: 65.7895 %

S-TCAT/C FEATURES

SYSTEM TEST VERSION OF TCAT/C -- S-TCAT/C

- SIMILAR TO TCAT/C FOR SINGLE/MULTIPLE MODULE
- SLANTED TO NEEDS OF INTEGRATION/SYSTEM TESTING

S1 METRIC

- ALL CALLER-CALLEE PAIRS EXERCISED
- STRONGER THAN "EVERY MODULE CALLED"

MEASUREMENT TECHNIQUE

- SEMI-INVASIVE INSTRUMENTATION
- RUN-TIME PACKAGE
- TRACEFILES
- "STANDARD" COVER ANALYZER
- SPECIAL INTERACTIVE UTILITIES

ADDITIONAL REPORTS

- FULL CALL-PAIR ANALYSIS
- COMPLETE CALLING TREE
- SYSTEM STRUCTURE STATISTICS

IMPLEMENTATION BASE

- UNIX ENVIRONMENTS
  - Berkeley Unix
  - AT&T Unix
  - Xenix
- PC-DOS

STRUCTURAL TEST PLANNING -- ADVANCED TECHNIQUES

- BASIC APPROACH
  - PROGRAM FLOWCHART -- DIGRAPH
  - PROGRAM SECTOR GRAPH
  - HIERARCHICAL DECOMPOSITION
    - SUCCESSION
    - IF ELSE END
    - WHILE END WHILE
  - LOOP ENUMERATION
  - PATH REPRESENTATION
- TECHNICAL LIMITS
  - NON-PURE-SP PROGRAMS
  - COMBINATORICS
  - PATH COMPLEXITY
- OPPORTUNITIES FOR AUTOMATION
  - FINDING NEXT USEFUL TEST
  - SELECTING "RELIABLE" TEST VALUES
  - INTERACTION WITH DEBUGGER
    - CONVENTIONAL LEVEL
    - ADVANCED LEVEL

SRA APPROACH TO SOFTWARE QUALITY MANAGEMENT

- Recognition of realities of problem
  - Knowledge of technology
  - Availability of information
  - Research and development capacity
- Cognizance of client needs and wants
  - Budget constraints
  - Personnel issues
  - Trust
  - Confidentiality
- Productivity gain through automation
  - Recordkeeping
  - Test planning
  - Coverage analysis
  - Re-testing (regression)
  - Electronic communication
- Quality gain through automation
  - Interactive inspections
  - Automated static analysis
  - Automated unit testing
  - Coverage analysis

QA-SRA-0.6

A UNIX BASED SOFTWARE DEVELOPMENT PROBLEM TRACKING SYSTEM

Gordon Staley

Software Quality Engineer

Portable Computer Division

Hewlett Packard

ABSTRACT

As more software development is being done in Unix environments, the need for a Unix based problem tracking system has come about.

This paper addresses the approach taken to set up an on-line software tracking and reporting system for use during the development of products; and discusses the potential areas for improvement and expansion in the future. The goal in developing this system was to improve productivity of the development and test engineers by providing an on-line problem tracking system that gave them easy access to problem information and status. PCD software engineers are currently using the first implementation on a widespread basis.

1. INTRODUCTION

During the testing phase of software development there is a need to have a reporting and tracking system for software problems encountered in the product. Often these inputs come from individuals in organizations other than the development organization, and in some cases from individuals at remote locations with respect to the development organization. Additionally, the development may be supported by individuals in different locations and in different organizations.

As the testing phase continues there is a need to track the progress of the action taken on reported problems. This status information is often needed on an on-demand basis (e.g. has problem X been fixed in the new release?).

As the project nears completion there is a need to evaluate the current risk of the code. This requires the extraction of information from data collected to date concerning problems.

Prior to the implementation of the Unix problem tracking system, most problem tracking was done manually. This resulted in some problems not getting formally reported due to extensive administrative overhead. This in turn caused a possible understatement of the true level of problem detection occurring in a project. Additionally, any status of the current reliability of the product had to be manually generated. These factors reduced the productivity of the testers, developers and software quality engineers, as well as making the assessment of the reliability of the code for the product difficult at best.

2. GOALS

The need for a system that would automate the software development reporting and tracking of problems was identified. The system needed to meet the following goals:

- provide easy access for testers and developers
- allow for easy problem information entry and retrieval
- allow for better risk assessment data
- able to be run on any multi-user system supporting Unix
- less than four weeks total development effort

3. SOLUTION

After an investigation, the solution that appeared to meet all of the goals was to highly leverage the existing software available on the Unix systems. By so doing, the development effort could be minimized and the result would be as supportable as possible given the tight schedule constraints.

The solution implemented made use of three different features of Unix, the notes system, shell scripts and the lexical analysis preprocessor (lex).

3.1 Unix Notes

Part of the solution was to use the Unix "notes" system to collect the raw data from the tester and developer. The notes system allows a user to place information on an electronic bulletin board. This electronic bulletin board is sub-divided into separate topic areas commonly referred to as notes groups. In our solution separate notes groups were used for different parts of different projects (e.g. O/S, BASIC). This made it easier for the different developers to keep track of the problems that applied to them and not have to wade through reports of problems that had no bearing on what they were doing.

The Unix notes system provides for an administrator (director) that can edit and delete notes and change the access controls for the particular notes group. This feature allows the director to limit the access of a particular notes group to only the developers or any group of users that might be desired. This can ensure that only the developers are entering solution data into the system.

3.2 Shell Script

In order to provide some control on the data being entered, a "script" (system program) was written to provide a consistent structure to the input. This script automatically enters the current date and time and prompts the user for the information needed to reproduce the problem. This script also placed consistent flags in the data to allow for programmatic extraction of key information by a lexical analysis preprocessor.

3.3 Lex

Programs were then written in a lexical analysis preprocessor language (lex) to extract pertinent summary and management data from the submitted notes. The output of the lex preprocessor is a compilable C program. Lex allows for embedded C commands in the lex source, making it easier to customize the extraction of the data, and reduces the effort to generate a one-time report. This extracted data allows anyone involved with the project to have current information on the status of any particular release.

3.4 Basic Use of System

When a user detects and wishes to post a problem into the system, they run the script called bugs. This script prompts them for the information needed to reproduce and troubleshoot a particular problem. The system is flexible enough to allow the different projects being tracked to ask for different information or to put various input restrictions on the information. The Unix system provides some of the desired information (current date, unique tracking number), with the balance being supplied by the user (see exhibit 1 at the end of this paper for a sample problem input).

A public account was provided for those potential users of the system that did not have a regular account on a computer that supported the problem tracking system. This proved to be particularly useful for the testers that were in other organizations or different geographical locations.

The notes system allows for notes files to be "networked". This way users on multiple Unix systems can track the status of software under development.

The developer responsible for a reported problem submits a response to the problem report (responses are built-in feature of the notes system) outlining the corrective action taken, if any (see exhibit 2 at the end of this paper for a sample response input). The absence of a response to problem report would flag the project manager that a solution has not yet been found to a particular problem.

This information can then be viewed by developers and testers to keep up on problems discovered and their current status.

The software quality engineer uses this system to generate summary reports for the project leader (see exhibits 3 & 4

at the end of this paper for sample reports). These reports detail the problems into four areas:

1. problems that do not have a response posted yet (exhibit 3)

2. problems that are under investigation

3. problems where no change action was taken (duplicate, user misunderstanding)

4. problems where corrective action was taken (exhibit 4)

Exhibit 4 is based on actual data with the names changed and descriptions deleted. The bug number is a unique number assigned to a problem report. This is done to allow for tracking of problems across notes systems, as notes usually do not have the same note number on all systems. The SEV column represents the severity of the problem discovered (one # being trivial, four # being critical). CLS refers to classification (totals shown at the bottom of the page). The DATE is date the problem was reported. The version of the software that the problem was found in is under VER. The last two columns indicate the level of difficulty of the repair, and the lines of code required to make the change.

4. RESULTS

The tracking system has been in place for 18 months. In that time it has been used to track five different projects at Hewlett Packard's Portable Computer Division. There has been widespread acceptance of the system in the division. We have seen interest in the system from other HP divisions and other companies in the Northwest.

The system had enough acceptance that Manufacturing set up the same system that contains the information on problems discovered. In addition they chose to add problems reported from the field to yield a current problem database for the released product.

There have been a number of benefits identified since the first implementation. The rapid implementation was very beneficial in that we were working on a fast-track project and did not have a lot of time to spend developing an automated tracking system.

The system really encouraged good communications between the developers, testers and Quality Department. The current

status of a particular problem was more visible than it had ever been before. Anyone with access to the machine with the tracking system on it could read about the known problems and the current status of the fix. This led to less confusion about the current status of the code.

By allowing for multiple responses to a particular report the testers were more likely to add amplifying information to the original reports. This feature was also used by the developers to acknowledge that a particular problem was being addressed, although not yet fixed. An initial "under investigation" response would be filed, and later a final response would be filed.

With all of the data being in a machine readable form, we were able to generate data for risk analysis much more quickly than we had been able to in the past. Initially there were many challenges here as a lot of the data was entered in free form. As we needed more statistical data from the system it became necessary to restrict some of the input to allow for programmatic extraction of this data. In later versions we would ask for input from a menu of selections. One selection "other" would allow for free form input in a case where none of the menu items were appropriate.

The only major drawback to the system is that of maintenance. The source has been modified by 6 different authors. Any time a new version of one of the projects needs to be tracked the source must change. To add a new project the source must change. With support for so many projects and having so many authors the modifications are likely to cause a problem in the system. There have been some occasions where the user is mysteriously dropped out of the system. A solution to this drawback is discussed in the next section.

In spite of the problem listed above the system has been a success. This early version has been a good prototype to get usage information from, but now needs to be upgraded to a fully documented and supported system.

5. IMPROVEMENT/EXPANSION

As with any system developed in a short period of time there is much room for improvement. Many details have been resolved by use of early versions of the tracking system. The following is a list of the known areas that could use improvement in this system:

- Convert the shell scripts to programs written in a more supportable and faster executing high level language.
- Allow for easier update to the list of products (and their modules) that are being tracked at any point in time.
- Preprocess input data to prevent > 80 character lines.
- Provide a facility that directly posts a response to a problem report (now manually done by the developer).
- Provide editing capabilities for public access users that may not be familiar with a particular editor.
- Allow for on-line access to summary status information.
- Have as many inputs as possible converted to menu prompting for inputs rather than requiring free form input.
- Add a usage tracking system, something that allow testers to easily enter their testing hours/module/configuration.

BIOGRAPHY

Gordon Staley

Gordon Staley is a software quality engineer with Hewlett Packard in Corvallis, Oregon. He has been with HP for six years, the last two in the Software Quality department. His educational background includes a BS in computer science from the University of Utah and an MBA from the University of Oregon.

Exhibit 1. Sample Completed Problem Report

<< PROBLEM ID >>

5068

<< REPORT DATE >>

Fri Jun 7 08:32:16 PST 1985

<< PROBLEM DESCRIPTION >>

The software test (for compatibility) includes a test for the Non-maskable interrupt by doing the instruction: INT 2. On the target machine, this function does an iret. On XXXXXXXX, since there can never be a hardware Non-maskable interrupt, this instruction sends the unit into never-never-land.

<< REPORT SUBMITTER >>

Annie

<< SYSTEM IDENTIFICATION >>

PP

<< SYSTEM CONFIGURATION >>

any

<< SOFTWARE VERSION >>

QA2

Exhibit 2. Sample Completed Problem Response

```
<< LAB ENGINEER ASSIGNED >>

 Annie

<< PROBLEM CLASSIFICATION >>

 1 - New problem (design)

<< PROBLEM SEVERITY >>

 4 - Trivial

<< AFFECTED MODULE(S) >>

interr.asm
rombios.asm

<< PROBLEM WORKAROUND >>

do not enable those interrupts without taking over their interrupt
vectors.

<< SOFTWARE VERSION FIXED >>

QA3

<< FIX DESCRIPTION >>

At boot, these  interrupts will point to a routine that will clear
any interrupts that come in on these lines.

<< REPAIR DIFFICULTY >>

 1 - Simple

<< TYPE OF ERROR >>

 6 - Error condition not trapped

<< NUMBER OF LINES OF CODE CHANGED >>

10

<< FIX VERIFICATION >>

tested on the 64000 & traced on the interrupts being cleared.
```

Exhibit 3. Sample No Response Problem Summary

Fri Jun 7 05:07:39 PST 1985

BUGS WITHOUT RESPONSES

BUG#	ENGINEER	SEV	CLS	DATE	SUBMITTER	VER	NOTE#
13				850603	Howard	LP3	15
25				850607	Howard	LP3	32
5028				850623	Eric	LP4	95
42				850625	Joni	LP4	66
62				850706	Joni	LP3	86
94				850720	Ken	QA1	122
109				850722	John	QA1	138
5043				850725	Phil	QA1	144
124				850727	Joni	QA1	161
125				850728	Joni	QA1	166
128				850801	Joni	QA1	170
127				850801	John	QA1	169
141				850805	Jim	QA2	185
140				850805	Howard	QA2	184
5057				850806	Jim	QA2	186
5062				850806	Bill	QA1	191
5063				850806	Bill	QA1	192
5061				850806	Craig	QA2	190
5058				850806	Annie	QA2	187
5059				850806	Annie	QA2	188
5060				850806	Annie	QA2	189
142				850807	Jim	QA2	194
143				850807	Joni	QA2	197
144				850807	Joni	QA2	198
5065				850807	Annie	QA2	195
5066				850807	Annie	QA2	196

Total Number of unanswered bugs = 26

Exhibit 4. Sample Fixed Problem Summary

Fri Jun 7 05:07:39 PST 1985

BUGS WITH ACTION TAKEN

BUG#	ENGINEER	SEV	CLS	DATE	SUBMITTER	VER	NOTE#	DIF	LINES
5032	Phil	#	0	850625	Phil	LP4	65	1	5
101	Allyn	##	0	850721	Ken	QA1	130	1	5
105	Jim	###	0	850721	Jim	QA1	134	1	10
5051	Annie	#	0	850728	Annie	QA1	163	1	5
134	Allyn	##	0	850804	Ken	QA2	177	1	3
5002	Leon	##	1	850602	Leon	LP3	3	2	15
5019	Leon	#	1	850610	Leon	LP3	36	1	1
61	Phil	##	1	850706	Joni	LP3	85	1	4
5038	Allyn	###	1	850713	Annie	QA1	100	2	40
81	Phil	####	1	850717	John	QA1	110	2	10
115	Phil	###	1	850726	Ken	QA1	152	3	20
5048	Annie	#	1	850726	Annie	QA1	150	1	2
131	Leon	##	1	850804	John	QA2	174	2	16
3	Allyn	##	2	841113	Ken	LP3	1	1	1
5003	Phil	##	2	850602	Leon	LP3	4	1	2
17	Allyn	#	2	850604	Jim	LP3	18	1	5
18	Allyn	#	2	850604	Gwen	LP3	41	1	1
21	Allyn	##	2	850607	Dan	LP3	42	1	5
36	Annie	####	2	850618	Eric	LP4	53	1	6
5029	Leon	###	2	850623	Annie	LP4	59	1	1
5031	Phil	##	2	850625	Phil	LP4	64	1	1
48	Allyn	##	2	850630	Mike	LP4	73	1	1
50	Allyn	###	2	850631	Mike	LP4	76	1	10
54	Allyn	###	2	850704	Karen	LP4	79	1	10
69	Allyn	##	2	850707	Eric	LP4	93	1	1
5035	Allyn	#	2	850707	Leon	LP4	89	1	1
72	Allyn	###	2	850712	John	LP4	98	1	6
73	Leon	###	2	850713	Andy	LP4	103	2	10
76	Allyn	##	2	850713	Ken	LP4	106	1	1
5040	Annie	###	2	850713	Annie	QA1	102	1	2
83	Annie	##	2	850719	Jim	QA1	112	2	1
107	Allyn	#	2	850722	Jim	QA1	136	1	1
111	Allyn	#	2	850724	John	QA1	140	1	1
117	Allyn	####	2	850726	John	QA1	154	1	1
5045	Annie	###	2	850726	Annie	QA1	147	1	2
119	Jim	###	2	850727	Jim	QA1	156	1	3
138	Allyn	##	2	850804	Ken	QA2	181	1	1
92	Phil	####	3	850720	Sherry	QA1	120	1	5
133	Leon	###	3	850804	Jennefer	QA2	176	3	20

Total number of new requirements discovered in the design = 5
Total number of new problems discovered in the design = 8
Total number of new problems found in the code = 24
Total number of side effects of previous changes = 2

A UNIX BASED SOFTWARE DEVELOPMENT PROBLEM TRACKING SYSTEM

PORTABLE COMPUTERS

THE NEED:

- SYSTEM FOR REPORTING PROBLEMS
- SYSTEM TO TRACK STATUS OF PROBLEMS
- ACCESS SUMMARY DATA

THE GOALS:

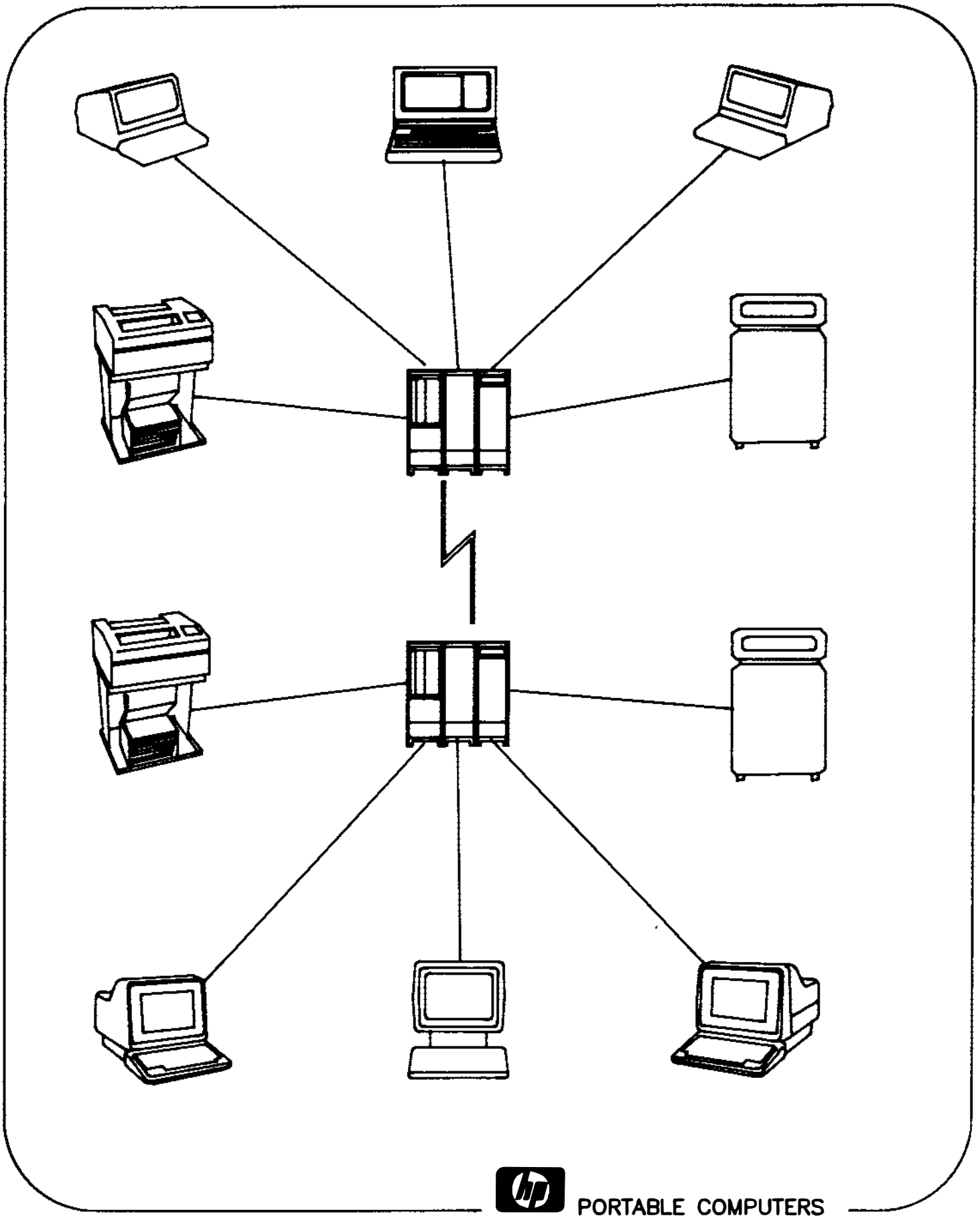
- **EASY ACCESS FOR TESTERS AND DEVELOPERS**
- **EASY PROBLEM INFORMATION ENTRY AND RETRIEVAL**
- **BETTER RISK ASSESSMENT DATA**
- **ABLE TO RUN ON LAB DEVELOPMENT/SUPPORT SYSTEM**
- **LESS THAN FOUR WEEKS TOTAL DEVELOPMENT EFFORT**

THE SOLUTION:

- **UNIX NOTESFILES (DATABASE)**

- **SHELL SCRIPT (USER INTERFACE/DATA STRUCTURE)**

- **LEX (DATA EXTRACTION)**



THE RESULTS:

- ACCEPTED BY LAB
- IMPROVED COMMUNICATIONS
  - DEVELOPERS
  - TESTERS
  - QUALITY DEPARTMENT
- IMPROVED PROBLEM STATUS ACCESS
- IMPROVED DATA FOR RISK ANALYSIS
- A GOOD ONE TO THROW AWAY

ROOM FOR IMPROVEMENT:

- CONVERT TO HIGH LEVEL LANGUAGE

- ALLOW FOR EASIER UPDATE OF PRODUCTS BEING TRACKED

- PROVIDE EDITING CAPABILITIES FOR PUBLIC USERS

- ADD MORE STRUCTURE TO RESPONSES TO ALLOW FOR EXTENDED DATA ANALYSIS

- ON-LINE ACCESS TO PROJECT SUMMARY INFORMATION

Session 5

DEVELOPMENT TOOLS

Titles and Speakers:

"Software Design Using BCS Argus"
Bill Hodges, Boeing Computer Services

"The System Engineering Environment PROMOD"
Peter Hruschka, Promod, Inc.

"Locating Suspect Software and Documentation by Monitoring Basic Information About Changes to the Source Files"
David Vomocil, Hewlett Packard Co.

SOFTWARE DESIGN USING BCS-ARGUS

William M. Hodges

Boeing Computer Services Company
P. O. Box 24346
Seattle, Washington 98124

The paper, "Software Design Using BCS-ARGUS", describes the use of a Boeing developed set of tools that aid in the specification and design stages of software projects. The process of using an architected set of mechanized tools is described with a number of major advantages highlighted. A brief discussion of the underlying architecture reveals the ease with which new tools can be added to mechanize more of the software development process.

Background

BCS-ARGUS is the code name for a tool under development at Boeing Computer Services. It is implemented as a desktop environment intended to mechanize and integrate many of the activities performed during the software life cycle by analysts, designers, programmers, and managers. This activity is considered proprietary at this time. However, it is expected that it will be packaged as a commercial offering in the future.

The heritage of BCS-ARGUS lies in Boeing research into software methodologies that started as early as 1974. These research activities produced two distinctly different prototypes intended to support the specification phases of the software life cycle.

One prototype is called SWIFT. SWIFT is functionally similar to PSL/PSA in that it uses specification language to capture system specifications. Like PSL/PSA, SWIFT can be regarded as a data dictionary system with added features. Key elements of its architecture include the following:

- A universal SWIFT language whose sublanguages include syntactic meta-language, a specification language, a PDL or pseudocode, and a relational query language.
- A user interface from which software tools can be invoked via commands or menus.
- A set of modular software tools accessing a central data store through a retained relational query language called SQL.
- A carefully structured relational data base supported by a full-function DBMS.

- A report capability for tabular and indented tree reports.

The second prototype is called ARGUS. It partially automates software development activities spanning the entire application life cycle. Through a sophisticated set of menus, users can:

- Enter software specifications in graphical (data flow diagram) or textual form from an on line terminal into a set of UNIX files. Some of these files comprise a relational data base.
- Create and update specially formatted UNIX files, including phone lists and schedules.
- Invoke basic UNIX functions for file control and document preparation.
- Connect to various host computers as an intelligent terminal.
- Produce high-quality plotter output of data flow diagrams.
- Perform analyses specifications captured in the data base.

ARGUS has been ported from the ONYX-based UNIX System 3 to DEC VAX systems running Berkeley 4.2 and System 5 UNIX systems. It is currently used by several Boeing projects.

BCS-ARGUS Overview

The experience gained since 1974 has allowed Boeing to develop BCS-ARGUS which exploits current hardware and software capabilities to produce a software development capability on the IBM PC/XT and PC/AT. BCS-ARGUS provides consistency of environment, user interface, and methodology regardless of the type of host system, implementation language or type of application, (i.e., real time or IMS COBOL). (The real time constructs for the methodology and the VAX VMS host capability is planned to be available in 1986).

The theory underlying BCS-ARGUS is that software is developed in stages, with each stage describing the software in more detail than the previous stage. Each stage culminates in a document expressing the design. The document is then reviewed to evaluate the completeness and adequacy of the design. The theory further assumes that the design documents will be updated to reflect the current software implementation throughout the life of the software.

The architecture of BCS-ARGUS allows systems to be specified, designed, and created on it and be implemented on some other host. The design of BCS-ARGUS allows a large project to centralize its specifications and integrated data dictionary on a large-scale host. Specifications may be checked out to the PC or checked in to the host as required. The host may also provide language processors and a test environment for the developed systems. Functions supported on the host include:

- Work package management
- Relational data base management
- Report generation
- Document generation

BCS-ARGUS hardware features a three-button mouse, an IRMA board, and a multi-function card incorporated on an IBM PC/XT or PC/AT. Color or monochrome monitors are supported. BCS-ARGUS tools include a data flow diagram editor, a commercial word processor to produce documentation, an SPF look alike editor to capture code, and a relational data base to tie everything together. An SNA communication capability provides the tools necessary to receive work packages from the host and return completed ones.

This unique architecture allows projects that are too large to run on the PC to be completed in parts and assembled on the host. Traceability and auditability of requirements are supported throughout the two-level data base implementation, thus allowing the completeness of an allocated baseline to be verified.

The current version of BCS-ARGUS is intended to be used either in a standalone configuration or connected to an IBM mainframe via an IBM 3274 telecommunications controller.

Work Package Initiation

The remaining discussion will assume that a systems analyst who currently uses an IBM 3278/9 terminal, supplemented with data flow diagrams for software design, decides to change to a PC/XT equipped with BCS-ARGUS. He would plug his existing coax cable into his PC/XT, then log in to a password-protected ARGUS account when the UNIX log-in prompt appears.

After logging in, our hypothetical systems analyst finds himself in the BCS-ARGUS top-level menu. At this level he can do a number of things, in any sequence, capture a specification or design for a work package:

- He may connect to the host to download data dictionary entries or requirements paragraphs from a higher level document.
- He may enter the documentation system to prepare portions of his document.
- He may enter the specification system via data flow diagrams or mini-specification data.

Communications

If the systems analyst chooses to communicate with the host, he can either request a complete package to be transferred from the host or log on to TSO

on the host and query the data base, depending upon the data that he wants. He can use these capabilities in whatever sequence he chooses.

Documentation

If the analyst chooses to enter the documentation system, he can use the WYSIWYG (what you see is what you get) word processor to prepare a section of his document. He can specify that pages containing data flow diagrams be included at certain points in the text. Likewise, if he were documenting after the construction of the data flow diagrams and mini-specifications, he could specify that analysis reports be included at certain positions.

Specification Entry

When the analyst constructs a data flow diagram, he can use the mouse to interactively position symbols depicting processes, interfaces, and data stores on the 132-by 60-character virtual screen. Once the entries are positioned and annotated with their descriptions, he can add the data flows and their labels.

At any time during this process, the analyst can selectively enter a mini-specification for any one of the entities that have been placed on the screen. Mini-specifications specify attributes, as follows:

- For a process
    - Siblings
    - Long description
    - Procedural descriptions of algorithms and/or transition logic
    - Etc.
- For an interface
    - Description
    - Data items
    - Edit requirements
    - Etc.
- For a data store
    - Component of
    - Contains
    - Description
- For a data flow
    - Component of
    - Contains
    - Description
    - Etc.

Analysis

If mini-specification data are entered at each level, the balancing activity will be accomplished as each successive level is defined during the decomposition process.

Once the specification is complete, the following reports can be produced for the design using the local data base:

- List
- Structure
- Analysis
- Dictionary
- Attributes
- Data Flow Diagram
- Requirements traceability

These reports will help assess the completeness and consistency of the current design.

Work Package Completion

If a particular software development task is a one-person task, the facilities on the IBM PC/XT workstation are adequate to do the complete job. If the task is a part of a larger task, it may be necessary to check in the portion of that design from the PC/XT to the BCS-ARGUS system on the host computer.

On check in, the host software will load the relational data into the relational data base on the host and prepare them for subsequent processes. The same set of analysis reports available on the IBM PC/XT workstation can be accomplished across the integrated database.

Benefits

In addition to intangible benefits resulting from the availability of data on the desktop and the responsiveness of the local capability on the PC/XT, BCS-ARGUS increases productivity throughout the software life cycle. It improves the response time of the system, captures data the first time they are keyed, provides aids for completeness and consistency checking, and allows errors to be detected early in the life cycle. The impact of these benefits is determined by the number of people on a project, the life of the system, and the way an organization did the job before BCS-ARGUS

During the early stages of the life cycle, BCS-ARGUS will dramatically reduce the time required to capture a design and refine it. (Studies have shown that

each data flow diagram in a system specification undergoes a range of 9 to 22 modifications). In addition, the system will ease the creation and production of documentation. Last but certainly not least, it will provide analysis techniques to ensure the accuracy of a design.

BCS-ARGUS editors will aid in the production of code in the latter portions of the software development life cycle. An on-line data dictionary will also aid in code production. Furthermore, the documentation and analysis tools described earlier will be available to support the later stages.

The maintenance stage of the life cycle will be aided by the existence of consistent, error-free documentation. Documentation and code are tied together via the relational data base, which will make it easier to identify areas related to each other by function or data. With this identification, it will be a straightforward task to isolate bugs, provide a complete and consistent fix, and identify the impact of proposed changes.

Later versions of BCS-ARGUS will provide an integrated project management system as well as a comprehensive configuration management systems. These systems will span all stages of the project life cycle.

Shown below is a typical breakdown for the life cycle cost of a large formal project that has a long life. This table also shows the estimated improvements that BCS-ARGUS is expected to provide.

	% Life Cycle	Improvement %
Requirements	11	50
Design	11	50
Construction	10	20
Testing	18	20
Maintenance	50	40

These estimates are probably conservative, since there are currently no data available to determine the synergism that will result from having a single set of integrated tools operating on a common data base throughout a system's life. Furthermore, they do not include the reduced training costs resulting from the use of a consistent workstation on all projects.

Clearly, BCS-ARGUS would provide significant productivity gains and a shortened development cycle to its users.

BIOGRAPHY

William H. Hodges

William H. Hodges holds a bachelor's degree in mechanical engineering from Oklahoma State University and his MS in administration from Wichita State University. He has been employed by the Boeing Computer Services Company since 1965, serving in various capacities of mechanical engineering, operations research, and software engineering roles. In the last two years that he has been in the Software Engineering Support Center, he has directed the ARGUS II Product Development activity. In addition, he has established directions for the Company relative to UNIX and IBM PCs.

BOEING COMPUTER SERVICES ARGUS

FROM A

SOFTWARE

QUALITY

PERSPECTIVE

THIRD ANNUAL PACIFIC NORTHWEST
SOFTWARE QUALITY CONFERENCE
27 SEPTEMBER 1985

W. HODGES
BOEING

BCS ARGUS

- OVERVIEW OF BOEING COMPUTER SERVICES SOFTWARE QUALITY PROGRAM
- OVERVIEW OF BOEING COMPUTER SERVICES ARGUS
- IMPACT OF BOEING COMPUTER SERVICES ARGUS SOFTWARE QUALITY
- SUMMARY

BCS ARGUS

OVERVIEW OF BCS SOFTWARE QUALITY OBJECTIVES

- OBTAIN QUALITY OBJECTIVES FROM USER REQUIREMENTS

- CONTINUAL ASSESSMENT OF PROGRESS

BCS ARGUS

OVERVIEW OF BCS SOFTWARE QUALITY OBJECTIVES

IMPLEMENTATION CONCEPTS

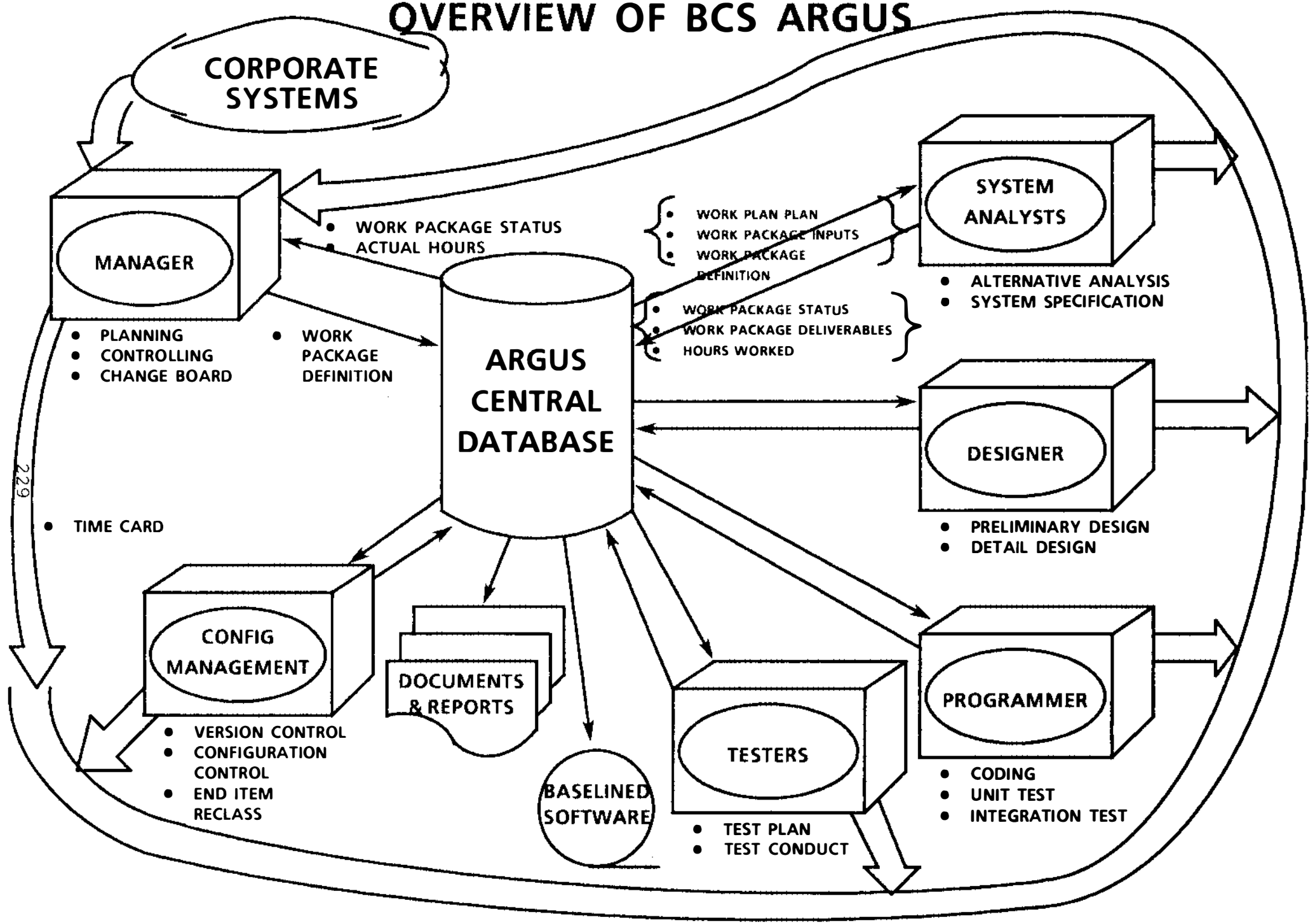
- SQA PLAN
- DISCIPLINE EMBEDDED IN PROCESS
- IDENTIFY AND MONITOR SQA ACTIVITIES
- DELINEATE TASKS IN COST-EFFECTIVE MANNER
- INDEPENDENT EVALUATIONS

BCS ARGUS

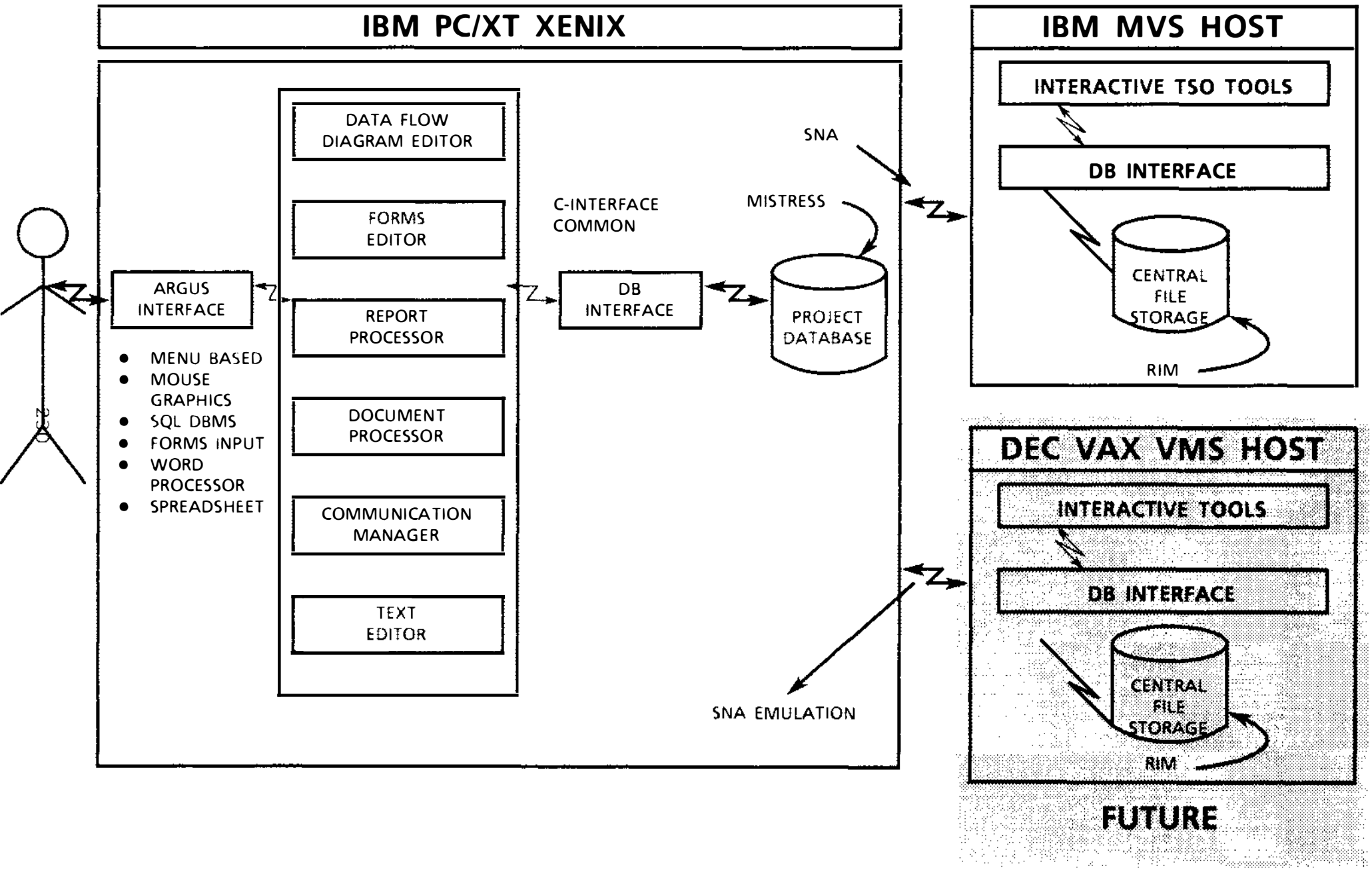
OVERVIEW OF BCS SOFTWARE QUALITY OBJECTIVES

ACQUISITION CONCERN	USER ISSUE	APPLICABLE QUALITY FACTOR
PERFORMANCE -- HOW WELL DOES IT FUNCTION?	HOW WELL DOES IT UTILIZE A RESOURCE?	EFFICIENCY
	HOW SECURE IS IT?	INTEGRITY
	HOW WELL WILL IT PERFORM UNDER ADVERSE CONDITIONS?	SURVIVABILITY
	HOW EASY IS IT TO USE?	USABILITY
	WHAT CONFIDENCE CAN BE PLACED IN WHAT IT DOES?	RELIABILITY
DESIGN -- HOW VALID IS THE DESIGN?	HOW WELL DOES IT CONFORM TO THE REQUIREMENTS?	CORRECTNESS
	HOW EASY IS IT TO REPAIR?	MAINTAINABILITY
	HOW EASY IS IT TO VERIFY ITS PERFORMANCE?	VERIFIABILITY
	HOW EASY IS IT TO EXPAND OR UPGRADE ITS CAPABILITY OR PERFORMANCE?	EXPANDABILITY
	HOW EASY IS IT TO CHANGE?	FLEXIBILITY
ADAPTATION -- HOW ADAPTABLE IS IT?	HOW EASY IS IT TO INTERFACE WITH ANOTHER SYSTEM?	INTEROPERABILITY
	HOW EASY IS IT TO TRANSPORT?	PORTABILITY
	HOW EASY IS IT TO CONVERT FOR USE IN ANOTHER APPLICATION?	REUSABILITY

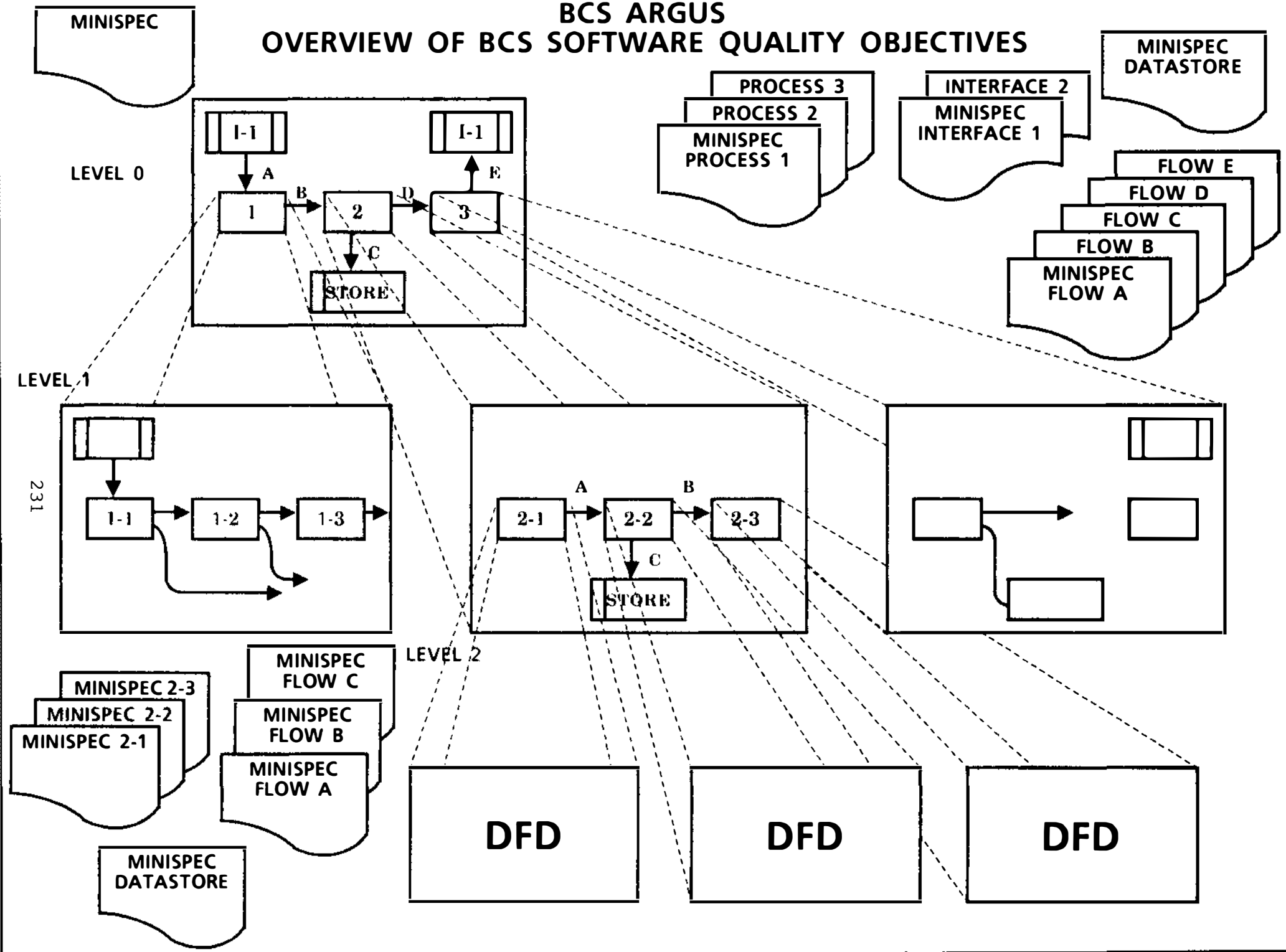
BCS ARGUS
OVERVIEW OF BCS ARGUS



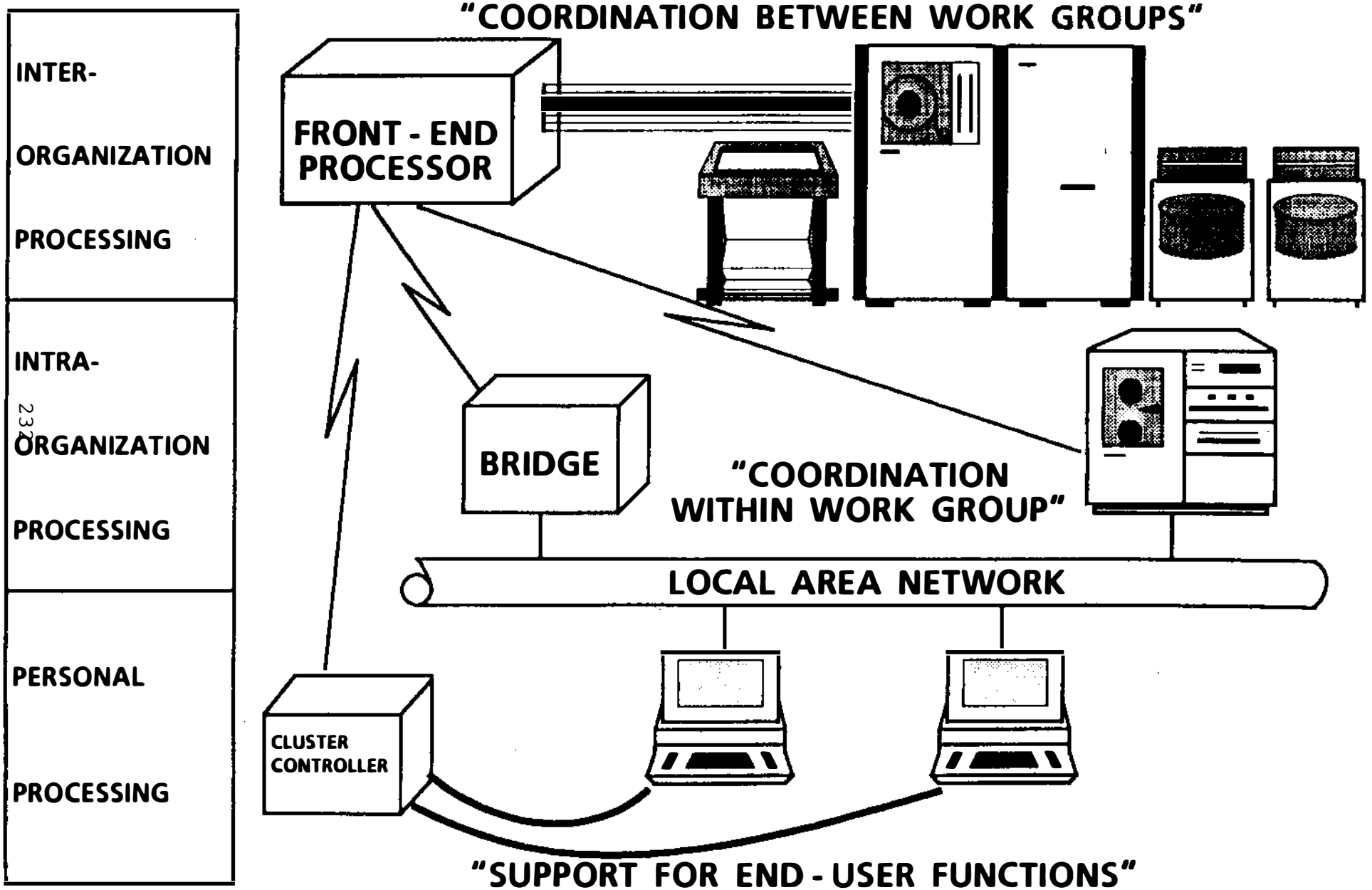
BCS ARGUS
OVERVIEW OF BCS ARGUS



BCS ARGUS
OVERVIEW OF BCS SOFTWARE QUALITY OBJECTIVES



OVERVIEW OF BCS ARGUS



BCS ARGUS
IMPACT OF BCS ARGUS ON SOFTWARE QUALITY

BCS ARGUS ATTRIBUTE	QUALITY FACTOR				
	RELIABILITY	CORRECTNESS	MAINTAINABILITY	VERIFIABILITY	EXPANDABILITY
• CONSISTENT METHODOLOGY	X	X			
• DATA DICTIONARY	X	X		X	X
• CONSISTENTLY/COMPLETENESS CHECKING	X	X	X		
• EMBEDDED DOCUMENTATION		X	X	X	
• REQUIREMENT TRACIBILITY		X		X	

BCS ARGUS
SUMMARY

- BOEING COMPUTER SERVICES ARGUS CURRENTLY SUPPORTS SOFTWARE QUALITY
- BOEING COMPUTER SERVICES ARGUS WILL ADDRESS MORE FACTORS
- BOEING COMPUTER SERVICES ARGUS WILL INCORPORATE SQA METRICS

The System Engineering Environment PROMOD

Peter Hruschka

Promod, Inc.

Abstract:

During the last years GEI developed a system engineering environment called PROMOD, (short for: project model). PROMOD comprises a set of wellknown techniques and methods (Structured Analysis, Modular Design, Pseudocode, ...) to guide developers from problem analysis to acceptance test, and it comprises effective interactive tools to give immediate feedback at every stage of the development of systems. Errors are detected and reported as early as possible; various reports can be generated according to the specific needs in every phase of the project. Although oriented towards modern programming languages PROMOD is language independent.

About 100 different licenses in the U.S., in Germany, the Netherlands and Great Britain are proving daily that PROMOD can improve the quality of a delivered system, can save delelopment time and money and increase productivity.

The environment PROMOD is constantly enhanced to cover more and more areas and thus successfully helping to improve the quality of system development.

1. Introduction

PROMOD is a system engineering environment developed to serve as the natural equipment of any system developer. It is applicable for any kind of system - hardware, software or organisational sytems, or any combination thereof.

PROMOD follows a life cycle oriented approach, offering tools for different phases during the creation of a new system or the improvement of an existing system. The major phases supported are:

- the analysis and definition of the requirements for a system (the analysis phase)
- the definition of the architecture of the system (the system design phase)
- the definiton of the detailed programs of a system (the program design phase)

For each phase PROMOD supports selected structured methodologies, all of which have been proven successfully in many large, industrial projects. DeMarco's STRUCTURED ANALYSIS /1/ is supported for the analysis phase, MODULAR DESIGN /2/,/3/ helps in finding the solution for a given problem and PSEUDOCODE /4/,/5/ is used for specifying the system in detail.

Studies have shown that using structured methodologies, i.e. obeying a set of rules during the analysis, design and implementation phase of a project, can double the productivity. The major reason is that the rigor of these methodologies allows errors to be found earlier and corrected immediately, before they become significant cost factors in the project.

The costs for finding and correcting an error that occured in the analysis phase might be one Dollar if it is corrected immediately, it might be $10, if you only find it in the architectural design phase; it is already $100 if you encounter it in detailed design, $1000 during coding and may even be $10000 if only recognized in the systems test and integration phase.

Other studies have verified that productivity can again be doubled by using tools supporting modern methodologies. One major drawback of methodologies usually is, that the analyst and the designer still manually create and update their documents, which is time comsuming and boring; another major drawback is that the principles of structured methodologies work fine as long as the sheer amount of information (in number of pages or number of documents) does not overwhelm the project team.

Both drawbacks can be overcome by effective tools, utilizing computer power where manpower is not as efficient. PROMOD takes over all the clerical work in a project associated with updating documentation and keeping track of changes. Management and technical staff may automatically produce a wide variety of up-to-date reports at any time and in any project phase they need them.

So, PROMOD helps in two ways to improve productivity in projects and ensure high quality of systems: it helps analysts and designers to "stick to the methodology", thus preventing errors to be propagated and it helps management cutting the costs for implementation, test, integration and maintenance by providing solid documentation, updated reports and cross references.

In the following the methodologies supported are sketched and the tools of PROMOD are discussed in detail.

2. Analysing Requirements

Many engineering disciplines emphasize the importance of establishing models before really developing products, e.g. models of new cars, new bridges, new towns, etc.. Structured Analysis applies this principle to the development of systems. These system models consist of 3 major components:

Data-flow diagrams give a graphical representation of the major functions of the system and their interconnections (i.e. the information or data flowing between the functions). The diagrams are organized in a hierarchy to display the system on different levels of abstraction and to limit the amount of information that has to be perceived at one time to a reasonable size for humans.

The data dictionary gives more detailed explanations about all the data used in the diagrams; about their structure, their components, their size, their usage and any other information that is relevant for the system.

Mini-specs give detailed descriptions of the functions drawn in the diagram. Since the diagrams decompose a large system into a set of well defined smaller systems, mini-specs (usually half a page to a page of text) can now be written instead of the traditional victorian novel style specifications for complete systems.

The following figure shows the components of a system model and the relation between them.

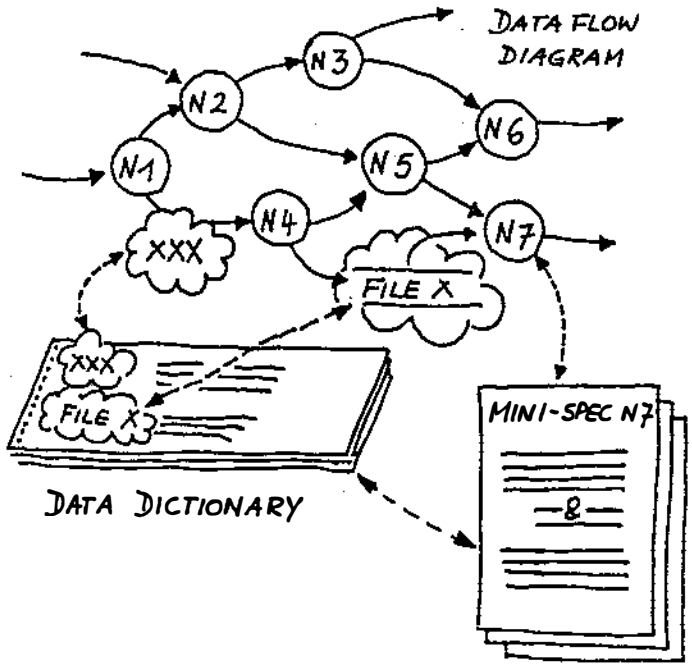


Fig. 1: A Structured Analysis System Model

PROMOD help in various ways preparing a system model according to Structured Analysis.
To draw the data-flow diagrams a graphic editor is at the users disposal. The analyst may flexibly draw nodes (functions), data-stores, terminators and data-flows, change such digrams by moving around components or changing their names, deleting objects, zooming into different levels (i.e in other diagrams) etc. During editing and before storing such diagrams in a central project library PROMOD checks the rules defined in Structured Analysis. E.g. it is verified that each node at least has one incoming and one outgoing data-flow, that no data-flow is drawn between two data-stores without going through a node, that every data flow has a proper name, etc. Thus PROMOD helps to follow the rules of the methodology.

For writing entries into the data dictionary and for writing mini-specs PROMOD offers syntax directed text editors. Also with these editors immediate local checks on the edited objects are performed.

The most powerful tool to assist in developing a system model is the SA-Analyzer. This analyzer checks for the global consistency and completeness of all analysis data collected in the project library (or of selected subparts of it). It makes sure that the hierarchy of diagrams is balanced, i.e. that inputs and outputs on one level are consistently refined at the next level and nothing has been added or missed. It also checks that all data used in diagrams or mini-specs are well defined in the data dictionary. And you receive warnings if your mini-spec descriptions do not conform to the information in the diagrams (e.g. you tried to access a data element, which is not input or output of the corresponding node in the diagram). Many more checks are performed to ensure that the model is consistent and complete. If the hierarchy of diagrams has only 3 to 4 levels this task would already be very time comsuming for humans and because of the constant changes occuring during the collection of requirements it is nearly impossible to do it without computer aid.

The analyzer can print a variety of documents, thereby generating automatically a table of contents, the system hierarchy including the numbering scheme, cross reference listings for all data, complete tree lists of all data refinement ever done in the analysis phase, error lists, etc.

The amount of documentation can be chosen, so that you can get more redundant documents for reviews (e.g. all the local data printed immediately after each diagram) or minimal documents for final versions (just one alphabetically ordered data dictionary at the end of the document).

3. Sketching the architecture of a system

After defining a given problem using the modeling techniques of Structured Analysis the solution under given constraints has to be found. Such constraints most often are resource limitations, e.g. predefined hardware, limited budgets, existing staff, ... PROMOD helps in suggesting the architecture for the solution by automatically transforming the systems model into a hierarchy of modules and functions. This hierarchy is built obeying criteria as suggested by Parnas and Liskov /2/,/6/. Usually - after a well done analysis phase this suggestion only has to be augmented by additional functions as needed during the design process.

The major building blocks a system designer is dealing with in this phase are modules, functions and subsystems.

Modules are program units responsible to solve a given part of the problem. They contain a collection of functions and local module data, which deliver specified results to their "boss" whenever they are asked to do it. Delivering results to a module further up in the hierarchy is called exporting functions of this module. Whenever necessary modules also factor out work to other (subordinate) modules; i.e. they ask a module further down in the hierarchy to do certain functions. This is called importing functions from other modules.

Export and import definitions make up the module's interface specification. These interface specifications are sufficiently detailed to allow cooperation of analysts, designers and users in this phase. The inside knowledge about modules is not necessary to make decisions about the structure of the system in the large. Many important questions about the suggested solution of a system can be discussed looking at the interface specifications only.

So, in other words, a module can be considered as a fence around a group of related functions and data. It is the systems designers job to describe that fence; the program designer later on will step into the modules and describe the functions in more detail.

Occasionally designs become so large that the module hierarchy can no longer be easily understood. PROMOD offers an additional structuring facility called subsystems to do this. You can combine any number of modules into a subsystem, thereby introducing higher levels of abstraction. Besides introducing higher level abstractions these subsystems can also be used for additional purposes, e.g. to combine the modules of a certain release, or to combine all module specifications written by one person, and many more.

The tools supporting this phase are similar to the tools for requirements engineering. Interactive, syntax directed editors are available to edit modules with their interface definitions

and subsystems. Again immediate checks are performed on the correctness of these descriptions. PROMOD warns you if you try to define functions in multiple ways, if your module data lists contain errors, etc.. The data dictionary of the requirements phase has been transformed into a data type dictionary for the design phase. These data types - simular to modern programming languages - allow for additional ways of expressing design constraints thus enabling PROMOD to perform additional checks on the validity of parameters of functions, the scope of data, the access to data elements and many more.

The more powerful checks are done by the Modular Design analyzer. It checks for interface consistency over larger parts of the system design or the overall system design. It finds contradictions between exports and imports, e.g. if somebody tries to call functions which are not defined and exported by another module; it also finds discrepancies in parameter lists of functions.

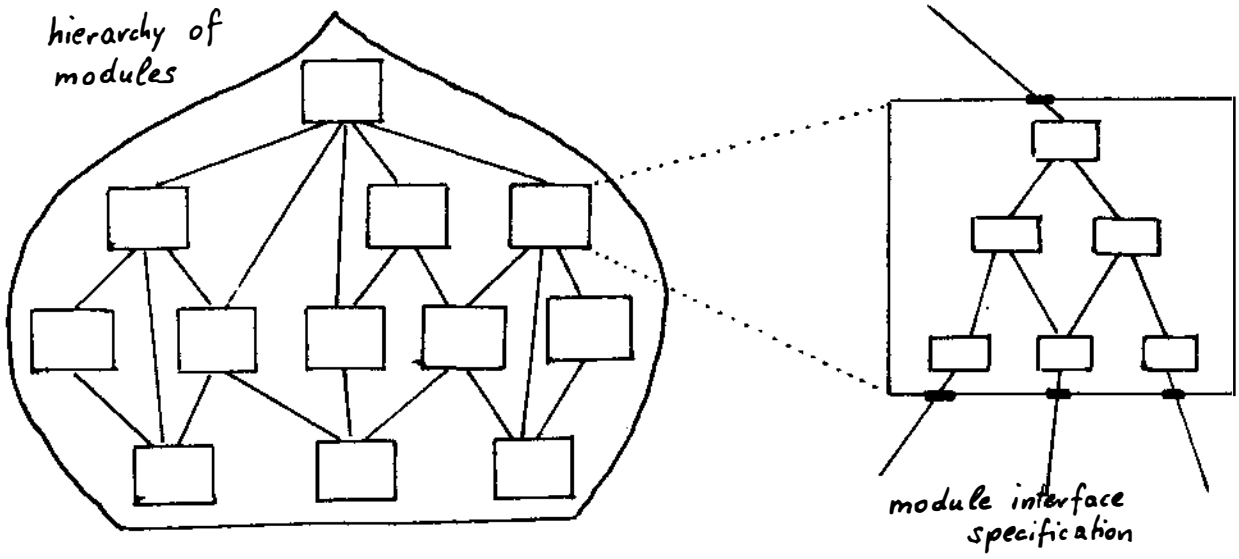


Fig. 2: The Architectural Design Model

These high level interface checks are very important, since systems design is usually done by different persons or different teams, all working at the same time on the same system. The system design analyzer is the tool to ensure, that everybody has the correct interface specifications, that others are informed whenever changes occur, and that the overall architecture is a clearly defined hierarchy. The analyzer offers a wide variety of reports at different levels of detail, generates a table of contents, the module's hierarchy, many different cross reference lists, tree lists (call hierarchies) for functions, refinement structures of data and data types, and others. Everywhere in the

documents you find generated hints on where to find more detailed information, where to read on, if you are interested in special parts. These generated references make it very easy to review and discuss the documents.

4. Designing the details

After defining the overall design structure of a system, the details have to be specified. PROMOD uses the well known principle of top down decomposition of function and data /4/, /5/ to describe algorithms and data structures. This is not only done for all the functions already specified in the module's interfaces but also for newly introduced internal functions of modules, which are factored out of export functions to keep the function body short and understandable.

Algorithms are described using a simple pseudocode notation. This pseudocode expresses sequences of statements, loops and decision statements, function calls and extensive natural language text.

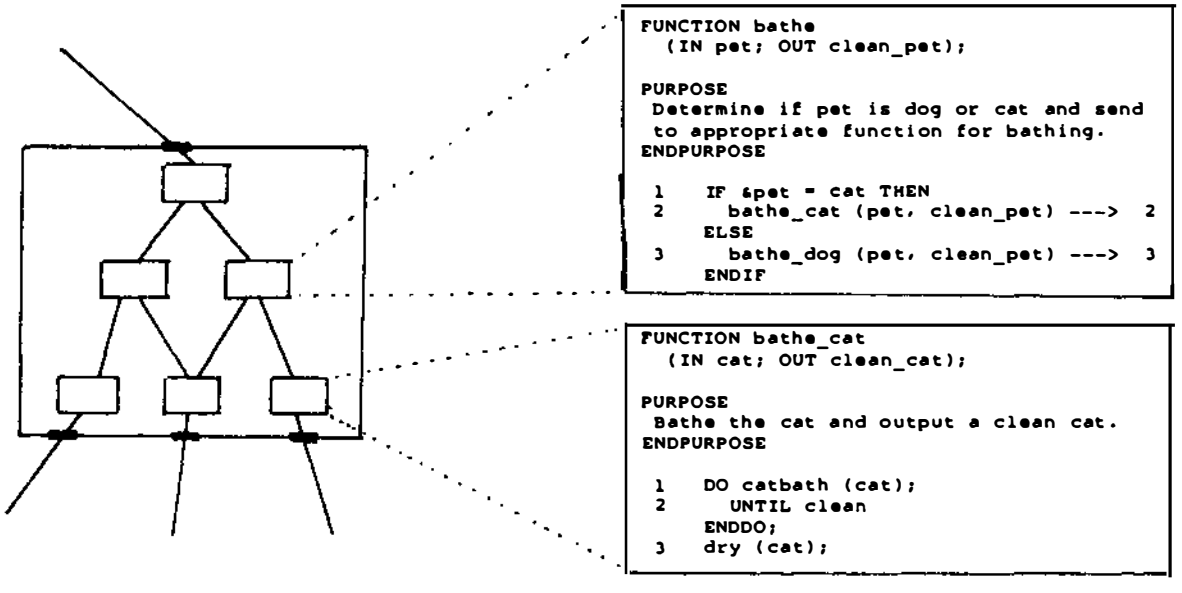


Fig. 3: The detailed design model

PROMOD again helps in this phase by offering an interactive pseudocode editor. While you describe a function the keywords of the pseudocode are recognized and the logical structure of the function is visualized by intending nested statements, aligning

keywords, and formatting the informal texts. Of course, if there are errors in your pseudocode structure PROMOD immediately reports these errors and allows you to correct the statements.

The analyzer for the program design phase - as its two predecessors - performs more global checks. It makes sure, that function calls within function bodies refer to defined functions, that the parameter lists conform in number and type, that all the data used in a function are accessible there and well defined, and many more. When printing documents of the detailed design, the pretty printed pseudocode is not only indented according to the logical structure; statement numbers are added, references to the pages of the document where you find the refinement of a function are generated and a variety of cross reference lists is available, e.g. for functions, data, data types, parameters. The hierarchical structure of the function calls can be generated, as well as hierarchies for the data and data types. Any deviation of the module hierarchy or any contradiction to the interface specification is reported to supply the designer with the information necessary for the next walkthrough or review meeting.

Over years pseudocode has proven to be a superb medium for naturally and easily expressing detailed designs for easy communication between designers and programmers. Pseudocode can help explain these details to non-EDP persons thus keeping the information exchange going that started in the analysis phase and insuring that the detailed program specifications still conform to the orignal requirements.

5. Implementing and testing a system

The detailed program specification in pseudocode is the basis for implementing a system in any chosen language. Since the pseudocode is a very detailed description on the one side it is very easy, nearly mechanical work to translate it into a programming language; on the other side the pseudocode is still general enough to allow translation in many different programming languages, from assembler to very modern languages like Ada. The checks that have already been performed in the detailed design phase nearly eliminate all errors in the coding phase with the exeption of trivial syntax errors that are easily caught by the compilers. In this phase the work previously done really pays off in time saved for testing.

Testing is additionally supported by the clearly defined functions of the system, small units allowing efficient separate tests to establish their correctness. The documentation supported for the test is mainly based on the user's input of the requirements, and on automatic transformations. Especially the data dictionary of the system model allows for a systematic generation of test data which are related to the original requirements from the user, and not to the defined solution of a programmer.

The systems integration phase can be a routine job, since PROMOD guarantees already since the architectural design phase, that the interface definitions between the larger units of the system (the modules) are correct and checked.

Especially for the maintenance of a system or the post-acceptance test evolution, which naturally comes about in every large system because of ever changing requirements, the documentation of PROMOD is very helpful. Since there are up-to-date models available of the requirements and the design it is easy for maintenance programmers to locate the parts of the system where changes or amendments are necessary. It is also easy to integrate these changes because of the information hiding principle used in the design phase. This principle allow to change single modules or functions in modules without effecting other parts of the system.

6. Integration - the key to success

The power of proMod lies in its integration. All the single tools like the DFD-editor or the pseudocode processor are powerful on its own. However, by utilizing information generated in one phase as input and basis for the next phase an optimum on integration is achieved. The project library - PROMOD's central database - is not only the repository of all information collected in a project, the individual objects in it also know about each other, are related in many different aspects, and therefore changes or amendments to one objects very often result in automatic changes and updates for other objects. Thus, a major portion of clerical work, which is usually loaded onto the developer, is easily done by PROMOD.

The requirement collected and modeled with Structured Analysis are used as basis for the suggested system architecture. Especially the hierarchy of diagrams is evaluated to suggest the hierarchy of modules, the files in the data dictionary are used to create abstract data types and the appropriate access functions a la Parnas /2/ or Liskov /6/; the nodes (or bubbles) of the diagrams are used to create functions of modules and the informal mini-spec texts are already in the right place as purpose descriptions for functions so that the designer easily can translate these texts into more formal pseudocode. Any change made to a data element in the data dictionary is automatically reflected everywhere the data element is used, e.g. in data-flow diagrams as an arc connecting nodes, or in other elements of the data dictionary or in the informal mini specs describing the tasks of the system. Changing parameters in a function results in an automatic update of the interface description of a module, changing a function's name automatically changes all the calls of that function.

These elegant and powerful features assist the system developer in that kind of work, that usually is not only laborious but also the source of many errors and troubles. The tools of PROMOD help the developer to concentrate on his or her most important job: to be creative.

7. Summary

PROMOD has been developed by a group of practical people to serve their own needs in system development support as well as the needs of their company. The internal goal was to provide adequate means ensuring the high standard and quality of systems and software developed by GEI . Because of its internal success it has been productized and is now available to help all systems analysts and designers. In many small, medium and very large projects PROMOD guided systems developers through the life cycle, showing them errors, suggesting solutions and preparing documents for reviews and presentations.

Literature:

/1/ T. DeMarco
Structured Analysis and System Specification
Yourdon Press, 1979

/2/ D.L. Parnas
On The Criteria To Be Used In Decomposing Systems Into Modules
CACM, Vol.5., No.12, Dec. 1972

/3/ G. J. Myers
Reliable Software Through Composite Design
Van Nostrand Reinhold, 1975

/4/ N. Wirth
Algorithms + Data Structures = Programs
Prentice Hall, 1976

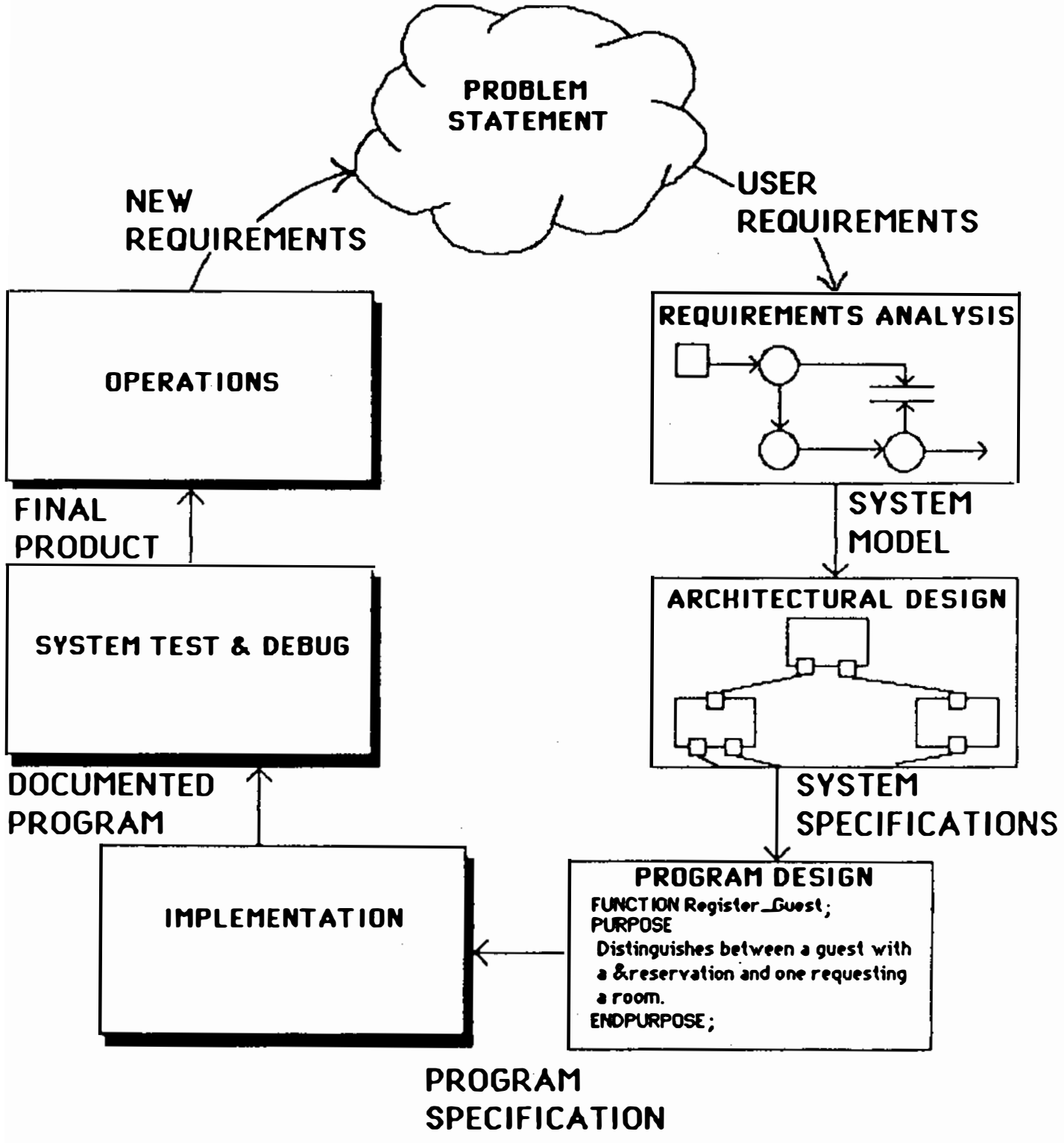
/5/ S. H. Caine, E. W. Gordon
PDL: A Tool For Software Design
in: AFIPS, Proc. NCC, Vol. 44, 1975

/6/ B. Liskov, S. Zilles
Programming with Abstract Data Types
ACM Sigplan Notices, Vol. 9, No. 4

BIOGRAPHY

Peter Hruschka

Peter Hruschka received his degrees in computer science from the Technical University of Vienna, Austria. He started working in the field of programming languages, participating in the definition, standardization, and implementation of the German real-time language PEARL. In 1979 Dr. Hruschka became Training Director at GEI Systems, mainly teaching seminars on software engineering and project management. During this period he developed the design tool DARTS (Design Aid for Real-Time Systems) and the concepts of the System Engineering Environment ProMod. Since 1982 he has been Product Manager for Promod, Inc.


PROBLEM STATEMENT
NEW REQUIREMENTS
USER REQUIREMENTS
OPERATIONS
REQUIREMENTS ANALYSIS
FINAL PRODUCT
SYSTEM MODEL
SYSTEM TEST & DEBUG
ARCHITECTURAL DESIGN
DOCUMENTED PROGRAM
SYSTEM SPECIFICATIONS
IMPLEMENTATION
PROGRAM DESIGN
FUNCTION Register_Guest;
PURPOSE
Distinguishes between a guest with a &reservation and one requesting a room.
ENDPURPOSE;
PROGRAM SPECIFICATION

WHAT IS IT?

>> INTEGRATED SYSTEMS ENGINEERING ENVIRONMENT

>> A SET OF COMPUTER DRIVEN TOOLS

- > REQUIREMENTS ANALYSIS
- > ARCHITECTURAL DESIGN
- > PROGRAM DESIGN

>>CONTROL PROJECT DESIGN

- > CENTRAL DATA BASE MANAGEMENT
- > DATA DICTIONARY
- > INTERACTIVE TEXT & GRAPHICS EDITORS
- > IMMEDIATE & GLOBAL ANALYZERS
- > REPORT GENERATORS
- > DOCUMENTATION & SPECIFICATIONS

WHERE DID IT COME FROM?

>> AN INTERNAL TOOL FOR G. E. I.

> MULTI-NATIONAL ORGANIZATION

> LEADING SYSTEMS & SOFTWARE HOUSE

> HEADQUARTERED IN WEST GERMANY

> CURRENTLY 400 EMPLOYEES, $30M REVENUE

>> THE PRODUCT HISTORY

> CONCEPTUAL DESIGN IN 1980

> INTERNAL TOOL IN 1981

> COMMERCIAL PRODUCT SINCE 1983

> CURRENTLY AVAILABLE ON VAX & IBM-PC

WHO SUPPLIES IT?

>> PROMOD INC.

> HEADQUARTERED IN LOS ANGELES

> CUSTOMER SUPPORT GROUP

> DEVELOPMENT ACTIVITIES

> SALES & MARKETING

WHO USES IT?

>> INTERNATIONAL CUSTOMER BASE

- > DIGITAL EQUIPMENT
- > BOEING
- > TELEDYNE
- > PHILIPS
- > SIEMENS
- > UNITED STATES NAVY
- > GRUMMAN

WHY USE PROMOD?

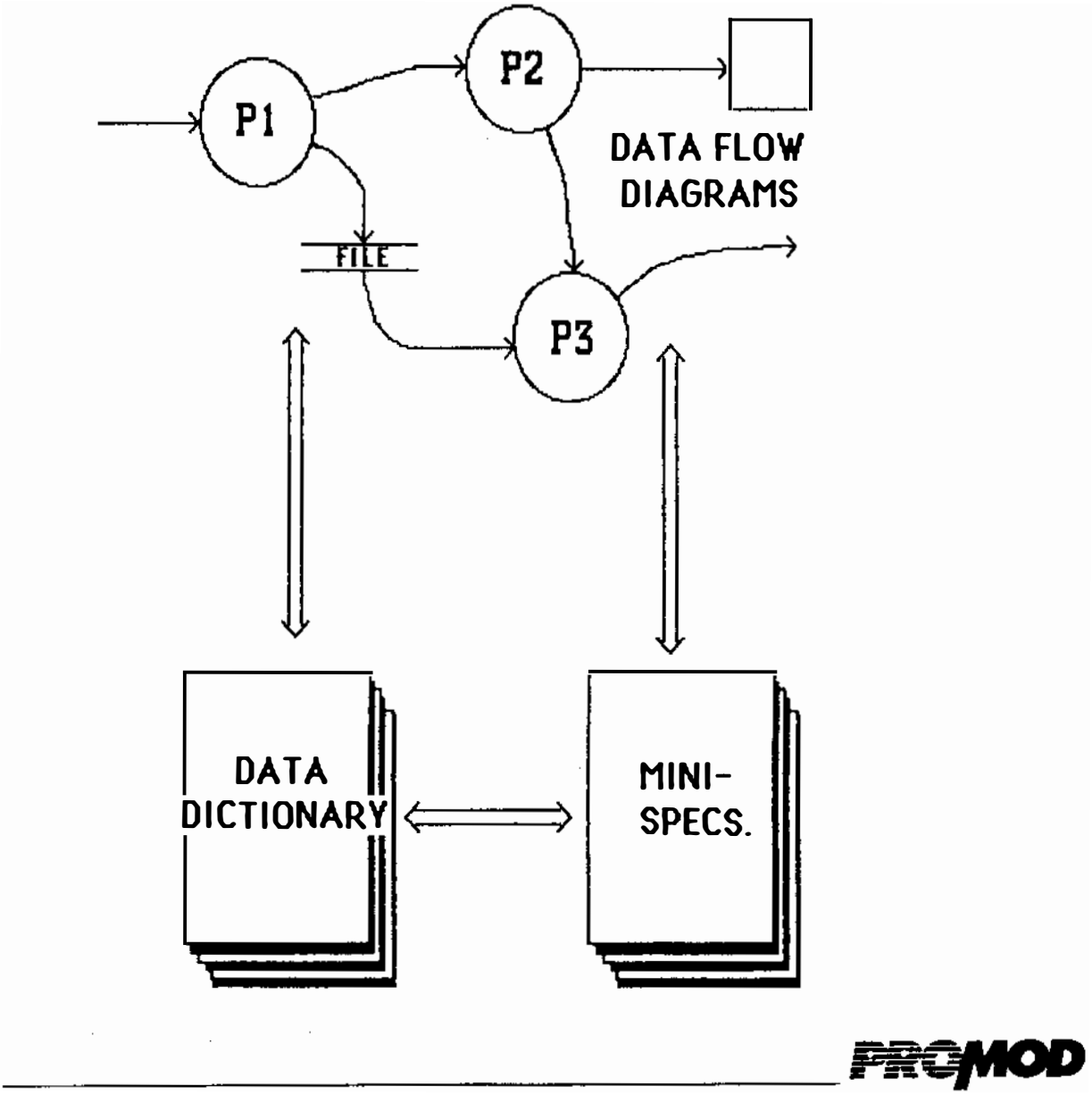
>> INCREASED PRODUCTIVITY

>> IMPROVED PRODUCT QUALITY

>> CURRENT & CONCISE DOCUMENTATION

REQUIREMENTS ANALYSIS

STRUCTURED ANALYSIS -- YOURDON



REQUIREMENTS ANALYSIS

THE TOOL SET

>> INTERACTIVE SYNTAX DIRECTED TEXTUAL & GRAPHICAL EDITING

>> IMMEDIATE CHECKS FOR SYNTAX & MEANING

>> GLOBAL ANALYSIS FOR CONSISTENCY & COMPLETENESS

REQUIREMENTS ANALYSIS BENEFITS

>> GUIDED TOUR THROUGH THE ANALYSIS PHASE

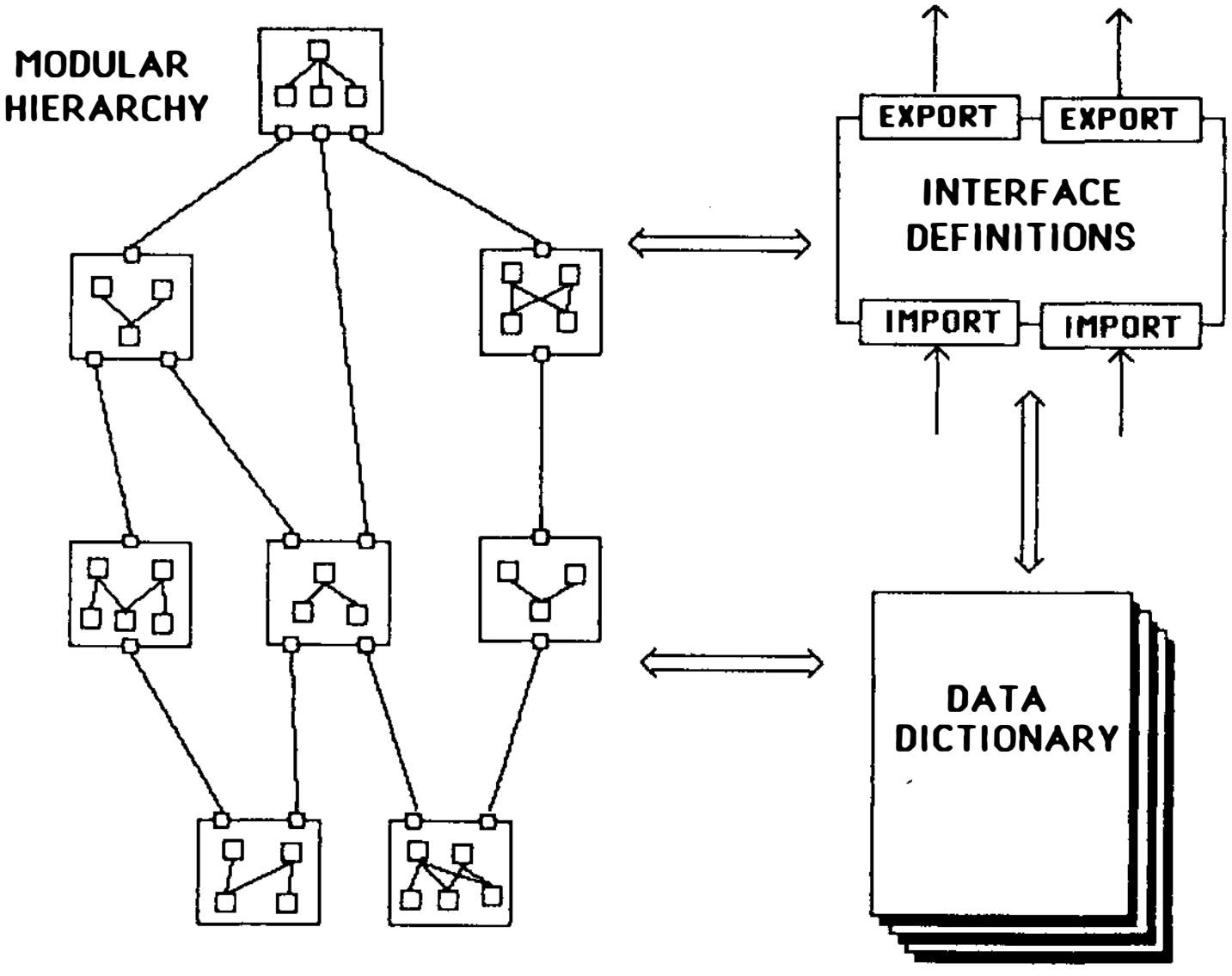
> PREDEFINED PROCEDURES & PRODUCTS

> WELL KNOWN & ACCEPTED METHODOLOGY

>> SHORTER, MORE PRECISE SPECIFICATIONS

>> STRUCTURED & VERIFIED DOCUMENTATION

ARCHITECTURAL DESIGN
MODULAR DESIGN -- PARNAS



ARCHITECTURAL DESIGN

THE TOOL SET

>> AUTOMATED TRANSITION FROM REQUIREMENTS ANALYSIS

>> IMMEDIATE LOCAL CHECKS OF INTERFACE DEFINITIONS

>> GLOBAL CHECKS FOR CONSISTENCY OF INTERFACES

>> STRUCTURED, CONCISE REPORTS

ARCHITECTURAL DESIGN BENEFITS

>> WIDELY STANDARDIZED

>> PRECISE HIGH LEVEL INTERFACES

(DISTRIBUTION OF WORK)

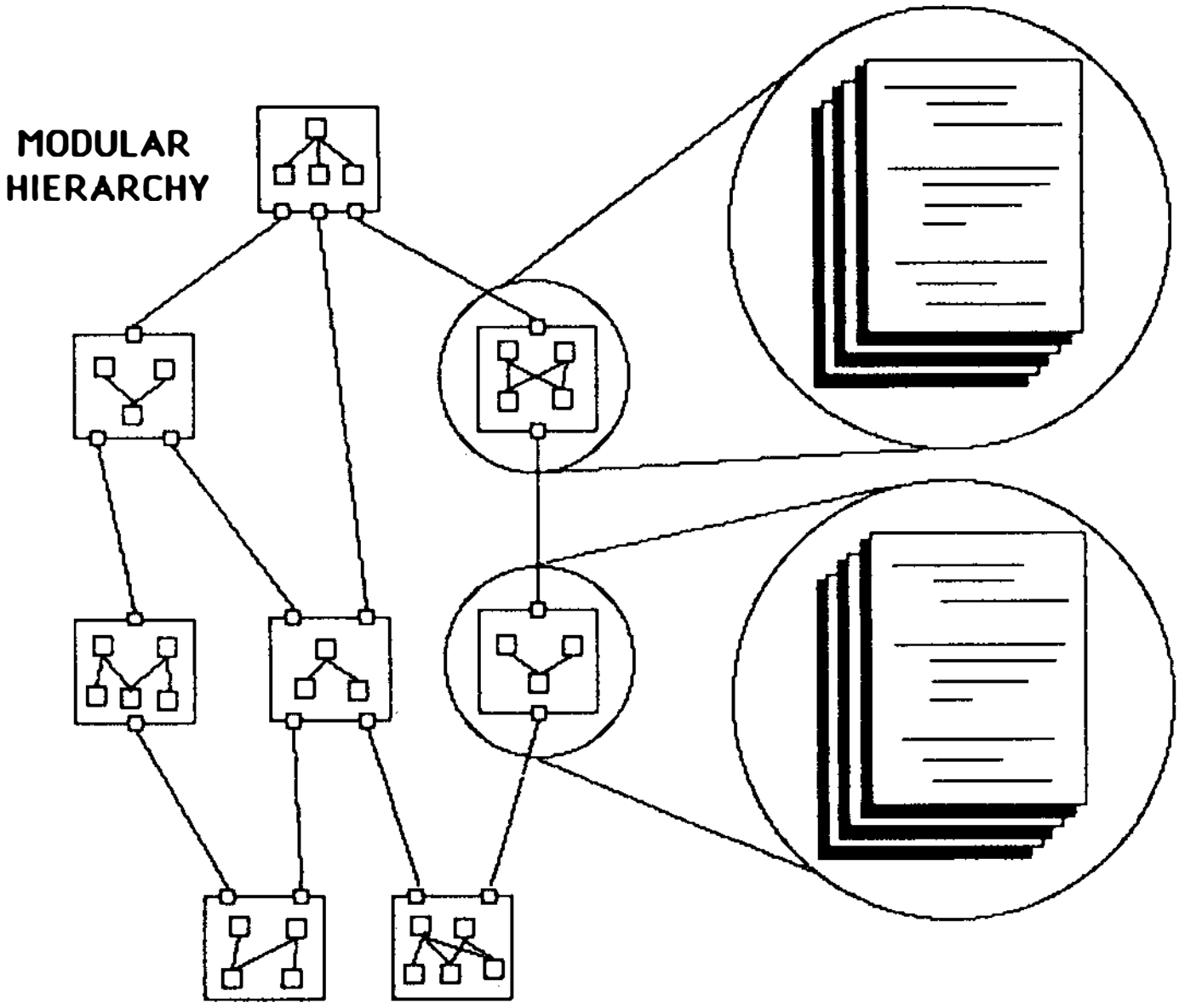
>> INFORMATION HIDING (BLACK BOXES)

>> INCREASED FLEXIBILTY IN MAINTENANCE PHASE

>> SMALL COMPREHENSIBLE UNITS

>> ADA COMPATIBLE

PROGRAM DESIGN
PSEUDOCODE -- CAINE & GORDON, WIRTH



PROGRAM DESIGN

THE TOOL SET

>> PROVIDES RECOGNITION OF KEYWORDS IN PSEUDOCODE

>> VISUALIZATION OF LOGICAL STRUCTURE

>> IMMEDIATE LOCAL CHECKS OF LOGICAL STRUCTURE

>> GLOBAL CONSISTENCY CHECKS WITH INTERFACES

>> STRUCTURED & CONCISE REPORTS

PROGRAM DESIGN BENEFITS

>> IMPROVED COMMUNICATION BETWEEN ANALYST & USER

>> CHANGES AND AMENDMENTS EASILY INCORPORATED

>> STRUCTURE IMPOSED ON NATURAL LANGUAGE WHILE ALLOWING ADEQUATE ROOM FOR CREATIVITY

>> EASY TO LEARN

INTEGRATED SYSTEMS ENGINEERING ENVIRONMENT

>> EASY TO INTRODUCE

> WELL KNOWN, WIDELY USED METHODS

> BASED ON HUMAN UNDERSTANDING

> IMPROVED MAN/MACHINE INTERFACE

>> EASY TO TEACH & LEARN

> PRECISE METHODS & PROCEDURES

> STANDARDIZED SCHEMAS

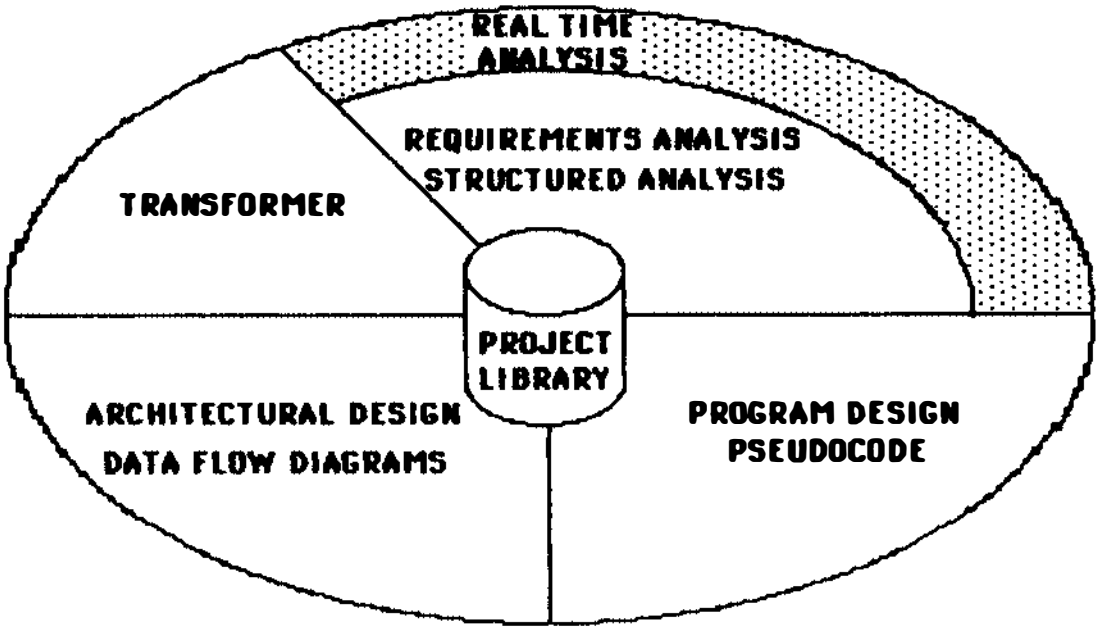
>> EASY TO USE

> UNIFORM TOOL INTERFACES

> MNEMONIC COMMANDS

> SELF EXPLANATORY MESSAGES

> EARLY ERROR DETECTION


REAL TIME
ANALYSIS
REQUIREMENTS ANALYSIS
STRUCTURED ANALYSIS
TRANSFORMER
PROJECT
LIBRARY
ARCHITECTURAL DESIGN
DATA FLOW DIAGRAMS
PROGRAM DESIGN
PSEUDOCODE

Locating Suspect Software and Documentation by Monitoring Basic Information About Changes to the Source Files

Dave Vomocil

July 31, 1985

We can gain useful insights about the status of software projects by monitoring relatively basic items. An instrumented source editor can be used to record module size at times of change, the number of lines added, the number of lines deleted, and other basic information items. When this data is tabulated or plotted against time, it becomes relatively easy to spot suspect modules.

The hypothesis is that a file, source or documentation, should undergo an increasing amount of change during the implementation phase. Then the rate of change should decrease and remain small relative to the size of the file.

This paper discusses the theory and how to implement it in a unix development environment.

Locating Suspect Software and Documentation by Monitoring Basic Information About Changes to Source Files

Dave Vomocil

Hewlett Packard, Corvallis, Oregon

1.1 INTRODUCTION

This paper intends to present a metric for locating problem modules and to demonstrate how easily this metric can be implemented in a unix1 programming environment. The metric graphically points out modules that are receiving an inordinate amount of attention, and statistics from it have been successfully used to argue that particular modules need to be rewritten. In addition, after the metric has been used for some time the results can be characterized and used to predict program size and release date. The paper presents some history, the ideas behind the metric and, primarily, what standard unix tools can be used to apply the metric in a software development environment.

We can improve quality control:

- without imposing time consuming and frequently inaccurate data entry requirements on engineers,
- without requiring an understanding of complex software metrics, and
- without building or buying expensive software tools.

1.2 HISTORY

The impetus for this work came from a paper presented in 1984 by Dan Lundberg of Hewlett Packard at Hewlett Packard's annual Software Productivity Conference. His paper discribed three ideas that had been studied by a Japanese company desiring improved statistical quality control.

The Japanese company looked first at program size as a metric to predict both release date and quality at release. They found, as many others have, they were unable to accurately predict program size early enough in the development phase to make the predictions valuable.

Secondly, the firm studied the effect of reusing tested modules. As a result of this study, they were able to develop tables allowing them to predict at release time (with reasonable confidence) the quality of the released product based on the percentage of the product that was reused code.

To facilitate these two studies they developed an instrumented editor and project management package

---

1unix is a trademark of Bell Labs.

that recorded vital statistics associated with programs. These statistics included a 1) history of module size and 2) history of modifications made to the module. For example, a statistics record would include date information, module size information, and a measure of the amount the module was changed. By plotting either of these items, size or amount of change, against time the final size of a program could be predicted at a reasonable time in the development phase. Additionally, these statistics were used during the support phase to indicate which programs needed complete rewrite.

To become able to predict program size and release date they first had to characterize the shape of the curves generated when either program size or cummulative changes were plotted against time. The technical content of a module determined the characteristic shape of the family of S-curves associated with the module. Once the curves had been characterized, they were able to predict final program size and quality at release time early in the coding phase.

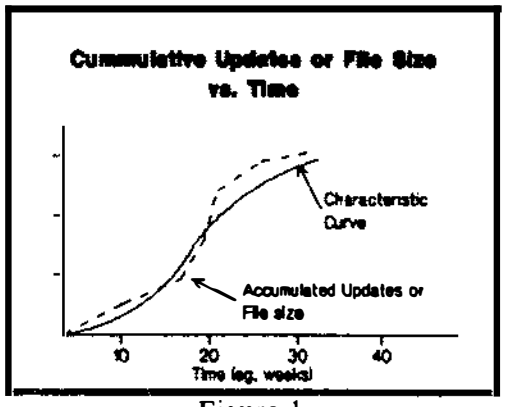


Figure 1

As an additional benefit, the statistics could be reset at release time. The statistics would then accumulate during the support phase. The charts produced from the activity during the support period were used to detect problem modules and argue successfully for rewrite of particular modules.

1.3 SCOPE OF THIS PAPER

In the past, engineers have been required to complete logs to supply data for use with statistical models. This data acquisition process was inherently inaccurate since it had little relation to the engineer's progress on his assigned project. Then the questionable data is piped into statistical model, and questionable conclusions are generated.

The Japenese used an instrumented editor to gather the data more accurately. Additionally, the statistical model used with this data is very easy to understand. Therefore, we wanted to test it in our environment.

Initialy we were frustrated because we did not have an *instrumented* editor, and we did not want to invent or buy one. After we moved to HP-UX, Hewlett Packard's unix operating system, we found that the Source Code Control System (SCCS) provided an excellent tool to collect the data. awk, another tool provided with HP-UX, could be used to extract the data from the **SCCS** files. A short C program was written to massage the output from the awk script. These steps leave the data in a form most chart presentation packages can use. The application of these standard unix tools to support the gathering and preparation of data for the metric is discussed in the remainder of this paper.

It should be pointed out that once the software engineering team has moved to a unix environment the tools needed are readily available. All of the parts can be mastered **and implemented** in a few days.

1.4 THE INSTRUMENTED EDITOR - SCCS

The first step is to collect the data. In many older software engineering environments, including our past environment, automating the data collection meant a major programming effort or a major purchase. Neither the effort nor the purchase was justified for an unproven tool. After adopting a unix engineering environment, we found the *Source Code Control System (SCCS)* could be used to collect the data. The *Revision Control System (RCS)* from Purdue could most likely be used just as well. We chose SCCS because it was readily available and running on our engineering machines.

SCCS is a standard unix tool that manages multiple versions of a text file with a single file. An example is provided in table 1. The left column of the example contains three versions of a short text file, the original version and versions that result from two short editing sessions. The right column contains the three respective SCCS files. (The third SCCS file has been truncated to keep the example on one page.)

The lines in the SCCS files prefaced with an '*s*' specify the number of lines that have been **added, deleted,** and **unchanged** respectively. The line following each '*s*' line (prefaced with a '*d*') contain date information indicating when the edited version was checked into the SCCS file.

The SCCS system is documented in most unix reference manuals. Basic use of the system involves mastering three commands.

1) **admin -i<file> s.<SCCS file name>**
   The admin command is used set up the initial SCCS file.

2) **get -e s.<SCCS file name>**
   The get command with the '*-e*' option is used to check out a version for editing.

3) **delta s.<SCCS file name>**
   The delta command is used to check in an edited version.

Source Files and SCCS File - Table 1

This is line 1 This is line 2 This is line 3 This is line 4	h14141 s 00004/00000/00000 d D 1.1 85/06/05 13:10:32 davev 1 0 c created 85/06/05 by davev e U u t T I 1 This is line 1 This is line 2 This is line 3 This is line 4 E 1
This is line 1 After line 1 This is line 2 This is line 3 This is line 4 End first edit	h30168 s 00002/00000/00004 d D 1.2 85/06/05 13:17:30 davev 2.1 c Result of first editting session. e s 00004/00000/00000 d D 1.1 85/06/05 13:10:32 davev 1.0 c created 85/06/05 by davev e u U t T I 1 This is line 1 I 2 After line 1 E 2 This is line 2 This is line 3 This is line 4 I 2 End first edit E 2 E 1
Delete and add. This is line 1 After line 1 This is line 3 This is line 4 End first edit One last line	H50689 s 00002/00001/00005 d D 1.3 85/06/05 13:20:31 davev 3 2 c Result of second editting session e s 00002/00000/00004 d D 1.2 85/06/05 13:17:30 davev 2 1 c Result of first editting session.

1.5 EXTRACTING THE DATA - AWK

The second step is to extract the data from an SCCS file. As explained above, only the '*s*' and '*d*' lines contain needed data. *awk* can be used to extract the data from those lines.

awk is a standard part of a unix environment. This programming language allows users to manipulate text and data. An *awk* program expects lines of input from standard in (usually a file), processes the line (eg. does arithmetic), and generates output. The output, which can optionally be formatted, is posted to standard out. Both standard in and standard out can be redirected to reference files. *awk* is documented in most unix reference manuals.

awk programs I have used to extract data from SCCS files are included below.

**** the extract script ****

```
awk -f awk1 <SCCS file | awk -F\/ -f awk2 >output file
```

**** the first awk program -- awk1 ****

```
$1 ~ /s/ {x = $2}
$1 ~ /d/ {printf "%8s/%17s\n", $4, x}
$1 ~ /u/ {exit}
```

**** the second awk script -- awk2 ****

```
{ printf "%2s%2s%2s %5s %5s %5s\n", $1, $2, $3, $4, $5, $6 }
```

The above contains three items. The first is a script which invokes awk twice, and the second and third are the awk programs. The first invocation of awk applies the first awk program directly to the SCCS file.

The three things it accomplished by the first invocation are:

- Anytime a line with starting with an 's' is found, the second item in the line is stored in the variable *x*. The second item in such a line is the number of lines added, number of lines deleted, and number of lines remaining unchanged. Since the default delimiter is a space and these counts are delimited by '/', the three counts are considered one item.

- Anytime a line starting with a 'd' is found, the second item is printed followed by a '/'. Then the information extracted from the previous 's' line, contained in the variable x, is printed. In this context *printed* means written to standard out.

- Finally the program exits if a line starting with a 'u' is encountered. This merely keeps the program from searching through the body of the SCCS file.

The output from the first invocation of awk is '*piped*' 2 into the second invocation of awk; then the second awk program is applied. The *-F* option on the second invocation sets the field delimiter to '/'. This second awk program merely formats the fields and separates them with spaces, i.e. makes them easier for a person to read.

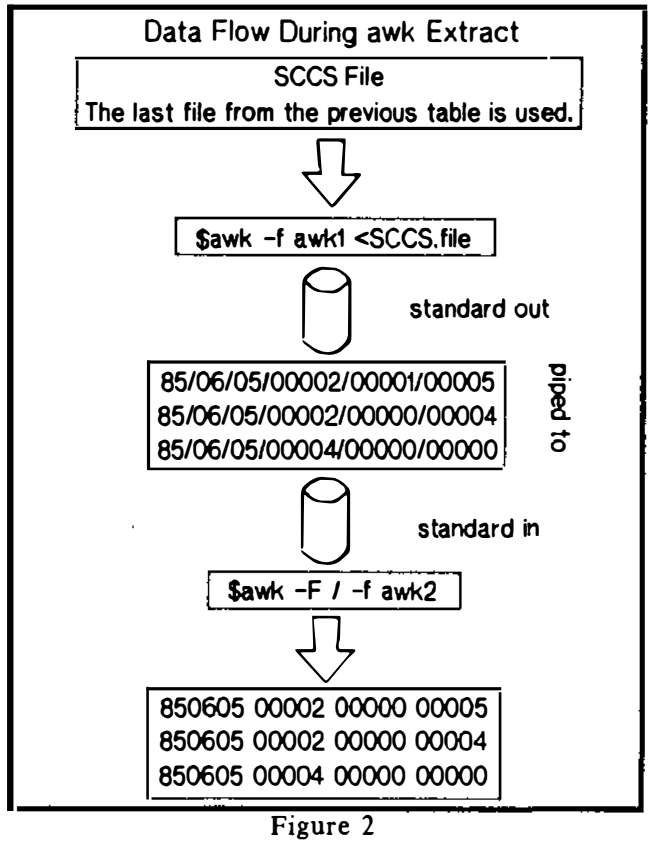


Figure 2

2The vertical bar "|" bewteen the two invocations of awk causes the output from the first to be used as the input for the second. This feature of unix, i.e. to be able to use the standard output of one program as the standard input of a second, is referred to as a pipe.

1.6 SORTING THE DATA

The awk scripts leave the data in the same order it appears in the SCCS file. That is, the first record contains the most recent information and the last record contains the oldest information. Most chart presentation packages will want the information in the reverse order. The unix *sort* utility will easily solve this ordering problem.

The output from the awk programs is what needs to be sorted. The lines of the output need to be sorted in ascending order based on the dates. Example output is given in the bottom of figure 3. The date is in the first field on each line. By default, the unix *sort* program uses the first field and sorts the lines in ascending order. Therefore, we can merely apply the sort program with no parameters to the output of the awk programs.

As with the two invocations of awk, a unix pipe can be used with the sort program. That is, we can actually combine the two previous awk calls with a call to sort, as pictured to the right, and get all the work done in one step. The SCCS file has been edited again for this example. In particular, the dates have been modified to give the sort program something to do.

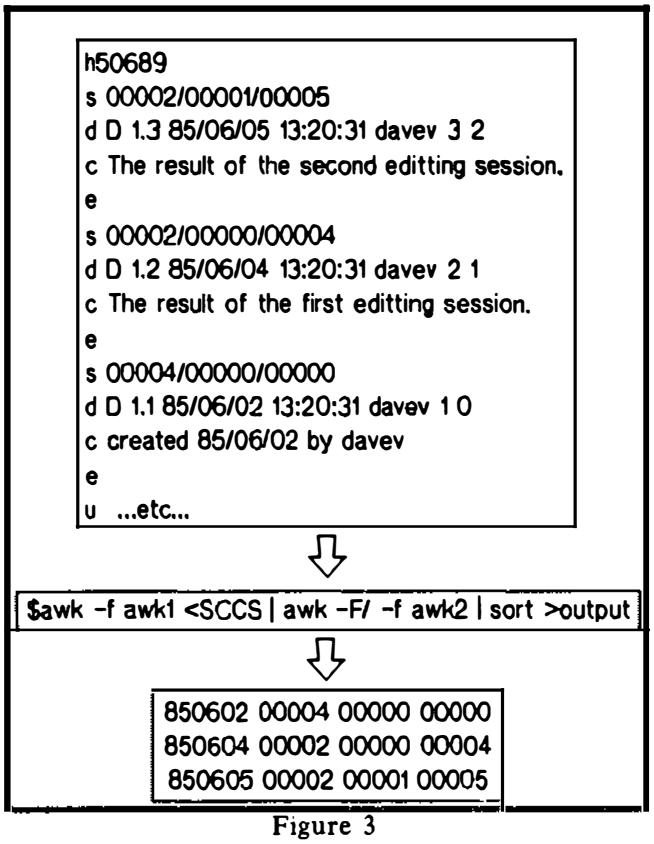


Figure 3

1.7 COUNTING THE DAYS - A C PROGRAM

The data is now extracted and sorted. If SCCS is used religiously once a day and every day, the data could be handed to a chart presentation package in its present state. The remaining problem is that SCCS is frequently not used this regularly, and one of the motivations behind this scheme is to not make such demands on the engineers. Therefore, to be able to place the data points correctly on the graph, the number of days between each datapoint needs to be calculated. Since many chart presentations packages cannot make such conversions, a C program was written to make the calculations. A copy of the C program is included in the appendix.

As before, this step can merely be added to the pipe. As is indicated by the flow diagram in figure 4, the C program adds a column of data in which each entry is the number of days since the beginning of the project. In addition to calculating the number of days between each datapoint, the program summarizes activity if there are multiple datapoints on a single day. You are referred to the C program for specifics on how the summary works. By reviewing the C program you might also appreciate the ease with which the summarizing could be customized.

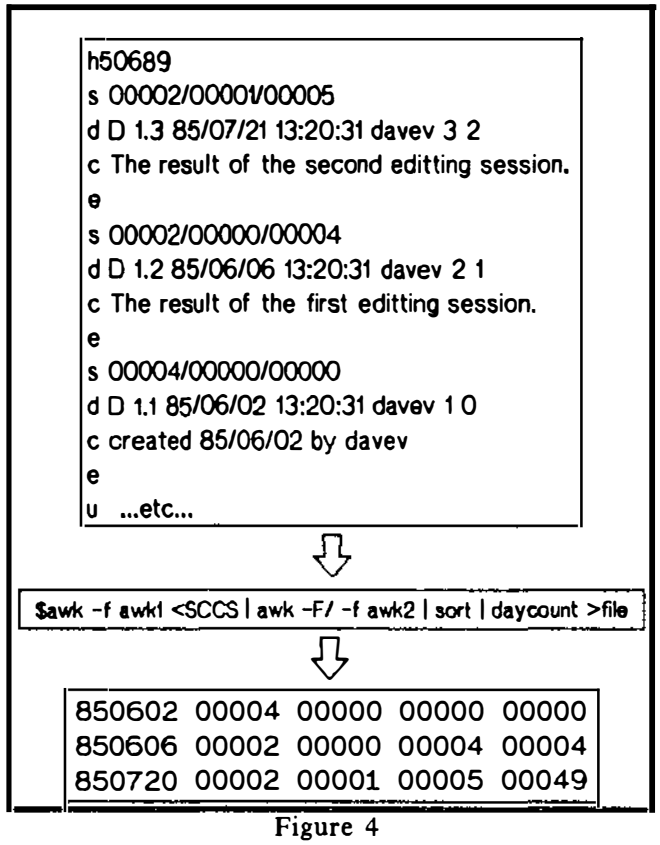


Figure 4

1.8 THE FINAL STEP - PRESENTING THE DATA

The data the above described procedures generates can be most easily interpreted when it is presented as a line graph. Most chart preparation packages (eg. Lotus/123, Picture Perfect, DSG/3000, to name a few) will accept the resultant file, that is the file created as output by the procedures described above, as input for creating a chart. Since the chart preparation tools available at different locations vary considerably, none is described in any detail here. At some sites a graphics package will be available on the host unix systems, and at other sites users will need to move their data to a PC or other host computer. If you need to move your data to another computer system to generate the charts, *kermit* is a reasonable tool to use; and it is available on the unix notes network and from universities.

1.9 CONCLUSION

This paper has presented both a metric for identifying modules that need to be rewritten and has described the standard tools available in a unix environment that can be used to implement the metric. The metric has merit in that:

- the data is easily collected. The collection involves no extra work by the engineer.
- the data is automatically stored and is accurate. The system does not rely on engineers and/or project managers remembering how much time was spent on various phases of the project.
- The metric is easy to understand and apply.

Equally important to this discussion is that the metric can be implemented with tools that are standard components of an unix environment. Building a similar system in many older environments meant paying for a medium to large project, and the results of many such past projects have been hard to use and nearly impossible to modify. In a unix environment, the pieces can be put together by a project manager in a short time., The resulting system is robust and easy to continue to modify.

1.10 APPENDIX - C SOURCE FOR DAYCOUNT

```
#include <stdio.h>

main()
{
        int yr, mo, dy, ins, del, unch;
        int yr1, mo1, dy1, ins1, del1, unch1;
        int days, unchanged, status;
        int day1, day2, tins, tdel, tunch, tdays;

        /* Determine initial conditions */
        /* i.e. initial date and initial program size. */
        scanf( "%2d%2d%2d %5d %5d %5d",
              &yr1, &mo1, &dy1, &ins1, &del1, &unch1);

        /* Check for further activity on day one. */
        scanf( "%2d%2d%2d %5d %5d %5d",
             &yr, &mo, &dy, &ins, &del, &unch);
        while ( yr == yr1 && mo == mo1 && dy == dy1)
                {
                if ( unch == 0 && del == 0 ) unchanged = ins;
                scanf( "%2d%2d%2d %5d %5d %5d",
                        &yr, &mo, &dy, &ins, &del, &unch);
                }
        printf( "%2d %2d %2d     0     0 %5d     0\n",yr1,mo1,dy1,unchanged);

        /* Proceed with rest of days logged in SCCS file. */
        tdays = 0;
        day1 = julian(yr1, mo1, dy1);
        do
        {
                /* initialize for present day */
                day2 = julian(yr, mo, dy);
                yr1 = yr; mo1 = mo; dy1 = dy;
                tins = ins; tdel = del; tunch = unch;

                /* scan for more activity on present day. */
                while (
                   (status = scanf( "%2d%2d%2d %5d %5d %5d",
                        &yr, &mo, &dy, &ins, &del, &unch)) != EOF &&
                   yr1 == yr && mo1 == mo && dy1 == dy )
                   {
                        tins += ins; tdel += del;
                        if( tunch > unch ) tunch = unch;
                   }

                /* Compute days since last activity
                  allowing for change of years. */
                if (day2 > day1) days = day2 - day1;
                else
```

```
                {
                days = ( 366 - day1 ) + day2;
                if ( (yr/4)*4 == yr ) days -= 1;
                }
            tdays += days;

            /* Post present day's activity to standard out. */
            printf ("%2d %2d %2d %5d %5d %5d %5d\n",
                yr1, mo1, dy1, tins, tdel, tunch, tdays);
            day1 = day2;
        }
        while (status != EOF);

}

julian(yr, mo, dy)

        int yr, mo, dy;
{
        static int months[] = {00,00,31,59,90,120,151,180,211,242,272,303,333};
        int i, days;

        days = months[mo] + dy;
        if (((yr/4)*4 == yr) && mo > 2) days += 1;
        return(days);
}
```

BIOGRAPHY

David Vomocil

David Vomocil earned his BS in science and mathematics from Portland State University in 1969 and his MS in computer science from Oregon State University in 1975. After a year at Cornell University and some time with NCR in New York, Mr. Vomocil came to work with Applied Theory Associates in Corvallis, Oregon. He is now at Hewlett Packard, where he supervises the computer services group for the Calculator Lab.

Session 6

TESTING AND PROBLEM REPORTING, II

Titles and Speakers:

"A Software Test Environment for Embedded Software"
David Rodgers and Ralph Gable, Boeing Commercial Airplane Company

"CLUE--A Program and Test Suite Evaluation Tool for C"
David Benson, BENTEC

"Tools for Problem Reporting"
Susan Bartlett, Metheus-CV, Inc.

A SOFTWARE TEST ENVIRONMENT FOR EMBEDDED SOFTWARE

BY DAVID A. RODGERS AND
RALPH GABLE

BOEING COMMERCIAL AIRPLANE COMPANY
P.O. Box 3707
M/S 77-21
SEATTLE, WASHINGTON 98124-2207

ABSTRACT

A software test environment is described that supports the testing of embedded, dual-dissimilar avionic control system software.

The environment design addresses the problems of testing a total software system. The design frees the software tester from operational test constraints (stop/start control, error introduction, etc.) often imposed by the hardware surrounding the embedded software. The environment provides input stimulus that is exact and repeatable for each operational cycle of the software under test. The software overall response is measurable on a cycle-by-cycle basis. The environment allows detailed monitoring of internal software events, for analysis by software designers and verifiers. The environment supports the dual-dissimilar nature of the software system to be tested.

The environment is designed to interface with and be user friendly to system engineers, who are cognizant of the fuctions to be performed by the software under the test but who may not be skilled in software techniques themselves.

The environment's test procedures are written in English language-like statements that use the terminology of the software system under test. The procedures tend to be self documenting. Software system test scenarios may be readily generated with economy of statements. Input test stimuli at bit level is generally invisible to the test writer and its generation is automated, providing reduced input errors. The environment can handle a complex set of digital discretes, analog and ARINC-429 signals. Output reports are generated that are readily interpreted by system engineers and software engineers alike.

TEST SYSTEM REQUIREMENTS

The software test environment described was developed to meet these requirements:

TABLE 1 - TEST ENVIRONMENT REQUIREMENTS

(1) It must support the verification of functional requirements of integrated software that is part of a dual-dissimilar system (see fig. 1) and that would later be embedded. Once embedded, the precise functional performance of the software would be difficult to verify due to considerations of timing control, repeatability, sensitivity and accuracy of hardware stimuli and measurement devices. ('Embedded' software is that which is an integral part of a hardware/software product and usually resides in ROM. 'Integrated' here implies the software is in its load form, as it would appear in ROM).

(2) It must (a) simulate the hardware in which the software under test (SUT) is to be later embedded (b) emulate the CPU on which it will be executed, in the final product for data collection and recording and (c) provide for data collection and recording. The input stimuli mechanism must support (1) up to five ARINC-429 channels each carrying up to five different labels, (2) five analog signals of up to twelve bits per signal and (3) up to sixty discrete signals. All input stimulus must be changeable at any and every basic cycle of the SUT. (An ARINC-429 channel carries 32-bit serial data messages. Each is comprised of an 8-bit label identifier, 2 bit status matrix, up to 21 bits of variable data and a parity bit). The host CPU emulator must support an Intel Z80 or a Motorola 6802.

(3) It must relieve the test writer as much as possible from the requirement for software skills. The writers must be given the opportunity to develop functional test scenario procedures using terminology that is familiar to the final product's system designers.

(4) The written procedures must be easy to interpret for audit and test maintenance purposes. Their format must lend themselves to precise expression of test stimuli, test operational steps and results measurments. The procedures must bemachine readable. Clerical support is to be minimized.

(5) Due to the number and complexity of test scenarios (over four hundred distributed over four separate SUT systems, each of which will go through four or five updates and each requiring verification), the translation of the test scenarios procedures into a form suitable for execution of the test , the test set operation and the formatting of measured results into test reports must be error-free automatic operations with minimal and simple manual intervention.

(6) The characteristics of the actual hardware/software interface of the embedded software will change during normal product development, forcing modification of the test environment's simulated hardware. The design of the test environment must comprise simple modules to accomodate these changes.

The Software under Test (SUT)

The software to be tested is dual-dissimilar (see fig.1). That is, the primary functional outputs of the system are supported by one CPU, say CPU-1, while the same functions are simultaneously generated by another CPU (CPU-2) of dissimilar architecture. Each CPU monitors the other's performance and each may individually disconnect the primary output in the event of unacceptable performance. Ideally, the software design and its implementation for each CPU are developed by separate design teams in order to reduce the probability of a common (or 'generic') error at any step of the software development process. Dissimilar CPUs are chosen to avoid operational generic faults.

The software architecture of one CPU is similar to that given in fig. 2. The foreground tasks are scheduled typically by two clock driven interrupts, one of which has priority over the other. The interrupt clocks of CPU-1 and CPU-2 run at the same frequency but are not synchronized. In real time the basic cycle interrupt initializes the primary input-process-output functions with the balance of the basic cycle time spent in background processing which typically comprises continuous ROM and RAM checking.

Special fast processing may be necessary on several occassions during the basic cycle. The fast process cycle is serviced by the higher rate, secondary interrupt which is synchronized to the basic cycle clock.

The problem then, was to provide a useful test environment for the software described and to meet the requirements of Table I.

TEST ENVIRONMENT OVERVIEW

The test system is shown in fig. 3. The system is comprised of two computer environments, (1) a VAX 11/780 and (2) a Tektronix 8002 emulator linked by a communication line.

Analysis showed that to verify the SUT's logical performance it was not necessary to execute the SUT in real time. Its logical performace could be measured in non-real-time providing (1) the execution sequence of the SUT was sufficiently similar to that experienced in real-time and (2) the SUT experienced stimuli similar to real-time stimuli. Also, in this case of dual-dissimilarity it was not necessary to verify each software system (CPU1, CPU2) at the same time. Under normal, non-fault conditions the

output from each CPU on a dissimilar system will be identical. Thus, with due regard to phase and polarity, a set of "pseudo" dissimilar CPU (say CPU2) output signals may be generated from a single system SUT (say, CPU1). The "pseudo" signals must be used as feedback input to the SUT itself, delayed by one cycle. A single system SUT thus may thus generate its own dissimilar channel inputs. This mechanism releives the test writer from having to predict the proper input to the single CPU SUT from the dissimilar channel. A means must however be provided to force 'incorrect' dissimilar CPU signals to simulate fault conditions.

The SUT is then a set of single CPU software, loaded into the Tektronix emulator memory and mapped into the same address space as when embedded in the final product. Resident with the SUT in emulated memory are (1) input stimuli data bases (one per hardware driver) derived from the input scenario of the test procedure, (2) a set of simulated hardware drivers (ARINC, analog and discrete) and (3) a special test operating system (Test O/S). The Test O/S, the drivers and the SUT are configured such that control passes first from the Test O/S to the drivers to establish the first (or next) cycle's input stimuli data at the SUT's hardware/software input interface and, secondly, to the SUT itself which attempts to execute in a normal manner. At the end of the cycle, in the background program, control is returned to the Test O/S. Output measurements are taken at the hardware/software output interface and written to the emulator disk. Optionally, the SUT software may be pre-modified to produce software interrupts so that the Test O/S records the value of some or all test scenario RAM variables as they exist at the completion of execution of previously specified SUT software modules. At test completion, the gathered data on disk is returned to the VAX by communication line and formatted (fig. 3) into a report.

INPUT DATA BASES

The input data is generated by the test writer in English language-like statements. The statement syntax rules are designed to give the test writer flexibility to express input stimuli in terminology used by the final product's system designers. For example, a discrete may be set by the statement:

```
HYD PRESS HIGH, HIGH, 1-29;
```

Here the Hydraulic Pressure High discrete is set to the "high" state (as shown on system drawings) for iterations (basic cycles) 1 through 29. At iteration 30 the discrete will be set to its default value.

The analog signal, Servo-Feedback, will be set to -2.98 degrees at iteration 3 and will remain at that value until otherwise specified by the statement:

```
SERVB = -2.98 DEGREES, 3;
```

The statement

RA, IRUC, 3.9, SM = NCD, P=B, 106;

will set the ARINC signal for RA (Roll Angle) on the IRUC (Inertial Reference Unit, Center) to 3.9 degrees with the SM (Status Matrix) to NCD (No Computed Data) with P (Parity) to the value B (Bad i.e. incorrect parity) from iteration 106 inward until set otherwise by another statement.

Commentary statements may be entered anywhere in the input statement stream. An example of a typical test procedure is given in fig. 4. Test control specification is embedded in such statements as: ITERMX (number of iterations this test), SELVAR (select variables to be measured), SELMOD (select modules after whose execution variable values will be measured) and NOMFIT (no measurements during iterations specified). Expected results are entered in comment format.

Test scenario source code is passed to translators written in Pascal and supported in the VAX environment. The translators produce compressed scenario data bases, ready for download to the emulator environment. Compression is achieved by only including data specifications at points of change rather than explicitly specifying data for each and every software cycle. An example of the data base format is given in fig. 5. The translators provide extensive error checking of statement syntax. The SUT linkmap, generated at SUT load generation time, its used both here and at output report generation time to correlate the mnemonics referenced with absolute SUT addressee.

THE SIMULATED HARDWARE DRIVERS

The drivers' function is to pass input scenario stimuli data from the appropriate input data base to the SUT in appropriate format and in a manner that sufficiently simulates the characteristics of the real hardware/software interface. The drivers are written in assembly language and are less than 1k bytes in size. Depending upon the SUT architecture, the drivers are either designed to be called (1) by the Test O/S, simulating a mechanism that pre-loads the DMA (direct-memory-access) memory for later access by the SUT or (2) by the SUT itself, simulating a mechanism that acquires data from a hardware I/O device using conventional I/O handshake protocol. In order to "hook" each driver into the SUT it is necessary to modify the SUT code instructions that normally supported the real hardware/software interface. In practice, such code corruption is minimal, with only a few I/O instructions being modified. The drivers are designed to detect abnormal calls by the SUT and to post error codes to the Test O/S. The drivers have the ability (1) to repeat the input stimuli scenario when the input data base is exhausted thus providing for stimuli with a cyclic characteristic (sine wave, square wave, ramp) and (2) to operate the SUT normally using default values..

THE TEST OPERATING SYSTEM

The function of the Test O/S is to control the test environment within the emulator. The Test O/S is comprised of (1) emulator JCL (job control language) procedures and (2) a Test O/S controller written in assembly language (3k-4k bytes in size) which interfaces with the SUT (again, with minimum code corruption). The Test O/S prepares the emulated SUT RAM areas, loads (1) the SUT, (2) the simulated hardware drivers, and (3) the downloaded test scenario data bases and test control requirements, ensures a proper load by checksum technique, performs the test by passing control to the Test O/S Controller, collects the specified measurements from the SUT and writes them to emulator disk, calls an on-line print driver that provides continuous monitoring of the SUT hardware/software output buffer (fig 6A) and provides the test set operator with continuous test status. The captured measurements are uploaded to the VAX and processed into a report (an example of which is shown in fig 6) and the actual results are compared to the expected (circled in fig 4 and fig 6).

OPERATIONAL EXPERIENCE

Two different dual-dissimilar systems (i.e. a total of four SUTs) were tested using the method described. A total of approximately four hundred different test scenarios were executed, the majority of which were non-trivial and often complex. During the test project, all four SUTs underwent change resulting in new SUT versions. Each new version was completely retested using, where necessary, updated test scenarios and Test O/S support software. The number of tests executed was in the order of two thousand.

It has been found that the advantages of this test system are: (1) the reduced need for in-depth software experience on the part of the test writer. The writer's experience can be primarily 'system' oriented. The test set operator needs minimal engineering skill since the test process is almost totally automated. Scarce software skill resources are directed to test system development/maintenance which has a lesser total cost in this case than that of test procedure preperation and results review. (2) Tests can be early rerun on new software versions to ensure previous level of confidence. (3) Tests can be quickly generated and de-bugged. (4) The test procedures can be more readily understood project-wide. (5) The procedures are self-documenting. (6) The test environment is modular indesign, lending itself to work partitioning in the test system development and on-going support phase. (7) During the development phase, once test procedure formats have been specified test procedure development can begin even though the test system is incomplete. The system is particularly useful in supporting software testing when no hardware laboratory facilities are available. The disadvantages notes are: (1) a major commitment has to be made for the test environment development, using skilled software personnel and (2) the development has to be carefully planned in order to attain timely delivery of the test results.

CONCLUSION

In conclusion the test system described meets the requirements of Table 1 satisfactorily. Consideration of its use in the future may be given in cases where precise and repeatable measurement of software response on a cycle-by-cycle basis is required, in situations where hardware is unavailable or where exact specification of signal acquisition is unimportant but where software signal processing and effect may be of interest. Clearly, the initial development cost consideration will be a major factor until off-the-shelf systems of this type are available and lend themselves to tailoring to individual needs. A business opportunity may exist for the entrepreneur.

BIOGRAPHY

David A. Rodgers

David A. Rodgers has worked as a computer software engineer for 18 years. His employers have included General Dynamics Corp., Infodata Corp., Xerox Corp., and most recently the Boeing Company. He has been responsible for the design, implementation and verification of real-time mini- and microcomputer systems for in-flight avionic system support (both commercial and military), commercial communications, and multi-program/multi-user order entry turnkey systems. He has 30 years' engineering experience, including work in England and Canada. His degree is in electrical engineering (UK).

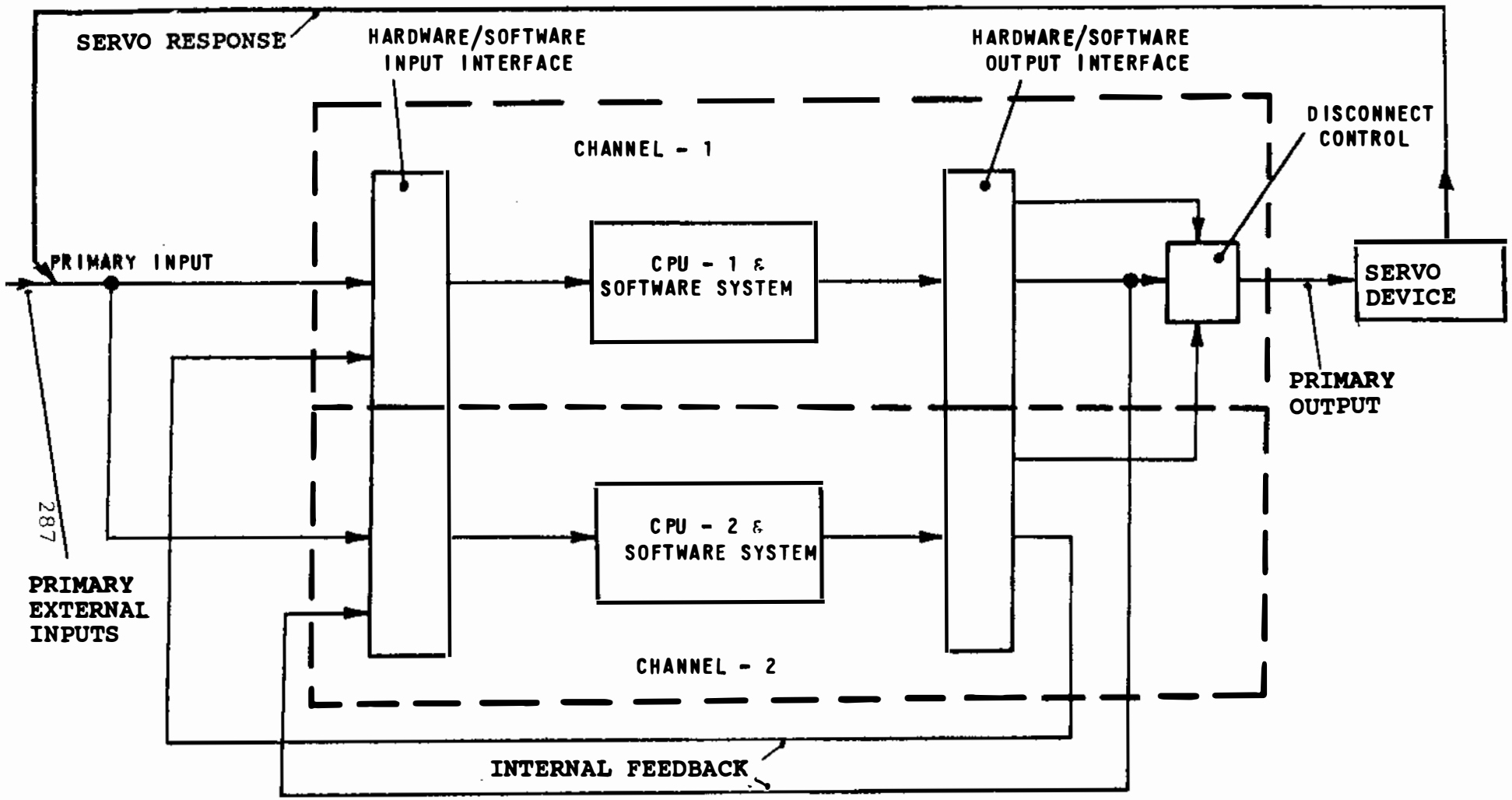


FIG 1 DUAL-DISSIMILAR SYSTEM

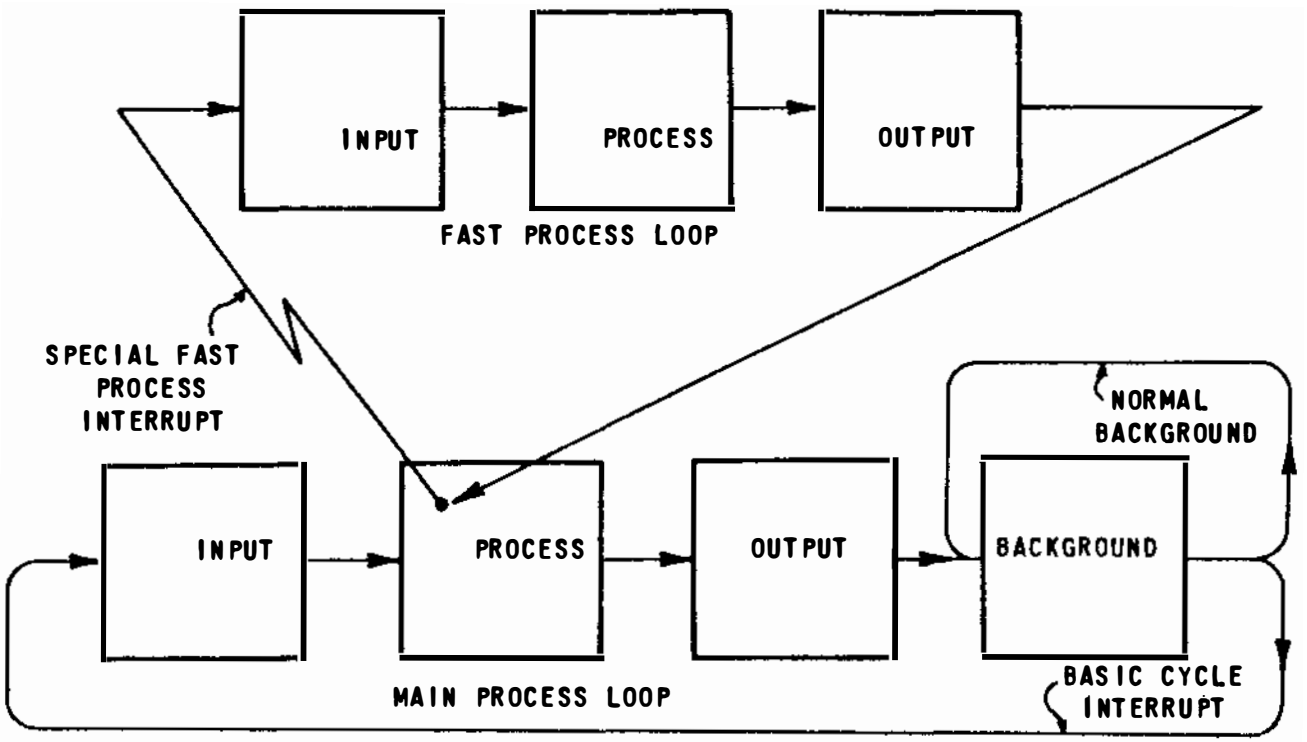


FIG 2 - SINGLE CPU SOFTWARE ARCHITECTURE

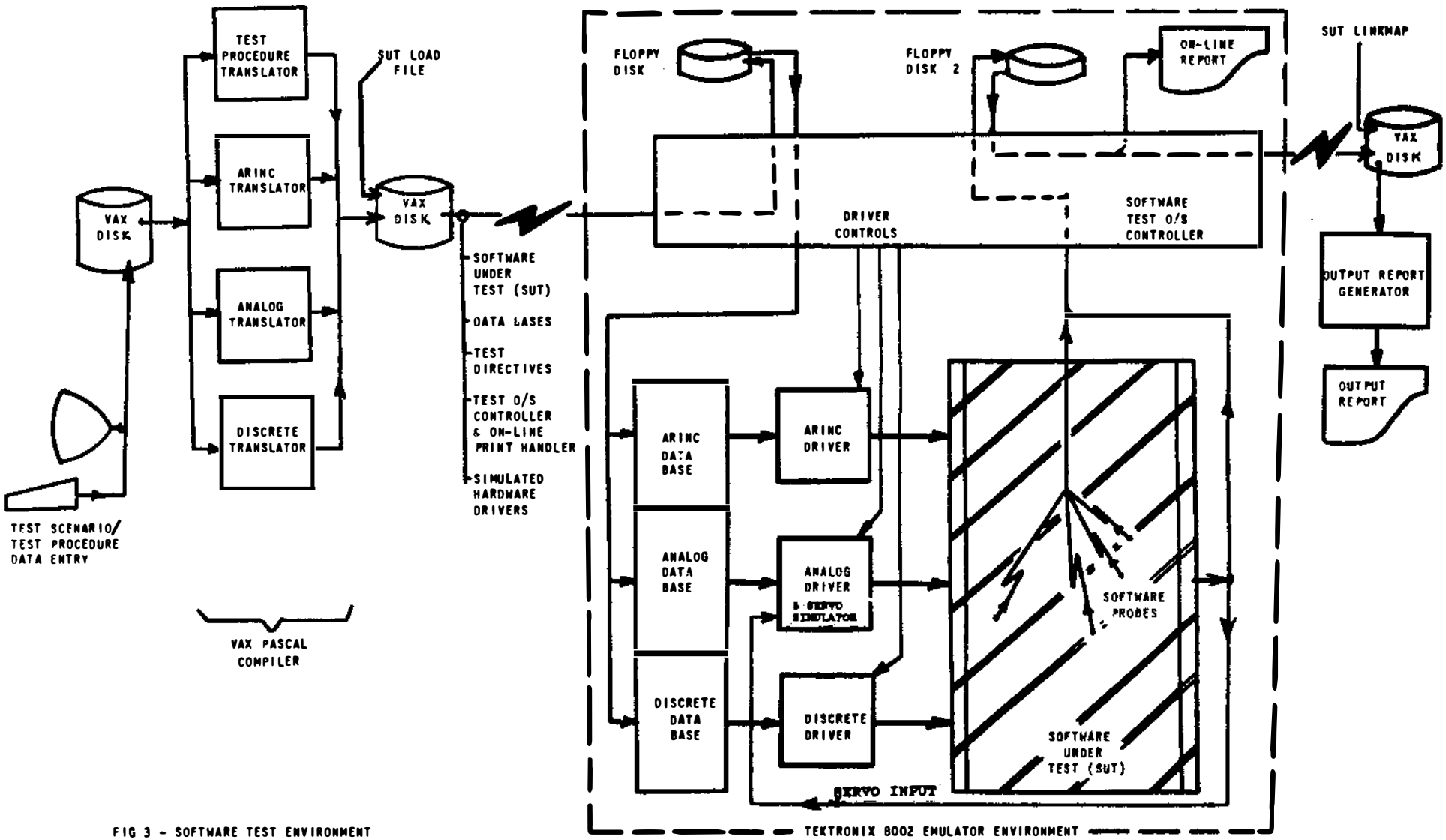


FIG 3 - SOFTWARE TEST ENVIRONMENT

FIG 4 - EXAMPLE OF TEST SCENARIO PROCEDURE

```
**** -----  BEGINNING OF TEST PROCEDURE -----
* TEST NUMBER:  XG6ØØ1DØØ
* 1Ø TEST OBJECTIVE:
*    THIS TEST WILL VERIFY THE TRIM MODE PRIORITY LOGIC
* 2Ø TEST APPROACH RATIONALE"
*    THIS TEST WILL CHECK THE TRIM MODE PRIORITY BY
*    SELECTING THE AUTO TRIM, MANUAL TRIM AND MACH/
*    SPEED TRIM MODE AND DEMONSTRATE THAT AUTO TRIM
*    SHALL OVERRIDE MANUAL TRIM.
* 21 TEST RESULTS/SUCCESS CRITERIA:
* RSC# ITER# ARMODE SELFCC AUTENG CAUTVD LAUTVD AUTTUA AUTTDA CONMOD MANMOD
*  1    41    Ø6     ØØ
*  2    42                                                      Ø6
*  3    47                                                 ØØ
*  4    5Ø                                                 Ø1
```

```
* 29   160      Ø4          Ø1                              Ø1            ØØ
* 3Ø   161                                                       Ø4
SYSTEM = SAMARM;                        * Name of system.
FNAME:   (XG6ØØ1DØØ.TPK);      * Name of procedure file.
LINKMAP: (DRCØ:[KAT.CPVCM.TOOLS]ARM1Ø1483.MTP);    * Name of linkmap.
*
IFILES:  ARINC = (XG6ØØ1DØØ).ACK    * Input scenario files.
           AID = (XG6ØØ1DØØ).IDK
        ANALOG = (XG6ØØ1DØØ).ALK
OFILES:   OUT1=(XG6ØØ1DØØ.ACT);     * Actual results file.
INTERMX=54Ø:                        * Number of iterations, this test.
* 3Ø    'SELVAR', 'SELMOD' AND 'NOMFIT' STATEMENT TO HERE.
SELMOD:   TRIM:                     * Select module "TRIM".
SELVAR:   ARMODE,SELFCC,AUTENG,CAUTVD,LAUTVD,AUTTUA,AUTTDA,
          CONMOD,MANMOD,VLDTDN,VLDTUP;            *   Measure   these
          variables.
NOMFIT:   1-39, 181-219,361-399;  * No measurements for iterations specified.
**** ----- END OF TEST PROCEDURE -----

**** ----- BEGINNING OF ARINC SCENARIO -----
FCC,FCCC,(TDA=Ø,TDC=Ø,AELS=Ø,AERS=Ø,TUA=Ø,TUC=Ø,GRO=Ø,UNSCHD+Ø),1:
FCC,FCCLR,(TDA=Ø,TDC=Ø,AELS=Ø,AERS=Ø,TUA=Ø,TUC=Ø,GRO=Ø,UNSCHD=Ø),1;
FCC,FCCLR,(TDA=1,TDC=1,AELS=1,AERS=1),41;

MC,DADCP,34Ø,1;       * Example:  Set ARINC signal MC on
MC,DADCS,34Ø,1;       * DADC Primary and Secondary channels to 340 millimach.
VC,DADCP,14Ø,1;
VC,DADCS,14Ø,1;       * Set ARINC signal AIRSPEED to 140 knots.
*
END;

**** ----- END OF ARINC SCENARIO -----
```

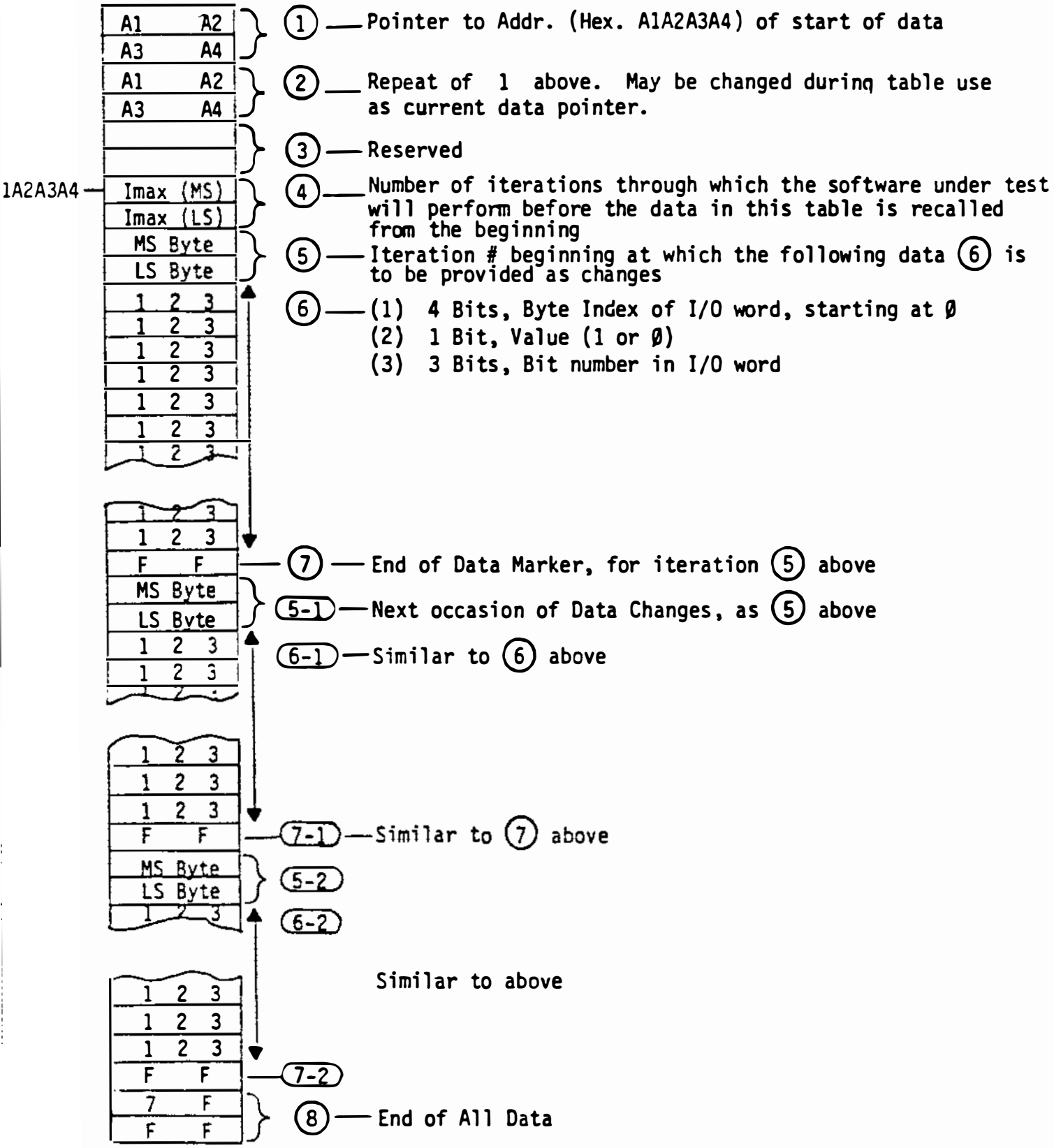
```
**** ----- BEGINNING OF AID SCENARIO -----


* SET TYPE CODE FOR 747-200
*
APL TYPE CODE - 0,1,1-180;            * Airplane type code.
APL TYPE CODE - 1,1,1-180
APL TYPE CODE - 2,0,1-180
APL TYPE CODE - P,1,1-180;
*
APL ON GROUND - 2,IN AIR,1-41;        * Put airplane in air.
APL ON GROUND - 1&2, IN AIR,1-41;
APL ON GROUND - 1, IN AIR,1-41;
*
VALID MANUAL COMD, MAN CMD, 80;       * Start manual trim.
*
* MANUAL TRIM DOWN
*
TRIM DOWN ARM CMD,TRIM DN,80-83;      * Exercise TRIM command.
TRIM DOWN CONT COMD,TRIM DN,80-83,
TRIM UP ARM CMD,NO TRIM UP,80-83,
TRIM UP CONT CMD,NO TRIM UP,80-83;
*
```

```
AUTOTRIM ARM-C,DISARMED,361;          * Exercise Autotrim.
AUTOTRIM ARM-C,ARMED,420;
AUTOTRIM ARM-C,DISARMED,440
AUTOTRIM ARM-C,ARMED,470;
AUTOTRIM ARM-C,DISARMED,490;
*
END;
**** ----- END OF AID SCENARIO -----


**** ----- BEGINNING OF ANALOG SCENARIO -----
*    THIS SCENARIO PROVIDES THE ANALOG FEEDBACK OF
*    RUDDER RATIO CHANGER FOR COINCIDENCE MONITORING
*         OF CONTROL AND ARMS CHANNELS.  STABILIZER
*    POSITION IS SET AT 0.0 DEGREES & PROGRAMMED IN
*    'SIMULATE' MODE TO PROVIDE THE DYNAMIC ANALOG
*    FEEDBACK OF THE STABILIZER HYDRAULIC MOTOR.
STABPO=SIM:IPOS=0,MRATE=0.2,IRATE=0.08,ITRIM=NO-TRIM,ISTATE=1,1;
PROVCA=TRACK-ON,1;
PROVCC=TRACK-ON,1;
END;
**** ----- END OF ANALOG SCENARIO -----
```

FIG 5 ANALOG INPUT DISCRETE DATA FILE FORMAT



```
;       CHANNEL:SAMARM                  VERSION:DRCØ:[KAT.KATI1Ø]ARM1Ø1483.TEK;1
;       TEST O/S: V1Ø  Ø8 NOV 83        TEST CASE: XG6001DDO          PLACE OF TEST:EMDC2
;                                                                LINKMAP:DRCØ:[KAT.CPVCM.TOOLS]ARM1Ø
ITERØØØ1-- ----------------------------------------------------------------------------
 I/O1N  Ø1
  OLD
  NEW


ITERØØ4Ø----------------------------------------------------------------------------
 TRIM   Ø1ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       ØØ        ØØ        ØØ        ØØ        ØØ        ØØ        ØØ        ØØ        ØØ
  NEW       Ø3        ØØ        Ø1        ØØ        ØØ        Ø1        Ø1        Ø3        Ø1


 I/OOUT Ø1OPRTØØ   OPRTØ1    OPRTØ2    OPRTØ3    OPRTØ4    OPRTØ5    OPRTØ6    $STABP    $STAB
  OLD
  NEW       AE        74        68        DA        25        Ø3        66        E8        4Ø


ITERØØ41----------------------------------------------------------------------------
 TRIM   Ø1ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       Ø3        ØØ        Ø1        ØØ        ØØ        Ø1        Ø1        Ø3        Ø1
  NEW       Ø6        ØØ        ØØ        ØØ        ØØ        Ø1        Ø1        Ø3        Ø1


 I/OOUT Ø1OPRTØØ   OPRTØ1    OPRTØ2    OPRTØ3    OPRTØ4    OPRTØ5    OPRTØ6    $STABP    $STABP+1
  OLD
  NEW       AE        74        68        DA        Ø5        Ø3        9C        E8        4Ø


ITERØØ42----------------------------------------------------------------------------
 TRIM   Ø1ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       Ø6        ØØ        ØØ        ØØ        ØØ        Ø1        Ø1        Ø3        Ø1
  NEW       Ø6        ØØ        ØØ        ØØ        ØØ        Ø1        Ø1        Ø6        Ø1


 I/OOUT Ø1OPRTØØ   OPRTØ1    OPRTØ2    OPRTØ3    OPRTØ4    OPRTØ5    OPRTØ6    $STABP    $STAB
  OLD
              AE        74        68        DA
```

FIG 6 - FORMATTED OUTPUT REPORT

SYSTEM: SAM ARM TEST NUMBER: XG6001D00 TIME: 05:21PM DATE: 11 01 84

ITER	SLRDY	SLOUT	TXINH	PGMVC	MEMLD	MTRS	AASEL	ATL/R	ATC	ATEN	TRIM	AILOV	AILOC	INSEL	WDOG	MODE	UNSCT	SAMF	PROGRAMMED VC (UNITS = INCHES) -1 -.5 0 .5 1	STAB POSTN
40	1	1	1	.	.	1	2	.	.	1	.	.	1	3	6	3			+ + X + AE4 + +	E84
41	1	1	1	.	.	1	2	.	.	.	.	.	1	3	9	6			+ + X + AE4 + +	E84
42	1	1	1	.	.	1	2	.	.	.	.	.	1	3	9	6			+ + X + AE4 + +	E84
43	1	1	1	.	.	1	2	.	.	.	.	.	1	3	9	6			+ + X + AE4 + +	E84
44	1	1	1	.	.	1	2	.	.	.	.	.	1	3	9	6			+ + X + AE4 + +	E84
45	1	1	1	.	.	1	2	.	.	.	.	.	1	3	C	6			+ + X + AE4 + +	E84
46	1	1	1	.	.	1	2	.	.	.	.	.	1	3	C	6			+ + X + AE4 + +	E84
47	1	1	1	.	.	1	2	.	.	.	D	.	1	3	C	6			+ + X + AE4 + +	E84
48	1	1	1	.	.	1	2	.	.	.	D	.	1	3	C	6			+ + X + AE4 + +	E8[illegible]
49	1	1	1	.	.	1	2	.	.	.	D	.	1	3	3	6			+ + X + AE4 + +	E8F
50	1	1	1	.	.	1	2	.	.	.	.	.	1	3	3	6			+ + X + AE4 + +	E95
51	1	1	1	.	.	1	2	.	.	.	.	.	1	3	3	6			+ + X + AE4 + +	E96
52	1	1	1	.	.	1	2	.	.	.	.	.	1	3	3	6			+ + X + AE4 + +	E96
53	1	1	1	.	.	1	2	.	.	.	.	.	1	3	6	6			+ + X + AE4 + +	E96
54	1	1	1	.	.	1	2	.	.	.	.	.	1	3	6	6			+ + X + AE4 + +	E96
55	1	1	1	.	.	1	2	.	.	.	.	.	1	3	6	6			+ + X + AE4 + +	E96
56	1	1	1	.	.	1	2	.	.	.	U	.	1	3	6	6			+ + X + AE4 + +	E96
57	1	1	1	.	.	1	2	.	.	.	U	.	1	3	9	6			+ + X + AE4 + +	E91
58	1	1	1	.	.	1	2	.	.	.	U	.	1	3	9	6			+ + X + AE4 + +	E8B
59	1	1	1	.	.	1	2	.	.	.	U	.	1	3	9	6			+ + X + AE4 + +	E85
60	1	1	1	.	.	1	2	.	.	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]			[illegible] X + AE4 + +	E7F

FIG 6A ON-LINE OUTPUT REPORT

A SOFTWARE TEST ENVIRONMENT

FOR EMBEDDED SOFTWARE

SEPTEMBER 27 1985

presented by:
David A. Rodgers and Ralph Gable

Boeing
Commercial Airplane Division
M/S 77-21 P.O. BOX 3707
Seattle, Washington 98124-2207

WHAT IS TO BE PRESENTED

- **A SOFTWARE TEST ENVIRONMENT FOR DUAL DISSIMILAR SOFTWARE**
  - **THE PROBLEM**
  - **THE SOFTWARE UNDER TEST**
  - **THE SOLUTION CHOSED**
  - **A TEST ENVIRONMENT OVERVIEW**
    - **INPUT SOURCE PROCEDURES**
    - **THE DATA BASES**
    - **OPERATIONAL SUPPORT SOFTWARE**
  - **OPERATIONAL EXPERIENCE**
  - **CONCLUSIONS**

THE PROBLEM

- **EMBEDDED SOFTWARE**
- **DUAL – DISSIMILAR**
- **PROBLEMS OF A PURE HARDWARE ENVIRONMENT**
- **SIMULATION vs EMULATION**
- **TEST SYSTEM REQUIREMENTS**

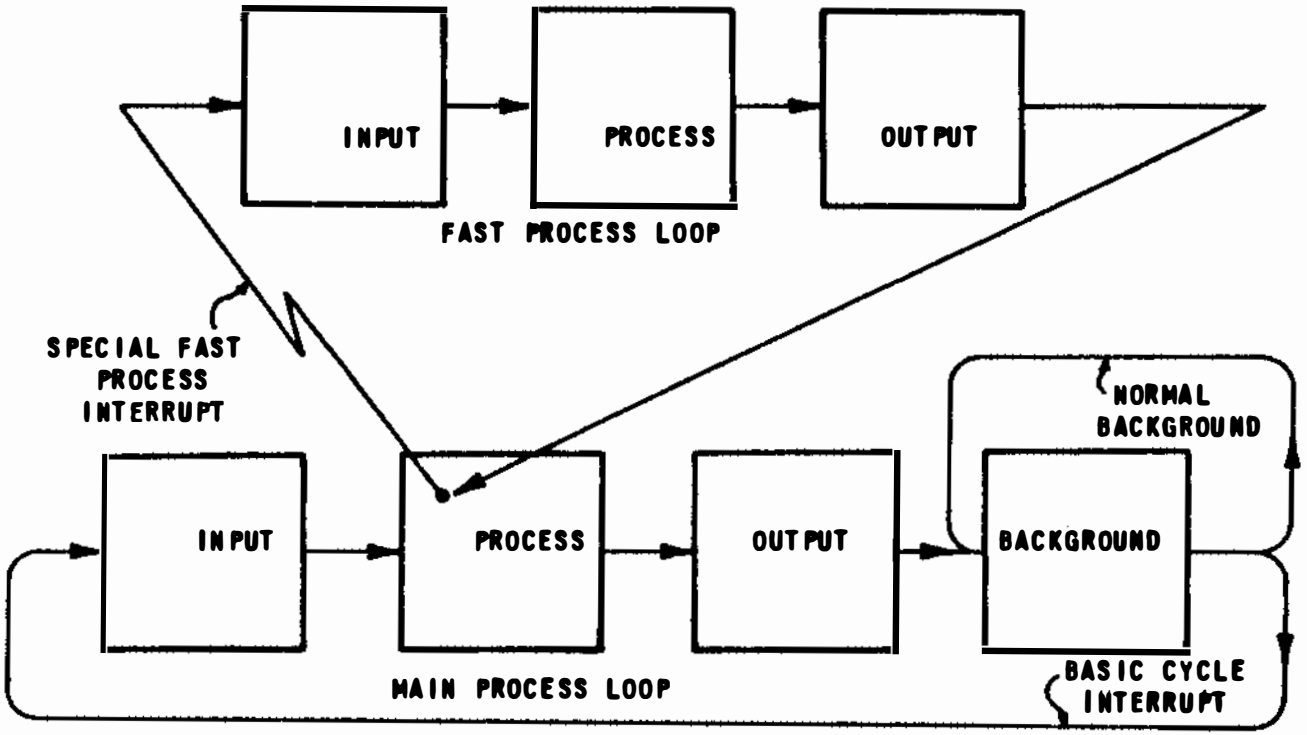


FIG 2 - SINGLE CPU SOFTWARE ARCHITECTURE

INPUT, EACH SUT

- INPUTS (AT HARDWARE / SOFTWARE INTERFACE)
  - 5 ARINC CHANNEL x5 LABELS / CHANNEL = 25 ARINC SIGNALS
    - PARITY
    - STATUS MATRIX
    - DATA
  - 5 ANALOG CHANNELS
    - 12 BITS / CHANNEL
  - 60 DISCRETES

INCLUDES AUTOMATIC FEEDBACK –

- CROSS-CHANNEL FEEDBACK, DISSIMMILAR CHANNEL
- ACTUATOR FEEDBACK

ALL INPUTS ARE TO BE SPECIFIABLE / OVERRIDEABLE BY INPUT PROCEDURE STATEMENTS, IF NECESSARY, AT EACH AND EVERY ITERATION.

OUTPUT, EACH SUT

- OUTPUTS
  - AT HARDWARE / SOFTWARE INTERFACE
    - 1 ARINC CHANNEL
    - 2 ANALOG CHANNELS
    - 30 DISCRETES
  - WITHIN SUT
    - 300 VARIABLES
    - 100 MODULES

300

ALL OUTPUTS ARE TO BE MEASUREABLE AS SPECIFIED BY INPUT PROCEDURE STATEMENTS, IF NECESSARY, AT EACH AND EVERY ITERATION.

TEST REQUIREMENTS

- **PRIMARY OBJECTIVE**
  - **TO GENERATE A TEST REPORT THAT**
    1. **DEMONSTRATED THAT THE TOTAL SOFTWARE HAD BEEN VERIFIED WITH "WHITE" BOX CONSIDERATIONS AGAINST SYSTEM REQUIREMENTS**
    2. **WOULD WITHSTAND AUDIT**
- **IN ADDITION**
  - **EASY TO GENERATE TESTS & TO OPERATE TEST RIG**
    - **SYSTEM ORIENTED TEST WRITERS NOT SKILLED IN SOFTWARE TECHNIQUES**
  - **EASY TO REVIEW & UNDERSTAND RESULTS**
  - **REPEATABLE RESULTS, MAYBE YEARS LATER**
  - **EASY TO MODIFY DURING PRODUCT LIFE-CYCLE**
  - **SUPPORT RELATIVELY LARGE TEST VOLUME & SUT VERSIONS**

PROBLEMS OF A HARDWARE TEST ENVIRONMENT

- CONTROL OF EXACTLY WHAT SCENARIO YOU WANT
- REPEATABILITY

} CAPTURE OF QUANTITY OF DATA IN SAME EVENT

- CONFIGURATION CONTROL

THE SOLUTION CHOSEN

- TOTAL SOFTWARE ENVIRONMENT
- ONE CHANNEL ONLY
  - DISSIMILAR CHANNEL BECOMES A "PHANTOM" CHANNEL
  - USE SUT's OWN OUTPUT TO GENERATE DISSIMILAR CHANNEL's SIGNALS
- VAX PLUS TEKTRONIX 8002 EMULATOR, LINKED BY COMMUNICATION LINE
- ENGLISH LANGUAGE INPUT
- PASCAL TRANSLATORS WITH ERROR CHECKING IN OFF-LINE MODE
- DATA BASE PLUS DRIVERS
- TEST OPERATING SYSTEM (TEST O/S)
  - TEKTRONIX JCL
  - TEST O/S CONTROLLER
- LOCAL DATA STORAGE WITH ON-LINE CONTINUOUS OUTPUT PRINT
- REMOTE OUTPUT REPORT GENERATION IN OFF-LINE MODE
- VAX CONFIGURATION MANAGEMENT & SUPPORT
- RESOURCE CONSIDERATIONS
  (STORAGE, RUNTIME)

SIMULATION vs EMULATION

- **2 FACTORS IN THIS CASE**
  - **CERTIFICATION AUTHORITIES REQUIRED USE OF A REAL CPU RATHER THAN A SIMULATED CPU**
  - **TIME TO EXECUTE FULL-UP SOFTWARE IS IN SIMULATOR ENVIRONMENT MUCH LONGER THAN IN EMULATOR ENVIRONMENT**

EMULATOR WAS CHOSEN

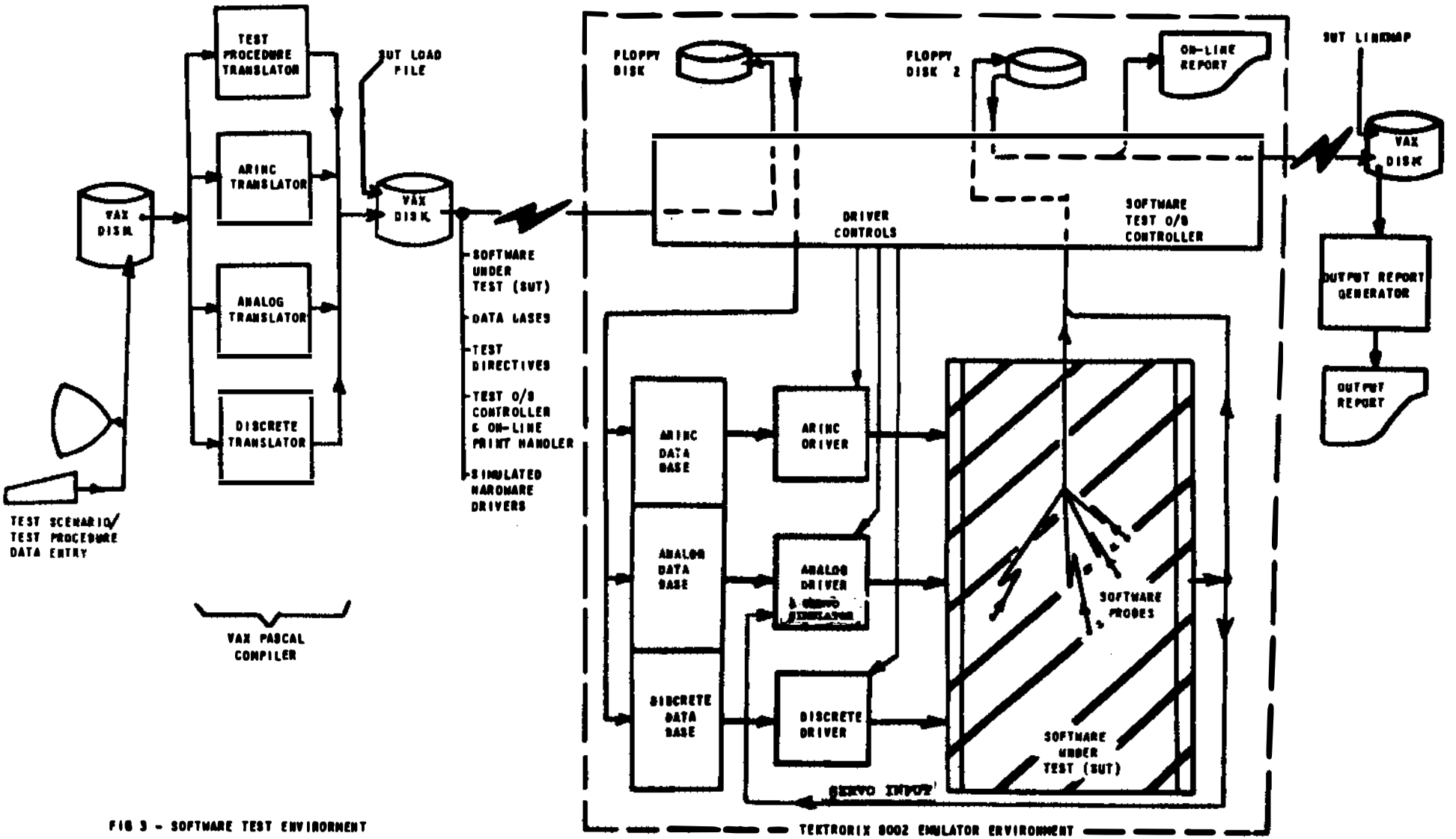


FIG 3 - SOFTWARE TEST ENVIRONMENT

TEST ENVIRONMENT OVERVIEW

- INPUT PROCEDURES
  - TEST CONTROL
  - SENARIO DESCRIPTION
    - ARINC
    - ANALOG
    - DISCRETE
- TRANSLATORS
- DATA BASES
- HARDWARE DRIVERS
- TEST O/S
- REPORT GENERATION
  - ON-LINE
  - OFF-LINE

WARNING

WARNING

THE PROPOSED SOLUTION

- ***SUT WILL NOT FULLY BEHAVE, IN SOME DETAILS, AS THOUGH REAL HARDWARE WERE ATTACHED (RELEGATE TO OTHER TEST PHASES)***
- ***WILL NOT RUN IN REAL TIME***
- ***USES A SMALL AMOUNT OF CODE CORRUPTION***
- ***ONLY APPROXIMATES INTERRUPTS***

HOWEVER

- **LOGICAL PERFORMANCE OF SOFTWARE WILL BE DEMONSTRATED**
- **THE ABOVE LIMITATIONS CAN BE RATIONALIZED**

INPUT PROCEDURES

- .SELF DOCUMENTING
- USE OF SYSTEM TERMINOLOGY AND ENGINEERING UNITS
- SYSTEM ENGINEER FRIENDLY

```
**** ----- BEGINNING OF AID SCENARIO -----


* SET TYPE CODE FOR 747-200
*
APL TYPE CODE - 0,1,1-180;              * Airplane type code.
APL TYPE CODE - 1,1,1-180
APL TYPE CODE - 2,0,1-180
APL TYPE CODE - P,1,1-180;
*
APL ON GROUND - 2,IN AIR,1-41;          * Put airplane in air.
APL ON GROUND - 1&2, IN AIR,1-41;
APL ON GROUND - 1, IN AIR,1-41;
*
VALID MANUAL COMD, MAN CMD, 80;         * Start manual trim.
*
* MANUAL TRIM DOWN
*
TRIM DOWN ARM CMD,TRIM DN,80-83;        * Exercise TRIM command.
TRIM DOWN CONT COMD,TRIM DN,80-83,
TRIM UP ARM CMD,NO TRIM UP,80-83,
TRIM UP CONT CMD,NO TRIM UP,80-83;
*
```

```
AUTOTRIM ARM-C,DISARMED,361;            * Exercise Autotrim.
AUTOTRIM ARM-C,ARMED,420;
AUTOTRIM ARM-C,DISARMED,440
AUTOTRIM ARM-C,ARMED,470;
AUTOTRIM ARM-C,DISARMED,490;
*
END;
**** ----- END OF AID SCENARIO -----


**** ----- BEGINNING OF ANALOG SCENARIO -----
*    THIS SCENARIO PROVIDES THE ANALOG FEEDBACK OF
*    RUDDER RATIO CHANGER FOR COINCIDENCE MONITORING
*         OF CONTROL AND ARMS CHANNELS.  STABILIZER
*    POSITION IS SET AT 0.0 DEGREES & PROGRAMMED IN
*    'SIMULATE' MODE TO PROVIDE THE DYNAMIC ANALOG
*    FEEDBACK OF THE STABILIZER HYDRAULIC MOTOR.
STABPO=SIM:IPOS=0,MRATE=0.2,IRATE=0.08,ITRIM=NO-TRIM,ISTATE=1,1;
PROVCA=TRACK-ON,1;
PROVCC=TRACK-ON,1;
END;
**** ----- END OF ANALOG SCENARIO -----
```

FIG 4 - EXAMPLE OF TEST SCENARIO PROCEDURE

```
**** -----  BEGINNING OF TEST PROCEDURE -----
* TEST NUMBER:  XG6ØØ1DØØ
* 1Ø TEST OBJECTIVE:
*    THIS TEST WILL VERIFY THE TRIM MODE PRIORITY LOGIC
* 2Ø TEST APPROACH RATIONALE"
*    THIS TEST WILL CHECK THE TRIM MODE PRIORITY BY
*    SELECTING THE AUTO TRIM, MANUAL TRIM AND MACH/
*    SPEED TRIM MODE AND DEMONSTRATE THAT AUTO TRIM
*    SHALL OVERRIDE MANUAL TRIM.
* 21 TEST RESULTS/SUCCESS CRITERIA:
* RSC# ITER# ARMODE SELFCC AUTENG CAUTVD LAUTVD AUTTUA AU TDA CONMOD MANMOD
*  1    41    Ø6          ØØ
*  2    42                                                      Ø6
*  3    47                                                ØØ
*  4    5Ø                                                Ø1
```

```
* 29   160      Ø4          Ø1                              Ø1            ØØ
* 3Ø   161                                                       Ø4
SYSTEM = SAMARM;                        * Name of system.
FNAME:   (XG6ØØ1DØØ.TPK);      * Name of procedure file.
LINKMAP: (DRCØ:[KAT.CPVCM.TOOLS]ARM1Ø1483.MTP);    * Name of linkmap.
*
IFILES:  ARINC = (XG6ØØ1DØØ).ACK    * Input scenario files.
           AID = (XG6ØØ1DØØ).IDK
        ANALOG = (XG6ØØ1DØØ).ALK
OFILES:   OUT1=(XG6ØØ1DØØ.ACT);     * Actual results file.
INTERMX=54Ø:                        * Number of iterations, this test.
* 3Ø    'SELVAR', 'SELMOD' AND 'NOMFIT' STATEMENT TO HERE.
SELMOD:   TRIM:                     * Select module "TRIM".
SELVAR:   ARMODE,SELFCC,AUTENG,CAUTVD,LAUTVD,AUTTUA,AUTTDA,
          CONMOD,MANMOD,VLDTDN,VLDTUP;            *  Measure  these
          variables.
NOMFIT:   1-39, 181-219,361-399;  * No measurements for iterations specified.
**** ----- END OF TEST PROCEDURE -----

**** ----- BEGINNING OF ARINC SCENARIO -----
FCC,FCCC,(TDA=Ø,TDC=Ø,AELS=Ø,AERS=Ø,TUA=Ø,TUC=Ø,GRO=Ø,UNSCHD+Ø),1:
FCC,FCCLR,(TDA=Ø,TDC=Ø,AELS=Ø,AERS=Ø,TUA=Ø,TUC=Ø,GRO=Ø,UNSCHD=Ø),1;
FCC,FCCLR,(TDA=1,TDC=1,AELS=1,AERS=1),41;

MC,DADCP,34Ø,1;       * Example:  Set ARINC signal MC on
MC,DADCS,34Ø,1;       * DADC Primary and Secondary channels to 340 millimach.
VC,DADCP,14Ø,1;
VC,DADCS,14Ø,1;       * Set ARINC signal AIRSPEED to 140 knots.
*
END;

**** ----- END OF ARINC SCENARIO -----
```

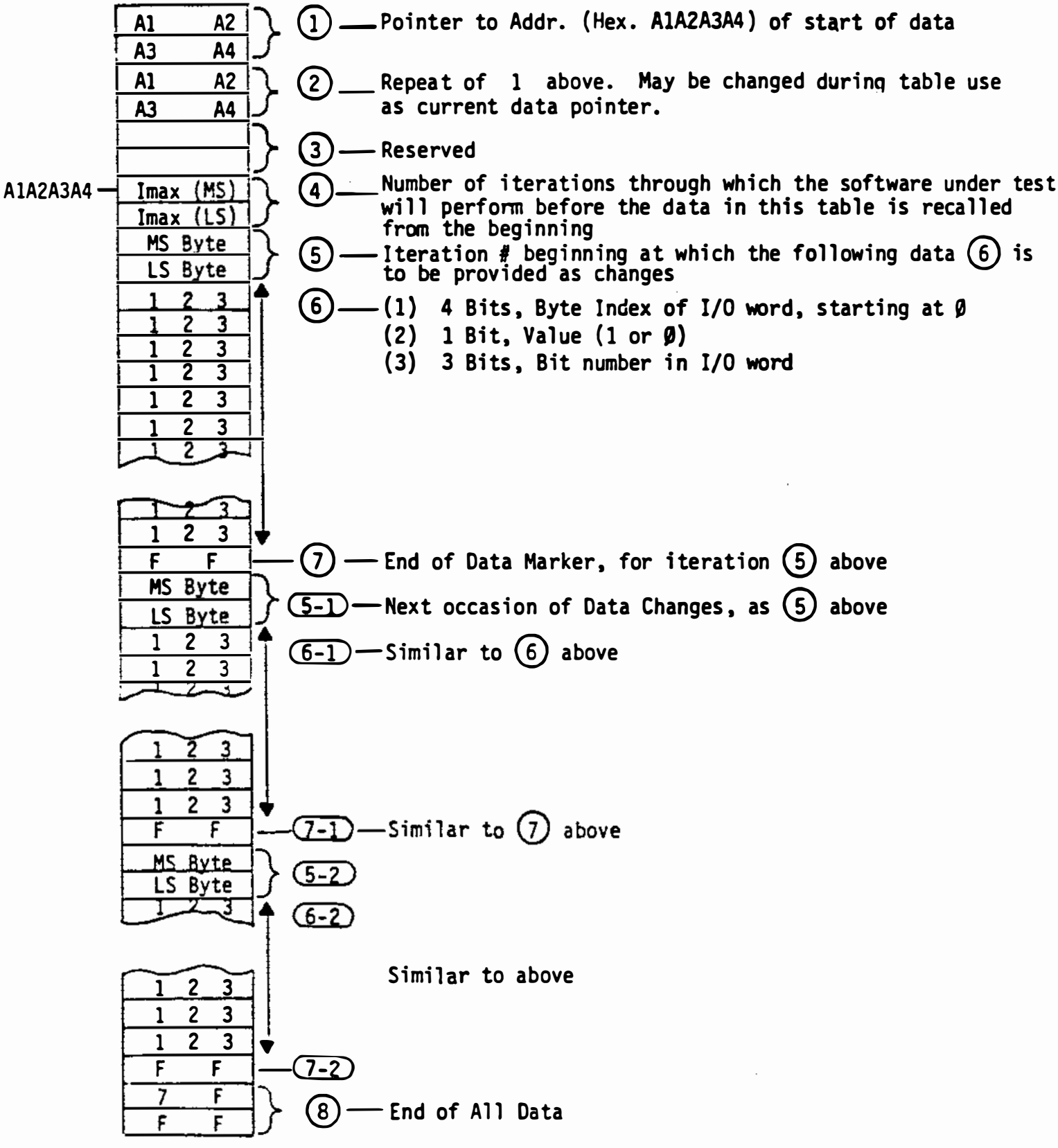
TRANSLATORS

- **OFF-LINE GENERATED IN VAX**
- **MODULAR**
- **ERROR CHECKING**

DATA BASES

- **TRANSITIONAL BASIS**
- **PACKED FORMAT**
- **CYCLICAL OPTION**

FIG 5 ANALOG INPUT DISCRETE DATA FILE FORMAT

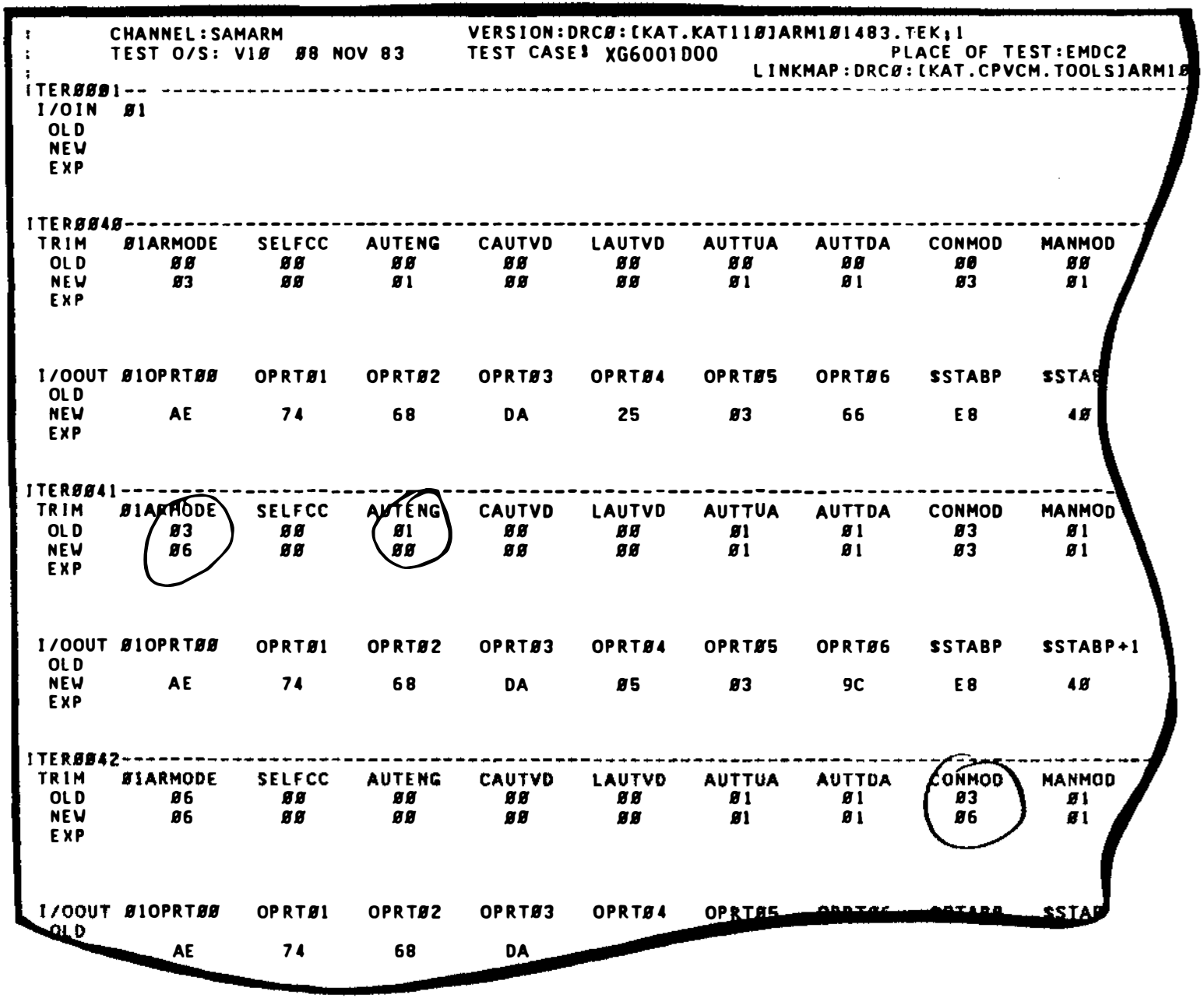


HARDWARE DRIVERS

- MODULAR
- SIMULATE ACTUAL HARDWARE
- OUTPUT
    - DATA BASE DRIVEN
    - OUTPUT DEPENDENT
- HOOKS
    - CODE CORRUPTION
- DESIGN
    - CALLED BY TEST O/S
    - CALLED BY SUT
    - ASSEMBLY LANGUAGE
        - CPU DEPENDENT
        - SUT DEPENDENT

TEST O/S

- BASIC FUNCTIONS
  - PREPARE RAM AREAS
  - LOAD
    - SUT
    - DRIVERS
    - D/B's
  - CHECK PROPER LOAD
  - PERFORM & CONTROL TEST
    - CYCLE COUNT & ON-OFF MEASUREMENT SYSTEM
    - COLLECT DATA
      - KNOWLEDGE OF MODULE BEING EXERCISED
      - VARIABLES TO BE MEASURED
    - CONTROL ON-LINE PRINTER
  - UPLOAD RESULTS



```
;       CHANNEL:SAMARM                  VERSION:DRC0:[KAT.KAT110]ARM101483.TEK;1
;       TEST O/S: V10  08 NOV 83        TEST CASE: XG6001D00            PLACE OF TEST:EMDC2
;                                                                LINKMAP:DRC0:[KAT.CPVCM.TOOLS]ARM10
ITER0001-- ------------------------------------------------------------------------------
 I/OIN  01
  OLD
  NEW
  EXP


ITER0040-------------------------------------------------------------------------------
 TRIM    01ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       00        00        00        00        00        00        00        00        00
  NEW       03        00        01        00        00        01        01        03        01
  EXP


 I/OOUT 01OPRT00   OPRT01    OPRT02    OPRT03    OPRT04    OPRT05    OPRT06    $STABP    $STAB
  OLD
  NEW       AE        74        68        DA        25        03        66        E8        40
  EXP


ITER0041-------------------------------------------------------------------------------
 TRIM    01ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       03        00        01        00        00        01        01        03        01
  NEW       06        00        00        00        00        01        01        03        01
  EXP


 I/OOUT 01OPRT00   OPRT01    OPRT02    OPRT03    OPRT04    OPRT05    OPRT06    $STABP    $STABP+1
  OLD
  NEW       AE        74        68        DA        05        03        9C        E8        40
  EXP


ITER0042-------------------------------------------------------------------------------
 TRIM    01ARMODE    SELFCC    AUTENG    CAUTVD    LAUTVD    AUTTUA    AUTTDA    CONMOD    MANMOD
  OLD       06        00        00        00        00        01        01        03        01
  NEW       06        00        00        00        00        01        01        06        01
  EXP


 I/OOUT 01OPRT00   OPRT01    OPRT02    OPRT03    OPRT04    OPRT05    OPRT06    $STABP    $STAB
  OLD
            AE        74        68        DA
```

```
SYSTEM: SAM ARM        TEST NUMBER:   XG6001D00          TIME:  05:21PM     DATE: 11 01 84

      S S T P M M A A A A T A A I W M U S - - - - - - - - - - - - - - - - - - - - - - S P
      L L X G E T A T T T R I I N D O N A       PROGRAMMED VC (UNITS = INCHES)        T O
      R O I M M   S L       I L L S O D S M - - - - - - - - - - - - - - - - - - - - - A S
ITER  D U N V L R E / C E M O O E G E C     -1        -.5        0         .5        1 B T
      Y T H C D S L R   N   V C L       T F  . . . . . . . . . . . . . . . . . . . . .   N
                                             !          !          !          !      !

 40   1 1 1 . . 1 2 . . 1 . . 1 3 6 3        +          +   X      +    AE4   +      + E84
 41   1 1 1 . . 1 2 . . . . . 1 3 9 6        +          +   X      +    AE4   +      + E84
 42   1 1 1 . . 1 2 . . . . . 1 3 9 6        +          +   X      +    AE4   +      + E84
 43   1 1 1 . . 1 2 . . . . . 1 3 9 6        +          +   X      +    AE4   +      + E84
 44   1 1 1 . . 1 2 . . . . . 1 3 9 6        +          +   X      +    AE4   +      + E84
 45   1 1 1 . . 1 2 . . . . . 1 3 C 6        +          +   X      +    AE4   +      + E84
 46   1 1 1 . . 1 2 . . . . . 1 3 C 6        +          +   X      +    AE4   +      + E94
 47   1 1 1 . . 1 2 . . . D . 1 3 C 6        +          +   X      +    AE4   +      + E84
 48   1 1 1 . . 1 2 . . . D . 1 3 C 6        +          +   X      +    AE4   +      + E87
 49   1 1 1 . . 1 2 . . . D . 1 3 3 6        +          +   X      +    AE4   +      + E8F
 50   1 1 1 . . 1 2 . . . . . 1 3 3 6        +          +   X      +    AE4   +      + E95
 51   1 1 1 . . 1 2 . . . . . 1 3 3 6        +          +   X      +    AE4   +      + E96
 52   1 1 1 . . 1 2 . . . . . 1 3 3 6        +          +   X      +    AE4   +      + E96
 53   1 1 1 . . 1 2 . . . . . 1 3 6 6        +          +   X      +    AE4   +      + E96
 54   1 1 1 . . 1 2 . . . . . 1 3 6 6        +          +   X      +    AE4   +      + E96
 55   1 1 1 . . 1 2 . . . . . 1 3 6 6        +          +   X      +    AE4   +      + E96
 56   1 1 1 . . 1 2 . . . U . 1 3 6 6        +          +   X      +    AE4   +      + E96
 57   1 1 1 . . 1 2 . . . U . 1 3 9 6        +          +   X      +    AE4   +      + E91
 58   1 1 1 . . 1 2 . . . U . 1 3 9 6        +          +   X      +    AE4   +      + E8B
 59   1 1 1 . . 1 2 . . . U . 1 3 9 6        +          +   X      +    AE4   +      + E85
 60   1 1 1 . . 1 2 . .                                              +    AE4   +      + E7F
```

FIG 6A ON-LINE OUTPUT REPORT

OPERATIONAL EXPERIENCE

- 2 DUAL – DISSIMILAR SYSTEMS
  - 4 INDIVIDUAL SUT's
    - 4 TRANSLATORS } PASCAL
    - 4 TEST O/S
    - 16 DRIVERS
    - 4 ON – LINE PRINT CONTROLLERS

    } ASSEMBLY LANGUAGE
  - 400 TESTS
  - 5 – 6 VERSIONS EACH SUT
    - 2000 SETS OF PROCEDURES
  - C – M SYSTEM
  - TEST S/R SYSTEM

ADVANTAGES AND DISADVANTAGES

ADVANTAGES FOUND

- LOW SOFTWARE SKILL, HIGH SYSTEM KNOWLEDGE SKILLS
- LOW ENGINEERING EXPERIENCE TO OPERATE TEST SET
- FAST TEST GENERATION & DE-BUG
- TEST PROCEDURES UNDERSTANDABLE PROJECT WIDE
- SELF-DOCUMENTING
- REPEATABLE
- MACHINE STORABLE
- TEST SYSTEM MODULAR IN DESIGN

DISADVANTAGES FOUND

- HIGH INITIAL INVESTMENT
- SPECIAL SKILLS TO DEVELOP & MAINTAIN TEST SUPPORT SOFTWARE

CONCLUSIONS

- **INITIAL REQUIREMENTS MET**

- **FUTURE USE**

CLUE
a program and test suite evaluation tool for C

Dr. David B. Benson

BENTEC
NE 615 Campus Street
Pullman, Washington 99163

Abstract

CLUE is a statement count profiler for C programs in Unix (tm AT&T) environments. Statement count profiles are used in debugging and evaluating software and determining the extent of code coverage by a test suite. CLUE instruments C language source in a manner which does not change the functionality of the software being evaluated. CLUE uses the C compiler available at the test site. This means the same compiler used for design and coding is used in the evaluation. CLUE is easy to use, requiring minimal reading before starting. The paper explains some of the uses of CLUE via an extended example, and gives a detailed evaluation of CLUE.

Profiling for Evaluation

Profiling serves an important role in software quality assurance. Typically profiling is done for timing measurements. However, counting the number of times lines, statements, or routines are invoked enables the evaluator to determine the adequacy of the tests performed and the extent to which the program is exercised by the test suite. The counts may be used to determine whether the tests exercise all of the code, which portions of the program are exercised at all, and whether the algorithms embodied in the code are performing as expected. Thus execution counts are used to evaluate the test suite and the program at the same time.

Execution count profilers may count routines, lines, or statements. The count of routine calls during test provide an overall coarse-grained view of program execution. Routine call counts are an important tool for the software designer and the software evaluator. The 'gprof' profiler available in Unix bsd 4.2 provides call counts together with other information. Other aspects of evaluation require a fine-grained view of program execution; the individual statements forming the grain size. Line counts are adequate for the study of small programs, but software engineering principles require the counts to be accumulated for each statement, even if several statements are placed on the same line of the source code. This requires reformatting the source code when producing the report. C language routines need not return in the Unix environment since the routine body may invoke exit(), _exit(), longjmp(), or may fail, transferring to a signal processing routine. Therefore the line of

code

```
x = foo(x);     y = foo(y);
```

in the source file needs to be displayed in the statement count report as

```
100     x = foo(x);
 99     y = foo(y);
```

and in this hypothetical case, some call to foo() failed to return. CLUE provides a reformatted report so that each executable statement appears on its own line with the count of the number of executions of that statement. The source appearing in the report is beautified, to maintain or enhance the readability of the original source text.

The statement count profile report provides the basis for a number of other reports useful to the evaluator. Code coverage is a basic measure of the adequacy of the test suite used to evaluate the software. CLUE provides a code coverage report by program, source file, and function. The evaluator is also interested in code which has either unusually large or unusually small execution counts. CLUE provides a filter enabling the evaluator to easily locate the statements reporting any percentage range of the total counts. These and other report types are discussed in the sequel.

The evaluator typically uses the code coverage report to determine test suite adequacy. If the test suite is inadequate, the report of code sections not executed by the test suite aids the evaluator in devising additional tests for the test suite. Of course, some code may not be executable by any test, in which case the designer or coder needs to be informed. Code with large statement counts is also to be viewed with suspicion.

If the software product is performing poorly, the algorithms may need changing. Occasionally, large statement counts are simply the result of poor or incorrect coding, even if the product meets the time performance specification.

In addition, there are various standards checks which are based on the statement counts. I give an example when discussing some evaluations based on CLUE.

Using CLUE

One first must produce the instrumented program from the C source files. Simply use 'procc' wherever 'cc' would ordinarily appear. For example, a makefile may contain the CC macro format. The CLUE user can simply change this line to

```
CC = procc
```

and then make the program in the usual way. If the makefile uses the .c.o dependency it suffices to modify this line to read

```
.c.o: procc <whatever cc arguments already appear.>
```

If 'cc' is used for loading, the above changes will suffice in most cases. If 'ld' is used to produce the executable image, one must change the occurrences of 'ld ...' to 'ld ... -lclue' to include the CLUE instrumentation runtime support. If the makefile uses the LD form it is best to change this line to

```
LD = procc
```

to avoid various confusions about the loader.

Occasionally one runs into problems with using a library containing 'main'. Since the CLUE runtime instrumentation support must be the first to gain control, the libclue.a library contains a definition of 'main'. Therefore it is necessary that -lclue appear before any other -l flags to the loader for libraries which contain a definition of 'main'. The command 'procc' places -lclue last, so that all the .o files made by procc have the rest of the runtime instrumentation linked in. These restrictions make it necessary to directly invoke 'cc' or 'ld' to obtain the executable images. This annoyance will be fixed in a later release of CLUE.

There are flags for procc so that only routines listed in the 'procc' command line are instrumented. Instrumented .o files may be linked with ordinary .o files in forming the resulting executable image. This feature results in smaller files, shorter reports, and faster execution times. The usual evaluation practice is to instrument all the routines, selecting the desired information from the resulting report.

Once the instrumented program has been made, there is a .i file in the making directory for every .c file used in the make. The .i files have all the preprocessor includes expanded, just in case there is any executable code in the include files. All C source in the include files will appear in the statement count report.

The instrumented program is now run on one or more test cases. The profiling information is accumulated in a file named 'profile.lc'. This file is highly condensed in order to save file write time. The information is appended to the file, so that summary data from a test suite is particularly easy to obtain. The CLUE user might wish to move the file after one series before beginning another. The report generator has facilities to cope with several files profiling information.

Finally, the CLUE user runs the report generator and filters. The report generation command 'prolc' takes the statement count information from the profile.lc file and the C source from the .i files to produce a report on the files in which any function has been executed during the test run or runs. If the information from several renamed profiling information files is desired, the command form is

```
prolc -db [file ...]
```

and the command acts as if the files were concatenated. The statement counts appearing in the report are the sums of the counts from all the listed files.

For evaluation one usually requires a report based on all the C files comprising the system. For programs with only a few files one may simply list all the .i files after the -f flag. For example,

```
prolc -f c.i d.i
```

will base the report on the files c.i and d.i, even if no routines in one of the C files are executed in the course of the tests. Large systems require too many C files to make the -f flag practical. CLUE enables the report to be based on one or more "listfiles" via the -l flag. The form is

```
prolc -l listfile ...
```

A listfile is a list of .i file names on which the report is to be based. Anything else may also appear in the listfile. A listfile is readily derived from a makefile by replacing all occurrences of '.o' by '.i' within the makefile. The Unix stream editor is quite useful here, allowing the listfiles to depend upon all the makefiles in all directories defining a system. Indeed, for large projects, we recommend that the listfiles be created in a make which keeps track of the dependencies upon the entire collection of makefiles defining the system.

A small example of a makefile appears in Listing 1. This makefile has been set to use 'procc -C' via the CC line. The -C flag means that comments will appear in the .i file and so also in the report. The makefile was edited to produce a file named 'fnamesf', appearing in Listing 2. The only editing was to globally replace '.o' by '.i'. This listfile is then used in the command

```
prolc -l fnamesf
```

to define the C files comprising this system to the report generator.

```
$ cat makefile
CFLAGS =
CC = procc -C

system: c.o d.o
        ${CC} c.o d.o -o system

c.o: c.h
```

Listing 1.

The statement count report consists of several fields of information, one row for each line of source in the defining .i files, and additional lines

```
$ cat fnamesf
CFLAGS =
CC = procc -C

system: c.i d.i
        ${CC} c.i d.i -o system

c.i: c.h
```

Listing 2.

whenever more than one executable statement appears on the same line of the .i file. The first field defines the line type via the one character type key:

F	file name
c	text outside function definitions
f	beginning of function definition
n	nonexecutable text in a function definition
x	executable statement with non-zero count
z	executable statement with zero execution count

The line type key makes special report generation easy. The second field repeats the name of the function throughout the function definition. The third field numbers each line in each function definition. The fourth field is the execution count for executable statements. The final field is the source text derived from the .i files. Examples of the report are in Listings 3, 7, and 8.

CLUE includes several filters and generators to present particular information from the statement count report. The most popular generator is 'lcp', which produces a code coverage report by system, by file and by function. Listing 4 provides an example. The most popular filter is 'lcf', which easily enables the user to reformat the statement count report, selecting information of particular interest to present. The generator 'lch' generates histograms of statement count data. The generator 'lct' provides statement count totals by system and function. The filter 'lcr' enables the user to select a range of counts for which the corresponding C statements are of interest.

Debugging Example

This example occurred when I was first preparing the next example for the paper. I wrote the little program 'bad.c' and attempting to execute it in preparation for running CLUE to produce the intended example. As Listing 3 shows, the program died with a bus error. My experience has been that I can find the fault faster by using CLUE than by using a symbolic debugger.

First I removed 'profile.lc' just in case it was filled with information from a previous use of CLUE. Then I ran 'procc -C' to instrument the program.

The -C option was included to keep the comments for the illustration in Listing 3. Ordinarily I do not use the -C option. I then ran 'prolc' to produce the report shown in Listing 3. Notice that in routine 'main', lines 6 through 11 are the body of a for loop which obviously should be executed exactly 6 times. Clearly the bus error occurred during the sixth call to 'malloc'. Inspection of the argument to 'malloc' shows that the argument to 'sizeof' returns the size of a pointer. This argument should be the size of the structure. Repairing this by removing the extraneous '*' results in a running program, the basis for the next example.

Evaluation Example

This example results from the evaluation of a useful software package. I have reduced the problem to its essence to form the example, keeping the file and program structure faithful to the original. The package involves many C files, which I have reduced to two for the example. In addition there is one header file included. The header file and the file 'c.c' are given in listing 4. The file 'd.d' is shown in Listing 5. The package ran correctly, but was intolerably slow for long inputs. The example will show why. The evaluation using CLUE begins by making a copy of all pertinent files in a directory for the CLUE evaluation. In this case I used the subdirectory 'example' of the directory 'test'. I began by modifying the makefile to use 'procc -C'. The resulting makefile is given in Listing 1. I then made the listfile named 'fnamesf' in Listing 2 by changing all occurrences of '.o' to '.i'. I next instrumented and ran the program. The script is given in Listing 6. I keyed the 'make system' and make responded with the invocations of 'procc'. I then keyed 'rm profile.lc' to be sure that the standard statement count information file was removed, since the instrumentation always appends to 'profile.lc'. Finally, I keyed 'prolc | lcp' to obtain the code coverage percentage report by piping the statement count report produced by 'prolc' directly into 'lcp'.

The report in Listing 6 shows a dismaying low percentage of code executed. In the entire system, only 65 of the executable statements were executed, the exact number being 13 executed and 7 unexecuted. In the first C file, about 81 of the executable statements were executed. Within this file, the routine 'process_node' had 2 statements unexecuted, the routine 'main' was completely executed, and the routine 'post_finish' was not executed at all. In the second C file, there is only one routine, which was not executed. All this suggests which routines to look for problems. The CLUE filter 'lcf' would enable one to look at one routine at a time. This example is short enough, however, that I chose to look at the entire statement count report. The report appears in Listing 7 and Listing 8. It has been split into two listings since it is too long to fit on one page.

The routine 'main' begins halfway down Listing 7. The routine builds a list of 6 nodes containing information. In this example the information is simply the node number. In the system upon which the example is based, the length and content of the list depended upon the input. At the end of 'main', the routine 'process_nodes' is called. This routine appears in the top half of listing 5. This routine was intended to process the information in each node exactly once. However, the routine is recursively called, with a total

```
$ cc bad.c -o bad
$ bad
Bus error (core dumped)
$ procc -C bad.c
$ rm profile.lc
$ a.out
CLUE: Abnormal termination with signal 10
      Line counts saved.
$ prolc
F /users/dbenson/test/example/bad.i
c c00lbad.i              0          struct node_list {
c c00lbad.i              1              int node;
c c00lbad.i              2              struct node_list *next;
c c00lbad.i              3              };
f process_nodes          0        0 void
n process_nodes          1          process_nodes(list)
n process_nodes          2              struct node_list *list;
n process_nodes          3          {
z process_nodes          4        0     if(list==0)
z process_nodes          5        0         return;
n process_nodes          6                  /* obtain information from node...
 */
z process_nodes          7        0     while(list->next!=0)
n process_nodes          8              {
z process_nodes          9        0         process_nodes(list->next);
z process_nodes         10        0         list = list->next;
n process_nodes         11              }
z process_nodes         12        0     return;
n process_nodes         13          }
f main                   0        1 main() {
n main                   1              int i;
n main                   2              struct node_list * head, *cur;
n main                   3
x main                   4        1     cur = (struct node_list *) 0;
x main                   5        1     for( i=0; i<6; i++ )
n main                   6              {
x main                   7        6         head = (struct node_list *) malloc
(sizeof(struct node_list *));
x main                   8        5         head->node = i;
x main                   9        5         head->next = cur;
x main                  10        5         cur = head;
n main                  11              }
n main                  12
n main                  13              /* other statements... */
z main                  14        0     ;
n main                  15
z main                  16        0     process_nodes(head);
n main                  17          }
```

Listing 3.

```
$ cat c.h
#define NIL 0
$ cat c.c
#include "c.h"
struct node_list {
        int node;
        struct node_list *next;
        };

void
process_nodes(list)
        struct node_list *list;
{
        if(list==NIL) {
                post_finish();
                return;
                }
        /* obtain information from node... */
        while(list->next!=NIL) {
                process_nodes(list->next);
                list = list->next;
                }
        return;
}

main() {
        int i;
        struct node_list * head, *cur;

        cur = (struct node_list *) NIL;
        for( i=0; i<6; i++ ) {
                head = (struct node_list *) malloc(sizeof(struct node_list));
                head->node = i;
                head->next = cur;
                cur = head;
                }
        /* other statements... */ ;
        process_nodes(head);
}

post_finish() {
        remove_node_list();
}
```

Listing 4.

of 32 calls. Noting that 2**(6-1) = 32, we guess that the nodes are actually visited as if they formed a binary tree. The problem is in lines 10 and 12 of 'process_nodes'. One designer had decided the routine should recursively traverse the list while another had decided to iteratively traverse the list.

```
$ cat d.c

struct node_list {
        int node;
        struct node_list *next;
        };

remove_node_list(head)
        struct node_list * head;
{
        struct node_list * next;

        while(head!=0) {
                next = head->next;
                free(head);
                head = next;
                }
}
```

Listing 5.

```
$ make system
procc -C  -c c.c
procc -C  -c d.c
procc -C c.o d.o -o system
$ rm profile.lc
rm: profile.lc nonexistent
$ system
$ prolc -l fnamesf | lcp
Code Coverage x, z, x:(x+z)              13    7    65.00 %

   /users/dbenson/test/example/c.i       13    3    81.25 %
      process_nodes                       5    2    71.43 %
      main                                8    0   100.00 %
      post_finish                         0    1     0.00 %
   /users/dbenson/test/example/d.i        0    4     0.00 %
      remove_node_list                    0    4     0.00 %
```

Listing 6.

The result was the unbelievably slow performance of the actual system for lists of length 8 or more. This problem is repaired be replacing the 'while' in line 10 by 'if'.

There is another problem resulting from the structure of this routine. Since the execution of the while loop is dependent upon the existence of

```
$ prolc -l fnamesf
F /users/dbenson/test/example/c.i
c c00lc.i                0          struct node_list {
c c00lc.i                1              int node;
c c00lc.i                2              struct node_list *next;
c c00lc.i                3              };
c c00lc.i                4
f process_nodes          0       32 void
n process_nodes          1          process_nodes(list)
n process_nodes          2              struct node_list *list;
n process_nodes          3          {
x process_nodes          4       32     if(list==0)
n process_nodes          5              {
z process_nodes          6        0         post_finish();
z process_nodes          7        0         return;
n process_nodes          8              }
n process_nodes          9              /* obtain information from node... */
x process_nodes         10       32     while(list->next!=0)
n process_nodes         11              {
x process_nodes         12       31         process_nodes(list->next);
x process_nodes         13       31         list = list->next;
n process_nodes         14              }
x process_nodes         15       32     return;
n process_nodes         16         }
c c002c.i                0
f main                   0        1 main() {
n main                   1              int i;
n main                   2              struct node_list * head, *cur;
n main                   3
x main                   4        1     cur = (struct node_list *) 0;
x main                   5        1     for( i=0; i<6; i++ )
n main                   6              {
x main                   7        6         head = (struct node_list *) malloc
(sizeof(struct node_list));
x main                   8        6         head->node = i;
x main                   9        6         head->next = cur;
x main                  10        6         cur = head;
n main                  11              }
n main                  12              /* other statements... */
x main                  13        1     ;
x main                  14        1     process_nodes(head);
n main                  15         }
c c003c.i                0
f post_finish            0        0 post_finish() {
z post_finish            1        0     remove_node_list();
n post_finish            2         }
```

Listing 7.

```
F /users/dbenson/test/example/d.i
c c005d.i                0
c c005d.i                1          struct node_list {
c c005d.i                2              int node;
c c005d.i                3              struct node_list *next;
c c005d.i                4              };
c c005d.i                5
f remove_node_list       0        0 remove_node_list(head)
n remove_node_list       1              struct node_list * head;
n remove_node_list       2          {
n remove_node_list       3              struct node_list * next;
n remove_node_list       4
z remove_node_list       5        0     while(head!=0)
n remove_node_list       6              {
z remove_node_list       7        0         next = head->next;
z remove_node_list       8        0         free(head);
z remove_node_list       9        0         head = next;
n remove_node_list      10              }
n remove_node_list      11          }
```

Listing 8.

another node in the list, 'process_nodes' is never called with an empty list, so the routine 'post_finish' was never called on line 6 of 'process_nodes'. The routine 'post_finish' is at the bottom of Listing 7. In the actual system there was considerable cleanup activity. I have just shown the call to 'remove_node_list' in the example.

The routine 'remove_node_list' is shown in Listing 8. This routine simply frees the entire node list, preparing for another round of input. The problems with 'process_nodes' meant that it was never called.

The problem of failing to free all dynamically allocated storage happens in many software projects. It is a source of subtle errors as well as the frustrating out-of-memory error. With the increasing use of virtual storage, it is often difficult to detect this problem during evaluation. The CLUE statement count report provides a simple means to assure that all dynamically allocated storage has been freed. The idea is straightforward: Sum the counts of all calls to 'malloc', sum the counts of all calls to 'free', and compare the totals. Listing 9 is a sample shell script to do this. The statement count reports are '.scr' files by convention, so the command line input to 'balance' is just the project name. The shell script uses the stream editor 'sed' to select just the lines of the report in which 'malloc' appears, placing these in a '.malloc' file. The shell script uses 'awk' to compute the sum of the statement counts in the '.malloc' file. The command file for 'awk' is shown in Listing 10.

A similar process is carried out for lines containing 'free'. If the totals are equal a pleasant message is printed and the extra files are

```
$ cat balance
: check that number of 'malloc' calls equal the number of 'free' calls.
sed -n -e /malloc/p $1.scr >$1.malloc
mallocs=`awk -f awktotal $1.malloc`
echo "mallocs: $mallocs"
sed -n -e /free/p $1.scr >$1.free
frees=`awk -f awktotal $1.free`
echo "frees: $frees"
if test $mallocs -eq $frees
then
  echo 'number of malloc calls equals number of free calls'
  rm $1.mallocs
  rm $1.frees
else
  echo '*****************************************************************'
  echo 'STANDARDS VIOLATION: number of malloc calls and free calls differ'
  echo ''
  echo 'mallocs:'
  cat $1.malloc
  echo ''
  echo 'frees:'
  cat $1.free
fi
```

Listing 9.

```
$ cat awktotal
{ total = total + $4 }
END {print total}
```

Listing 10.

removed. If the totals are not equal, a less pleasant message is printed and the two files are listed for the evaluator. Listing 11 shows the run of 'balance' on our example system.

The shell script 'balance' also illustrates the variety of tools available in Unix to process text files such as the CLUE statement count report. We have included several filters and generators in the CLUE package, but urge each evaluation group to develop additional generators such as 'balance'. These are easily written with the Unix utilities such as 'sh', 'sed' and 'awk'. Widely used generators will be incorporated in subsequent releases of CLUE.

The CLUE filter 'lcf' is used to select information from the statement count report, reformat the statement count report, and change the type keys to

```
$ prolc >system.scr
$ balance system
mallocs: 6
frees: 0
*****************************************************************
STANDARDS VIOLATION: number of malloc calls and free calls differ

mallocs:
x main                  7       6       head = (struct node_list *) malloc

frees:
z remove_node_list      8       0       free(head);
```

Listing 11.

preferred characters. Listing 12 demonstrates these features. The command line selects just the routine 'process_nodes' via the -s flag. In addition, we select just the type keys (ty), a blank (b), and the source line (sl) via the -f flag. Finally, the type keys are translated via the -t flag. All the type keys which type lines in a function definition are translated to blank, except the 'z' type key which is translated to '*'. The result is a code coverage report in which the unexecuted statements are conspicuous.

```
$ prolc | lcf -s process_nodes -f ty b sl -t f:  n:  x:  z:*
  void
  process_nodes(list)
      struct node_list *list;
  {
      if(list==0)
      {
*         post_finish();
*         return;
      }
      /* obtain information from node... */
      while(list->next!=0)
      {
          process_nodes(list->next);
          list = list->next;
      }
      return;
  }
```

Listing 12.

Additional reports may be obtained from the CLUE generators 'lct', 'lch', and the CLUE filter 'lcr'. When considering performance, we are partial to

reports which highlight heavily used statements. The filter 'lcr' selects a range of counts, by percentage of the total counts, for highlighting. The type keys within the range remain 'x' in the output of 'lcr' while the type keys of executed statements outside the range are changed to 'p'. The result is usually piped through 'lcf' to reformat before viewing.

In Listing 13 we have an example in which the most frequently executed 85% of lines are selected by 'lcr -lb 15' (the lower bound of the desired range is 15%) and the result piped to 'lcf' In 'lcf' the selection is done by the type keys via the -k option. The report keeps only the 'x' type lines. In addition, the resulting report consists only of the function name field (fn), a blank (b), and the source line (sl), on the command line after the -f flag.

```
$ prolc | lcr -lb 15 | lcf -k x -f fn b sl
process_nodes        if(list==0)
process_nodes        while(list->next!=0)
process_nodes            process_nodes(list->next);
process_nodes            list = list->next;
process_nodes        return;
```

Listing 13.

Of course in our example, almost all executions occur in the function 'process_nodes'. All the remaining counts fall in the lowest 15% of all executions and 'lcr' has converted the type key on these lines to 'p'. So the selection in 'lcf' eliminates such lines from the resulting report. In more substantial programs, similar reports are often quite valuable and surprising.

Our last example is a call count report, given in Listing 14. Each type 'f' line in the statement count report begins a routine definition. Counts of function entries reported on these lines. We select the file name lines and the function header lines of the report via '-k F f' and reformat them as the statement count (lc), a blank (b), and the function name (fn) after the -f flag. Notice that the file name lines are not reformatted. The result is a report of the number of times each routine was called, headed by the file in which the routine is to be found.

```
$ prolc | lcf -k F f -f lc b fn
F /users/dbenson/test/example/c.i
       32 process_nodes
        1 main
        0 post_finish
F /users/dbenson/test/example/d.i
        0 remove_node_list
```

Listing 14.

There are still more uses for a properly designed statement count report. The CLUE User's Manual describes several additional uses for the CLUE statement count report in connection with the filters and generators. References [1] and [2] present other uses of statement count profilers.

Design Criteria: Evaluation of CLUE

Foremost, a statement count profiler must provide accurate counts under all conditions, while maintaining the functionality of the original code. CLUE maintains the original function of the code with a few insignificant exceptions: CLUE writes an additional file for the profiling information. CLUE issues signals to trap all the terminating errors so that the profiling information file can be written before program termination. Any signals issued by the original code override the CLUE signals, so the original function of the signal processing routines in the instrumented code is maintained. CLUE requires a working 'malloc' to provide storage for the count accumulations. Thus the instrumented code uses more storage than the uninstrumented code. If the original code functioned correctly only in isolation with its own pattern of 'malloc' storage allocation calls, then it is unlikely to function correctly when instrumented by CLUE. I view this positively, since any change to 'malloc' is likely to cause such a program to stop working. Such a program is not robust.

CLUE provides accurate counts, again with a few minor exceptions: statement counts of one billion (10**9) or more are reported as "infinity" in the report. CLUE uses the 'sigalrm' alarm clock signal to time writes to the profiling information file in order to guarantee accurate counts for programs which run a very long time. Each time the alarm goes off (currently set at 20 minutes) the counts are written out and the counts reset to zero. If the original program uses the 'sigalrm' signal, this protection is lost and the counters could conceivably overflow, losing count accuracy. CLUE will report counts of all forked processes provided all the descendant processes terminate before the report generator is run. Finally, if CLUE is used to instrument 'malloc', the counts will include the uses of 'malloc' by the instrumentation. Similarly, if any of the other operating system services which CLUE requires are instrumented, the counts will include the uses by CLUE. CLUE is designed to function under the most stringent of conditions. The experience to date suggests that it does.

The second design criterion for CLUE was simplicity for users. While simplicity is certainly a matter of individual judgement, we feel we have succeeded in making CLUE easy to understand and use. The software engineer instruments programs by using 'procc' wherever 'cc' is ordinarily used to compile C programs. After the executable image is run, the report is obtained by invoking 'prolc'. This suffices to begin using CLUE. The additional features can easily be acquired as one uses CLUE by reading the on-line manual pages provided. The CLUE User's Manual contains all the details, but like most manuals, tends only to be consulted when unusual uncertainties arise. The additional features are specifically intended for the professional evaluator. As need for yet further features arise, we intend to add such to later releases of CLUE.

Users have mentioned that the use of the .i files clutters their directories. A future release of CLUE will eliminate the .i files, simplifying CLUE at the expense of additional time to run the report generator, 'prolc'.

The last design criterion was speed. The instrumented code runs longer than the original software. Long instrumented programs require about 120% of the original time. Very short programs can take up to twice as long to run when instrumented, as the profiling information file write time dominates. The counts are incremented only once per "block" of straight-line code to cut down on the excess time due to the instrumentation. This helps a bit, but C code rarely has long sequences of statements without a function call, and we begin a new block after each function call. The majority of the excess time is the result of writing the profiling information file. Not much can be done to improve the file write time while still maintaining the strict accuracy of the statement counts, the simplicity of use, and the clean directories.

Instrumenting C source with 'procc' requires about two and one-half as much time as just compiling via 'cc', running under bsd 4.2. The ratio is better when running under Eunice (tm The Wollogong Group). We believe that eliminating the .i file will improve the performance of 'procc'.

While speed is appreciated, robustness and simplicity are our primary goals. The software designer and evaluator will have little difficulty in using the tool and will have confidence in the results.

Other Evaluations of CLUE

CLUE is regularly used to instrument itself. We have a regression test suite of about 200 tests. Running the instrumented version of CLUE on the regression test suite results in an eight megabyte 'profile.lc' file. We use the statement count report to determine what portions of the code have not been exercised by the regression test suite, and where the inefficiencies lie. When a new release of CLUE is made, the regression testing statement count report may suggest new tests to cover the revised code.

CLUE is regularly used for much these same purposes by software engineers in other organizations. In general it has performed well over the last twelve months. The largest system that CLUE has instrumented to date, as far as I know, is a 190,000 line software product. CLUE failed to instrument two of the modules because the block nesting of these modules was already near the limit of the compiler. CLUE adds additional levels of block nesting to maintain the original functionality of the C code. On the remaining modules, CLUE provided the required information. Size is not an issue for CLUE, so long as enough file space is provided.

A few faults with CLUE have been uncovered in the twelve months since the conclusion of the beta testing. These have been reworked. Of course, test cases for these faults have been added to the regression test suite.

Overall, the design seems to be appreciated by CLUE users. CLUE has proved to be a robust and simple tool for debugging and evaluation. It is

well integrated into the Unix environment, incorporating the Unix style of simple programs which do one thing well and which fit together easily. The practicing software engineer and software evaluator will enjoy using CLUE and have high confidence in the results.

Acknowledgements

I heartily thank Doug Gregory, Keith Koplitz, Craig Thomas, Brian Carlson, and Clay Breshears for their work on CLUE. I sincerely appreciate the assistance of Tektronics Logic Design Systems and DATA I/O in the beta test, and thank the software engineers at both organizations for their aid and suggestions. I also want to thank Kelly Whitmill of Burroughs and Robert Wells of BBN for suggestions and patience.

Availability

CLUE is currently available for the Unix operating system varieties

bsd 4.1
bsd 4.2
Eunice

on VAX (tm DEC) hosts. Object and source licenses may be obtained only from the distributor:

Oasys
60 Aberdeen Avenue
Cambridge, Massachusetts 02138.

References

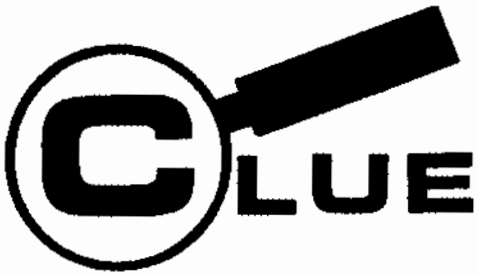
[1] J. L. Bentley, Writing Efficient Programs, Prentice_Hall, 1982.

[2] L. R. Power, Design and use of a program execution analyzer, IBM Systems Journal v. 22 (1983), 271-294.

BIOGRAPHY

David Benson

David B. Benson received his BS, MS, and PhD in engineering, science and mathematics from the California Institute of Technology. He spent some time in the defense industry designing information systems during this period. Dr. Benson has taught at the University of North Carolina at Chapel Hill, Washington State University at Pullman, the University of Colorado at Boulder, and in 1983 was briefly at the University of Edinburgh in Scotland. Since 1979, David Benson has been Professor of Computer Science at Washington State University. He has published over 30 professional papers and lectured in Canada, Europe, Japan, and India. In 1984 he formed BENTEC, of which he is a general partner, to provide high-quality software tools to the software industry.



statement count profiler

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

CLUE

consists of

prolc
procc

and

lcf lcp

lch lcr lct

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

a.out

counts appended to

profile.lc

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

prolc

statement count report

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

.i files

libclue.a
-lclue

BENTEC

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

```
$ cat makefile
CFLAGS =
CC = procc -C

system: c.o d.o
        ${CC} c.o d.o -o system

c.o: c.h
```

```
$ cat fnamesf
CFLAGS =
CC = procc -C

system: c.i d.i
        ${CC} c.i d.i -o system

c.i: c.h
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
cc bad.c -o bad
bad
Bus error (core dumped)
procc -C bad.c
rm profile.lc
a.out
CLUE: Abnormal termination with signal 10
      Line counts saved.
prolc
F /users/dbenson/test/example/bad.i
c c00lbad.i              0          struct node_list {
c c00lbad.i              1              int node;
c c00lbad.i              2              struct node_list *next;
c c00lbad.i              3              };
f process_nodes          0        0 void
n process_nodes          1          process_nodes(list)
n process_nodes          2              struct node_list *list;
n process_nodes          3          {
z process_nodes          4        0     if(list==0)
z process_nodes          5        0         return;
n process_nodes          6                  /* obtain information from node... */
z process_nodes          7        0     while(list->next!=0)
n process_nodes          8                  {
z process_nodes          9        0         process_nodes(list->next);
z process_nodes         10        0         list = list->next;
n process_nodes         11                  }
z process_nodes         12        0     return;
n process_nodes         13          }
f main                   0        1 main() {
n main                   1              int i;
n main                   2              struct node_list * head, *cur;
n main                   3
x main                   4        1     cur = (struct node_list *) 0;
x main                   5        1     for( i=0; i<6; i++ )
n main                   6                  {
x main                   7        6         head = (struct node_list *) malloc(
sizeof(struct node_list *));
x main                   8        5         head->node = i;
x main                   9        5         head->next = cur;
x main                  10        5         cur = head;
n main                  11                  }
n main                  12
n main                  13                  /* other statements... */
z main                  14        0     ;
n main                  15
z main                  16        0     process_nodes(head);
n main                  17          }
```

BENTEC
NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ cat c.h
#define NIL 0
$ cat c.c
#include "c.h"
struct node_list {
        int node;
        struct node_list *next;
        };

void
process_nodes(list)
        struct node_list *list;
{
        if(list==NIL) {
                post_finish();
                return;
                }
        /* obtain information from node... */
        while(list->next!=NIL) {
                process_nodes(list->next);
                list = list->next;
                }
        return;
}

main() {
        int i;
        struct node_list * head, *cur;

        cur = (struct node_list *) NIL;
        for( i=0; i<6; i++ ) {
                head = (struct node_list *) malloc(sizeof(struct node_list));
                head->node = i;
                head->next = cur;
                cur = head;
                }
        /* other statements... */ ;
        process_nodes(head);
}

post_finish() {
        remove_node_list();
}
```

NE 615 Campus Street
Pullman, WA 99163
(509) 332-3164

```
$ cat d.c

struct node_list {
        int node;
        struct node_list *next;
        };

remove_node_list(head)
        struct node_list * head;
{
        struct node_list * next;

        while(head!=0) {
                next = head->next;
                free(head);
                head = next;
                }
}
```

NE 61S CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
make system
/users/dbenson/clue/src/procc -C  -c c.c
/users/dbenson/clue/src/procc -C  -c d.c
/users/dbenson/clue/src/procc -C c.o d.o -o system
rm profile.lc
rm: profile.lc nonexistent
system
prolc -l fnamesf | lcp
Code Coverage x, z, x:(x+z)              13    7    65.00 %

   /users/dbenson/test/example/c.i       13    3    81.25 %
      process_nodes                       5    2    71.43 %
      main                                8    0   100.00 %
      post_finish                         0    1     0.00 %
   /users/dbenson/test/example/d.i        0    4     0.00 %
      remove_node_lis                     0    4     0.00 %
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
prolc -l fnamesf
F /users/dbenson/test/example/c.i
c c001c.i                0          struct node_list {
c c001c.i                1              int node;
c c001c.i                2              struct node_list *next;
c c001c.i                3              };
c c001c.i                4
f process_nodes          0       32 void
n process_nodes          1          process_nodes(list)
n process_nodes          2              struct node_list *list;
n process_nodes          3          {
x process_nodes          4       32     if(list==0)
n process_nodes          5              {
z process_nodes          6        0         post_finish();
z process_nodes          7        0         return;
n process_nodes          8              }
n process_nodes          9              /* obtain information from node... */
x process_nodes         10       32     while(list->next!=0)
n process_nodes         11              {
x process_nodes         12       31         process_nodes(list->next);
x process_nodes         13       31         list = list->next;
n process_nodes         14              }
x process_nodes         15       32     return;
n process_nodes         16         }
c c002c.i                0
f main                   0        1 main() {
n main                   1              int i;
n main                   2              struct node_list * head, *cur;
n main                   3
x main                   4        1     cur = (struct node_list *) 0;
x main                   5        1     for( i=0; i<6; i++ )
n main                   6              {
x main                   7        6         head = (struct node_list *) malloc(
sizeof(struct node_list));
x main                   8        6         head->node = i;
x main                   9        6         head->next = cur;
x main                  10        6         cur = head;
n main                  11              }
n main                  12              /* other statements... */
x main                  13        1     ;
x main                  14        1     process_nodes(head);
n main                  15         }
c c003c.i                0
f post_finish            0        0 post_finish() {
z post_finish            1        0     remove_node_list();
n post_finish            2         }
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
F /users/dbenson/test/example/d.i
c c005d.i                0
c c005d.i                1          struct node_list {
c c005d.i                2              int node;
c c005d.i                3              struct node_list *next;
c c005d.i                4              };
c c005d.i                5
f remove_node_list       0        0 remove_node_list(head)
n remove_node_list       1              struct node_list * head;
n remove_node_list       2          {
n remove_node_list       3              struct node_list * next;
n remove_node_list       4
z remove_node_list       5        0     while(head!=0)
n remove_node_list       6              {
z remove_node_list       7        0         next = head->next;
z remove_node_list       8        0         free(head);
z remove_node_list       9        0         head = next;
n remove_node_list      10              }
n remove_node_list      11          }
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ cat balance
: check that number of 'malloc' calls equal the number of 'free' calls.
sed -n -e /malloc/p $1.scr >$1.malloc
mallocs=`awk -f awktotal $1.malloc`
echo "mallocs: $mallocs"
sed -n -e /free/p $1.scr >$1.free
frees=`awk -f awktotal $1.free`
echo "frees: $frees"
if test $mallocs -eq $frees
then
  echo 'number of malloc calls equals number of free calls'
  rm $1.mallocs
  rm $1.frees
else
  echo '*****************************************************************'
  echo 'STANDARDS VIOLATION: number of malloc calls and free calls differ'
  echo ''
  echo 'mallocs:'
  cat $1.malloc
  echo ''
  echo 'frees:'
  cat $1.free
fi
```

```
$ cat awktotal
{ total = total + $4 }
END {print total}
```

BENTEC

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ prolc >system.scr
$ balance system
mallocs: 6
frees: 0
**********************************************************************
STANDARDS VIOLATION: number of malloc calls and free calls differ

mallocs:
x main                  7       6       head = (struct node_list *) malloc

frees:
z remove_node_list      8       0       free(head);
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ prolc | lcf -s process_nodes -f ty b sl -t f:  n:  x:  z:*
  void
  process_nodes(list)
      struct node_list *list;
  {
      if(list==0)
      {
*         post_finish();
*         return;
      }
      /* obtain information from node... */
      while(list->next!=0)
      {
          process_nodes(list->next);
          list = list->next;
      }
      return;
  }
```

BENTEC

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ prolc | lcr -lb 15 | lcf -k x -f fn b sl
process_nodes          if(list==0)
process_nodes          while(list->next!=0)
process_nodes              process_nodes(list->next);
process_nodes              list = list->next;
process_nodes          return;
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

```
$ prolc | lcf -k F f -f lc b fn
F /users/dbenson/test/example/c.i
       32 process_nodes
        1 main
        0 post_finish
F /users/dbenson/test/example/d.i
        0 remove_node_list
```

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

CLUE

Simple to use

Clean directories

ROBUST

Evaluation reports

NE 615 CAMPUS STREET
PULLMAN, WA 99163
(509) 332-3164

TOOLS FOR PROBLEM REPORTING

Susan V. Bartlett
Metheus-CV, Inc.
Hillsboro, OR

Ways To Track Problems

Benefits of Data Base Management Systems

Determining Your Needs

Example of Our Implementation

Choosing Your Own DBMS

WAYS TO TRACK BUGS

1) Do Nothing

2) Manual Paper Systems

3) Design Your Own Computerized System

4) Use An Existing DBMS as a Base

WHY WE USE AN ON-LINE DBMS

- Central Location
- Instant Access
- Global View of Product Status
- Increases Visibility to Management
- Time Saving in Tracking Status
- Automatic Follow-up

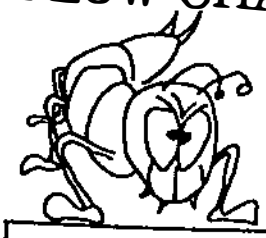
WHY WE USE AN ON-LINE DBMS (continued)

- Problems Aren't Lost
- Formalizes Methodology
- Enforce Entry of Needed Information
- Faster to Apply Metrics, Reporting
- Data Entry Time Same as Paper (or less)
- Unique Number for Cross Reference

DETERMINE YOUR NEEDS

- Look at the Big Picture
- What Are Your Needs Now?
- Try To Anticipate Future Needs
- Who Else Might Be Interested?

PROBLEM REPORTING SYSTEM
FLOW CHART



Find Problem

User → **Input Problem Manager** → **Report to Manager**

Manager → **Assign for Analyzing** → **Report to Engineer**

Engineer → **Analyze & Suggest Action** → **Report to Board**

Eng. Mgr. QA Mktg. → Board → **Determine Priority, Action, Schedule** → **Report to Engineer**

Engineer → **Fix, Delete or Defer (Enhancement)**

QA → **Record Closed** → **Report**

Three types of reports are automatically generated and mailed to assigned person:

1) Newly entered problems.

2) Problems to be analyzed.

3) Problems requiring actions.

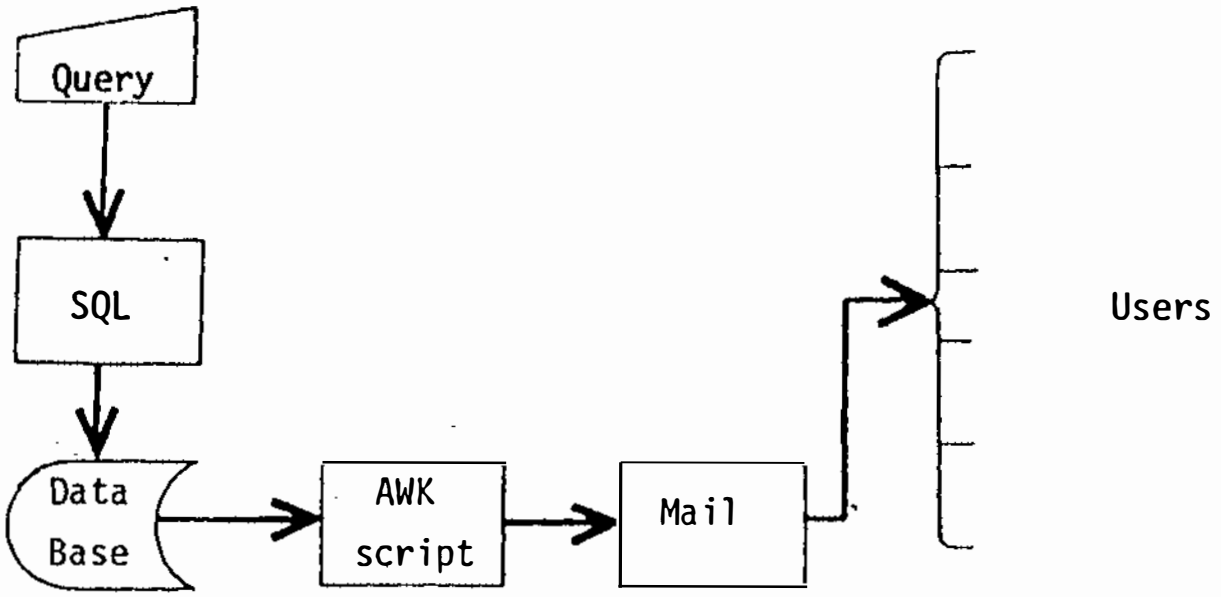


Figure 8. Process Flow for Notification Program

OUR PROBLEM REPORT RECORD

DATA ENTRY SCREEN

- Release Number
- Users View of Priority
- Description of Problem

ANALYSIS SCREEN

- Responsible Engineer
- Effort To Fix
- Analysis
- Recommendations

STATUS SCREEN

- Final Priority
- Activity and Engineer
- Target Date
- Complete Date

CHOOSING YOUR D.B.M.S.

- Set-up and Maintenance Utilities
- Query Language: On-line and Batch
- Access to Operating System
- Report / Formatting Utility
- Aggregate Functions (sum, count, avg)
- Pre and Post Processing of Data Entry
- Good Documentation with Examples
- Security

CHOOSING YOUR D.B.M.S.
(continued)

- Diagrams and Pictures
- Tie-in to Configuration Management
- Variable Length Fields
- Editing of Fields

TIME INVESTMENT

System Development and Enhancments

Development	~100 hours
Add Reports	5 min. -> 4 hours
Add New Record	2 days

System Monitoring

Monitor	~1 hour/week
CCB Meetings	Varies

BENEFITS

- Gives A Clear Picture Instantly
- Saves Time in Tracking Problems
- Automate Problem Tracking Procedures
- Reports are Easily Generated
- Flexible, Easy to Implement and Change
- Allows Better Response to Customers

Tools for Problem Reporting

Susan V. Bartlett

Project Leader for Software Test and Evaluation
Metheus-CV Inc.
Hillsboro, Oregon 97124

ABSTRACT

It has been said that all software has bugs. For companies in the software business, this premise translates into a need for methods to deal with known problems. The informal methods of word-of-mouth and paper memos have the disadvantages of temporary (or permanent) lapses of memory, misfiling, and the information being dispersed instead of being centrally available for queries.

We have obtained a database management system which we feel satisfies the requirements of problem reporting, problem tracking and problem follow-up. This paper discusses our application of the DBMS and its many benefits. The online database itself will be covered, with the ease of defining and updating the schema. Tools provided with the DBMS fulfill several needs: The definable screens and menus allow an easy-to-use interface to those who need to input information into the system. There is a choice of querying methods which are used for different levels of access, allowing users to see what problems have been reported and what their status is, along with other relevant information. We will present our use of the report writing system, along with the interface to the operating system which allows access to UNIX‡ mail and other utilities.

‡ UNIX is a trademark of Bell Laboratories.

Benefits of Error Reporting Systems

The benefits of error tracking systems are comprehensive. The fact that a system of this sort exists in a company indicates the realistic acceptance of the fact that errors can exist. The extent to which it is used and supported indicates the extent of the company's understanding of the quality problem and their commitment to maintenance of their product.

Ideally, a problem reporting system should include:

(1) A way to enter information which is easy to learn and use and thereby encourage its use.

(2) A form which captures all the information needed to reproduce and evaluate the problem.

(3) A way of assigning responsibility for the problem.

(4) A mechanism to confirm the problem.

(5) A mechanism to determine if the problem can be fixed and how.

(6) A mechanism to determine if the problem should be fixed.

(7) A tickler system to keep the ball from dropping.

(8) A way to make sure that the fix gets to the customers (both to those who report it and to future customers)

(9) A way to track the problem and determine its status at any time by anyone allowed access to the information.

(10) Security to insure that only those allowed access to the information can get to it.

(11) A way to apply metrics to all the problems as a whole in an effort to reduce problems in future products or determine the current "quality" of the product.

(12) Some way to determine the correspondence between changes to the product and the problems reported (ie: these lines of code were changed to fix problem number N, which was reported by...).

The Unify† Data Base Management System is the tool we have chosen to automate our Problem Reporting System. It has the flexibility and integrated utilities that allow fulfillment of most of the items in the wish list.

One of the biggest benefits of the DBMS is its reporting capabilities. The system is on line and real time and so allows the storage of reported bugs in a central location for general queries, but also allows a person to have instant access to up-to-date data at a terminal or on hard copy.

Some Drawbacks to Problem Reporting Systems

There are several drawbacks to problem reporting systems.

(1) Some programmers are reluctant to report bugs because it is an admission that their software is imperfect.

(2) Someone must monitor the system AND take responsibility for it.

(3) Bugs must be entered to be fixed.

(4) As with paper systems, managers and engineers must take it seriously and provide resources for maintenance to make it useful.

† Unify is a Trademark of Unify Corporation.

Definition of terms

Following are definitions for some common data base terms:

Field: The smallest significant unit in this data base. For example: "program name" may be a field and would be of type string (characters) of length 16, and "telephone number" would be a field of type numeric (integer) of length 10. (Note: DATE is a defined field type in Unify.)

Record Type: This is generally an entity which is comprised of related fields. For example: we have a "Problem Report" record type. This consists of the definition of all the fields which we considered necessary to provide complete information on a problem.

Record: A record is an instance of the record type. It consists of the data which describes the particular problem and it exists in the form described by the record type.

Schema: Definition of the information to be stored in the data base. In this case, the schema consists of the record types and the fields in each record type.

Query: We use this word to mean a description in a formal language (SQL) of the type of information we want, based on stated restrictions.

Use of Screens and the Schema

We chose to make one record type for our released software and one for unreleased software. Both record types consist of three categories of data: the submittal data, analysis of the incident and the current status of activities relating to the problem.

Figure 1 is an example problem report which has most of the fields represented. Refer to it for the following discussion.

The first category is the submittal fields. These include most of the information needed from the person who found or is reporting the problem.

Second, is the analysis fields: the information provided by the engineer assigned to evaluate the problem. This can include what the engineer perceives the problems to be, whether it is a problem, an enhancement or an improper use of the system, how long it would take to fix the problem, and the engineer's view of the priority.

The third category is what is determined in a Configuration Control Board (CCB) meeting. Members of the CCB are the managers of the engineering group, a person representing QA and a person representing Marketing. They look at the engineer's evaluation of the problem, the customer's perception of the importance of the problem, the engineering resources and the marketing priorities and come to an agreement on a status (bug, enhancement, duplicate or delete), the final priority, an action to be taken, a target date for that action to be completed and the person responsible.

Defining the Configuration of the System

The next few sections describe a little how easy it is to define the system you wish to create. Please refer to the examples. Figure 2 is a picture of the system maintenance menu to give you an idea of what kind of utilities are provided.

Metheus-CV Configuration Management System
Problem Report

07/19/85

Problem Report #: 327
Program Name: cif2phl
Release_ID: 3.1.pa
Resp. Group (sys/fe/be/sim/man): be
Company (if any): MCV
Priority (HOT/critical/major/minor): major

Name of Submitter: howard
Date of Occurrence: 08/07/84
Problem Type (bug/enh): bug
Problem Duplicated (y/n): y
Supporting Documents (y/n):

Summary: "cif2phl -150 -t foo" core dumps. "cif2phl -l 50 -t foo" doesn't. The options parser seems to be buggy.
Description: If there is no space after the "-l" in a cif2phl command line, the program can core dump trying to read the next argument as an integer.

ANALYSIS SECTION

Responsible Engineer: jay
Analysis Date: 08/21/84
Eng. Priority (critical/major/minor): minor
Effort to Fix (manhours): 4
Recommended Action - Software Change (y/n): y
Manual Change (y/n): n
Delete (not a bug): n
Change to Enhancement: n

Analysis: This is a minor problem with the command line options in cif2phl. I should be no problem to fix

CCB STATUS SECTION

CCB Date: 08/21/84
Final Priority (critical/major/minor): minor
CCB Action (fix/enh/dup/del): fix

Activity	Responsibility	Target Date	Completion Date
software fix	jay	11/15/84	11/20/84
software test	jay	11/15/84	11/20/84
check-in for 3.0	jay	11/20/84	11/21/84
		//**	**/**/**
		//**	**/**/**
		//**	**/**/**

Closure Date: 11/21/84

Figure 1. Example Problem Report

Schema Definition

The schema is easily defined just by typing in the name of the field, type of field and descriptive name. Figure 3 shows what the on line entry screen for the schema fields look like.

Enough space is allowed for the description to explain the field. It is a true relational data base, and you may use combination fields to link up with a field in another record type. Figure 4 shows the one page of the schema listing for the problem report record type.

Screen Definition

Once you have what you think you want, a utility is provided to reconfigure the data base and with that done, you can start defining your screens. Screen entry is quite easy. You can let the system give you a default screen, or you can use the paint

---

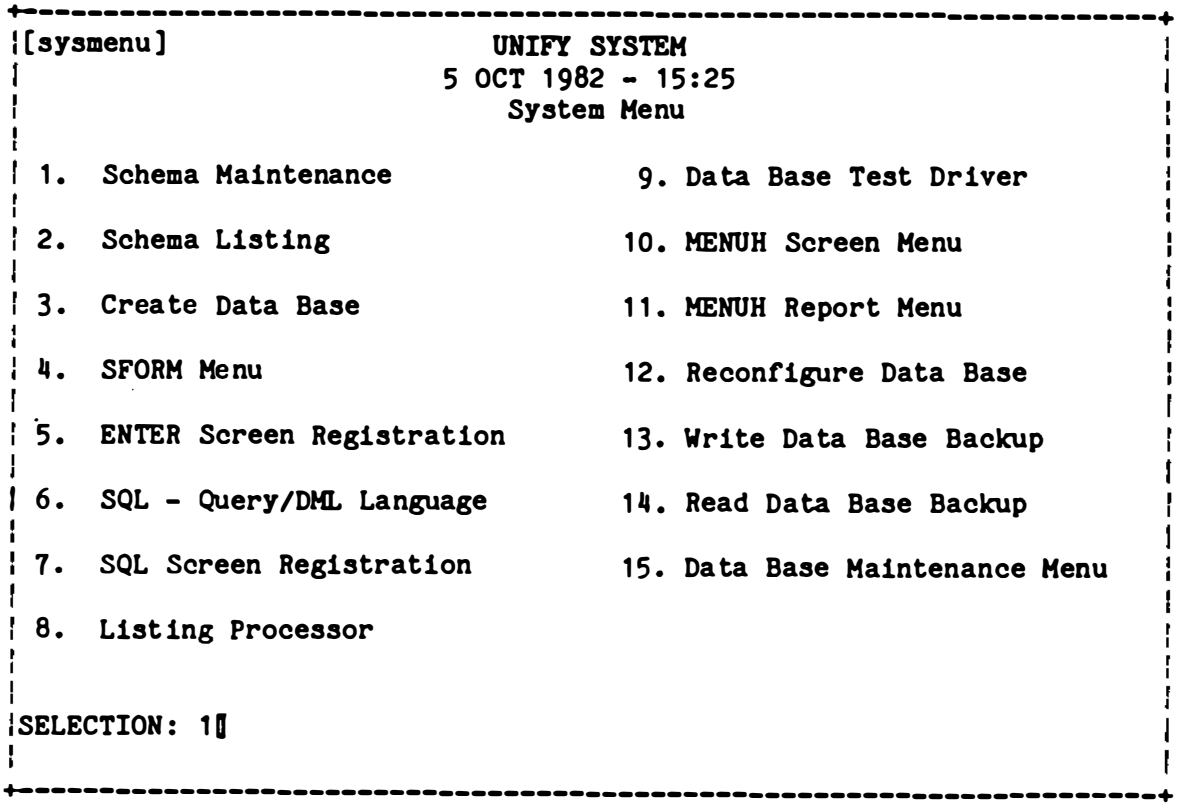


Figure 2. System Maintenance Menu

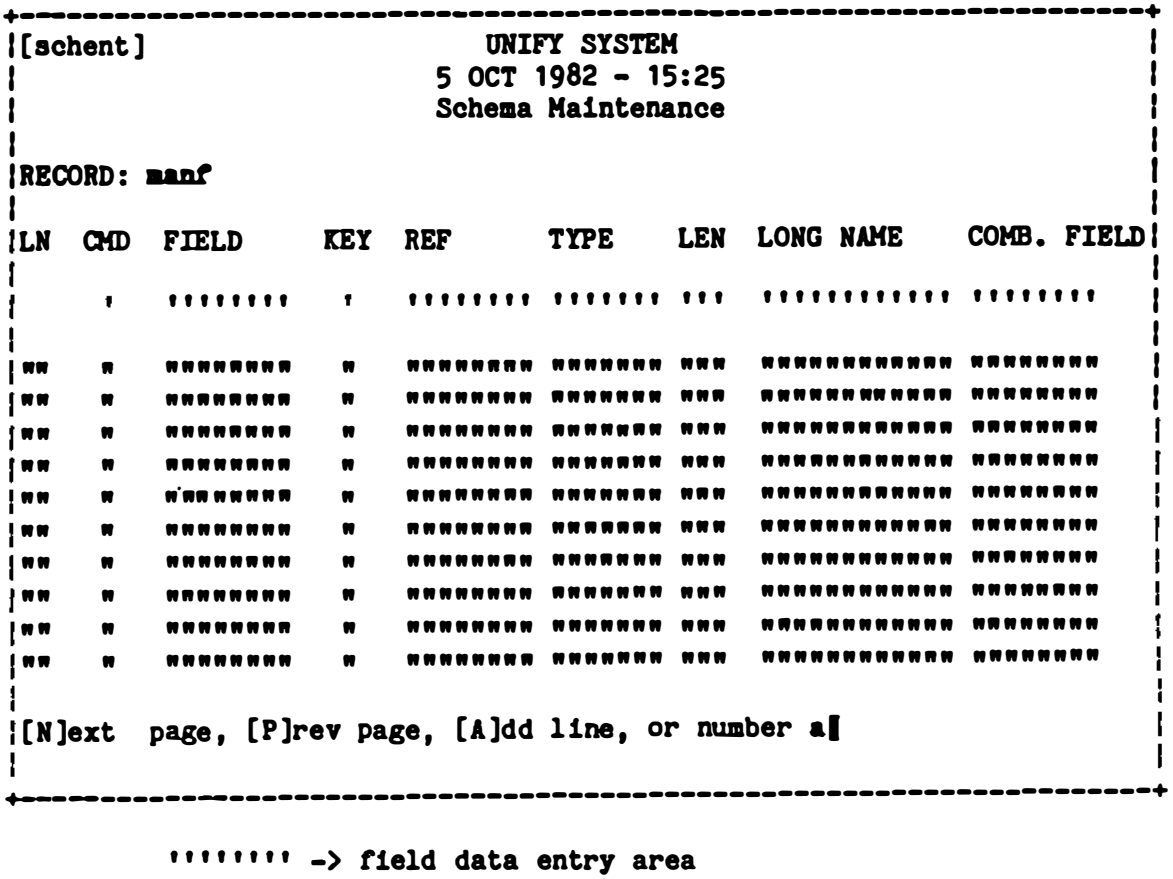


Figure 3. Field Data Entry

```
ipr              2000                                  interim_pr
    *iprno                          NUMERIC     5     pr_number
     submtr                         STRING      8     submitter_name
     prnm                           STRING     20     program_name
     relid                          STRING      8     release_id
     rcomp                          STRING     24     rptg_company
     sum1                           STRING     70     summary_line1
     sum2                           STRING     78     summary_line2
     desc1                          STRING     66     description1
     desc2                          STRING     78     description2
     desc3                          STRING     78     description3
     desc4                          STRING     78     description4
     desc5                          STRING     78     description5
     desc6                          STRING     78     description6
     prdt                           DATE              pr_date
     analdt                         DATE              analysis_date
     respeng                        STRING      8     responsible_eng
     engpri                         STRING      8     engineers_pri
     swchange                       STRING      1     software_change

DATE : 07/18/85  TIME : 20:42:31
                                SCHEMA REPORTS
                                Schema Listing

RECORD/FIELD        REF         TYPE      LEN   LONG NAME

     mpchange                   STRING      1   manpage_change
     delete                     STRING      1   del
     enh                        STRING      1   enhancement
     anal1                      STRING     69   analysis1
     anal2                      STRING     78   analysis2
     anal3                      STRING     78   analysis3
     anal4                      STRING     78   analysis4
     anal5                      STRING     78   analysis5
     anal6                      STRING     78   analysis6
     ccbdt                      DATE            ccb_date
     finpri                     STRING      8   final_priority
     ccbact                     STRING      3   ccb_action
     act1                       STRING     32   activity1
     act2                       STRING     32   activity2
     act3                       STRING     32   activity3
     act4                       STRING     32   activity4
     act5                       STRING     32   activity5
     act6                       STRING     32   activity6
     resp1                      STRING      8   resp1
     resp2                      STRING      8   resp2
     resp3                      STRING      8   resp3
     resp4                      STRING      8   resp4
     resp5                      STRING      8   resp5
     resp6                      STRING      8   resp6
     tgtdt1                     DATE            target_dt1
     tgtdt2                     DATE            target_dt2
     tgtdt3                     DATE            target_dt3
     tgtdt4                     DATE            target_dt4
     tgtdt5                     DATE            target_dt5
     tgtdt6                     DATE            target_dt6
     comodt1                    DATE            complete dt1
```

Figure 4. Schema Listing

facility. Figure 5 is an example from the Unify Tutorial Manual which shows a default screen built with a record type which has three fields.

The default screen takes all the fields in the record type and using the long name for the prompt, puts them in columns on the screen.

The paint facility lets you enter your own prompts and field positions anywhere you wish. It uses commands similar to *vi*, the screen oriented editor in Unix‡. For example: 'a' is to append, 'w' moves you across the line by word, and 'q' is quit. Figure 6 is the listing of the status screen for our problem report. It took about an hour to enter this screen in paint.

When you finally have it the way you want it, you register it by executing another utility and start entering data.

```
[manf]                          UNIFY SYSTEM
                              5 OCT 1983 - 15:25
                            Manufacturer Maintenance

number :
name   :
address:









[I]NQUIRE, [A]DD, [M]ODIFY, [D]ELETE
```

Figure 5. Default Screen Example

‡ Unix is a Trademark of Bell Laboratories.

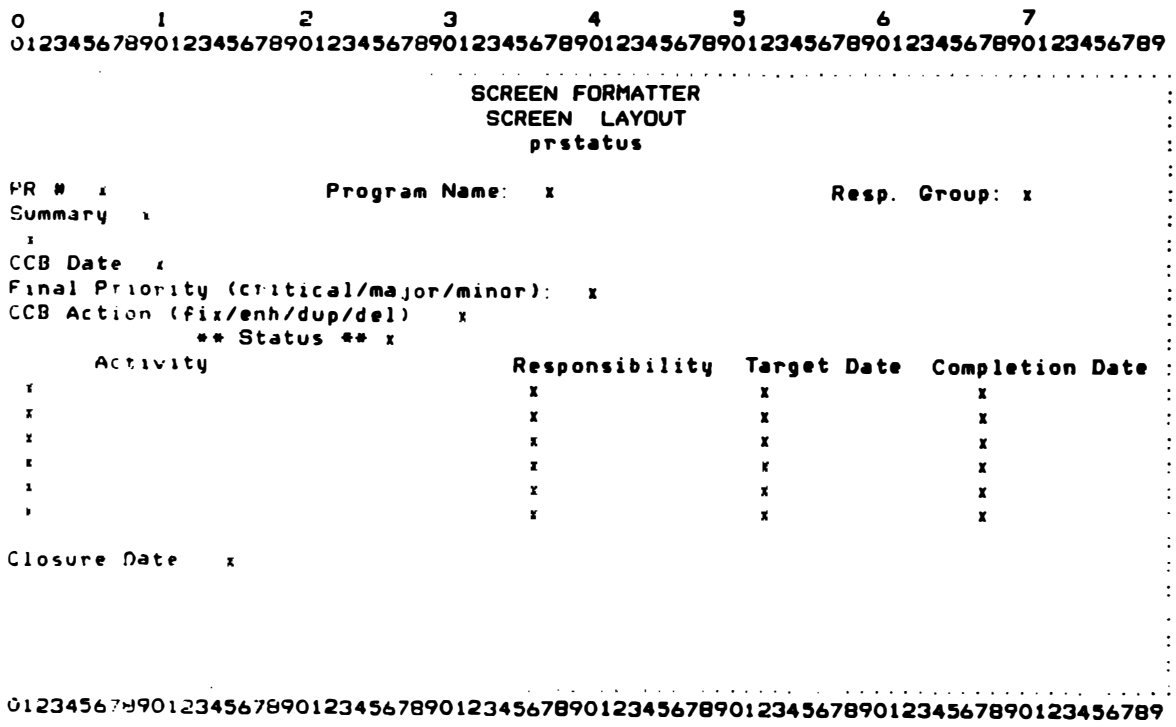


Figure 6. Problem Status Screen

Changing the Schema (Schema maintenance)

Generally, once you've designed a data base and then started using it, you find lots of things that you want to change There is very little difficulty in maintaining this data base. Changing it is even easier than defining it: just modify the schema by deleting, adding or changing the field you want. Then reconfigure the data base and you are done. The screen is changed in a similar manner: delete or add the field and re-register the screen.

Input

Data input is accomplished by one of two means: input through the screen you just built or through a data base load (batch method) which uses ascii files. The only rules you have to remember for input through a screen is that carriage return gets you to the next field and <control> U gets you back. When in doubt, <control> U like crazy and you will get out.

Querying and Report Feature

The querying feature is a very powerful tool. It is an implementation of the IBM Sequel(SQL) relational inquiry and data manipulation language based on an English keyword syntax. Together with the report writer (RPT) it's just about all you need to get whatever information you want out of the data base.

You can query on any field, match keywords, ranges, etc. The results of the query can be dumped to the screen, a unix file, or a printer, or you can write a C program or shell script and pipe it through any utility you like.

Figure 7 is an example of what kind of queries can be generated.

It is a Bourne shell script (batch command processor program on Unix) which echoes SQL syntax into a temporary file based upon the user's choices and then executes that shell script and pipes it through the report formatter and to the printer. This is executed from a menu within the Unify environment. This is only one of several ways that this type of report can be done. There are actually easier ways, but this was an early attempt and one easily copied.

Using Unix utilities

We use 'awk' scripts (a pattern scanning and processing language) and Unix mail to notify or remind people of the action items which have been assigned to them. Figure 8 is a diagram which demonstrates the process flow.

We have three different kinds of mailings. The first queries the data base for all new problems (those not assigned to anyone) and then divides them up by development group and mails off a report to each manager informing them of the new problems and ask that they assign someone to each one.

The second mailing looks at what reports have not yet been analyzed by an engineer, but have been assigned. Mail is then sent to the assigned engineer with the information of which problems need to be analyzed by them.

The last mailing queries for all the action items which have been assigned to someone and which have not yet been completed. It sends to the assigned engineer a list of the problems which have actions assigned to them as a reminder. It also sends to each manager, the entire list of open action items assigned to their group. This is all run once a week. Emergency bugs go through this process as well but are generally expedited with a walkthrough by the concerned party. To keep things from getting lost in a black hole, all the items which have not been assigned, analyzed, been through a CCB meeting or are incomplete with past due target dates are put together on a report once a week and go to the CCB meeting.

The possibilities are mostly limited by the resources you wish to tie up in development to enhance the problem reporting system

Administrative Problems

As usual, there were some who found fault with our system. One of the perceived limitations was the fixed field lengths. To have a description which will accommodate a lot of data, you would have to define a large amount of space in the record type just to give space to the few who need it. We have just limited our description to six lines on the screen and encourage use of ascii Unix files in a related directory for any

```
echo
echo
echo "This report brings a complete copy of the records you have chosen to the"
echo "line printer.  You may have searches made with the following keys:"
echo "Program name, responsible group, responsible engineer, "
echo "and problems which have or have not been fixed (based on closure date)."
echo
echo "Would you like to specify a program name(pn), responsible group(rg),"
echo "responsible engineer(eng), all problems(all) "
echo -n "or would you like to quit(q)?  "
read choice
echo
if test $choice = "q" ; then
        exit
else
        cp lpfull.s lpfull.i
        echo "Please specify if you want problems reports which are closed(cl)"
        echo -n "(ie: fixed and checked in) or not closed(nc) or both(b):  "
        read status
fi
if test $choice = "pn" ; then
        echo
        echo -n "Please specify program name desired(eg. phled, sch*):  "
        read prog
        prog=\'$prog\*\'
        case $status in
                "nc") echo where program_name = $prog and closure_dt \< 1/1/80 /
                "cl") echo where program_name = $prog and closure_dt \> 1/1/80 /
                *) echo where program_name = $prog / >> lpfull.i
        esac
elif test $choice = "eng" ; then
        echo
        echo -n "Please specify responsible engineer(login name):  "
        read name
        name=\'$name\*\'
        case $status in
                "nc") echo where  [[ responsible_eng = $name and analysis_date \
                        echo resp1 = $name or resp2 = $name or resp3 = $name
                        echo or resp4 = $name or resp5 = $name or resp6 = $name]
                        echo and closure_dt \< 1/1/80 / >>lpfull.i ;;
                "cl") echo where  [responsible_eng = $name or  >> lpfull.i
                        echo resp1 = $name or resp2 = $name or resp3 = $name
                        echo or resp4 = $name or resp5 = $name or resp6 = $name]
                        echo and closure_dt \> 1/1/80 / >> lpfull.i ;;
                *) echo where  [responsible_eng = $name or  >> lpfull.i
                        echo resp1 = $name or resp2 = $name or resp3 = $name ;;
                        echo or resp4 = $name or resp5 = $name or resp6 = $name]
        esac
elif test $choice = "rg" ; then
        echo
        echo -n "Please specify responsible group(sys, fe, be, sim, man):  "
        read rgroup
        rgroup=\'$rgroup\*\'
        case $status in
                "nc") echo where respn_group = $rgroup and closure_dt \< 1/1/80
                "cl") echo where respn_group = $rgroup and closure_dt \> 1/1/80
                *) echo where respn_group = $rgroup / >> lpfull.i
        esac
elif test $choice = "all" ; then
        case $status in
                "nc") echo where closure_dt \< 1/1/80 / >>lpfull.i ;;
                "cl") echo where closure_dt \> 1/1/80 / >> lpfull.i ;;
                *) echo / >> lpfull.i
        esac
else
        echo
        echo "You have not entered a valid choice, please try again."
        exit
fi
echo
echo [running]
SQL lpfull.i |RPT fr.rpt -|lpr
```

Figure 7. Example Script for Generating Reports

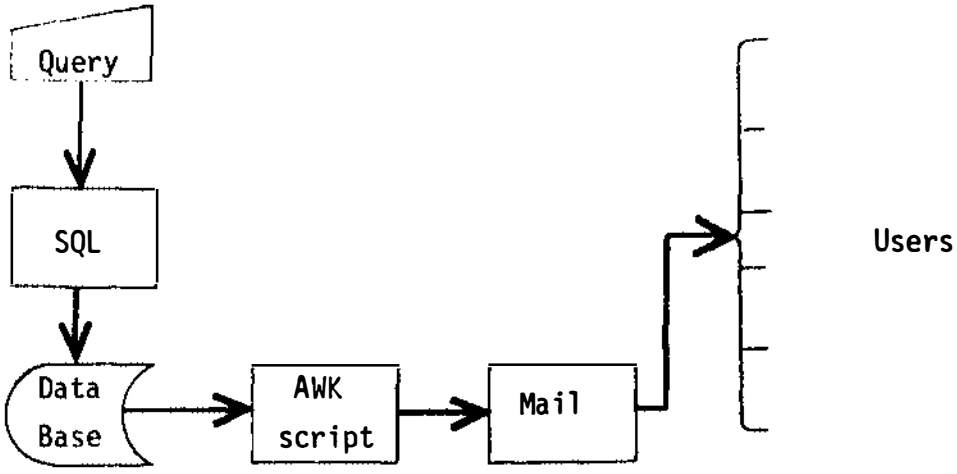


Figure 8. Process Flow for Notification Program

additional documentation. This also encourages short descriptions which are to the point, which are usually more desirable. I have found that maybe 1 of 30 reported problems really need more space. We overcame this problem by providing a directory for any additional documentation needed to describe the problem.

Also there are no editing capabilities on a field unless you program it in. Thus if you make a typing mistake, you have to retype the whole field instead of just editing the mistakes. This is irritating but not disastrous since none of the fields are larger than one screen line (80 characters).

The last perceived problem is that there doesn't seem to be a way to change screens and keep working with the same record, without going through the process of backing out to the menu, choosing the menu and the mode of operation and the problem. We have not yet found a way, although we believe the problem can be solved through the optional programming.

How Much Time Do I Have To Devote?

To design and implement this system including learning Unify took around 100 hours. I would estimate that to maintain this system on a minimal basis has taken an average of one hour a week or less. To monitor the data entered is trivial due to the reporting capabilities of Unify. It's a matter of reading a report and acting on the information.

Writing a new report is a function of what is already there. If the output format is the same, then it might take two minutes to devise an SQL script to pull the information out that you want. On the other end of the scale, to put together a report for a new record type, it will probably take four hours or more depending upon the complexity

of the information you want.

Concerning enhancements, I just added a new record type to our data base. It took me about two days (16 person hours) to enter the schema, set up the screens and set up one report.

The only other time consuming item left is CCB meetings. This is probably the most time consuming part of the system (besides actually fixing the bugs) because everyone has to discuss the problem. But it is likewise an important function because of the ideas it generates and the awareness of how the system as a whole functions as well the need for maintenance plans. This can take an hour a week if you have one meeting and run it efficiently or an hour a week per engineering group, depending on how you wish to schedule the meetings.

Conclusion

We have found the Unify DBMS system with the Unix access to be invaluable tools. Time saving in problem status tracking alone has probably amounted to the work of one full time person or more. When the tools are not used bugs seem to get lost. The flexibility of a good DBMS allows for changes like added projects, added fields, and varying reports as needs change.

This system provides a good record of the current status of projects, in terms of quality, and provides up-to-date information to customers and managers alike.

Good tools do exist, and good use can be made of them. However, they are ineffective without a joint commitment from management and engineering to produce a quality product.

BIOGRAPHY

Susan Bartlett

Susan Bartlett joined Metheus Corporation in 1983. Previously she worked with a software testing group at Johnson Controls after receiving a BS degree in computer science at the University of Wisconsin at Madison. She is the project leader for software test and evaluation at Metheus-CV, Inc., Hillsboro, Oregon.