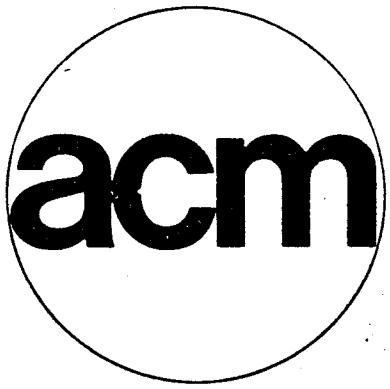
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging

Edited by
Mark Scott Johnson

1983

21680



SOFTWARE ENGINEERING Notes
Volume 8, Number 4 August 1983

SIGPLAN Notices
Volume 18, Number 8 August 1983

Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging

Edited by
Mark Scott Johnson

Pacific Grove, California
March 20-23, 1983

The Association for Computing Machinery, Inc.
11 West 42nd Street
New York, New York 10036

ISBN 0-89791-111-3

Additional copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

Price:
Members: $13.50
All Others: $18.00

ACM Order Number: 593830

ACM SIGSOFT/SIGPLAN Software Engineering Symposium

on High-Level Debugging

March 20-23, 1983

Asilomar Conference Center—Pacific Grove, California

General Chair:

Richard E. Fairley, Wang Institute of Graduate Studies

Program Chair and Proceedings Editor:

Mark Scott Johnson, Hewlett-Packard Laboratories

Program Committee:

Russ Atkinson, Xerox Palo Alto Research Center

Robert M. Balzer, Information Sciences Institute

R. Stockton Gaines, Palyn Associates, Inc.

David R. Hanson, University of Arizona

John L. Hennessy, Stanford University

Mark N. Wegman, IBM T.J. Watson Research Center

John R. White, Xerox Palo Alto Research Center

General Chair's Preface

It has been my pleasure to serve as General Chairman of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on High-Level Debugging, which was held at the Asilomar Conference Center in Pacific Grove, California, on March 21-23, 1983. The symposium was one in a continuing series of Software Engineering Symposia sponsored by ACM.

The symposium was limited to approximately 90 attendees; roughly twice that number indicated an interest in attending. Attendees were chosen on the basis of position statements submitted to the program committee. The attendees came from Canada, China, France, Germany, Italy, Scandinavia, the United Kingdom, and all regions of the United States.

Publication of these proceedings substantially increases the literature in debugging. To my knowledge, this is the first symposium on debugging since the Courant Symposium in 1970.* Many new issues and many new approaches to old issues are reported here.

These proceedings contain session summaries and papers submitted by the participants. They constitute the formal record of the symposium. The informal discussion, the live and videotaped demonstrations of debugging tools, and the spirit of collegiality among the participants were important aspects of the symposium that are not recorded here.

Thanks are due the program committee, the session recorders, and the participants who all contributed to the success of the symposium. Special thanks to Mark Scott Johnson who served as program chairman. He also handled local arrangements. Finally, a word of appreciation to the staff of Asilomar. They provided a pleasant, professional setting for the symposium.

---

*_Debugging Techniques in Large Systems_. Edited by Randall Rustin; Prentice-Hall, 1971.

Richard E. Fairley
General Chairman

Program Chair's Preface

The Program Committee met on November 8, 1982, to organize the Symposium and to select participants. One hundred twenty submissions (representing one hundred sixty people) were received, numbering over eleven hundred pages of material. To maintain the workshop character of the symposium, only half of these could be accommodated.

The submissions fell into three groups: position statements (60%), working papers (30%), and research abstracts (10%). The proceedings contains the revised papers and abstracts of all invited participants; the position statements are not included. Versions of several of the papers have been submitted for publication in referred journals.

The Symposium was organized into seven sessions based on the submissions: Debugging Methodology, Knowledge-Based Debugging, Requirements/Design Debugging, Integrated Environments, Distributed Debugging, Implementation Issues, and Demonstrations. The proceedings are likewise divided into these seven categories; each begins with a summary of the session as it unfolded at Asilomar, followed by the papers and abstracts that most relate to the session. It should be noted, however, that few papers were formally presented at the Symposium. Most sessions were organized as panels, short papers, or directed group discussions.

My thanks to the members of the Program Committee, who survived admirably the grueling one-day long committee meeting. Despite the difficulty of running a meeting of so many talented (and opinionated) people, much was accomplished in relatively short time. I trust it was their expert preparation, and not my tyrannical leadership, that made this possible.

Seven graduate students graciously consented to act as session recorders during the Symposium. Their thoro and professional efforts are greatly appreciated:

- Peter C. Bates, University of Massachusetts
- Wayne C. Gramlich, MIT
- Thomas Gross, Stanford University
- Michael S. Kenniston, Stanford University
- Insup Lee, University of Wisconsin, Madison
- Mark A. Linton, University of California
- Janice Cynthia Weber, University of Toronto

Bert Raphael, Director of Hewlett-Packard's Computer Science Laboratory, supported the Symposium thru both my time and subsidy of the Symposium's expenses. My thanks to him and to Barbara Alles, who assisted me with the veritable deluge of correspondence that emanated from my computer.

I approached Dick Fairley with the idea for the Symposium over two years ago. He enthusiastically accepted the idea. We worked over the phone and thru the mail for a year and a half before meeting in person, however. It has been a pleasure and an honor to work with Dick.

Mark Scott Johnson
Program Committee Chairman

ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging

Symposium Schedule

Sunday, 1983 March 20

3:00- 6:00PM	Check-In
6:00- 7:30PM	Dinner
7:30-10:00PM	Reception

Monday, 1983 March 21

7:30- 8:30AM	Breakfast
8:30-10:15AM	Session: Debugging Methodology, R. Stockton Gaines
10:15-10:30AM	Coffee Break
10:30-12:00N	Session: Knowledge-Based Debugging, Robert M. Balzer
12:00- 1:30PM	Lunch
1:30- 3:00PM	Session: Integrated Environments, Richard E. Fairley
3:00- 3:30PM	Coffee Break
3:30- 5:00PM	Cedar Programming Environment videotape, Warren Teitelman
5:00- 6:00PM	Free Time
6:00- 7:30PM	Dinner
7:30-10:00PM	Demonstrations

Tuesday, 1983 March 22

7:30- 8:30AM	Breakfast
8:30-10:00AM	Session: Distributed Debugging, John R. White
10:00-10:30AM	Coffee Break
10:30-12:00N	Session: Distributed Debugging, John R. White
12:00- 2:30PM	Lunch and Free Time
2:30- 4:00PM	Session: Requirements/Design Debugging, Robert M. Balzer
4:00- 4:30PM	Coffee Break
4:30- 6:00PM	Session: Methodology, R. Stockton Gaines
6:00- 7:30PM	Banquet
7:30-10:00PM	Demonstrations

Wednesday, 1983 March 23

7:30-12:00N	Check-Out
7:30- 8:30AM	Breakfast
8:30-10:00AM	Session: Implementation Issues, Mark N. Wegman
10:00-10:30AM	Coffee Break
10:30-12:00N	Session: Implementation Issues, Russ Atkinson
12:00- 1:00PM	Lunch

ACM SIGSOFT/SIGPLAN Symposium on High-Level Debugging

Table of Content

Implementation Issues

Demonstrations

Debugging Methodology

(Session Summary)

Wayne C. Gramlich

M.I.T.

Panel Participants:

R. Stockton Gaines (moderator)
Dan Moore
Elaine Weyuker
Elliot Soloway
Richard Hamlet
Jonathan Goldblatt

Two sessions were held on the general topic of debugging methodology. Both were moderated by R. Stockton Gaines. During the first session five speakers presented short talks on their workshop submissions. Each talk was followed by a brief question and answer period. The format of the second session was that of an open discussion. The talks presented in the first session have been briefly summarized. The remaining notes of these two sessions have been arranged topically rather than chronologically.

1. Talk Summaries

The first debugging methodology session was opened with the following list of questions:

1. What is hard about debugging?
2. What are programmers' thought processes during debugging?
3. How can bugs be classified?
4. How do people approach debugging?
5. Why are debugging facilities used as little as they are?
6. Why has so little happened in the field of debugging in the last ten years?
7. How does high-level debugging differ from low-level debugging?
8. How does the programming environment (batch, interactive, real-time, concurrent, distributed, etc...) affect debugging?

Neither question six nor question eight received much discussion during the two sessions.

Dan Moore from Bell Laboratories addressed the questions of "why is debugging hard?" and "what are programmers thought processes during debugging?" His observation was that debugging appeared to require both analytic and intuitive thought modes. See his workshop submission for more details.) Recent psycological evidence suggests that analytical thought occurs in one hemisphere of the brain while intuitive thought occurs in the other. It was Moore's hypothesis that debugging is hard because it is difficult to transfer the locus of thought from one hemisphere to the other.

Elaine Weyuker from New York University addressed the question of "How do people approach debugging?" She collected data on the kinds of bugs made by a group of three professional programmers working for an industrial firm. She pointed out that there is very little of this kind of information in existence. For this particular software project no debugger was used despite the fact that a debugger existed. 90% of the bugs were found by desk checks and test runs, while the remaining bugs were found using program dumps.

Elliot Soloway from Yale University addressed the question of "How can bugs be classified?" He examined the bugs made by novice programmers and concluded that bugs could

not be classified without knowing what the programmer had in mind.

Richard Hamlet from the University of Maryland addressed the question of "How does high-level debugging differ from low-level debugging?" He contended that low-level debugging provided the user with a great deal of data but relied on the user to extract useful information from the data. Program dumps and DDT (an interactive memory examination program) were examples of low-level debugging. He suggested that high-level debugging would have the computer extract the useful information about the bug for the user. In particular, the user would deal with the debugger in the same abstractions in which the program was written.

Jonathan Goldblatt from Intermetrics discussed the experience gained from debugging the software for the NASA space shuttle. The space shuttle requires high reliability software because a catastrophic software failure could potentially cause the loss of a billion dollar piece of hardware. The approach taken was to use system simulation to extensively test the shuttle software.

2. Major Issues

The major issues discussed in the two debugging methodology sessions are listed in the following paragraphs.

How can bugs be classified?

Early in the workshop people classified bugs into the two broad categories of easy bugs and hard bugs. An easy bug is relatively easy to find and a hard bug is relatively difficult to find. Elaine Weyuker's talk presented information that showed most bugs were found and corrected in approximately an hour. There is another set of bugs that takes on the order of a day to find and correct. Someone in the audience suggested that an 80/20 rule might apply to program bug classification. That is, 80% of the bugs are easy bugs and 20% of the bugs are hard bugs. However, since hard bugs take longer to find and correct, the programmer can spend more time trying to find and correct hard bugs rather than easy bugs. Several participants suggested that a frequent cause for a hard bug is that the programmer develops a mind set as to how the program is supposed to work when in fact it works differently.

Is bug classification useful?

Someone claimed that bug classification is not useful because the bug classifications seem not to be a useful aid in developing debuggers. Someone else pointed out that simply classifying bugs is not enough. It is also necessary to know how frequent each bug class occurs before bug classification can be used to decide what features to insert into a debugger.

What is hard about debugging?

While there was general agreement that debugging is hard, there is very little consensus as to why it is hard. Here is a list of suggested reasons:

- Debugging is hard because it requires repeated switches between intuitive and analytic thought modes.
- Debugging is hard because programmers develop a mind set that their program should work one way when in fact it does not.
- Debugging is hard because the available debugging tools are not adequate.
- Debugging is hard because the program semantics change as the user finds and corrects bugs. This makes it difficult for the user to develop a consistent model of program behavior.

What is high-level vs. low-level debugging?

While the workshop was on high-level debugging, it was very difficult to come up with an agreed upon definition for high-level debugging. There seemed to be a consensus that low-level debugging consisted of things like program dumps and DDT. Someone belatedly suggested that high-level debugging is anything that is not low-level debugging. Someone else suggested that high-level debugging is debugging in the problem domain whereas low-level

debugging is debugging in the solution domain (i.e. programming language.) Yet someone else suggested that high-level debugging consists of tools added to a low-level debugger to make debugging easier. Someone suggested that an important distinction is whether the debugging occurred in the same programming language or below the programming language level.

Why are debugging facilities used as little as they are?

No one really contested that debugging facilities were used very little. While most people in the audience regularly used debuggers, the audience did not represent a typical selection of programmers. Some suggested reasons for why debugging facilities are not used are:

- In many situations the debugger simply does not exist. Alternatively, sometimes the debugger and the program to be debugged cannot both be put into the same address space.
- Debuggers frequently display the wrong kind of information (for example, displaying a string as hexadecimal bytes.)
- Debuggers are frequently designed long after the language system has been designed. This results in a debugger that is not integrated into the system.
- The debugger frequently does not use the same syntax as the programming language. This makes the debugger more difficult to learn to use.
- Sometimes using the debugger modifies the program behavior. This is the Heisenberg Uncertainty Principle as applied to debugging (an instance of such a bug was called a "Heisenbug" by one participant.)

Is it possible to make a portable debugger?

There was a mixed reaction to whether debuggers can be made portable in the same sense that languages are made portable. Some people claimed success in porting a debugger and other people did not succeed. One person explained that their group was unsuccessful in porting a debugger to some machines because the various machines did not support needed features. Breakpoints were used as an example. Someone made the comment that the only way that their group succeeded in porting a debugger was to design the debugger and the language implementation at the same time. Someone else commented that porting debuggers would probably get harder as the debuggers became more sophisticated in capabilities.

What unusual tools were useful for debugging?

Someone in the audience asked the question of whether anyone used some tools for debugging that were not explicitly designed for the task. Someone displayed the binary image of the program directly onto a bit-map screen. Someone else found that a program history management system was useful. In one case a program that translated a program into English was also useful. There were several comments that graphically displaying the state of the program in real-time was a useful feature.

DEBUGGING "LEVEL": STEP-WISE DEBUGGING

Dick Hamlet

Department of Computer Science
University of Maryland*
College Park, MD 20742

Abstract

Debugging techniques originated with low-level programming languages, where the memory dump and interactive word-by-word examination of memory were the primary tools. "High-level" debugging is often no more than low-level techniques adapted to high-level languages. For example, to examine the execution of an operational specification one state at a time by setting breakpoints, is superior to doing the same thing to a machine-language program, but only because the language level has improved; the debugging remains primitive. This paper attempts a radical definition of debugging level, and illustrates it with a technique for ordering the execution of concurrent processes in a way that follows their design structure.

Division of a program into a collection of cooperating processes is a means of controlling the complexity of each process. However, in execution the program-development structure is ignored, with the result that the advantages of decomposition are lost. What the designer has divided and conquered, the debugger sees as an overwhelming monolith. The technique proposed here causes the focus of execution to follow the design structure, in a way that does not require detailed user direction.

1. Debugging Level

The two major low-level debugging techniques are memory dumps and interactive examination of memory. Both require breakpointing, the ability to temporarily halt an executing program, to be effective. Almost every system has a "snapshot" dump facility; today many also have some variant of DDT [4]. These techniques can be adapted to high-level languages in an obvious way: dumps can be formatted according to storage declarations and can show execution history in source terms [9]; interactive debuggers can employ type information to display the source program's view; both can set breakpoints based on complex conditions, perhaps using expressions written in the source language itself. However, the debugging paradigm of halting just before some difficulty occurs and examining everything to see what might be wrong, is the same at any language level, and can only be called low-level debugging.

In attempting a definition of debugging "level" it is easiest to characterize the existing low level:

a) The person involved must understand the details of execution sequence, in order to properly set breakpoints.

b) The information interchange between person and computer system is extensive; this is obvious in the case of a dump, but interactive sequences of long duration are no improvement.

c) Computer processing of the information presented to the person is minimal, little more than formatting.

Two improvements can occur when the language level is raised:

i) Breakpoints can be set "associatively," to halt the program when a condition expressed in term of the program state occurs rather than when a particular control point is reached. This is a partial answer to point a) above.

*Part of this work was performed while on leave at Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia. It was partially supported by AFOSR grant F49620-80-C-0004.

ii) Debugging commands may be given in source terms, with the full power of the programming language available. For example, to trace down a linked list, the debugger might use a program fragment that moves from link to link just as the source program would, printing nodes as it goes. This bears on point c).

In the repair of a given bug, the debugging level is determined by the information interchange between person and program. It rises when less information is exchanged. This definition is imprecise, but can serve as a guideline since it concentrates on point b) above.

Consider the case of a concurrent-process system. It is low-level debugging to "just let it run" and study the history of execution interleaving and communication, because far too much information passes from program to user. It is also low-level for a person to interactively control the scheduling and communication, because too much information passes in the other direction. In the remainder of this paper a high-level approach is described, in which each process is isolated from the others, and a user implicitly controls execution by describing the process structure, without knowledge of execution details.

2. Decomposition of Programs

"Divide and conquer" is acknowledged to be the best strategy for combating program complexity. In its least structured form, this philosophy involves dividing a program into "modules," each so small that it can be understood easily. Unfortunately, a bad division can be counterproductive in that the data interfaces between modules are so broad, and the control connections so chaotic, that understanding of the whole does not come from understanding of the parts. The method of structuring decomposition considered here is cooperation between autonomous processes. (However, the technique may also be applied to step-wise refinement [1].)

Cooperating sequential processes can be viewed as complete programs that communicate by passing messages [2]. A graph in which nodes represent processes (programs) and the arcs are potential message-interchange connections describes the system structure. The system complexity is more likely to reside in the nodes than in the communication connections. In the extreme case represented by UNIX pipes [3] the graph is a chain, and the nodes may be sophisticated programs. The idea of process decomposition is that each process can be understood alone, and the behavior of the whole is obvious given that of the parts. Ideally, understanding an N-module system requires only dealing with a fixed number K (K << N) of modules at a time.

During system development, each module can be assigned to a separate designer or programmer, who needs only information about immediately connected modules. Units are difficult to debug in isolation because intermodule interfaces are less precise than actual designs or code for the other modules. The most common schemes are to wait for all modules to be completed, and test them in a bottom-up fashion so that all communications are supplied as they will actually be in the complete system, or to test top-down using stubs that do not behave much like the missing modules, but merely (say) announce their presence. This paper considers how to do a better job of top-down debugging. The essence of the idea is to use modules in isolation, in stages: the information that should be supplied by missing modules is given at random in stage 0, creating outputs that are used as inputs in stage 1, and so on until the process converges (if it does) to a useful test.

Step-wise debugging requires the support of a bookkeeping tool, because the volume of data and its proper labeling is difficult for a person to handle. In some cases it is useful for a human user to monitor details of the debugging process; more often it is desirable that all detail be hidden, for which machine assistance is essential. When a user supplies only the design structure and allows a machine system to control execution, then the technique is truly high-level, since no execution detail is supplied by the use, and none conveyed to him.

3. Step-wise Debugging of Cooperating Sequential Processes

A module-communication graph does not wholy capture the structure of processes in so-called real-time systems. In such systems each process is cyclic (perhaps implemented using a never-ending loop) with its body devoted to interacting with other processes and performing calculations based on those interactions. The "output" of a process is the messages it sends to other processes, and its "input" is received from them. The external world can be thought of as another process interacting with the computer system. (This view is probably due to Fitzwater [5] and has been used as the basis of a specification language [6].)

An important feature of cooperation among processes is synchronization of communication, which can be incorporated in message passing by allowing a process to test for a waiting incoming message. The details of message-passing primitives are not critical to the method presented here, but choosing particular primitives makes the description better. Therefore, assume that communication occurs as follows: (1) processes communicate by name; (2) process output is immediate, and is not queued if not accepted before another output; (3) process input waits for the corresponding output; (4) a process may test for input waiting. To combine separately executed processes so that their messages could actually be exchanged during execution, it is sufficient to associate a time stamp with each message and input-waiting test.

Step-wise debugging is defined in a series of stages, as follows. Given a system input, at stage 0, each process body is executed once in isolation, with incoming messages randomly generated (but see below), and decisions about waiting input treated as two-valued random choices. Each message and choice is stamped with the execution time of the process at which it occurs. It may happen that

some stage-0 outputs can serve as inputs later in stage 0 (and similarly, later stages may sometimes use the outputs from earlier stages) if the time stamps permit. If process S sends a message to process R stamped t, it can be used by R at time u only if all of R's input-waiting tests on S before t and before u have failed, and all those after t and before u have succeeded. Should such an output be used, as input it is time stamped the greater of t and u, reflecting the fact that R had to wait for it. Should t be greater than u, execution time for R is also updated to t. At the end of stage 0, two lists are available by module: outputs produced, and inputs consumed. Most items in these lists are the consequences of random choices, and so must be marked as dubious.

At stage 1, the stage-0 input-consumed lists are considered for each process. Each entry is required as output from another process P. P is then executed (using stage-0 and earlier stage-1 values where available, otherwise generating inputs at random) until the needed value is produced. (This may require more than one repetition of P's body, and indeed the necessary message may never be forthcoming, as discussed below.) When a needed message occurs, the calculation in which it is to be used is repeated, including the necessary revisions in time stamps discussed above. As examination of the stage-0 input-consumed list proceeds, a new stage-1 list of input consumed and output produced is amassed, in which some of the dubious marks from stage 0 do not occur. Continuing in this way, at stage N there is an input-consumed list from stage N-1, to be sought using the most recent messages where possible, then substituted into the most recent earlier computations. This may force yet earlier substitutions, and creates the two lists for stage N.

Step-wise debugging may terminate even though execution of a set of cooperating processes does not. That is, a stage may be reached at which all the dubious marks have been eliminated, and no new messages are needed. This means that a particular sequence of interactions has been discovered that is time-consistent, and so represents a possible actual execution of the system. Each process body has been executed an integral number of times, and no inputs are missing. Continuing to execute any process may begin a new (possibly different) cycle.

The execution sequence that results from a given system input depends on the order in which processes receive control. This scheduling is implicit in step-wise debugging, determined by a user-supplied process structure. If the process structure is a tree, then the modules are considered at each stage in breadth-first order. Those near the root of the tree dominate the execution sequence in the sense that other processes "give them what they want." It is this implicit scheduling control that eliminates most of the detailed information interchange between program and user, and makes the method high-level according to the definition in Section 1.

When a process P should produce a message at stage N, it can fail, either because the message cannot be produced, or because P has not itself been given the proper messages to yield the needed result. In the former case a bug has been found in P; this may be true in the latter case as well, but the difficulty may be deferred by copying the entry that caused the trouble to the input-consumed list at stage N, so that it will be tried again at stage N+1.

To illustrate step-wise debugging, each process may be written as a Pascal-like program, named by a "process header." Communication is via the procedures:

```
send(link, message)
```

with the meaning that "message" is transmitted to process "link", to be overrun if not received before another send, and the caller immediately resumes control; and,

```
receive(link, variable)
```

meaning that the caller P waits until process "link" has directed a message to P, whose contents are available in "variable".

Synchronization uses the Boolean function

```
ready(link)
```

which returns TRUE if process "link" has sent the caller a message that may now be received without waiting, otherwise FALSE. These constructions may be used with a process named EXT, not provided explicitly, representing the system environment. Thus receive(EXT, ...) is an external input, and send(EXT, ...) an output.

The three-process system {A, B, C} below illustrates the effect of structuring on step-wise debugging. The processes attempt to distribute work among themselves. Constants and variables are of a suitable "message" type that is irrelevant to the example.

```
process A;
const free, busy;
var x
begin {wait for work sent by process C}
  while true do
    begin
    send(C, free);
    receive(C, x);
    send(C, busy);
    {do the work indicated by x)}
    end
end.
```

```
process B;
const free, busy;
var x;
begin {if process C has sent work, do it;
       otherwise get work from EXT}
  while true do
    begin
    send(C, free);
    if ready(C) then
      receive(C, x);
    else
      receive(EXT, x);
    send(C, busy);
    {do the work indicated by x}
    end
end.

process C;
const free;
var x;
begin {parcel out work to A and B when
       they can accept it, or do it here}
  while true do
    begin
    receive(EXT, x);
    receive(A, state);
    if state=free then
      send(A, x)
    else
      begin
      receive(B, state);
      if state=free then
        send(B, x)
      else
        {do the work indicated by x}
      end
    end
end.
```

This system is probably intended to have C at the top level, with A and B subordinate. Using that structure, and with an input represented by x y z , where x takes two units of time to process, y takes 4, and z takes 5, measured in statement count units, at stage 0 we have:

Time	Process C	A	B
1	EXT -> x	free -> C	free -> C
2	A -> busy	C -> ?	"ready C
3		busy -> C	EXT -> y
4	B -> free	(do ?)	busy -> C
5			(do y)
6	x -> B		

where the notation "P -> m" in a process column means that P sends message m to this process; "m -> P" means that this process sends m to P. At stage 0, only the actions at time 1 are accurate; the others are generated by random choices. At stage 1 we have:

Time	Process C	A	B
1	EXT -> x	free -> C	free -> C
2	A -> free		"ready C
3			EXT -> y
4	x -> A	C -> x	busy -> C
5	EXT -> z	busy -> C	(do y)
6	A -> busy	(do x)	
8	B -> busy		
10	(do z)		

where now all actions are solid except the last three entries under C. At stage 2 these turn out to be correct, and the procedure is complete. This display is similar to one that could have been obtained by just letting the system run, or by controlling the scheduling explicitly [7]. The difference is that because process C is treated first, it dominates the execution pattern. From the viewpoint of high-level debugging the more important point is that such a table would not be printed at all, but merely summarized (here perhaps as:

Input	A	B	C
x y z	x@6	y@5	z@10

or showing explicit output only). Thus the information interchange is drastically reduced.

If we restructure the same system so that B is dominant, at stage 0:

Time	Process B	C	A
1	free -> C	EXT -> x	free ->C
2	ready C	A -> busy	C -> ?
3	C -> ?		busy -> C
4	busy -> C	B -> free	(do ?)
5	(do ?)		
6		x -> B	

with only the time-1 actions sure. At stage 1:

Time	Process B	C	A
1	free -> C	EXT -> y	free ->C
2	"ready C	A -> free	
3	EXT -> x		
4	busy -> C	y -> A	C -> y
5	(do x)		busy -> C
6			(do y)
7	free -> C		
8	ready C		
9	C -> ?		
10	busy -> C		
11	(do ?)		

which is sure through time 7. The summary result is:

Input	A	B	C
x y z	y@6	x@5 z@11	

rather different than when the probable design

structure was followed. Again note the limited exchange of information to obtain the new case: the user has only to specify that B dominates C and A.

4. Computer Support

Even the trivial examples of section 3 show that step-wise debugging cannot be attempted without automatic bookkeeping. A support implementation is straightforward. The techniques of self-contained interpreter, preprocessor, or run-time library with compiler modification are all applicable. For a conventional programming language (and for most program design languages) the run-time library approach can be easily added to a conventional debugging system with procedure-call tracing. Since each message is transmitted by a procedure call, and since the trace routine gets control at each call, it can be modified to generate values and record the required lists. Instead of allowing the program itself to execute, a phony main program is added that invokes the instrumented procedure code, according to the method's stages. Essentially the same techniques have been used for an automatic testing system [8], and in an concurrent-process debugging system using formatted dumps [9].

5. Summary

A method of high-level debugging has been described which structures the usual chaotic process of testing a large piece of software. The resulting test run emphasizes processes at the top level of the system's structural description. The method suggests a straightforward computer tool to support the extensive bookkeeping required.

References

1. N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

2. C. A. R. Hoare, Communicating sequential processes, CACM 21 (Aug., 1978), 666-677.

3. D. M. Ritchie and K. Thompson, The UNIX time sharing system, CACM 17 (July, 1974), 365-375.

4. A. Kotok, DEC debugging tape, Technical Memo MIT-1, Massachusetts Institute of Technology, 1961.

5. D. R. Fitzwater, A decomposition of the complexity of system development processes, COMPSAC 78, Chicago, 424-429.

6. P. Zave, An operational approach to requirements specification for embedded systems, IEEE Transactions on Software Engineering SE-8 (May, 1982), 250-269.

7. P. Zave, Testing incomplete specifications of distributed systems, Proc. ACM Symposium on Principles of Distributed Computing, Ottawa, 1982, 42-48.

8. R. G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering SE-3 (July, 1977), 279-290.

9. R. G. Hamlet, Single-language, small-processor systems, Information Processing 77, B. Gilchrist, Ed., North Holland (1977), 969-974.